*Transactions in GIS*, 2007, 11(2): 233–253

Research Article

# GML for Representing Data from Spatio-Historical Databases: A Case Study

Fang Siyuan, Tony Griffiths and Norman W Paton
*School of Computer Science, University of Manchester*

**Abstract**
Many applications, in areas such as land use, traffic management and location aware services, involve the storage, analysis and sharing of spatio-temporal data. The need to represent such data in a way that eases sharing across applications, has led to the development of the Geography Markup Language (GML), which provides a rich collection of constructs for representing spatial and associated aspatial data as XML documents. However, although there are a growing number of applications and tools that make use of GML, there are surprisingly few experience reports on the representation of data from existing applications or models using GML constructs. This paper provides one such report, describing the use of GML as an exchange format for the Tripod spatio-historical database. This in turn involves identifying mappings between Tripod and GML constructs, and the development of a generic architecture for carrying out such mappings. The experience demonstrates that even though GML provides rich modelling facilities, the development of mappings from pre-existing models can be challenging, as related constructs often provide semantically distinct capabilities.

## 1 Introduction

Spatial database systems aim to provide Geographic Information Systems (GIS), or other spatial data handling applications with facilities for the efficient storage and retrieval of geographic data (Rigaux et al. 2001). In addition, change to geographic data over time is important to many applications. This situation gives rise to a requirement for database systems with capabilities for handling both the spatial and temporal aspects of information. These have been termed spatio-temporal database management systems (ST-DBMS) (Abraham et al. 1999)[1]. Tripod is the first complete prototype of an *Object* ST-DBMS (Griffiths et al. 2002), with its data model extending that of the Object Data

**Address for correspondence:** Fang Siyuan, School of Computer Science, University of Manchester, Kilburn Building, Oxford Road, Manchester M13 9PL, United Kingdom. E-mail: fangs@cs.man.ac.uk

Management Group (ODMG) (Cattell et al. 2000) object database standard with constructs for capturing changes to both the spatial and aspatial properties of entities over time.

With the development of GIS, many spatial databases have been developed for different domains. These databases often have different data models and storage structures, which impedes data communication between these databases. To help address this problem, the Open GIS Consortium (OGC) has proposed the Geography Markup Language (GML) (OpenGIS 2003), as the standard for XML encoding of geographical data. The purpose of GML is to provide an application-neural format for spatial data storage and exchange, especially for web-based applications, with the most recent version (GML 3.0) providing explicit support for the representation of time-varying spatio-historical information.

GML provides features to model real-world objects. The concept of a feature has been proposed by the International Organization for Standardization (ISO), and a detailed description is available in OpenGIS (1999). Features are categorized into particular feature types, with user-defined feature types being specified in applications schemas. GML features can be concrete, such as rivers and cites, or abstract such as observations. A GML feature is characterized by a set of properties, which are essentially XML elements of the feature XML element. The properties can be of spatial (geometry) types or non-spatial types.

Other standards for encoding geographic data do exist, e.g. GDF (Geographic Data Files) (ISO14825, 2002) and SDTS (Spatial Data Transfer Standard; ANSI 1998); however, these are designed for spatial data exchange for specific application domains (GDF is designed to describe road networks and road-related data in Europe and SDTS is a standard for transferring digital spatial data between distributed computer systems). In contrast, GML is an application-independent standard based on XML. Application-independence is achieved by GML documents referencing a schema (written using the XML Schema language (W3C 2001a) using constructs from the GML core schemas), whose purpose is to provide a machine and human readable description of the document, against which the document can be verified.

In addition, all the technologies developed for XML can be used with GML, e.g. data contained in GML documents can be transformed using XSLT (W3C 1999); distributed resources can be integrated into GML documents using XLink (W3C 2001d); and the spatial data contained in GML can be rendered by Scalable Vector Graphics (SVG) (W3C 2001c).

Although GML has received significant attention, and is now comprehensive in scope, there is surprisingly few papers that provide detailed experience on its use. The main purpose of this paper is to evaluate GML as a meta-language for encoding data from spatio-temporal databases, using Tripod as an exemplar. To achieve this aim, a mapping between the Tripod data model and that of GML has been developed, together with GML import/export tools for Tripod. These tasks are representative of those that other applications will have to perform to transform data from their particular data model into GML.

The remainder of this paper is organized as follows. Section 2 presents an overview of GML. Section 3 introduces the Tripod spatio-historical data model, together with an application that is used as a motivating example throughout the paper. Section 4 describes the mapping issues from the Tripod spatio-historical object model to the GML feature model. Section 5 introduces the implementation of GML import and export facilities for Tripod. Section 6 presents conclusions and a summary of the paper.

## 2 An Overview of GML

GML is a mark-up language that is used to describe objects that have geographic properties. A GML specification consists of two components: a GML schema, and a collection of document instances that conform to this schema. GML schemas are written using the XML Schema language (W3C 2001a), using element and attribute tags from both the core XML and GML schemas. GML instances are XML documents that describe geographic information based on the structures and constructors provided by these schemas.

GML provides several schemas (known as the *core* schemas) allowing developers to choose the particular features that they want to use in their application. Schemas developed for a particular application (or domain) are known as GML *application* schemas. For example, developers may define an application schema for use in the oil production industry. The GML core schemas provide frameworks and basic data types that can be used by the GML application schemas.

GML application schemas can be viewed as application-specific meta-languages that define the "vocabularies" for the particular application domain. Application schemas always have a target namespace (W3C 2001e), which is used to uniquely identify the schema, and hence avoid naming conflicts between different application schemas. In order to make use of the GML core schemas, the application schemas import the required core schemas. In addition, GML application schemas can import one another to make use of a particular feature, allowing the vocabulary of an application domain to be defined using several different application schemas.

Figure 3 presents a fragment of the GML application schema for representing the Tripod database schema of Figure 1. This fragment defines a feature type called *Building* which represents the *Building* object type. The *Building* has three property elements: *Building_attr_name*, *Building_attr_footprint*, *Building_rela_in_street* and *Building_rela_has_premises_of*, that represent the four properties declared in the Building object type, respectively. The target namespace of this application schema is *http://www.cs.man.ac.uk/schemas/businesses* and the associated prefix is *ex*. Throughout the remainder of this paper, `courier font` is used to refer to Tripod object model concepts, whereas *italic font* is used to refer to GML schema concepts.

## 3 The Tripod Spatio-Historical Object Database

Tripod (Griffiths et al. 2001) is a complete implementation of a spatio-historical object database management system (ST-ODBMS) that extends the ODMG standard for object databases. Tripod's spatio-historical object data model (Griffiths et al. 2004a) extends the ODMG object model with two important categories of primitive types: *spatial* types and *timestamp* types. In addition, a specialized mechanism called a *History*, is used to capture changes to object properties and their relationships with other objects over time. Figure 2 illustrates an overview of the Tripod architecture.

The basic modeling primitives of the ODMG Object Model are literals and objects, which can be characterized by their types. The ODMG Object Model provides a collection of predefined literal types, (e.g. `float` and `string`), and collection types (e.g. `set` and `list`). With these literal types, users can create their own object types to represent, for example, companies in our motivating example. Users can then define relationships

```
forward class Street;
forward class Building;
forward class Company;

historical<timeIntervals, DAY> class Street (extent Streets) {
 attribute string name;
 historical<timeIntervals, DAY> attribute lines central_line;
 historical<timeIntervals, DAY> relationship set<Building> has_buildings


                                        inverse Building::in_street;
};

historical<timeIntervals, DAY> class Building (extent Buildings) {
 attribute string name;
 historical<timeIntervals, DAY> attribute regions footprint;
 historical<timeIntervals, DAY> relationship in_street


                                        inverse Street::has_buildings;
 historical<timeIntervals, DAY> relationship set<Company> premises_of


                                        inverse Company::has_premises;
};

historical<timeIntervals, DAY> class Company (extent Companies) {
 attribute string name;
 attribute string type;
 historical<timeIntervals, DAY> attribute string manager;
 historical<timeIntervals, DAY> attribute string address;
 historical<timeIntervals, DAY> attribute string telephone;
 historical<timeIntervals, DAY> relationship set <Building> has_premises

       inverse Building::premises_of;
};
```

**Figure 1** Specification of the `Businesses` Tripod database schema

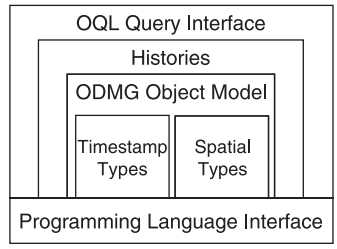


**Figure 2** Tripod architecture overview

between these object types. Each object type is characterized by its *state* (i.e. *attributes*, which are values such as character strings or integers, and *relationships* defined between object types, i.e. the object Fred has the relationship *wife* with the object Katy) and *behavior* (a collection of *operations*, e.g. functionality to move location). The collection of objects of these user-defined types constitutes an object database.

```
<schema targetNamespace="http://www.cs.man.ac.uk/schemas/businesses"
       xmlns="http://www.w3.org/2001/XMLSchema"
       xmlns:ex="http://www.cs.man.ac.uk/schemas/businesses"
       xmlns:gml="http://www.opengis.net/gml"
       xmlns:gml_ex="http://www.cs.man.ac.uk/schemas/gml_extension"
       xmlns:xlink="http://www.w3.org/1999/xlink"
       xmlns:xs="http://www.w3.org/2001/XMLSchema">
         …
<element name="Building" type="ex:Building_Type"/>
<!--=========================================================================-->
<complexType name="Building_Type">
   <complexContent>
      <extension base="gml_ex:AbstractHistoricalObjectType">
         <all>
            <element minOccurs="0" ref="ex:Building_attr_name"/>
            <element  minOccurs="0" ref="ex:Building_attr_footprint"/>
            <element minOccurs="0" ref="ex:Building_rela_in_street"/>
            <element minOccurs="0" ref="ex:Building_rela_premises_of"/>
         </all>
         <attribute fixed="DAY" name="granularityType" type="gml_ex:granuType"/>
      </extension>
   </complexContent>
</complexType>
         …
</schema>
```

**Figure 3** GML application schema for the business application

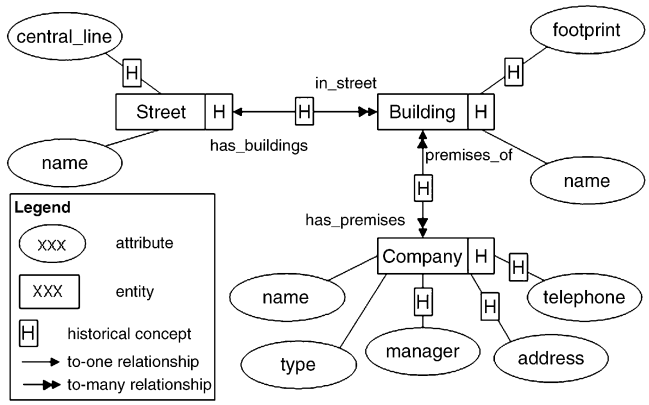


**Figure 4** Schema representing business information

### *3.1 A Motivating Example*

The motivating example running throughout this paper is that of a simple business location system. The features in this application are representative of those of many GIS, in that it captures changes to both spatial and non-spatial information over time. A much simplified schema (adapting the notation in Cattell et al. 2000, with extensions to represent historical data) showing the main classes of interest in the application is presented in Figure 4. Entities and relationships that can change over time are marked with a 'H' symbol (i.e. are historical). It should be noted that changes to some of a building's properties over time are captured (e.g. `footprint`), whereas others (e.g. `name`) are not. In addition to tracking the changes to such properties, the values of

relationships between entities can also be tracked (e.g. the collection of companies that are located in a building).

The information stored by the system can be used for many purposes. For example, the database can be queried to create a themed map showing the change in the distribution of types of company over time.

### 3.2 The Tripod Spatial Types

Tripod extends the ODMG Object Model with six new spatial data types (SDTs), that are based on the ROSE (Robust Spatial Extensions) algebra (Güting and Schneider 1995). It is important to understand that of these six spatial data types, `Points`, `Lines` and `Regions` are *collection-based* types (i.e. a single `Regions` value can be used to model a building that has a non-contiguous footprint). The underlying domain of the ROSE algebra is a finite integer grid referred to as a *realm*. Roughly speaking, each element of a `Points` value is a pair of coordinates from the realm; each element of a `Lines` value is a collection of connected line segments; and each element of a `Regions` value is a collection of polygons that may contain finite holes, which themselves can also contain a collection of polygons. In addition, three *singleton* valued SDTs, `Point`, `Line` and `Region`, are added by Tripod. These are equivalent to the individual elements in their collection-based counterparts. Tripod provides a comprehensive collection of operations over these SDTs. Figure 5 presents an example of spatial data values that can exist in our example application using these SDTs. The `Regions` value, **R1**, represents the footprint a building that consists of two disjoint parts, one of which contains a hole containing another component. The `Lines` value **L1** represents the central-line of a street.

### 3.3 The Tripod Timestamp Types

Tripod provides four timestamp types: `Instants` and `TimeIntervals`, that are collection-based; and `Instant` and `TimeInterval`, that are singleton values. These timestamp types

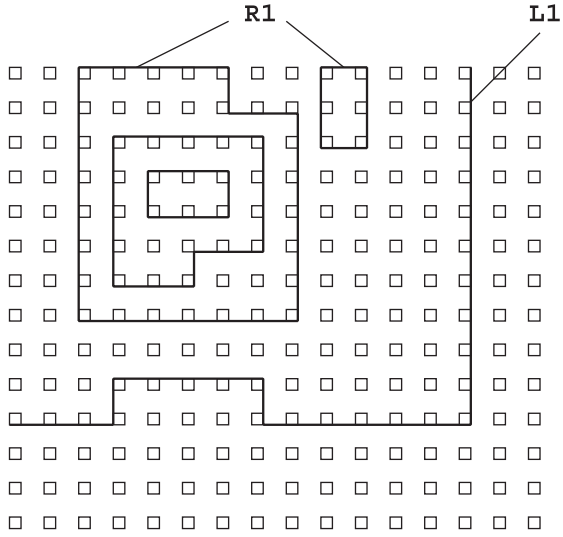


**Figure 5** A realm and example spatial data values

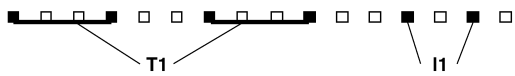


**Figure 6** A temporal realm and example timestamp values

can be viewed as one-dimensional analogs of the spatial types, and the underlying temporal realm is thus a finite set of integers that constitute a time line. Each element in an `Instants` value is a time point, which is an integer in the temporal realm. Each element in a `TimeIntervals` value is a pair of time points, that denote the start and end time of the interval respectively. The singleton spatial timestamp types, `Instant` and `TimeInterval`, are equivalent to individual elements in `Instants` and `TimeIntervals` values. Tripod's timestamp types provide a broad-spectrum of operations, including ordering predicates based on the underlying integer domain. In addition, Tripod utilises the standard Gregorian calendar to represent dates. Timestamps can be defined at a range of granularities, such as `Year`, `Month`, `Day`, `Hour`, `Minute` and `Second`. Figure 6 presents some examples of timestamp values. **T1** is a `TimeIntervals` value (composed of two non-contiguous intervals) and **I1** is an `Instants` value (composed of two time points).

### *3.4 Histories*

Tripod defines the concept of a *history* to model the time-varying properties of an entity. Using the history mechanism, changes over time to any Tripod construct (e.g. attributes and relationships) can be captured. In essence, a history is an ordered collection of timestamp value pairs, referred to as *states*. In a particular history, the timestamp is either of type `Instants` or `TimeIntervals`, and the associated value (referred to as a *snapshot*) is either a primitive value or the identifier of another object within the database. An important feature of Tripod histories is that the collection-valued nature of Tripod timestamps is used to group together snapshots with equal values into a single state (a process known as *coalescing*). The history,

```
{ < [1990 - 1995, 2001 - 2005), "Tony" > , < 1995 - 2001, "Norman" > }
```

can therefore be used to represent the three separate database transactions:

1. 1990–1995, manager is Tony
2. 1995–2001, manager is Norman, and
3. 2001–2005 manager is again Tony

If an object type (e.g. `Street`) is defined using the keyword `historical`, Tripod automatically creates and maintains an attribute called *lifespan* that records the *valid time* of the object (i.e. when the object is active or inactive in the modeled system). For example, if a company was founded in 1990 and dissolved in 1995, but was subsequently re-activated in 2000, then the lifespan of this company object in the database would be the history: {<1990 – 1995, `true`>, <2000 – `uc`, `true`>}, where `uc` is a special timestamp that indicates that this information is valid *until changed*. Tripod utilises the object's lifespan to ensure that a particular object can only have properties when the object exists in the application.

It is important to note that an object can have properties that asynchronously change over time. For example, Figure 7 shows that a company `tripod.com` has

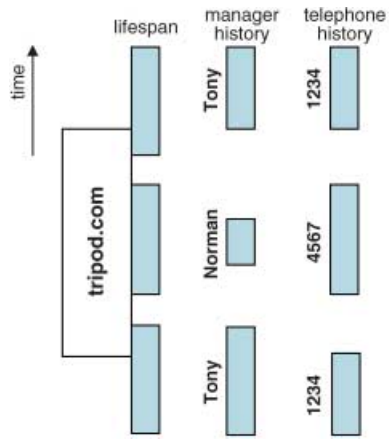


**Figure 7** Asynchronous change in object historical properties

histories associated with its `manager` and `telephone` attributes. Note that the values for both these histories are contained within `tripod.com`'s lifespan. The snapshots within the `manager` and `telephone` histories, however, change at different points in time, and indeed have different periods where they have no recorded value.

Developers use a declarative schema definition language (Object Definition Language (ODL)) for creating a database structure for their application. In the case of our example application, they create a database, called `Businesses`, using the statements given in Figure 1. It can be seen that each class, attribute, or relationship in Figure 1 corresponds to an entity, attribute or relationship from Figure 4.

## 4 GML Representation of Tripod Data

This section presents how GML can be used to represent the data from a spatio-historical database as exemplified by Tripod. The basic data types provided by the GML core schemas have been used to represent the Tripod literal types. However, some Tripod literal types do not have GML equivalents. In this situation, extensions to GML have to be made to cater for the Tripod context. The extended data types for GML are defined in the schema: *gml_extension.xsd*, whose target namespace is *http://cs.man.ac.uk/schemas/gml_extension*, with an associated prefix *gml_ex*. For reasons of space, the GML extension schema is available as a separate download from http://www.cs.man.ac.uk/tripod/download.jsp. In this paper, all the elements with a prefix *gml* (associated with the GML target namespace *http://www.opengis.net/gml*) are from the GML core schemas.

### *4.1 Mapping the Literal Types*

The Tripod literal types can be classified into three categories: atomic, collection and structured. Tripod's atomic literal types can be further divided into three groups: the literal types from the ODMG object model such as `string` and `long`, the spatial literal types and the timestamp literal types. The collection and structured literal types group together atomic literal types.

**Table 1** Mapping atomic literal types

| ODMG atomic literal types | GML data types (XML schema data types) |
|---|---|
| `String` | *String* |
| `Boolean` | *Boolean* |
| `Char` | *String* |
| `Short` | *Short* |
| `unsigned short` | *unsignedShort* |
| `Long` | *Long* |
| `unsigned long` | *unsignedLong* |
| `Float` | *Float* |
| `Double` | *Double* |
| `octet char` | *String* |

### *4.1.1 Atomic literal types*

The ODMG atomic literal types can be represented by the built-in XML schema data types (W3C 2001b). Table 1 presents the mapping from the ODMG atomic literal types to their XML schema equivalents.

### *4.1.2 Spatial literal types*

Since one of the primary motivations behind the creation of GML is to provide a mark-up language to describe the geographic properties of an object, it is desirable to utilise a direct mapping between the spatial types of Tripod and those of GML.

The singleton Tripod spatial literal types can be directly represented using the GML geometry types. Tripod `Point` can be directly mapped to *gml:Point*, and Tripod `Points` can be represented by the aggregated type *gml:MultiPoint*. However, the `Line` (`Lines`) literal types cannot be represented by any of the existing GML geometry types. GML provides four one-dimensional geometry types: *gml:LineString*, *gml:Curve*, *gml:CompositeCurve*, and *gml:OrientableCurve*. *gml:LineString* is used in environments that require only simple geometry. A *gml:LineString* is composed of a finite set of points, which are called control points. These control points are linearly interpolated. By contrast, a Tripod `Line` value is composed of several connected segments, rather than points. *gml:LineString* cannot directly represent the structure of a Tripod `Line` value such as that illustrated in Figure 8. The remaining GML geometric types do not provide a direct mapping, as:

1. The individual segments of these types are curves not line segments, with each curve being a non-linear interpolation between control points, and
2. In *gml:CompositeCurve* (as shown in Figure 9) the end point of the previous curve must coincide with the start point of the next curve. By contrast, the segments contained in a Tripod Line value are required to be connected, but do not have such a restriction.
3. *gml:OrientableCurve* is used to model directional one dimensional elements, whereas, a Tripod `Line` value is un-oriented. Thus, *gml:OrientableCurve* is not suitable for representing `Line` values.

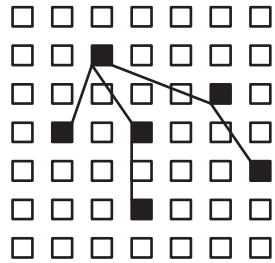

**Figure 8** Example of a Tripod Line value

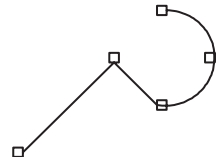

**Figure 9** Example of gml:CompositeCurve

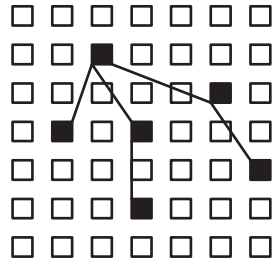

**Figure 10** A Tripod `Line` value

The lack of a suitable direct mapping necessitates the creating of a new geometry type for the Tripod `Line` and `Lines` types. To accommodate the required semantics, the new type should be a subtype of *gml:Curve* and consist of only *gml:LineStringSegment* elements, as this is the standard GML type for representing a linearly interpolated line segment. The qualified name of this new type is *gml_ex:Line*. *gml_ex:Line* derives from *gml:AbstractCurveType* and could be substitutable by *gml:_Curve*. *gml_ex:Line* can contain a *gml:LineStringSegments*, which is a container of *gml:LineStringSegment*. Figures 10 and 11 illustrate these mappings.

The aggregate type of *gml_ex:Line*, namely *gml_ex:MultiLine*, is defined to represent a Tripod `Lines` value. In addition, in order to link the *gml_ex:Line* (*gml_ex:MultiLine*) into geometry-valued property elements of the GML features, *gml_ex:LinePropertyType* (*gml_ex:MultiLinePropertyType*) has been created. The type of any property element whose value is of *gml_ex:Line* (*gml_ex:MultiLine*) type should be defined using (*gml_ex:MultiLinePropertyType*). The *gml_ex:LinePropertyType* (*gml_ex:MultiLine-PropertyType*) is a property type that has a *gml_ex:Line* (*gml_ex:MultiLine*) as its value. The GML application schema fragment presented in the definition of a property element whose value is of *gml_ex:MultiLine* type. For example,

```
<gml_ex:Line>
  <gml_ex:LineStringSegments>
    <gml:LineStringSegment>
        <gml:pos>1 3</gml:pos>
        <gml:pos>2 5</gml:pos>
    </gml:LineStringSegment>
    <gml:LineStringSegment>
        <gml:pos>2 5</gml:pos>
        <gml:pos>3  3</gml:pos>
    </gml:LineStringSegment>
    <gml:LineStringSegment>
        <gml:pos>3 3</gml:pos>
        <gml:pos>3  1</gml:pos>
     </gml:LineStringSegment>
     <gml:LineStringSegment>
        <gml:pos>2 5</gml:pos>
        <gml:pos>5 4</gml:pos>
    </gml:LineStringSegment>
     <gml:LineStringSegment>
        <gml:pos>5 4</gml:pos>
        <gml:pos>6 2</gml:pos>
    </gml:LineStringSegment>
    </gml_ex:LineStringSegments>
</gml_ex:Line>
```

**Figure 11** The corresponding *gml_ex:Line* Instance

```
<road_line>
  <gml_ex:Line>
    <gml_ex:LineStringSegments>
        <gml:LineStringSegment>
          <gml:pos>1 2</gml:pos>
          <gml:pos>2 2</gml:pos>
        </gml:LineStringSegment>
     </gml_ex:LineStringSegments>
  </gml_ex:Line>
</road_line>
```

**Figure 12** Example of a *gml_ex:LinePropertyType*

```
< element name="road_line" type="gml_ex:LinePropertyType"/ >
```

A valid GML instance according to the above schema is shown in Figure 12.

For similar reasons, the `Region` (`Regions`) type cannot be directly mapped to the existing GML geometry types. GML provides four two-dimensional geometry types: *gml:polygon*, *gml:CompositeSurface*, *gml:OrientableSurface*, and *gml:Surface*.

*gml:polygon* provides the closest mapping. It is a flat connected surface, where flat means all the vertices in a *gml:polygon* are in the same plane. *gml:Polygon* is analogous to the Tripod `Region`. However, each hole in a `Region` value can itself contain other holes, whereas, in *gml:Polygon*, nothing can be contained in the interior boundary. Therefore, it will cause information loss if *gml:Polygon* is used to represent Tripod `Region`.

The main difference between the *gml:Surface and* Tripod `Region` is that the component surface patch of *gml:Surface* may be in different planes. The remaining possibilities for a `Regions` mapping cannot therefore be used.

A new geometry type must therefore be created for Tripod Region values. `Region` is defined recursively, as an inner construct of a `Region` value can contain other inner constructs. Therefore, a recursive GML element is defined to represent such recursion. The qualified name of this new geometry type is *gml_ex:Region* which derives from *gml:AbstractSurfaceType*. *gml_ex:Region* contains at most one *gml_ex:outerBoundary* value and zero or more *gml_ex:innerBoundary* values. The contour of the *gml_ex:outerBoundary* is characterized by a *gml:LinearRing*, which is composed of at least three points. The start point of *gml:LinearRing* must coincide with the end point. The *gml_ex:innerBoundary* has a child element called *gml_ex:innerPatch*, which is defined recursively. *gml_ex:innerPatch* can also contain its own *gml_ex:outerBoundary* and *gml_ex:innerBoundary*. The *gml_ex:Region* has not been defined as the recursive element because the inner construct in a Tripod `Region` value is not an independent entity. In other words, the inner construct in a Tripod `Region` value is not another Tripod `Region` value. So, in order to indicate this difference, the *gml_ex:innerPatch* is added for the recursion. Moreover, to prevent infinite recursion, the minimum occurrence of *gml_ex:innerBoundary* is set to be zero.

In addition, *gml_ex:RegionPropertyType* is defined for linking the *gml_ex:Region* into geometry-valued property elements of the GML features. The *gml_ex:RegionPropertyType* is a property type that has a *gml_ex:Region* as its value. The aggregate type of *gml_ex:Region*, called *gml_ex:MultiRegion*, is defined to represent the Tripod `Regions` values. Similar to *gml_ex:RegionPropertyType*, *gml_ex:MultiRegionPropertyType* is created for linking the *gml_ex:MultiRegion* into geometry-valued property elements in the GML features. Figure 13 illustrates an example of *gml_ex:MultiRegion* which represents the `Regions` value **R1** presented in Figure 5.

### *4.1.3 Timestamp literal types*

The Tripod timestamp types `Instant` and `TimeInterval` can be directly represented by GML types *gml:TimeInstant* and *gml:TimePeriod*. However, GML does not provide types for collection-based timestamp types, such as Tripod `Instants` and `TimeIntervals`. The collection-based Tripod timestamp types can however be represented by collections of GML timestamp types.

### *4.1.4 Collection literal types*

Tripod's collection literal types can be represented by the XML (GML) types defined in the GML application schemas. Unlike the Tripod object types and structured literal types, the GML collection literal types do not have their own names. The result of this is that the XML (GML) types for representing collection literal types are not defined separately, but are anonymously nested within a GML feature's attribute elements whose values are of the collection literal types. For example, assume the following Tripod ODL definition:

```
attribute list<string>alias;
```

Figure 15 presents the XML (GML) element definition of this attribute, with Figure 14 showing a corresponding instance. The *type* attribute of the element *collection* is used to denote the five collection categories: `bag`, `set`, `list`, `array` and `dictionary`. The *index* attribute of the *item* element is used to preserve the order of ordered collection types.

```
<gml_ex:MultiRegion>
   <gml_ex:RegionMember>
      <gml_ex:Region>
         <gml_ex:outerBoundary>
            <gml:LinearRing>
               <gml:pos>2 6</gml:pos>
               <gml:pos>2 13</gml:pos>
               <gml:pos>6 13</gml:pos>
               <gml:pos>6 12</gml:pos>
               <gml:pos>8 12</gml:pos>
               <gml:pos>8 6</gml:pos>
               <gml:pos>2 6</gml:pos>
            </gml:LinearRing>
         </gml_ex:outerBoundary>
         <gml_ex:innerBoundary>
            <gml_ex:innerPatch>
               <gml_ex:outerBoundary>
                  <gml:LinearRing>
                     <gml:pos>3 7</gml:pos>
                     <gml:pos>3 11</gml:pos>
                     <gml:pos>7 11</gml:pos>
                     <gml:pos>7 8</gml:pos>
                     <gml:pos>5 8</gml:pos>
                     <gml:pos>5 7</gml:pos>
                     <gml:pos>3 7</gml:pos>
                  </gml:LinearRing>
               </gml_ex:outerBoundary>
               <gml_ex:innerBoundary>
                  <gml_ex:innerPatch>
                     <gml_ex:outerBoundary>
                        <gml:LinearRing>
                           <gml:pos>4 9</gml:pos>
                           <gml:pos>4 10</gml:pos>
                           <gml:pos>6 10</gml:pos>
                           <gml:pos>6 9</gml:pos>
                           <gml:pos>4 9</gml:pos>
                        </gml:LinearRing>
                     </gml_ex:outerBoundary>
                  </gml_ex:innerPatch>
               </gml_ex:innerBoundary>
            </gml_ex:innerPatch>
         </gml_ex:innerBoundary>
      </gml_ex:Region>
   </gml_ex:RegionMember>
   <gml_ex:RegionMember>
      <gml_ex:Region>
         <gml_ex:outerBoundary>
            <gml:LinearRing>
               <gml:pos>9 11</gml:pos>
               <gml:pos>9 13</gml:pos>
               <gml:pos>10 13</gml:pos>
               <gml:pos>10 11</gml:pos>
               <gml:pos>9 11</gml:pos>
            </gml:LinearRing>
         </gml_ex:outerBoundary>
      </gml_ex:Region>
   </gml_ex:RegionMember>
</gml_ex:MultiRegion>
```

**Figure 13** An instance of *gml_ex:MultiRegion*

```
<alias>
  <collection type="list">
    <item index="1">
      <item_value>Norman</item_value>
    </item>
    <item index="2">
      <item_value>Siyuan</item_value>
    </item>
    <item index="3">
      <item_value>Tony</item_value>
    </item>
  </collection>
</alias>
```

**Figure 14** An instance of a collection literal type

```
<element name="alias">
   <complexType>
      <sequence>
         <element name="collection" minOccurs="0">
            <complexType>
               <sequence>
                  <element name="item" maxOccurs="unbounded" minOccurs="0">
                     <complexType>
                        <sequence>
                           <element name="item_value" type="xs:string"/>
                        </sequence>
                        <attribute name="index" type="xs:short" se="optional"/>
                     </complexType>
                  </element>
               </sequence>
               <attribute name="type" fixed="list"/>
            </complexType>
         </element>
      </sequence>
   </complexType>
</element>
```

**Figure 15** Example `Collection` literal type mapping

### *4.2 Tripod Object Model Representation*

Tripod objects are broadly equivalent to GML features, with Tripod object properties being equivalent to GML feature properties. In addition to a generic feature type, GML provides two other special feature types: (1) *feature collection* type, that is used to represent features that may contain other features; and (2) *dynamic feature* type, that is used to model time varying objects. Both the feature collection and dynamic feature types present difficulties for use as a means of directly representing a Tripod object. This is because:

1. A Tripod object may have a collection of relationships that indicate *associations* to referenced objects. However, a feature association in a GML feature collection indicates *inclusion*, indicating that the contained feature is a part of the feature collection.
2. According to the definition of the root object type of the dynamic feature types, a dynamic feature can only possess one GML *history* property. As shown in Section 3,

**Table 2** Property types for encoding Tripod types

| Tripod literal type | GML data type | GML property type |
|---|---|---|
| `Point` | *gml:Point* | *gml:PointPropertyType* |
| `Points` | *gml:MultiPoint* | *gml:MultiPointPropertyType* |
| `Line` | *gml_ex:Line* | *gml_ex:LinePropertyType* |
| `Lines` | *gml_ex:MultiLine* | *gml_ex:MultiLinePropertyType* |
| `Region` | *gml_ex:Region* | *gml_ex:RegionPropertyType* |
| `Regions` | *gml_ex:MultiRegion* | *gml_ex:MultiRegionPropertyType* |
| `Instant` | *gml:TimeInstant* | *gml:TimeInstantPropertyType* |
| `TimeInterval` | *gml:TimePeriod* | *gml:TimePeriodPropertyType* |

a Tripod historical object may contain many asynchronously changing historical properties, whereas a GML *history* is a complete snapshot of the state of a feature at a particular point in time.

We are therefore forced to create a new generic feature type to represent Tripod objects in GML. To this end, a root object type called *gml_ex:AbstractHistoricalObjectType* is created to provide a common interface for representing the Tripod historical object types. This type derives from the *gml:AbstractFeatureType* (the root object type for all user-defined GML feature types). A child element called *gml_ex:lifespan* has been added to represent the Tripod `lifespan` attribute.

A GML feature must have an attribute called *gml:id* that is used to uniquely identify that feature. Each Tripod object has an object identifier (OID) to uniquely identify it within a particular database that can be used for this purpose.

### *4.2.1 Non-historical attribute representation*

Tripod non-historical attributes are represented by the GML elements whose value types are declared using the types from the GML, XML, and Tripod GML extensions. Table 2 presents these GML property types.

### *4.2.2 Non-historical relationship representation*

In general, a Tripod relationship can be expressed by a GML property element nested in a GML feature element. Such property elements can contain one or more reference elements. The reference element uses XLink constructs to specify the referenced feature, rather than using containment. All Tripod relationships are bi-directional, with the additional property that the related GML feature may be the current element (i.e. a so-called *homogeneous* relationship). The reference element has a simple XLink attribute group, that includes the properties: *xlink:href*, *xlink:role*, *xlink:arcrole*, *xlink:title*, *xlink:show* and *xlink:actuate*. Of these attributes, *Xlink:href* is used to identify the remote feature that participates in the relationship. The value of the *Xlink:href* is the value of the *gml:id* attribute of the referenced feature plus the namespace of the referenced feature. In Tripod, there are three kinds of relationship: `ref`, `set`, and `list`. A ref

```
<app:site gml:id ="001">
  <id>site01</id>
  <app:owner>
    <app:owner_reference xlink:href="http://www.cs.man.ac.uk/schemas/app#010"/>
  </app:owner>
</app:site>
...
<app:site gml:id ="002">
  <id>site02</id>
  <app:owner>
    <app:owner_reference xlink:href="http://www.cs.man.ac.uk/schemas/app#010"/>
  </app:owner>
</app:site>
  ...
<app:person gml:id ="010">
  <name>Tom</name>
  <address>...</address>
  <app:owning>
    <app:owning_reference  xlink:href=http://www.cs.man.ac.uk/schemas/app#001
                           xlink:title="1"/>
  <app:owning_reference xlink:href="http://www.cs.man.ac.uk/schemas/app#002"
                        xlink:title="2"/>
  </app:owning>
</app:person>
```

**Figure 16** GML representation of a `list` relationship

type is a *to-one* relationship, whereas both `set` (unordered with no duplicates) and `list` (ordered) denote cardinality *to-many* relationships.

A GML property element with at most one reference element is used to represent the `ref` relationship, whereas a property element with unbounded reference elements is used to represent the `set` relationship. The `list` relationship is mapped to another attribute *xlink:title* in the simple XLink attribute group, to preserve the order of the associated objects. Figure 16 presents an example of the GML representation of a `list` relationship.

### *4.2.3 Historical property representation*

The time varying properties of a GML feature can be represented by a GML *history* whose value is a collection of *time slices*. As such, a GML *history* can be used to represent a Tripod `history`, and *time slices* can represent the Tripod states. Figure 17 presents an instance of a GML *history* type called *Street_rela_has_Buildings*, that represents the historical relationship `has_Buildings` in the object type `Street` from the Tripod `businesses` schema as shown in Figure 1.

However using GML *histories* and *time slices* to represent a Tripod history and states raises two issues.

1. Firstly, as discussed in Section 3, within a history, Tripod coalesces value-equal snapshot values into a single state using a collection-valued timestamp. GML, however, utilises the non-collection-valued timestamps, *gml:TimeInstant* and *gml:TimePeriod*. When exporting data from a GML source to Tripod, each Tripod state must therefore be mapped into several GML *time slices*. Things are simpler in the reverse direction, since when importing data from GML into Tripod, the *time slices* with the same snapshot are automatically coalesced into a single *state* by the Tripod runtime algebra.

```
<ex:Street_rela_has_Buildings>
   <ex:Street_rela_has_Buildings_state>
      <gml:timeStamp>
         <gml:TimePeriod>
            <gml:begin>
               <gml:TimeInstant>
                  <gml:timePosition>
                     1973-03-03
                  </gml:timePosition>
               </gml:TimeInstant>
            </gml:begin>
            <gml:end>
               <gml:TimeInstant>
                  <gml:timePosition>
                     1973-03-07
                  </gml:timePosition>
               </gml:TimeInstant>
            </gml:end>
         </gml:TimePeriod>
      </gml:timeStamp>

   <ex:Street_rela_has_Buildings_snapshot>

   <ex:Street_rela_has_Buildings_reference
      xlink:href="http://www.cs.man.ac.uk/schemas/businesses#018"/>

   <ex:Street_rela_has_Buildings_reference
      xlink:href="http://www.cs.man.ac.uk/schemas/businesses#018"/>

   </ex:Street_rela_has_Buildings_snapshot>
   </ex:Street_rela_has_Buildings_state>
</ex:Street_rela_has_Buildings>
```

**Figure 17** Instance of *Street_rela_has_Buildings*

2. Secondly, because in GML the timestamps associated with time slices share the common base type, *gml:timeStamp*, and both *gml:TimeInstant* and *gml:TimePeriod* inherit from this type, GML time slices can be indexed by multiple timestamp types within a single *history*: a situation that is not allowed in a Tripod `history`. To solve the problem, when importing data from GML documents, the mapping program should verify the timestamp type of each GML time slice prior to insertion, and convert the timestamp as necessary.

### *4.2.4 Object extent and database representation*

In an object database, the collection of all instances of a particular object type is called its *extent*. An object database is therefore a collection of such extents. In order to represent a complete Tripod database, an XML (GML) element is defined as the root element in the GML application schema. This *database* element contains several extent elements, that correspond to data types defined in the Tripod database schema. Each extent element can contain zero to unbounded GML features. The XML schema for this database, called *businesses_DB.xsd*, representing the ODL database schema is available as a separate download from http://www.cs.man.ac.uk/tripod/download.jsp.

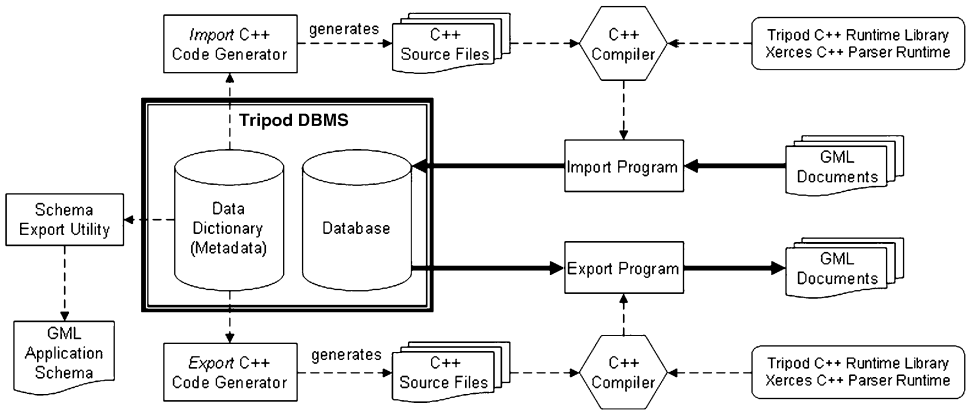


**Figure 18** Architecture of Tripod GML import/export utilities

## 5 GML Import and Export Utilities for Tripod

Based on the data model mapping presented in Section 4, utilities for exporting data from Tripod databases as GML documents, and for importing data from GML documents into a Tripod database have been implemented. Tripod provides a C++ language binding (API) (Griffiths et al. 2004b) to provide programmatic access to stored objects. The GML import and export utilities are implemented as C++ application programs that utilise these language bindings. A C++ implementation of the Document Object Model (DOM) (W3C 1998), the Xerces C++ parser (Apache 2003) for manipulating XML documents, has been used to manipulate GML documents. Figure 18 presents a high-level overview of the implemented utilities.

The GML import and export utilities consist of three parts: a *schema export utility*, an *export C++ code generator* for exporting data from a Tripod database to GML documents, and an *import C++ code generator* for importing data from GML documents to a Tripod database.

### *5.1 Schema Export Utility*

In Tripod, each database consists of a repository for stored objects plus a corresponding data dictionary, that contains metadata corresponding to the database schema. The information contained in the data dictionary is represented using a set of classes, such as *scope*, *attribute* and *class*. The schema export utility queries the data dictionary and generates a corresponding GML application schema. The exported GML application schemas always conform to certain naming conventions that are used to avoid name declaration collisions and meaningfully describe the characteristic of the corresponding Tripod entities. For example, the element corresponding to a Tripod attribute, called `name` contained in the object type `Building` is named as *Building_attr_name*.

### *5.2 Export C++ Code Generator*

Tripod's data export utility can be partitioned into two levels: (a) DOM output functions for each Tripod literal type; and (b) the C++ code generator for exporting Tripod data, that utilises these DOM output functions.

Each Tripod database is based on a specific database schema. Rather than force users to write a bespoke application program whose purpose is to export data according to a particular schema, we have take the approach of automatically generating such bespoke programs from the information contained in each database's metadata. The *export C++ code generator* reads the schema information from the data dictionary and automatically generates C++ source file for exporting data from Tripod to GML. These C++ source files are then compiled and linked with the Tripod and Xerces runtime libraries to produce an executable program. It is this program that, when executed, queries a Tripod database and generates GML documents that can be validated against the GML application schema produced by Tripod's schema export utility.

### *5.3 Import C++ Code Generator*

The import C++ code generator is a generic GML data import tool for Tripod databases. A common mechanism for bulk loading a database is to use a bespoke control file that specifies how data should be loaded into a database. This is the mechanism used by DBMSs such as Oracle. Tripod, on the other hand, adopts the orthogonal approach of automatically generating a data import program specialized for a particular Tripod database schema, using Tripod's import C++ code generator. Tripod users therefore do not have to specify how data should be loaded.

A problem that the import C++ code generator addresses is how to associate GML elements (tags) to the corresponding Tripod entities. This is achieved by creating a name resolution function, whose purpose is to translate the name of each GML element to its Tripod counterpart. To achieve this translation, the name resolution utilizes the previously discussed naming conventions.

In the current implementation, the GML import facility can only import GML documents that conform to these naming conventions. An enhancement to the system could use XSLT to transform GML into Tripod's standard input format.

## 6 Summary and Conclusions

Recent studies on GML have focused on efficient GML document storage (Córcoles et al. 2002) querying (Córcoles et al. 2001), and utilizing GML for particular domains (e.g. Shekhar et al. 2001, Zhang and Gruenwald 2001). Nevertheless, the primary use of GML is to assist in the exchange of data between heterogeneous geographic data sources. A key task in such activities is the creation of mappings between the data models that underpin such data and GML. However, there are to date few studies that have evaluated GML as an efficient and effective interchange standard for encoding data from disparate models. Fornari and Iochpe (2002) have developed a collection of mapping rules for translating an object-oriented model to GML; however, the developed mapping rules do not consider practical issues such as those involved with mapping literals from a specific object model to those of GML.

The work reported in this paper has demonstrated how the data from a spatio-historical database system (Tripod) can be represented using the structures and constructors provided by GML. In general, the atomic literal types that are provided by such database systems can be represented by the built-in XML schema data types or predefined GML data types. However, specific systems will invariably have literal types

that have no direct counterpart in GML, especially in the case of spatial data types, considering the diverse requirements of different geographical domains.

This paper has nevertheless shown that the various modelling primitives provided by GML do serve as a foundation upon which application developers can build for their particular domain. In particular, the GML feature model provides a generic mechanism for modeling geographical objects (both historical and non-historical) with spatial or aspatial properties. Currently, application developers need to create their own application schemas if they want to use GML. However, in the future, with the development of online schema registries, application developers may often be able to make use of the existing GML application schemas stored in such registries.

With specific reference to capturing historical change, GML provides a dynamic feature type. While this is sufficient to model the requirements of objects whose properties change at the same time, this paper has shown that this mechanism is not sufficient in itself to represent real world objects whose properties change asynchronously. We have shown how developers can extend GML to allow the modelling of such types of change.

Based on the described model mappings, utilities for importing to, and exporting from Tripod using GML have been presented. These show how GML can be used to capture data from independently developed models.

## Note

1 Throughout this paper the terms temporal and historical are used to refer to databases that pay special attention to handling time-varying data. The term historical, however, is used to refer to databases where only valid-time (and not transaction-time) semantics are supported, as is the case with the contribution of this paper.

## References

Abraham T and Roddick J F 1999 Survey of spatio-temporal databases. *GeoInformatica* 3: 61–99

ANSI 1998 Information technology – Spatial Data Transfer. ANSI INCITS 320-1998 (R2003)

Apache 2003 Xerces C++ Parser document. WWW document, http://xml.apache.org/xerces-c/

Cattell R G and Barry D K 2000 *The Object Database Standard: ODMG 3.0*. San Francisco, CA, Morgan Kaufmann

Córcoles J E and González P 2001 A specification of a Spatial Query Language over GML. In *Proceedings of the Ninth ACM International Symposium on Advances in Geographic Information Systems (ACM-GIS)*, Atlanta, Georgia: 112–7

Córcoles J E and González P 2002 Analysis of different approaches for storing GML documents. In *Proceedings of the Tenth ACM International Symposium on Advances in Geographic Information Systems (ACM-GIS)*, McLean, Virginia: 11–6

Fornari M R and Iochpe C 2002 Mapping of conceptual object oriented models to Geography Markup Language (GML). In *Proceedings of ICWI 2002*, Lisbon, Portugal: 444–51

Griffiths T, Fernandes A A A, Paton N W, Mason T, Huang B, Worboys M, Johnson C, and Stell J 2001 Tripod: A comprehensive system for the management of spatial and aspatial historical objects. In *Proceedings of the Ninth ACM International Symposium on Advances in Geographic Information Systems (ACM-GIS)*, Atlanta, Georgia: 118–23

Griffiths T, Fernandes A A A, Paton N W, Jeong S-H, Djafri N, Mason K T, Huang B, and Worboys M 2002 TRIPOD: A spatio-historical object database system. In Ladner R, Shaw M, and Abdelguerfi M (eds) *Mining Spatio-Temporal Information Systems*. Boston, MA, Kluwer: 127–46

Griffiths T, Fernandes A A A, Paton N W, and Barr R 2004a The Tripod spatio-historical data model. *Data and Knowledge Engineering* 49: 23–65

Griffiths T, Paton N W, Fernandes A A A, Jeong S-H, and Djafri N 2004b Language bindings for spatio-temporal database programming in Tripod. In Williams M H and MacKinnon L M (eds) *Proceedings of the Twenty-first Annual British National Conference on Databases*. Berlin, Springer Verlag Lecture Notes in Computer Science No. 3112: 216–33

Güting R H and Schneider M 1995 Realm-based spatial data types: The ROSE algebra. *Journal of Very Large Databases* 4: 243–86

ISO14825 2002 Intelligent Transport Systems – Geographic Data Files – Overall Data Specifications (GDF4.0), ISO/TC204 (Intelligent Transport Systems) 2002. Geneva, International Organization for Standardization

OpenGIS 1999 *The OpenGIS Abstract Specification Topic 5: Features*. Wayland, MA, OpenGIS Consortium

OpenGIS 2003 *OpenGIS Geography Markup Language (GML) Implementation Specification*. Wayland, MA, OpenGIS Consortium

Rigaux P, Scholl M O, and Voisard A 2001 *Spatial Databases: With Application to GIS*. San Francisco, CA, Morgan Kaufmann

Shekhar S, Vatsavai R R, Sahay N, Burk T E, and Lime S 2001 WMS and GML based Interoperable Web Mapping System. In *Proceedings of the Ninth ACM International Symposium on Advances in Geographic Information Systems (ACM-GIS)*, Atlanta, Georgia: 106–11

W3C 1998 Document Object Model (DOM) Level 1 Specification. WWW document, http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/

W3C 1999 XSL Transformations (XSLT) Version 1.0. WWW document, http://www.w3.org/TR/1999/REC-xslt-19991116

W3C 2001a XML Schema Part 0: Primer. WWW document, http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/

W3C 2001b *XML Schema Part 2: Datatypes*. WWW document, http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

W3C 2001c SVG 1.0 Recommendation. http://www.w3.org/TR/2001/REC-20010904/

W3C 2001d XML Linking Language (XLink) Version 1.0. WWW document, http://www.w3.org/TR/2001/REC-xlink-20010627/

W3C 2001e Namespaces in XML 1.1. WWW document, http://www.w3.org/TR/2004/REC-xml-names11-20040204/

Zhang J and Gruenwald L 2001 A GML-based open architecture for building a geographical information search engine over the Internet. In Claramunt C, Winiwarter W, Kambayashi Y, and Zhang Y (eds) *Proceedings of the Second International Conference on Web Information Systems Engineering (WISE'01), Kyota, Japan*. New York, IEEE Computer Society: 25–32