GQUERY : A QUERY LANGUAGE FOR GML

OMAR BOUCELMA François-Marie Colonna

LSIS, UMR-CNRS 6168, University of Provence
France

ABSTRACT

XML has established itself as the standard for representing data in scientific and business applications, leading to several dialects and tools for XML data processing. In this paper we are interested in querying geographic data encoded in GML, the Geography Markup Language adopted by the OpenGIS consortium (OGC) for the description, transmission and storage of geographic data. Although it is anticipated that GML will appeal to a large class of GIS users, OGC did not recommend a specific language for querying GML data. On the other hand, the database community, in both the academia and the industry, has been very pro-active in the design and implementation of XQuery, a query language for XML. In this paper, we present our XQuery based approach and system for querying GML data.

1. INTRODUCTION

XML has established itself as the standard for representing data in various applications, leading to several dialects and tools for XML data processing. The Geographic Information Systems (GIS) community, by way of the OpenGIS consortium *(Opengis Consortium 2003)* has adopted the Geography Markup Language (GML) - an XML encoding - for the transport and storage of geographic information. GML is the XML representation of an abstract spatial model defined by the OGC. The OpenGIS did not recommend a specific language for querying GML data. On the other hand, the database and Web communities, both in the academia and the industry, have been very pro-active in the design of a standardized query language for XML, known today as XQuery *(World Wide Web Consortium, 2003)*. In this paper, a spatial query language named GQuery is described. The language is based on XQuery and allows users to perform both attribute and spatial queries against GML documents. Spatial semantics is extracted and interpreted from XML tags without affecting XQuery syntax. GQuery may be used either for parsing spatial queries on GML files, or as a query language in the context of a geographic data integration system, to query several distributed heterogeneous GIS as it is currently deployed in the VirGIS *(O. Boucelma, Z. Lacroix and M.Essid 2002)* mediation system. The paper is organized as follows. Section 2 presents GML and highlights the issues faced when attempting to query GML data. Section 3 describes GQuery: examples of spatial operators and queries are given, and performance issues are illustrated. Section 4 presents our prototype, the languages and tools used for development purposes. In Section 5, we describe the role of GQuery in the VirGIS heterogeneous GIS integration system. A conclusion is presented in Section 6.

2. ISSUES IN QUERYING GML DATA

Geography Markup Language is an XML grammar written in XML Schema *(World Wide Web Consortium, 2003)* for the modelling, transport, and storage of geographic information.

GML data model

The key concepts used by GML to model the world are drawn from the OGC Abstract Specification. The basic concept is a Feature, i.e., an (object) abstraction of the real world phenomena, with spatial and non-spatial attributes. Figure 1 shows a town cut in four districts:

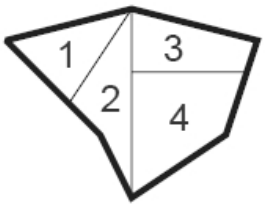

Figure 1: Town's districts

Figure 2 shows the UML schema of the town example used in this paper, with respect to the OpenGIS abstract model. Town and parcels inherits from Feature, and parcel has a Geometric property.

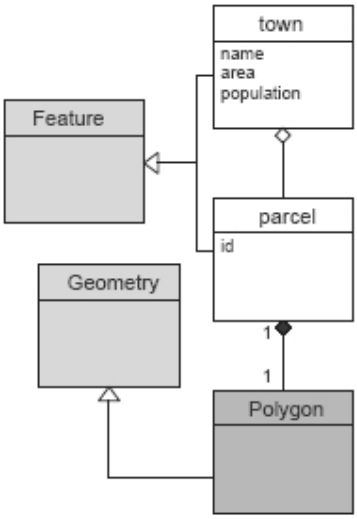

Figure 2: UML modelling of town's districts

A simple encoding of these features with GML only shows containment relationships between town and districts, as parcels tags are nested in town tag :
<town name='NY' area='500' population='10000'>
<parcel id='1'><polygon>...</polygon></parcel>

<parcel id='2'><polygon>...</polygon></parcel>
<parcel id='3'><polygon>...</polygon></parcel>
<parcel id='4'><polygon>...</polygon></parcel></town>
Not all information represented in Figure 1 is available in GML. For example, adjacency relationships between parcels need to be first defined in an XML schema, and then encoded in the GML document. When working with complex examples of thousands of features, expressing all relationships between them is prohibitive because it increases GML document complexity and size. What is needed is a query language for GML with spatial operators that capture the spatial semantics in GML code, in highlighting relationships not expressed explicitly in GML.

Related Work on Querying GML

Querying spatial data is a well known problem studied in the context of SQL and relational DBMS and resulted in languages such as SpatialSQL *(M. Egenhofer 1994)*, GeoSQL *(C.H. Wang Feng 2000)* or Oracle Spatial (*Oracle Corporation 2003)* born from extensions of SQL. They are slightly syntactic and semantic extensions of the original language. The choice of extending an existing language, instead of creating a new one is motivated by the fact that spatial databases contain both spatial and non-spatial data. Querying GML data is the same problem as seen before in relational databases. The query language for GML should be developed on the basis of the languages developed for XML. Several solutions already exist for querying GML data. Opengis Web Feature Service requests (WFS) *(Opengis Consortium 2003)* are used for describing data manipulation operations on geographic features posted to a Web Feature Server using HTTP requests. The Web Feature Server layer is in charge of manipulating data sources.

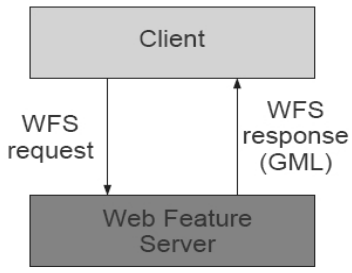

Figure 4: Web Feature Service

Web Feature Service interfaces allow geographic data extraction that is returned to the user as a GML document. When accessing a source, one may execute spatial operators provided by the underlying GIS. There are mainly two kinds of requests (GET and POST). The examples below query all instances of type “TOWN” in the spatial database. The first method uses keyword-value pairs (KVP) to encode the various parameters of a request:
http://www.someserver.com/wfs.cgi?service=WFS&version=1.0.0&request=GetFeature&typename=TOWN
The second method uses a specific XML encoding as a query language:
<?xmlversion="1.0"?><GetFeature version="1.0.0" service="WFS">
<Query typeName="TOWN"/></GetFeature>

In both cases, HTTP URLs or XML encoded WFS requests can express selections of data. Their expressive power is not equivalent to SQL or XQuery, and it is not easy to write and read such queries. WFS is not able to join data from two distinct GIS. Some spatial operators are available (equals, disjoint, crosses...) but their use depends on the underlying GIS capabilities. Using a query language over GML solves these problems. If a spatial operator is not available at a GIS data source, it can be applied over the GML document obtained from a data source through its WFS server. When executing an external join between two GIS, we only need to join two GML documents retrieved by the two local WFS servers: this is much more comfortable in case of heterogeneous data integration. When querying GML data, we need to perform spatial analysis (area of a polygon, length of a line) and highlight spatial relationships (containment, overlapping…). Languages that are currently used for querying XML data are fully dedicated to tree manipulations, can only get alphanumeric features and are not suitable for spatial calculus. XPath, and more recently XQuery, do not highlight the spatial properties of the set of tags representing Points or Polygons. With currently available tools, if we have two subsets of a GML document (Figure 7), that represent respectively a fountain (f) and a park (p), we can not test if the fountain lies within the park. We can only select parts of the two GML sub trees.

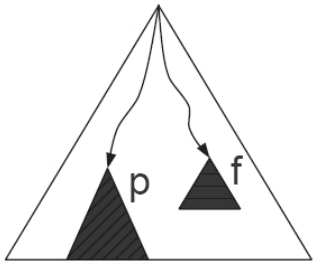

Figure 7: Relationships between subtrees

The GML tree structure is not suitable for geometric manipulation, and representing geometric data poses structural problems, like the number of tags needed for describing geographic features. This can led to use GML for data exchange only, but recent work in GIS data integration such as those conducted in the VirGIS project, increased the need of a query language for GML. Proposals were made for extension to spatial data of existing XML query languages *(Ranga Raju Vatsavai 2002)*, or extension of existing data models to spatial domain (*P.G and J.Corcoles 2002)*. Actually none of them gave birth to a functional implementation. One solution for querying GML is to come up with an implementation of all spatial functions in XQuery itself, but this is quite difficult, because XQuery is a functional language that doesn't offer data structures for implementing geometric algorithms. Another solution could be the translation of a complete GML document into another language, in order to easily manipulate their properties (for example by the translation, and insertion of a GML document into a database like PostGIS *(Postgis (2003)*). This could be possible for small documents, but becomes unfeasible with large documents that may contain thousands of features; first of all, because of the translation time, and we also lose the benefit of using XQuery for tree navigation, querying and merging XML datasets. The solution we propose consists

in adding spatial operators to XQuery, in order to capture the spatial semantics of a GML document.

3. GQUERY DATA MODEL

The data model we use is very simple, and is an extension of XQuery's one (Figure 8).Extensions of another data model for XML *(D.Beech, A.Malhotra and M.Rys 1999)* have been proposed (*P.G and J.Corcoles 2002)*; but an inconvenient is that implementation of the database model and the algebra has to be done from scratch.
As XQuery is about to be recommended by the W3C, we shall stick to the standard.
A GQuery query is composed of expressions. Each expression is made of built-in or user-defined functions. An expression has a value, or generates an error. The result of an expression can be the input of a new one. A value is an ordered sequence of items.
An item is a node or an atomic value. There is no distinction between an item and a sequence containing one value.
There are seven types of nodes: document, element, attribute, text, comment, processing-instruction and namespace nodes.
Writing a query consists of combining simple expression (like atomic values), path expressions (from XPath), FLOWER expression (For-Let-Where-Return), test expressions (it-then-return-else-return), or (pre or user defined) functions.
Non spatial operators are arithmetic operators (+, -, ×, /, mod), operators over sequences (concatenation, union, difference), comparison operators (between atomic values, nodes, and sequences), and boolean operators.

query ::= expression
expression ::= expression ○ expression | value | ERROR
value ::= (item_0, item_1,..., item_i)
item ::= atomic value | node

Figure 8: GQuery Expressions

Spatial operators are applied to sequences. We have three types of spatial operators, the first two categories perform spatial analysis, the third highlight spatial relationships:
operators that return numeric values:
area, length : sequence=(node) → numeric value
distance : sequence=(node, node) → numeric value
operators that return GML values :
convexhull, centroid : sequence=(node) → node
operators that return boolean values :
equal, within, touches : sequence=(node,node) → boolean
(in definitions above, node is a GML data node)

Each result of a GQuery expression is part of the data model. The input and output of every query or sub-expression within a query is an instance of the data model. GQuery is closed under this query data model. When computing area(node), if node is a Polygon, the function returns a numeric value, otherwise it raises an error. In both cases, results are instances of the data model. Spatial operators can be nested.

For example, within(convexhull(node1),node2) is correct according to the data model. convexhull(node1) is equal to node, and within(node,node2) returns a boolean.

Examples of Operators

Calculus of the city hall area
for $var in document('city.xml')/City/CityHall where $var/@number = '1'
return geo:getArea($var)

and the result is: 200.0

Calculus of the city hall centroide
for $var in document('city.xml')/City/CityHall where $var/@number = '1'
return geo:getCentroid($var)

wich gives:
<wfs:Point>
<wfs:coordinates>17.333333333333332, 84.66666666666667</wfs:coordinates>
</wfs:Point>

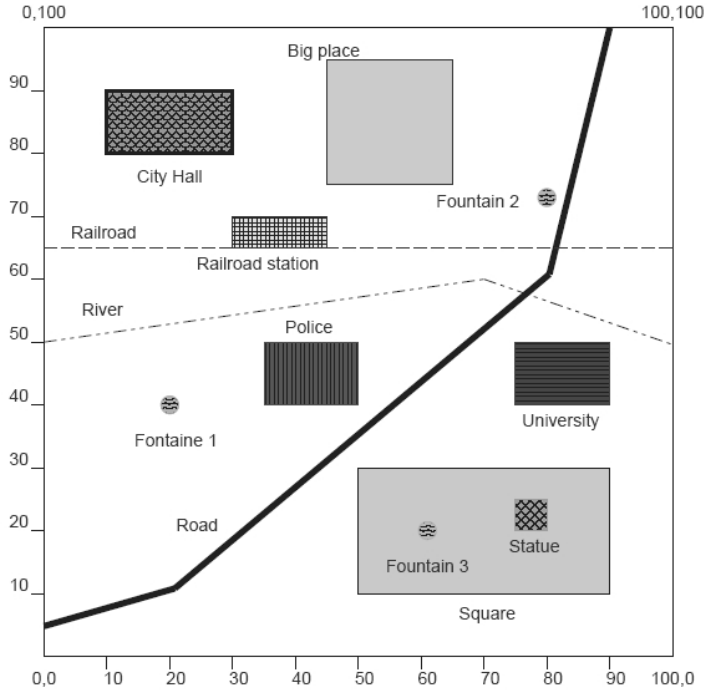

Figure 9: Cambridge's map

4. THE GQUERY PROTOTYPE

To illustrate operators and GQuery usage, we give examples of queries posed against the cambridge.xml document depicted in Figure 9. A portion of the corresponding GML document is represented in Appendix A.

As explained before, trees and sub trees manipulations are devoted to XQuery, while spatial processing is performed by means of geometry functions. In order to add spatial operators to XQuery according to the previous data model, we use external function calls. XQuery grammar *(W3 Consortium 2003)* proposed by W3C provides many built-in functions and the language allows user defined functions, especially external ones which we are using. A function is defined by its prototype and its content, like in Java or C++. Variable names in XQuery are defined using a qualified name (QName); a function name is a variable, so it is constructed using a QName, which is composed of a Prefix (optional) and a LocalPart (name of the function). For example, function geo:getArea computes the area of a geometric object. We use XQuery properties in order to manipulate GML data, and XQuery external (library) functions support for spatial calculus. XQuery is used for tree manipulations, selections of sub trees, transformations and reconstruction of XML documents.

Querying GML documents requires a translation step when calling a spatial function. Indeed, since we can not directly pass an XML tree as input of a spatial operator, we need to perform XML data marshalling. The translation step first flattens the GML tree into a list of coordinates, which can be used by a geographic library to compute spatial methods. We implemented a prototype using an XQuery parser, coupled with Java Topology Suite (JTS), an Open Source API providing spatial object model and fundamental geometric functions. It implements the geometry model defined in the OGC Simple Features Specification for SQL *(Opengis Consortium 2003)*.

When executing a query that contains a spatial operator, the parser needs to convert GML data into a JTS compatible format. This transformation is performed in two steps as described in Figure 10. First, we use SAX2 to transform GML tags and we obtain a Well Known Text Format string. The OpenGIS Consortium “Simple Features for SQL” specification defines the “well-known text"' representation to exchange geometry data in ASCII format. For example, the city hall extracted from Appendix A is transformed in POLYGON((10 80,10 90,30 90,30 80,10 80)). Then this string is read by a WktReader that creates the corresponding Geometry object. Spatial operators provided by JTS apply on this data type. The result is also transformed in order to be sent back to the XQuery parser. We modified JTS library by adding a ConvertToGML method to all geometric types: this method constructs GML tags from a JTS Geometry object.

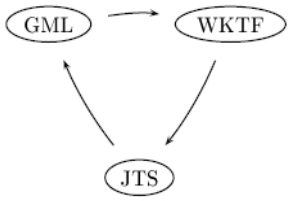

Figure 10: Converting GML data to JTS objects

We also modified the WktReader provided by JTS. The original WktReader cannot read floating values written in a mantissaEexponent format. Now the analyzer is compatible with this syntax, and we can manipulate big spatial objects. All our development is done in Java, using the Eclipse environment *(IBM Eclipse 2003)*.

During the tests, we noticed that queries using join operations took a long time to be executed. This is due to the large size of the documents and the complexity of joins

algorithms. As join operations occur very often when using a mediation system, we came up with a specific algorithm to gain performance.

A Geographic Efficient Join Algorithm

Since we are interested in a geographic mediation query engine, we are not dealing with records order. This notion is irrelevant in our case because we work on a virtual global schema of a set of real databases. So we cannot specify a record order because data are coming from several databases. In Geographic Information Systems, results are often displayed in a graphical form (such as Scalable Vector Graphics (SVG) or ESRI's shape) rather than in a textual form such as XML or relational tables. So we are just ignoring the record order. With these assumptions in mind, we designed a join algorithm, based on the JoinSort, which can compute the join operations in O(nlog(n)) time. To compare existing XQuery engines and our algorithm, we used a set of GML files with variable sizes.

The SortJoin Algorithm

As we already said, we used the SortJoin to compute Joins. This consists in sorting the GML (or XML) join entries according to the set of tags and attributes against which the join is performed. After, entries are scanned at the same time to extract only data which satisfy the join condition. Note that every entry is scanned only one time since it is sorted. In other word, suppose the algorithm is performing the i^{th} iteration (we look for all tuples in the second entry that satisfy the join condition) knowing that the last tuple in the second entry which has satisfied the join condition is at position y. So we don't re-scan the second entry from the beginning because we know that tuples that we are looking for can't exist in the first part of the second entry (part between the beginning of the entry and the y position). So search is only done on the second part (between y and the end of the entry). Another important point is that tuples which satisfy the join condition are necessarily consecutive. So we stop the scan when the join condition is no more satisfied. This Algorithm is illustrated below:

```
Input : two GML files named entry1 and entry2 | Output : a file containing the join result
Algorithm :
SortedEntry1 = EntrySort(entry1)
SortedEntry2 = EntrySort(entry2)
while ((node1 = nextitem(SortedEntry1)) != null){
value1 = GetJoinConditionTagValue(node1)
while((VerifJoinCondition(value1,value2) == false) and ((node2 = NextItem(SortedEntry1)) != null)){
value2 = GetJoinConditionTagValue(node2)}
if ((VerifJoinCondition(value1,value2) == true){
result = AddResult(node1,node2)
while((VerifJoinCondition(value1,value2) == true) and ((node2 = NextItem(SortedEntry1)) != null)){
result = AddResult(node1,node2)}
} else return result
} return result
```

Note that the EntrySort(Entry) function sorts the Entry according to the join parameters, NextItem(Entry) function returns the next node in the entry, GetJoinConditionTagValue(node) returns the value of the tag or the join attribute, VerifJoinCondition(value1,value2) returns true if the join condition is satisfied and AddResult(node1,node2) function adds a new node to the result built by node1 and node2 according to the return clause. There are mainly two steps in this algorithm: entry

sorting (this step consists of sorting the join entries according to the tags and attributes of the join condition. As we know, the best complexity of a sort algorithm is expressed in O(n log(n)). We used the xsltproc library to implement this step) and join computing (this step consists of computing the join result according to the explication above. Let n and m be respectively the number of tuples in the join entries. The complexity of this step is expressed in O(n+m)).
So, if we consider joins made on two entries of n tuples and m tuples, we can express the complexity of our algorithm in O(nlog(n) + mlog(m)).

Performance Measures

As we already said, we did run some tests on GML files coming from the example's databases. Each time, we changed the size of the files (by modifying the conditions) while making sure that they have the same number of tuples and we try to execute the the final query of the execution plan :

```
for $x in document(doc1.gml)/FeatureCollection/featureMember/Road,
    $y in document(doc2.gml)/FeatureCollection/featureMember/Road
where $x/id/node() = $y/id/node() return $x
```

These tests were made on a Linux platform with 3Ghz processor and 1Go RAM. We used two other parsers - denoted Engine1 and Engine2 - to do some performance comparison. Table 1 illustrates time results for the SortJoin, Engine1 and Engine2. NbTuples1 and NbTuples2 are the sizes of join entries. The two last columns show the percentage between the SortJoin execution time and the Engine1 and Engine2 ones.

Execution Times (Sc)			Join Entries		SJ Time Percentage	
Sort join	Engine 1	Engine 2	Tuples 1	Tuples 2	Engine 1	Engine 2
4.21	923.50	25419.44	4047	4047	0.456%	0.017%
4.13	831.05	21351.42	3818	3818	0.497%	0.019%
3.92	734.80	17106.76	3520	3520	0.533%	0.023%
3.62	629.79	13399.50	3263	3263	0.575%	0.027%
3.53	549.42	12594.46	3187	3187	0.642%	0.028%
3.46	517.60	11467.63	3071	3071	0.668%	0.030%
3.33	441.48	9563.73	2903	2903	0.754%	0.035%
3.22	386.45	7682.98	2689	2689	0.833%	0.042%
3.03	317.20	5846.72	2440	2440	0.955%	0.052%
2.99	298.05	5277.13	2353	2353	1.003%	0.057%
2.93	281.76	4665.27	2259	2259	1.040%	0.063%
2.81	236.91	3707.01	2078	2078	1.186%	0.076%
2.45	136.29	1790.64	1594	1594	1.798%	0.137%
2.19	76.74	876.82	1197	1197	2.854%	0.250%
2.15	71.14	746.42	1117	1117	3.022%	0.288%
1.9	43.69	422.62	874	874	4.349%	0.450%
1.57	8.41	61.19	352	352	18.668%	2.566%

Table 1: Different Engine Times

When analyzing these results, we notice that the SortJoin algorithm saves a considerable amount of time in the join execution; it's complexity is expressed in O(nlog(n)) against polynomial times of the Engine1 and Engine2 we tested. The only drawback of this algorithm resides in the ordered set of tuples returned. But in the geographic mediation setting, this factor isn't relevant, so we can advocate (for the time being) SortJoin algorithm.

5. USING GQUERY IN A MEDIATION SYSTEM

As mentioned in Section 1, one of the motivations behind GQuery is a query model for a geographic integration system developed in Marseille. The system is mainly composed of three layers: a GIS mediator, Web Feature Servers (WFS) and data sources. The GIS Mediator is composed of a Mapping Module, a Decomposition/Rewrite module, an Execution module, a Composition module, and a Source Manager module. The GIS mediator is in charge of analyzing GQuery expressions, including various transformations that involve (meta) schema information, performing some optimizations, and splitting the query into sub-queries, passing them to the right WFS for execution. The WFS layer is in charge of manipulating the data sources: it receives requests from the mediator, executes them, and returns the results.
A WFS request consists of a description of a query or data transformation operations that are to be applied to one or more features. The request is issued by the client and is posted to a Web server via HTTP. The WFS is invoked to service the request.
A request combines operations defined in the WFS specification. The WFS reads and executes the request and returns an answer to the user as a GML document.
Among those operations are:
GetCapabilities: a WFS source must be able to describe its capabilities.
Specifically, it must indicate which feature types it can service and what operations are supported on each feature type.
DescribeFeatureType: a WFS source must be able, upon request, to describe the structure of any feature it can service.
GetFeature: a WFS source must be able to service a request, and to retrieve feature instances and properties.
The main components of the GIS mediator are as follows:
The WFS translator receives a request that is an XML encoding of (a subset of) a SQL like query language derived from the OpenGIS CQL (Catalog Query Language). This request is translated into a GQuery expression. This module turns the VirGIS system into a WFS-enabled server.
The Mapping module uses integrated schema (meta) information in order to express user queries in terms of local source schemas.
The Decomposition/Rewrite module exploits information about source feature types and source capabilities to generate an execution plan.
A global GQuery expression is used as a container (place-holder) for collecting and integrating results coming from local data sources. The Execution module processes sub-queries it receives in sending them to the appropriate WFS. Note that sub-queries that require an operator that is not available at any of the integrated data sources are processed differently. The Composition module processes a GQuery expression and produces a GML document, i.e., the answer that should be returned to the client.
The Source Manager module is in charge of collecting information from the WFS sources. It builds the configuration files for the Integrated (Global) Schema, Source Capabilities and Cost Information.

6. CONCLUSION AND FUTURE WORK

In this paper, we described GQuery, a spatial query language we designed and implemented. To the best of our knowledge there is no such current implementation, i.e., an XQuery based geography query language. In *(P.G and J.Corcoles 2002)* for instance, a specification of a GML query language is suggested; the approach followed led to a geographic (semi-structured) data model, with a SQL like query language; but with no implementation. The GQuery prototype allows FLWR expression queries posed over GML data. Geographic operators are implemented in an ad-hoc way by means of external calls to functions provided by Java Topology Suite. The preliminary design and implementation choices we made so far turn out to be very satisfactory. Future work should include query optimization, and more experimentation in the VirGIS integration platform. It should also consist in prototype dissemination for the database/GIS community.

REFERENCES

D.Beech, A.Malhotra and M.Rys (1999) : A formal data model and algebra for XML.

O. Boucelma, Z. Lacroix and M.Essid (2002) : A WFS based mediation system for GIS interoperability, ACM GIS'02, November 2002.

M. Egenhofer (1994) : Spatial SQL, a query and presentation language, IEEE Transactions on Knowledge and Data Engineering, 1994.

IBM Eclipse (2003) : An open platform for tool integration, http://www.eclipse.org, 2003.

P.G and J.Corcoles (2002) : A specification of a spatial query language over GML, pages 112-117, ACM GIS'01, November 2002

Opengis Consortium (2003) : Opengis specifications, http://www.opengis.org, 2003

Oracle Corporation (2003) : Oracle Spatial Data Sheet, http://www.oracle.com, 2003

Postgis (2003) : Postgis, geographic objects for Postgres, http://postgis.refractions.net, 2003

Ranga Raju Vatsavai (2002) : GML-QL, A spatial query language specification for GML, UCGIS Summer 2002, Athens, Georgia, 2002

C.H. Wang Feng, Sha Jichang and Y. Shuqiang (2000) : Geosql, a spatial query language of object oriented gis, Second International Workshop on Computer Science and Information Technologies, CSIT, 2000.

W3 Consortium (2003) : W3C Specifications.

CV(S) OF THE AUTHORS

OMAR BOUCELMA IS PROFESSOR AT THE UNIVERSITY OF PROVENCE.

FRANÇOIS-MARIE COLONNA IS A PHD STUDENT.

CO-ORDINATES

Professor Omar Boucelma

Institution	:	LSIS, UMR-CNRS 6168, University of Provence
Address	:	Avenue Escadrille Normandie-Niemen
Postal Code-City	:	13397 Marseille Cedex 20
Country	:	France
Telephone number	:	
Fax number	:	

E-mail address : omar@cmi.univ-mrs.fr
Website :

François-Marie Colonna
Institution : LSIS, UMR-CNRS 6168, University of Provence
Address : Avenue Escadrille Normandie-Niemen
Postal Code-City : 13397 Marseille Cedex 20
Country : France
Telephone number :
Fax number :
E-mail address : fcolonna@cmi.univ-mrs.fr
Website :

APPENDIX A

```
<City xmlns:gml="http://www.opengis.net/gml">
<gml:name>Cambridge</gml:name>
<gml:boundedBy>
    <gml:Box srsName="http://www.opengis.net/gml/srs/epsg.xml#4326">
        <gml:coord><gml:X>0.0</gml:X><gml:Y>0.0</gml:Y></gml:coord>
     <gml:coord><gml:X>100.0</gml:X><gml:Y>100.0</gml:Y></gml:coord>
    </gml:Box>
</gml:boundedBy>

<Cityhall>
 <gml:name>City Hall</gml:name>
 <gml:description>The city hall of Cambridge</gml:description>
 <adress>22 City Hall Avenue</adress>
 <gml:Polygon srsName="http://www.opengis.net/gml/srs/epsg.xml#4">
    <gml:outerBoundaryIs>
      <gml:LinearRing>
        <gml:coord><gml:X>10</gml:X><gml:Y>80</gml:Y></gml:coord>
        <gml:coord><gml:X>10</gml:X><gml:Y>90</gml:Y></gml:coord>
        <gml:coord><gml:X>30</gml:X><gml:Y>90</gml:Y></gml:coord>
    <gml:coord><gml:X>30</gml:X><gml:Y>80</gml:Y></gml:coord>
    <gml:coord><gml:X>10</gml:X><gml:Y>80</gml:Y></gml:coord>
   </gml:LinearRing>
    </gml:outerBoundaryIs>
 </gml:Polygon>
 <classification>house</classification>
</Cityhall>

</City>
```