Integration of Heterogeneous GML Sources

Keywords: Heterogenous Databases, Feature Catalog, Schema Integration, Schema Analysis, XSLT, GML Browser, SVG, Geodata, Open Source

Gunnar Misund
Associate Professor
Østfold University College
Halden
Norway
gunnar.misund@hiof.no
http://www.ia.hiof.no/~gunnarmi/

Biography

Gunnar is teaching and researching at Østfold University College, Halden, Norway. His main research interests are the geoweb and distributed and mobile computing. He is the founder of Project OneMap.

Harald Vålerhaugen
M.Sc. Student
Østfold University College
Halden
Norway
harald.valerhaugen@norkart.no

Biography

Harald had just finished his Master thesis at Østfold University College, Halden, Norway, where he made valuable contributions to Project OneMap. He is now working at the mapping software company Norkart in Norway.

Abstract

Geography Markup Language (GML), as being an XML Schema meta standard for geospatial data, is becoming widely adopted for both storage and exchange of data between organizations and systems. In particular it is suited for open and seamless exchange of geospatial data on the web. Application schemas are designed based upon a set of GML schemas, allowing for extensions, restrictions and substitution of the base types. Instance documents

from different application domains may thus appear heterogeneous, and general-purpose tools are commonly depending upon complex configuration files to handle unknown dialects. We propose a simple dictionary approach enabling integration of differing GML instance documents. The dictionary is built by a cascading process: 1) If all schemas are available, they are analyzed to identify the origin of the specialized elements, or 2) else, if not all schemas are present, or if there are some missing schemas, the instance document(s) are analyzed using certain heuristics based on the base GML element structure, or, 3) if still some element relations are unresolved, the user is asked to assist the system by a manual mapping procedure. We also propose a non-intrusive strategy where we integrate features from differing sources with minimal tampering of the original data. The implemented proof-of-concept framework is far from complete. However, we demonstrate the main features by using the strategy to develop a simple SVG GML browser and apply on to heterogeneous GML data. The work is part of Project OneMap, a long term effort contributing to the fusion of standard web technologies and geographic content, often referred to as the GeoWeb.

Table of Contents

1. Introduction

Geography Markup Language (GML, **[5]**) is based on XML Schema technology **[3]** and is utilizing a very powerful framework for modeling design rules for XML documents. An instance document is a document that conforms to a particular schema or a set of schemas. The jargon used indicates a close relation to object-oriented principles. Vocabularies can indeed be described without much of an object-oriented approach, and of course, the best solution is often the simplest. However, XML Schema is superior speaking of data typing and more complex design compared to the more well-known Document Type Definition (DTD). GML 2 and 3 are entirely based on XML schemas and are utilizing type inheritance and type substitution to model geographic content. GML 3.0 is backwards compatible with GML 2.x instance documents, though deprecating some of the types. The work related to this paper is based on GML 2.x, but the principles may indeed apply to any existing and future GML version.

Many of the elements and data types in the base GML schemas are not intended for direct use in instance documents, but rather as a foundation for user defined types. `AbstractFeatureType` and `_Feature`, as modeled in the base schemas, are expected to be derived, either to found new and more specialized abstract features, or to serve as an instantiable feature for instance documents. The basic geometries in the GML schemas will typically be used as they are in instance documents, while features and feature collections are abstract, making it a requirement to subtype them. The base GML schemas only serves as a framework for developers to define their own vocabularies, often referred to as application schemas (see **gmlappschema**). Two application schemas might appear truly dissimilar even though they do have the same foundation types. Tools and applications are usually developed to serve or access only one vocabulary of GML, thus being unable to cope with GML based on 'unknown' schemas. If one only consider the top level schemas of different GML dialects, the data appear heterogeneous, making it not entirely trivial to find a common way to utilize them as what they are, data modeled from a mutual origin.

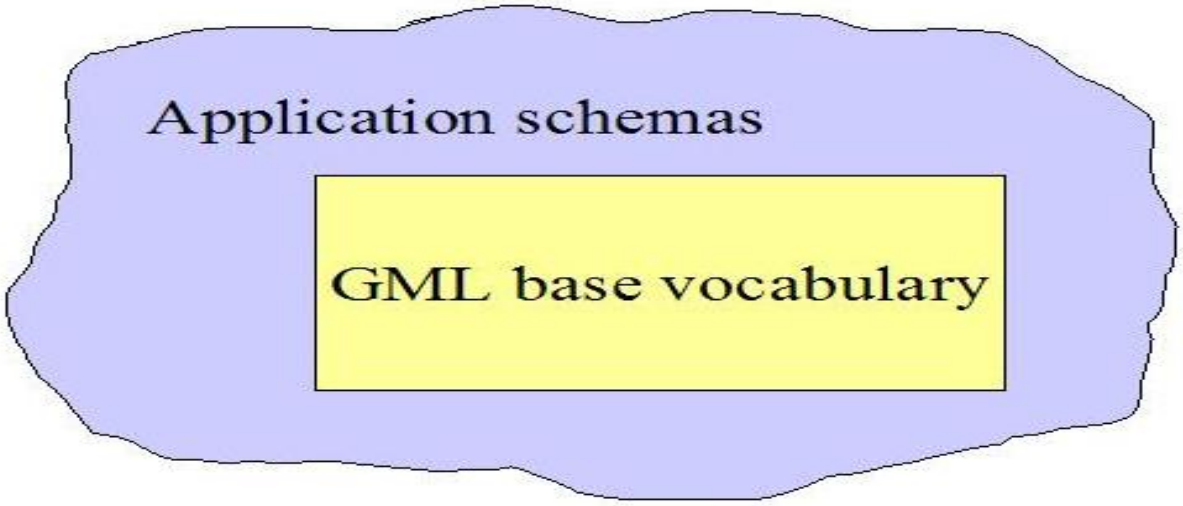

Figure 1: GML application schema

GML is designed to be very flexible and able to model almost any kind of feature, even those not thought of yet. Provided that the basic rules of structure and data-typing as explained in the GML specifications are conformed to, developers have a great freedom of choice, both considering how a documents elements should be nested and structured and how elements relates to the base schemas. These loose definitions is probably a reason for the popularity (in certain users communities) of GML. Most GIS systems can store or export their data to GML, thus making it possible to distribute and exchange data on a non-proprietary format.

As addressed by ESRI **[1]** , the richness that can be introduced in the feature model using GML is not necessarily helping to achieve interoperability. Paradoxically enough, while trying to embrace modeling rules for virtually every possible feature, restrictions ends up too loose and the strengths appear like weaknesses, especially considering interoperability. A profile of GML is a restriction of the capabilities of the base GML rules, either agreed upon or made physical through selecting, restricting and/or deleting functionality in base schema files, as illustrated in **gmlprofile** . Based on a common agreement that layers should contain homogeneous features, and that the number of sub-layers should be limited, ESRI has taken the initiative to define a standard profile of GML to avoiding awkward design. Upholding this profile, arbitrary GML can be utilized by a wide range of ESRI products.

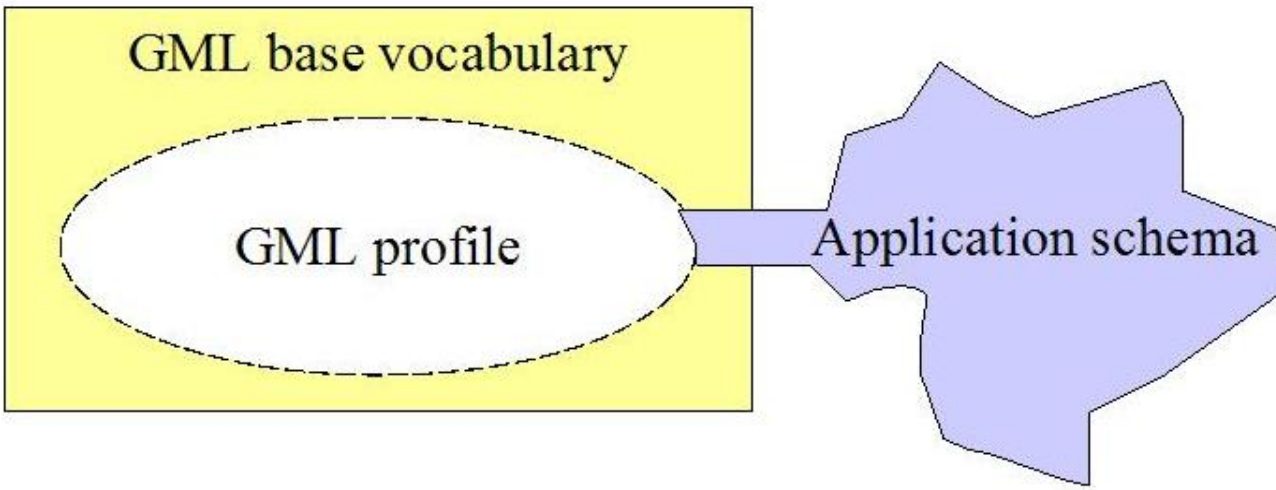

Figure 2: GML profile and application schema

There is not doubt that profiles are created for good reasons, trying to simplify interoperability. However, as long as the GML standard is fairly flexible considering how data is modeled, schemas not in adherence to best practice guidelines or profiles will be made. A totally generic tool will have to be able to interpret all kind of GML, maybe with the exception of incorrect GML. Integration of arbitrary GML is one of the challenges in Project OneMap where one of the main goals is to provide access to a comprehensive and detailed world map **[2]** . The data collection is done in an incremental and uncoordinated manner, by submissions from several contributors. Ideally, all submissions should be on one agreed up-on format. Of course, this option is excluded; GML is probably as close as we will get to a common format. To demand data on a certain format is unrealistic, and writing conversion code for a range of dialects is too time consuming, not too speak of the risk of loss of semantics and precicsion of the original data.

In this paper we approach this fundamentally complex problem of integrating heterogenous GML content from a pragmatic and realistic point of view. In the next section we point to some existing solutions and describe two of them in more detail. Then we introduce the concept of Cascading GML Analysis. With this tool at hand we model a GML framework enabling encapsulation of content on diverse dialects with minimal changes of the original data, following what we have termed a Lazy Integration approach. We then demonstrate a possible use of the combination of these two concepts by demonstrating a generic GML browser based on SVG technology. We close the paper by some conclusions and directions for further work.

In the more technical parts of the paper we assume that the reader has basic knowledge of core XML technologies, Scalable Vector Graphics (SVG) and the specifications from

the Open GIS Consortium (OGC), in particular Geography Markup Language (GML). For more details on these topics, see for instance **[3]** , **[4]** and **[5]** .

2. Related Work

Let there be no doubt that there exists tools and software to utilize heterogeneous GML, also automatically by data-/schema-analysis. Probably the most powerful ones, are however not open source. Snowflake Software has a developed a loader to populate Oracle Spatial/Locator databases with GML 2 data This is done by parsing the schemas, and the creation of tables and loading is done automatically **[6]** . Safe Software is offering products for data translation and conversion and has also developed software to utilize heterogeneous GML **[7]** . As a part of the GeoTools project a GML 2 WFS datastore is currently under construction/testing **[8]** . GeoTools is "The leading open source Java library for developing of OpenGIS solutions". This will hopefully lead to a solid implementation, able to deal with heterogeneous GML in an elegant manner.

2.1 Schema Mapping

Some applications utilizing arbitrary GML requires mapping files describing the vocabulary for the application. They offer no functionality to automatically generate these mapping files, leaving it as a manual task for the user. This is of course not an impossible job if you want the application to read your own GML vocabulary. However, setting up mapping files manually when you are supposed to access several sources of GML, possibly complicated ones unknown to the developer, this work is extensive and time consuming. Mapping is probably the only possible way to utilize heterogeneous GML without having to parse the schemas or analyze the structure of documents. One advantage can be mentioned though, when specifying mapping files you can include only the content you are interested in by leaving out the unwanted types. The following two sections present two applications using mapping files to utilize heterogeneous GML.

2.1.1 JUMP

The Unified Mapping Platform (JUMP) is an open source GUI-based application for viewing, editing and processing spatial data **[9]** . JUMP utilizes the Java Topology Suite (JTS) **[10]** , also developed by Vivid Solutions, to implement the OpenGIS Simple Features Specification (SFS). The JUMP Workbench is designed for both development of conflation algorithms and as a general purpose tool for the visualization and edition of spatial data. To be able to process heterogeneous GML, you have to specify a GML Input Template, identifying collections, features, geometry and non-spatial properties. By using the template you are able to extract a single FeatureCollection from a GML file, meaning that you have to specify multiple input templates in order to import more

than one collection. For additional functionality, the application can be extended by providing plugins. Users can also write their own drivers to different data sources, allowing the application to work with proprietary formats. The XML snippet underneath shows how a feature is mapped into the JUMP toolkit, specifying the feature collection element, feature element and the geometry that is to be drawn out in the application. In addition the properties must also be mapped, in order to be accessed through the application. The work associated with building such templates for your data, depends on the richness of features represented. If you want to map one or two feature types to JUMP, the effort required is minimal.

```
<?xml version="1.0" encoding="UTF-8"?>
<JCSGMLInputTemplate>
    <CollectionElement>CityModel</CollectionElement>
    <FeatureElement>Road</FeatureElement>
    <GeometryElement>linearGeometry</GeometryElement>
    <ColumnDefenitions>
        <column>
            <name>classification</name>
            <type>STRING</type>
            <valueelement elementname="classification"/>
            <valuelocation position="body"/>
        </column>
        <column>
            <name>number</name>
            <type>INTEGER</type>
            <valueelement elementname="number"/>
            <valuelocation position="body"/>
        </column>
    </ColumnDefenitions>
</JCSGMLInputTemplate>
```

2.1.2 Cleopatra

This project is a proof of concept for generating Scalable Vector Graphics (SVG) on the fly from GML. It is intended to act as a publishing layer between a GML data source and the end user. The conversion process is parameter driven and customizable **[11]** . The process of publishing generic GML data as SVG is not automatic. The plugin requires a configuration settings XML file, defining Xpaths to indicate which features and non-spatial data to expose. Listed underneath is a fraction of the configuration file for Cleopatra, where XML XPath constructs are used to point to specific parts of the

document. By pointing to external Cascading Style Sheets (CSS), the feature geometries are styled for viewing. Using XPath for retrieval of features is probably the best way to point to the features inside the document. This allows closer control of the data imported, also probably making it easier to utilize complex vocabularies. However, XPath is can turn out relatively complex, thus requiring highly skilled personnel.

```
[...]
<!-- this has various GML application Schema specific xpaths -->
<settings:xpaths>
    <!-- absolute xpath that will find features -->
    <settings:feature>//osgb:topographicMember</settings:feature>
    <!-- relative xpath from feature to feature type -->
    <settings:featureType>./*[1]/osgb:theme</settings:featureType>
    <!-- relative xpath from feature to attribute data-->
    <settings:attributeData>./*[1]/*[text() and count(text()) =
1]</settings:attributeData>
    <!-- relative xpath from attribute data to data name-->
    <settings:attributeDataName>local-name()</settings:attributeDataName>
    <!-- relative xpath from attribute data to data value-->
    <settings:attributeDataValue>./text()</settings:attributeDataValue>
</settings:xpaths>
[...]
```

3. Cascading GML Analysis

For most users, applications like JUMP and Cleopatra are used to work with one GML vocabulary only. Specifying the mapping files manually is therefore not a too significant obstacle to overcome. Nevertheless, if a tool was available for users, enabling them to analyze their schemas and at least do a partially automatic generation of these templates, this would be a significant improvement.

When GML is valid, and all schemas are available from the urls specified, information about the origin of application specific types can be extracted from the schemas. Schema parsing will thus be the primary source of meta information about GML vocabularies. However, relying on the schemas being available, especially when exchanging data over the Internet, requires a tad of naive optimism. Most applications that are meant to handle heterogeneous GML will probably succumb to broken schema links. Is it so that unknown GML is worthless to analyze if the application schema(s) are inaccessible? We

introduce a method to handle heterogeneous GML, that allows for missing meta information, either as a result of broken schema links or incongruity between schemas and instance documents. This method is cascading, invoking a chain of methods to analyze a document's elements.

By combining the forces of structural knowledge of all GML documents, and the specific knowledge of each vocabulary defined through the application schemas, we will now try to outline a robust solution for analysis of GML schemas and documents. The framework is extensible to encourage implementations of new methods for document analysis.

3.1 Schema analysis

Schema analysis is a pretty straight forward task, even though it is a cumbersome one. Validating parsers do for example have to parse schema vocabulary in order to check structure and values in an instance document. When dealing with GML schemas, we can be certain that the vocabularies has a `targetNamespace`, telling us which namespace is being described in the file(s). One file can contain the whole vocabulary, or it may use the `include` element to bring in other files also describing the same (or no) namespace. The schema can utilize the constructs of the included schema, just as constructs within the same file. A good example of this modular design is the GML 3 schemas, where developers usually utilize a subset of all the available schemas. However, bear in mind that the includes are recursive. If you want to bring in elements or types defined in another namespace, the files have to be linked to in your schema, using the `import` element. This element allows for utilization of another vocabulary, by specifying the desired namespace and the physical location of the schema file. **schemadesign** shows how the file components are related when working with XML schemas.

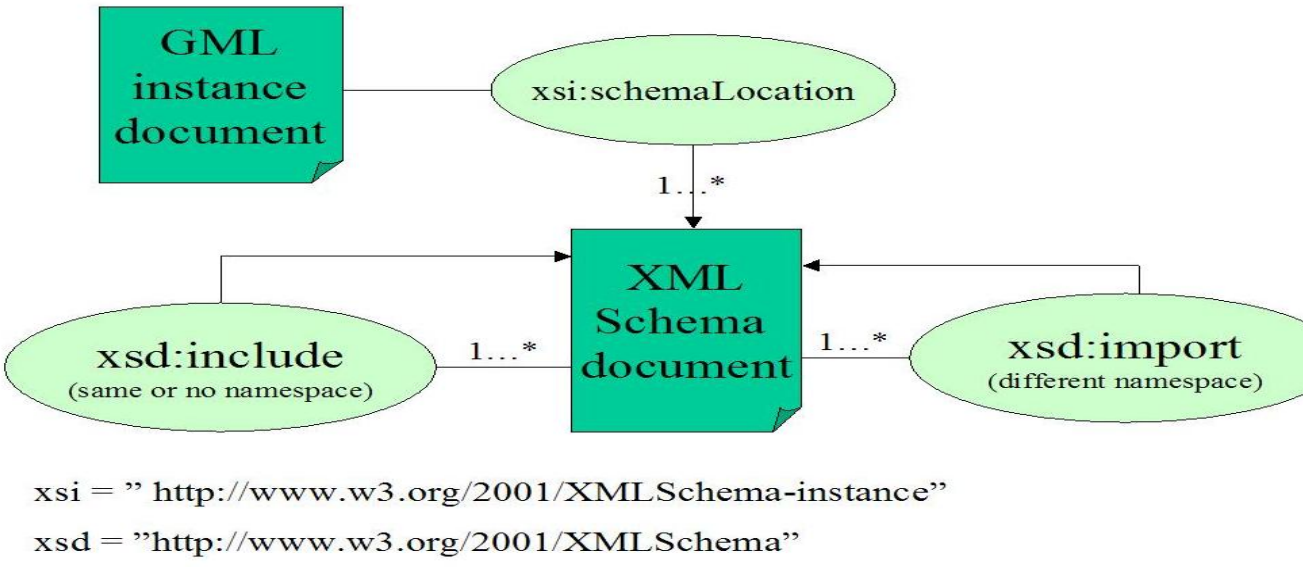

Figure 3: Defining a GML vocabulary

When analyzing schemas, they main objective is to find out how elements relate to other elements, and possibly if they er directly or indirectly derived from a GML type. By gaining easy access to this information, a utilizing application can treat elements depending on their base type. Features, feature collections, properties and other elements can be treated in a generic way, meaning that the application can work with heterogeneous GML documents in a sensible way. There is nothing mysterious about making a mapping file of a vocabulary, but it greatly simplifies meta-data access for applications. All element declarations in the schemas are described in an XML file which contains information about instance type, and possibly GML base type, substitution group and GML base substitution group. The following example shows an element, `NightSiteBar`, mapped from a schema into a mapping file. The `instanceOf` element contains the name and namespace for the type this element is an instantiation of. This can be a GML type, a user defined type, or maybe even one of the types defined in the XML Schema vocabulary. If the element is only indirectly descending from a GML type, the `gmlDerivedType` element contains the name and namespace (always being GML namespace) of the type it derives from. The same logic applies to the `substitutesFor` element and `baseSubstitutesFor` element. In this example it is obvious that `NightSiteBar` is a generic GML type, but the relationship is only visible through a chain of derivation. Part of the analyzed schema is listed underneath the mapping file, to illustrate how derivation is mapped to a `TypeMap` element.

```
<TypeMap id="d1e13">
        <appElement>
                <localname>NightSiteBar</localname>

        <namespace>no:hiof:onemap:gml:appschema:example1</namespace>
        </appElement>
        <instanceOf>
                <localname>NightSiteBarType</localname>

        <namespace>no:hiof:onemap:gml:appschema:example1</namespace>
        </instanceOf>
        <gmlDerivedType>
                <localname>AbstractFeatureType</localname>
                <namespace>http://www.opengis.net/gml</namespace>
        </gmlDerivedType>
        <substitutesFor>
                <localname>_NightSiteFeature</localname>

        <namespace>no:hiof:onemap:gml:appschema:example1</namespace>
        </substitutesFor>
        <baseSubstitutesFor>
```

```
                <localname>_Feature</localname>
                <namespace>http://www.opengis.net/gml</namespace>
        </baseSubstitutesFor>
</TypeMap>
```

```
[...]
<xs:element name="NightSiteBar" type="NightSiteBarType"
substitutionGroup="_NightSiteFeature"/>
<xs:element name="_NightSiteFeature" type="gml:AbstractFeatureType"
abstract="true" substitutionGroup="gml:_Feature"/>

<xs:complexType name="NightSiteBarType">
        <xs:complexContent>
                <xs:extension base="NightSiteType">
                        [...]
                </xs:extension>
        </xs:complexContent>
</xs:complexType>

<xs:complexType name="NightSiteType" abstract="true">
        <xs:complexContent>
                <xs:extension base="gml:AbstractFeatureType">
                        [...]
                </xs:extension>
        </xs:complexContent>
</xs:complexType>
[...]
```

When parsing an XML document into another one, XSLT can appear as an obvious choice of implementation language. Even though some operations can appear cumbersome, XSLT offers what is required. XSLT 2.0 is at the time of writing statused as a 'Last Call Working Draft' at W3C. The Saxon8 'basic' XSLT and XQuery processor, implements the "basic" conformance level for XSLT 2.0, XPath 2.0 and XQuery 1.0 processing **[12]** . Using some of the functionality from XSLT 2.0 to simplify the implementation, we have developed a schema parsing XSL Transformation stylesheet, that parses one or several vocabularies into one mapping file. The stylesheet can either follow the `schemaLocation` attribute value from an instance file, or alternatively a specified 'root' schema or a customly provided `schemaLocation` string provided as command line parameters to the stylesheet. All elements defined globally or inline in one

of the vocabularies, are mapped by the stylesheet, traversing all linked schemas to find the origins of application specific types.

A bundle with sample data and the transformation stylesheet is available for download and testing from our web server **[19]** . We have tried to collect a subset of test data, representing both simple application schemas, more complex ones with include and import statements, and finally one representing Lazy GML Integration (**lazy**) as it will be used in Project OneMap.

First, our small 'hello world' application schema and instance document. It models features in the small city of Halden, Norway. The vocabulary is fully defined with one schema file, and should impose no serious challenge for the schema parser. The elements are instantiated from types indirectly deriving and substituting for the base GML types. As we can see from figure **hbn_mapping** , the elements are mapped, typewise, but all restrictions and extensions done from the base types are not information available from the mapping file. It is important to note that if the schema-parsing is supposed to be used in an editing environment, the changes carried out on the properties, have to be checked for validity against the original schemas, before the data update is finalized.

Figure 4: OS MasterMap schema hierarchy

So, lets move on to a more complex dataset. A broad range of companies has embraced GML, and adopted it as their data interchange format. Ordnance Survey, the UK mapping agency, provides detailed, property rich, spatial and non-spatial data of UK as GML. The schemas are naturally quite complex, even though they have strived to keep them as simple and easy accessible as possible. The vocabulary is defined through a number of schemas, all in the same namespace, logically modularized (**osmm_schemas**).

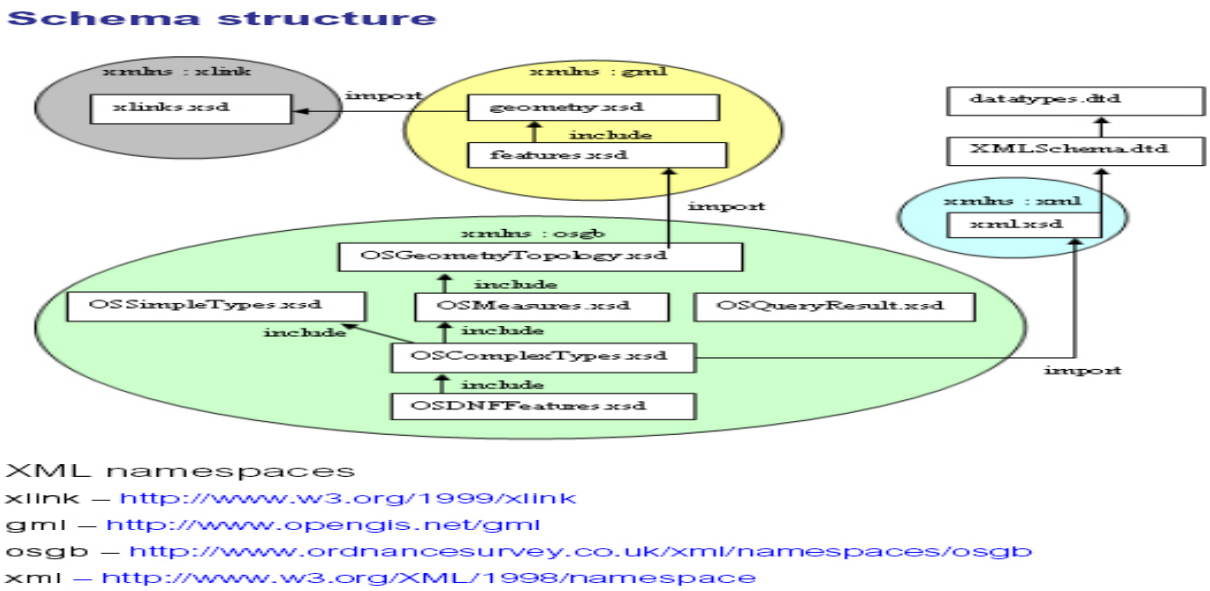

Figure 5: OS MasterMap schema hierarchy

Since the elements in the OS MasterMap schemas, are all in the same namespace, the issue here, is to navigate through the include statements, parsing the elements as they occur. If the schemas are parsed directly from their location at the Ordnance Survey web server, there is of course an issue of the reliability of internet connections and speed. However, as soon as the schemas are mapped, the mapping file would answer all questions regarding the relation between the application specific types and the base GML types. One of the OS MasterMap features, mapped to the dictionary format, is shown in the listing underneath. From the mapping information, it is obvious that the feature is created in accordance with the best practice guidline, (indirectly) deriving from `AbstractFeatureType` and (indirectly) substituting for `_Feature`.

```
[...]
<TypeMap id="d2e48">
 <appElement>
        <localname>BoundaryLine</localname>
        <namespace>http://www.ordnancesurvey.co.uk/xml/namespaces/osgb</namespace>
 </appElement>
 <instanceOf>
        <localname>BoundaryLineType</localname>
        <namespace>http://www.ordnancesurvey.co.uk/xml/namespaces/osgb</namespace>
 </instanceOf>
 <gmlDerivedType>
        <localname>AbstractFeatureType</localname>
```

```
        <namespace>http://www.opengis.net/gml</namespace>
 </gmlDerivedType>
 <substitutesFor>
        <localname>_BoundaryFeature</localname>
        <namespace>http://www.ordnancesurvey.co.uk/xml/namespaces/osgb</namespace>
 </substitutesFor>
 <baseSubstitutesFor>
        <localname>_Feature</localname>
        <namespace>http://www.opengis.net/gml</namespace>
 </baseSubstitutesFor>
</TypeMap>
[...]
```

If URLs to included or importet schemas are invalid, the information from these will of course not be mapped. This means that there will be missing vocabulary type-maps, for element declared in these files. In addition, type-tracking for element in available files, could prove incomplete, if their type hierarchy is fully or partially defined in these files. Using SAXON as XSLT engine, it will spit out an error if the document is unavailable, however this will not inflict on the further parsing of the vocabulary.

3.2 Structural analysis

A schema parser may in some cases fail to provide a complete mapping file for an application schema. There might be several reasons, including missing or unreachable schema files and not entirely consistent schemas, leading to ambiguous or incomplete results. In such cases we can attempt to parse the instance documents and analyze their content based on the structure of the elements. The GML specifications will offer us the basic rules, and the document can be parsed filling out missing pieces in the mapping dictionary.

GML documents should be built according with some basic structural rules (GML 2.x):

1. The root element must be directly or transitively descended from `gml:AbstractFeatureCollectionType`.
2. Relationships between classes (e.g. features/feature collections) should be represented through associations and/or properties, possibly restricting membership. A property can contain `simpleTypes` or other classes. This is the fundamental construction model. There is no basic restriction of how deep nestings can be.

In GML 3, however, it is bit more complex. However, as long as we stick to GML 2.x, the base framework is restricted enough for us to be able to do fairly simple structure analysis. A common way to model application schemas is to define new properties and

roles, describing the vocabulary more accurate in your 'own words', but stick to the base geometric constructs. To provide maximum interoperability between heterogeneous GML sources, developers should strive to inherit as specialized base GML types as possible. This way a generic parser can operate on the data more accurately.

Using what we know about type relationships, identification of elements should be possible based on their parent, children or neighbor elements. A framework including structural analysis is presented in **cascadingprocess** .

3.3 Manual analysis

When schemas are incomplete, inaccessible or instance documents are not in accordance with the information parsed from a schema, we will try to parse documents and analyze them based on their structure and the known types within. This will sometimes succeed, but can not be considered a fool-proof method. There might occur situations where there is a question whether an element is one of two possible, or maybe there aren't any presented options. This is where the software surrender, and we should present to the user the unidentified elements, trusting that he will fill out the missing pieces.

3.4 Cascading process

This section describes the process and framework for combining the forces of several analyzing methods. This process is based on SAX-parsing of instance documents, where the GML elements are mapped to an internal tree-model with name, namespace and meta-information. Information concerning the origin of elements is attempted revealed by so called resolvers, all implementing the interface `GMLTypeResolver`. The implementation is done in Java, using JAXP to SAX-parse the instance document **[13]** . SAX parsing is event based, meaning that events are generated by the parser when it reaches specific constructs in an XML document. By implementing `ContentHandler` interface the parser reports a number of events to this class. Among others, each start and end element is reported and caught by the registered `ContentHandler`, see **contenthandler** .

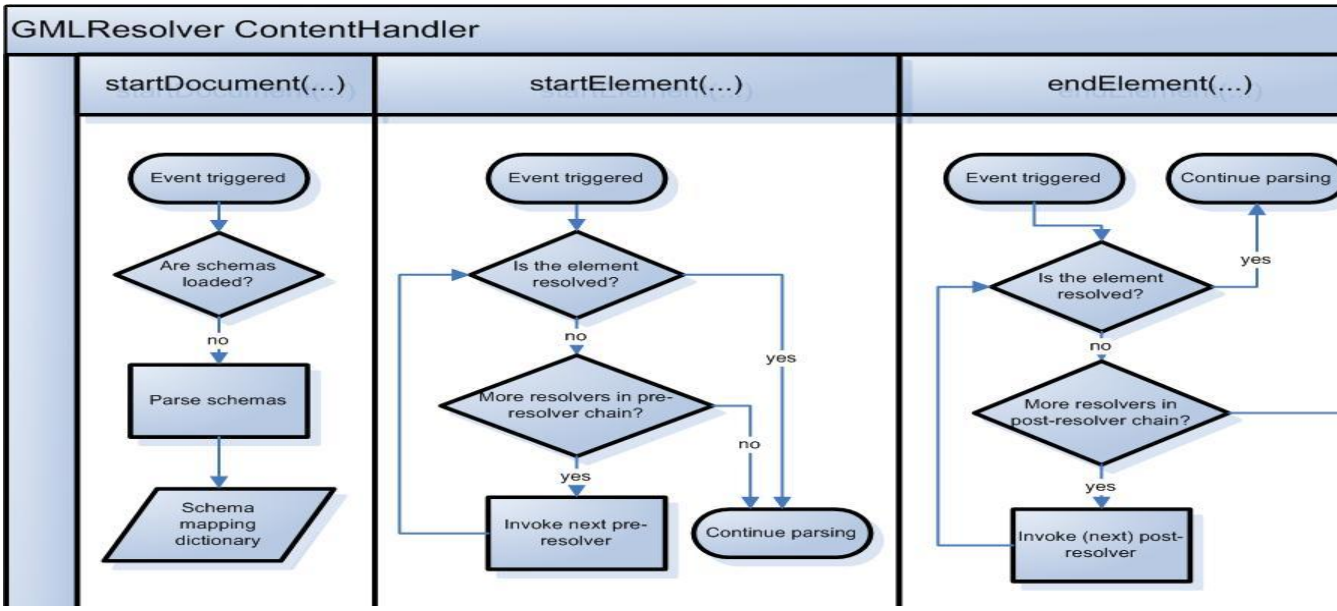

Figure 6: ContentHandler methods

In addition to the element being added to the internal tree-representation, the resolvers are invoked here. A chain of pre-resolvers attempts to identify the element in the `startElement` method, while all non-resolved elements are attempted resolved by a chain of post-resolvers when reaching the `endElement` method (**resolvers**). The number, order and type of resolvers are specified using an array of resolvers. If an element is fully identified, there will be no further attempts to resolve them, both pre and post. Therefore it is important that resolvers relying on qualified guesses do report correct types. The framework can be extended to support partially resolving, meaning that an element can be structurally identified as one of a number of types. Thereafter the element identification can be further limited by subsequent resolvers based on the pruning already done.

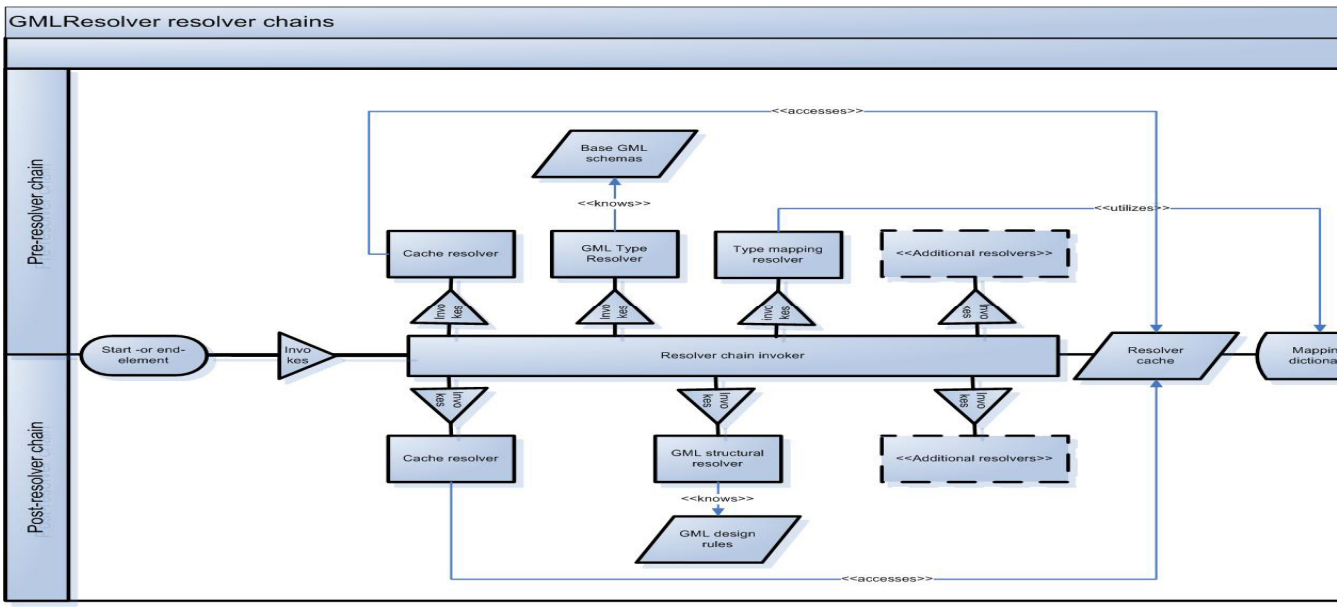

Figure 7: Resolver chains

4. Lazy GML Integration

In this section we describe an integration strategy used in Project OneMap. OneMap is a long term project contributing to the fusion of contemporary IT tools and techniques, in particular Web based ones, and geographic content **[2]** . The OneMap infrastructure is constituted by three main services. The Gateway is a WFS based query an retrieval interface interacting with a datastore, the Repository. In addition there is a ClearingHouse which handles contributions and integrate them into the Repository. In this sections we outline the rationale for the integration methodology and describe the GML modeling necessary to achieve our goals.

4.1 OneMap Repository

The main purpose of the OneMap Repository is to provide access to a world wide multiscale database of geodata. Our main focus is on data retrieval of GML formatted vector data. The content is modeled using a traditional layered map organization, where each layer is representing a certain feature type, for instance coastline, with global extent. The user may query and retrieve data according to feature type in a given area and with a specified resolution, typically through a simple WFS query. The Repository is not supporting complex queries defined by e.g. certain properties since we are assuming that the user will import the base data into their own application for further analysis and presentation.

The Repository is constructed in a bottom up and incremental manner. Data may be contributed by a wide range of parties, and is integrated into the framework using a peer review process. One possible integration strategy would be to convert the data to a common GML model. However, we have chosen an orthogonal approach governed by the principle of non-intrusive integration. Basically this means that we want to keep the contributed data as close to the original as possible. The main objective behind this strategy is to avoid loss of geometric and semantic precision, which is an unavoidable consequence of any format converting procedure. Another important reason for this choice is that we want to avoid the complicated and time consuming manual labor involved in a conversion process. For a more detailed description of the organization and construction of the Repository, see for instance **[14]** .

4.2 OneMap Integration

There are two main aspects of the integration methodology in OneMap; geometric and semantic integration. The former ensures geometric consistency when combining data

from diverse sources which describes (parts of) the same geographic entity, e.g. when building a global coastline based on chunks from national mapping agencies. The latter is to classify contributed features according to a common Feature Type Catalog. The OneMap Feature Type Catalog is built incrementally governed by peer review, and may be view as a thesaurus or a simplified ontology. More details on related approaches to semantic integration is found in **[15]** . The geometric integration corresponds to the problem often referred to as map conflation (see e.g. **[16]** and references therein), and is out of the scope of this paper. In the following we assume that each submitted feature or feature collection may be classified according to the OneMap Feature Catalog.

Our goal is to design a general strategy where we model each feature class in the Feature Type Catalog as an encapsulating GML class, substituting for the `_IntegratingFeatureCollection`, which again is substituting for the abstract `_FeatureCollection` element. The integrating featurecollection corresponds to a traditional map layer. Further, we want to restrict a given OneMap integrating featurecollection to contain only the kinds of features that are considered to be of the same class according to the Feature Type Catalog. A given integrating featurecollection, e.g. Buildings, may then contain a set of external feature types defined in the schemas of the contributing sources, and only these feature types. Another design goal is that is should be easy to include a new external feature type in a given integrating featurecollection.

A result of this method is that each original feature is preserves in the original state. The only alteration made to a contributing data set is that the featurecollections may be disassembled and distributed to the appropriate integrating featurecollections. The approach may be viewed as a minimal version of schema integration as known in the domain of federated databases **[17]** .

So, down to the bone, how do we choose to integrate the data? The theory is simple, we want to include features and GML types into our OneMap system. A feature should be included into an instance document as is, meaning that we have to deal with it in a generic way. The schema standard and namespaces does of course allow us to import as many namespaces into our application schemas as we desire, so the focus have to be how to integrate data to fit our needs.

We have chosen a common strategy for structuring the data in our system, as layers of related features. However, as you have probably understood by now, these features are not homogeneous in terms of their GML definition, nevertheless they are heterogeneous representations of the same real world objects.

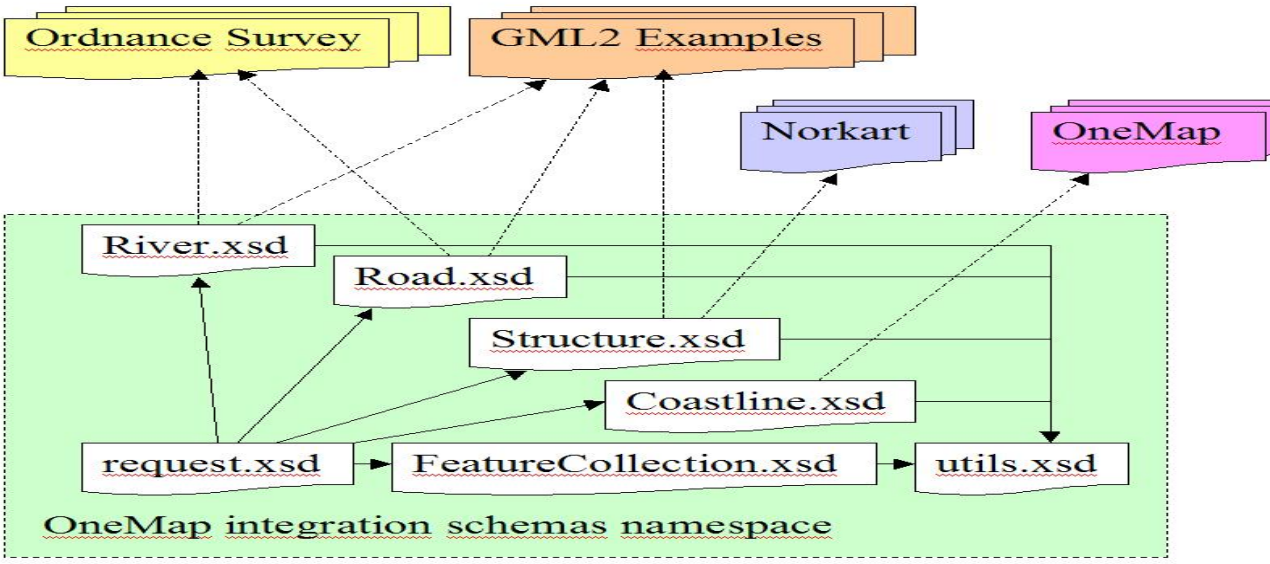

Figure 8: Integrated schema hierarchy

The figure **integratedschemas** , shows how the schema hierarchy for OneMap feature integration is constructed, providing one schema file for each layer. This is naturally just a question about modularization, since all integrating schema files are in the same namespace. All integrated features, are imported from different vocabularies. There are no non-spatial feature types defined in the integration schemas, all are actually integrated from different namespaces. Feature member membership are restricted using the '*barbarians at the gate*' approach, presented in the GML2 specification, where a `FeatureAssociationType` is restricted to contain an abstract or non-abstract feature, which other elements must substitute for in order to be a child of the association.

5. Generic GML Browser

In order to test both the cascading GML analysis and the lazy integration strategy, we have implemented a simplistic GML to SVG transformation. The main idea is to visualize the geometric constructs and provide easy acess to the the non-geometric properties of the features. Transformation are done on GML instances, and the SVG application can not load data from other sources. However, this is made to outline strategies for handling instance documents, when there are mapping files present.

By accessing a mapping file, constructed using the cascading method presented above, the transformation stylesheet can convert any valid GML 2.x instance document into a SVG document, see **generictool** . It is however required that the cascading analyzis succeeded in identifying the elements in the GML application schemas describing a document. The structure of the final SVG-document, is identical to the GML file, in

terms of nesting of features and feature collections. If the transformation comes over unknown elements, it will not continue parsing the sub-tree of this element.

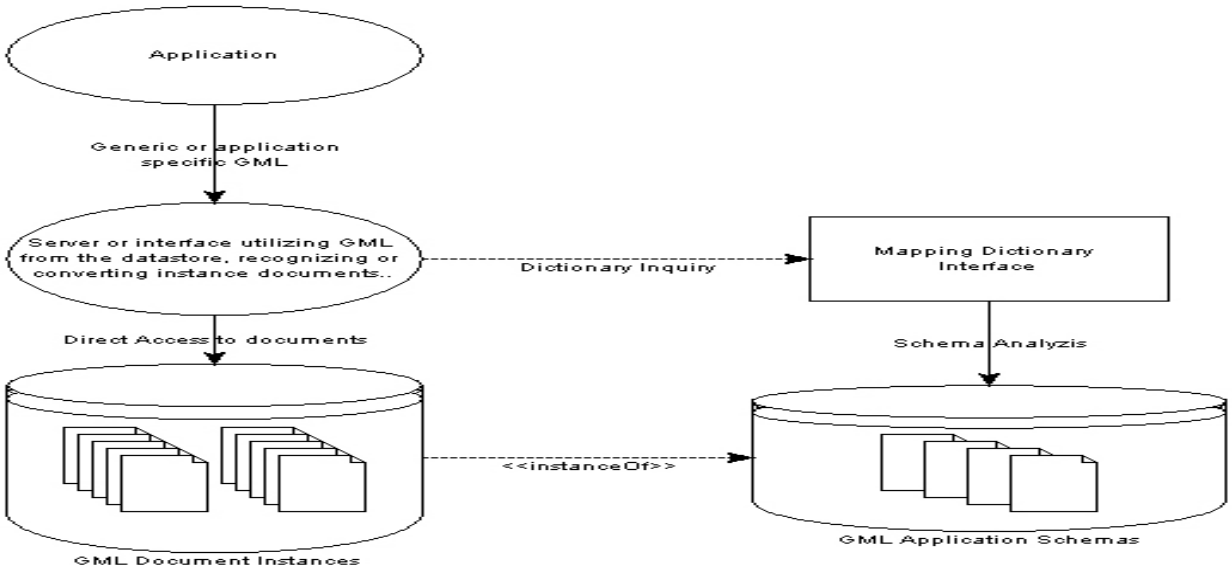

Figure 9: Utilizing dictionary to parse arbitrary GML

Styling of the different features, has not been an issue in this work. Therefore, we have only introduced a very limited way of styling, only making it possible to apply one style for all features from one namespace. This is of course not adequate if more than one type of feature from a namespace is integrated into a vocabulary. The *OneMap GML editor*, presented at SVG Open 2003 **[18]** , is a lightweight SVG editor for editing and displaying GML 2.1 compliant data. The server converts GML to SVG, for the client to display it and offer editing possibilites. One of the stated challenges for further work, was to develop a more robust method regarding what kind of data the application was able to utilize and edit. By implementing the next editor version, using the principles described in this article, the editor will be able to handle arbitrary GML, as opposed to only utlizing a specifically created GML format.

The integration example from the preceding section, has integrated features from *ordnance survey*, *GML2 spec example*, *Norkart* and *OneMap*. Applying the SVG transformation on these data, results in a map containing all integrated feature, see **integratedsvg** .

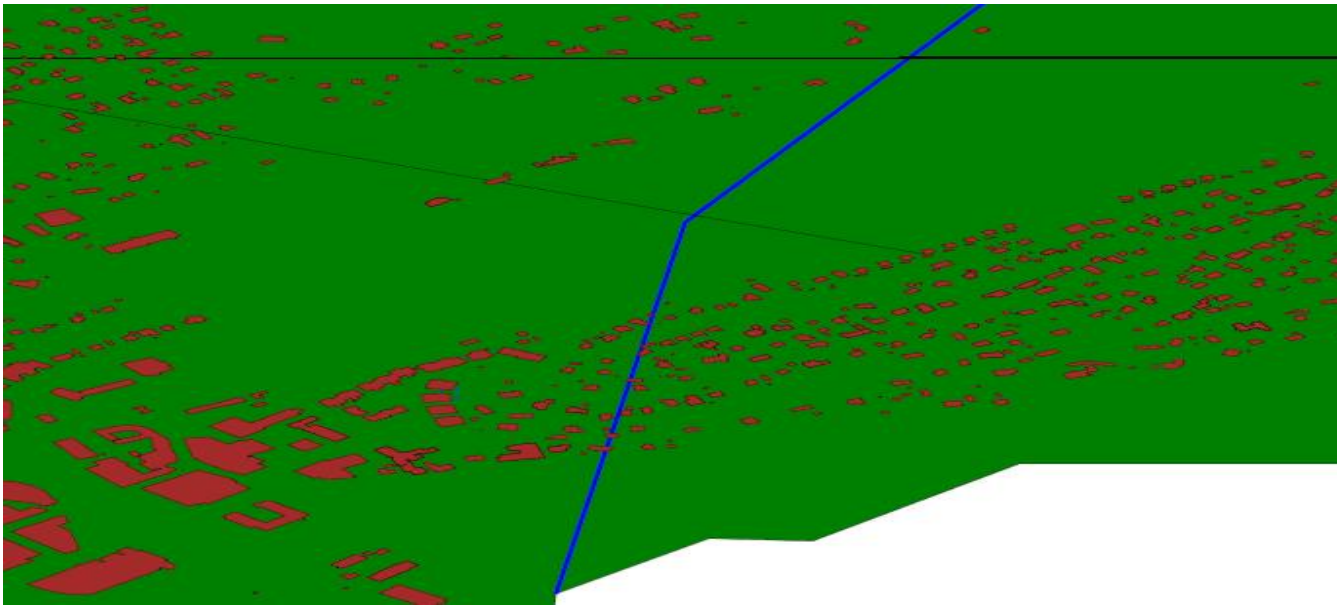

Figure 10: Integrated GML transformed to SVG

The styling is as simple as possible, allowing users to specify custom styles for each namespace present. This file is the specified when converting. All namespaces, that has not been applied a user style, will get a default style. The style to specify is identical to the value of the SVG *style attribute*, and is applied to all features using a named class.If the user wants to specify a custom style, a style as that listed underneath, will be stored in a separate file, then the filename is passed to the transformation as a command line argument.

```
<style:styles xmlns:style="userstyle" targetNamespace="userstyle">
        <style:style>
                <style:namespace>default</style:namespace>
                <style:stylestring>stroke:black; stroke-width: 0.05%;
fill:white; fill-opacity:0.0</style:stylestring>
        </style:style>
        <style:style>
                <style:namespace>http://www.onemap.net</style:namespace>
                <style:stylestring>stroke:black; stroke-width: 0.05%;
fill:green</style:stylestring>
        </style:style>
</style_styles>
```

Even though the GML to SVG transformation can be applied to all GML 2.x data, that there is a mapping file available for, the integration namespace has introduced an

attribute that can be used on a feature collections, representing a feature layer, e.g. *roads* or *rivers* (**integratedsvg**).

Figure 11: SVG integrated layer visibility

It is pretty trivial to draw the geometries of GML in SVG, considering that most geometry types in instance documents are original GML elements. The transformation do however also map the non-spatial element types and values, into the SVG file, making it possible for users to review their GML data. By clicking on the different features, information stored in the features, toghether with the type information, can be accessed (**finfo**). As for now, the feature type information given is pretty thorough, maybe a bit to extensive for an ordinary viewer, but as a valuable supplement for companies wanting to review their GML data, not having a proprietary viewer.

Figure 12: Feature information window

For quick viewing of GML data, the transformation can be applied to a type-mapped file, without the need of any styling at all. Default styling will then be applied to all features. Top10nl example data, will e.g. be converted into a SVG file, as shown in figure **top10_svg** .

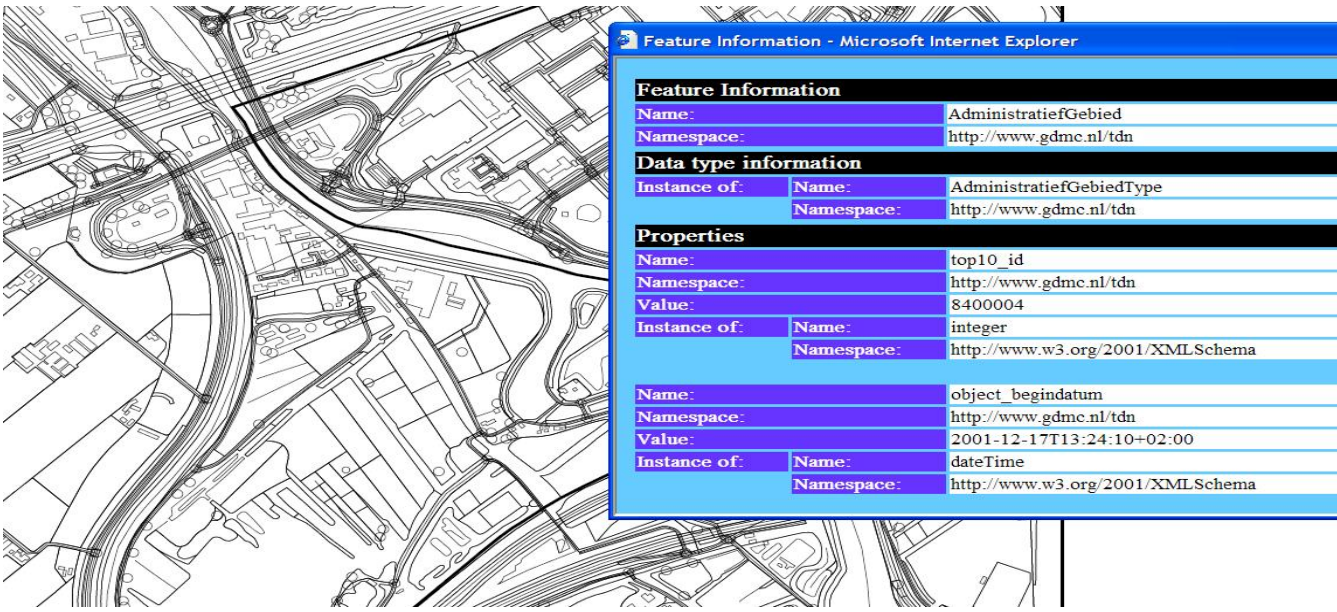

Figure 13: Ordnance survey data with default styling

6. Final Remarks

The Cascading GML Analysis is to be considered a proof of concept. We have demonstrated the ability to derive the roots of GML application elements by schema analysis, but also indicated that important information may be derived from the instance documents in the case of missing or corrupt schemas. This method results in a mapping file containing information about all elements occurring in a instance document. Their relationship, if any, to the GML framework types are appearant from this mapping. The mapping file can thereafter be used in a wide variety of applications dealing with mixed GML content from diverse sources in a generic way, for instance as demonstrated with the Generic GML Browser.

The framework is however very limited speaking of functionality and will be subject to further enhancement in upcoming OneMap projects. We encourage readers to download and try out the code and concepts presented in the paper and to suggest improvements **[19]** .

The transformation can get pretty difficult to follow when tracing `import` and `include` statements recursively. If element types must be traced into other files, there might be several identical mappings for the same element. The search is performed in a depth-first recursive manner, and the algorithm has no way to determine if a type already has been mapped when winding up and tracing other includes or imports. To clean files with identical mappings we have implemented a separate XSLT transformation called `wash.xslt`. A better solution to this problem should be found.

Namespaces may be changed throughout a file, defining new ones in nested elements, possibly also changing prefixes for them. The transformation has not been tested on schemas with such constructs, and will probably fail. Within a namespace, elements can be redefined. The mapping will not take into consideration elements that have been redefined.

The demonstrated GML browser transforms mixed GML content to SVG, benefiting from the mapping file generated in the cascading analysis. It was merely implemented as a proof of concept tool, to show how we can integrate and handle heterogeneous GML in a generic way. To make it more robust and useful, two major enhancements should be implemented:

1. Integration of GML sources using XLink constructs should be supported. Importing data from several sources physically into a wrapping instance document is obviously not the way to go.
2. Consideration of Spatial Reference Systems (SRS) or Coordinate Reference Systems (CRS) were not taken when implementing the system. This means that all the geometries being integrated or otherwise treated, must be given in one SRS only. Converting to and from these systems would bring the integration one step further.

Acknowledgements

The authors are grateful for the financial support offered by Østfold University College, which made it possible to attend GML Dev Days 2004.

Bibliography

[1]

GML Profiling: Why It's Important for Interoperability, ArcUser April-June 2003 (www.esri.com) http://www.esri.com/news/arcuser/0403/special-section/gml-profiling.pdf.

[2]

Project OneMap, Østfold University College, http://www.onemap.org.

[3]

Extensible Markup Language (XML), http://www.w3.org/TR/REC-xml/.

[4]

Scalable Vector Graphics (SVG) 1.1 Specification, http://www.w3.org/TR/SVG/.

[5]

Open GIS Consortium, http://www.opengis.org.

[6]

XML Schema: Reconciling Diversity with Standardisation, Eddie Curtis, Snowflake Software, http://www.snowflakesoft.co.uk/news/papers/xmlSchema.pdf.

[7]

http://www.safe.com/solutions/whitepapers/pdfs/GML%20-%20Experiences%20From%20the%20Field1.pdf, Don Murray and Juan Chu Chow, Safe Software http://www.safe.com.

[8]

GeoTools project, http://geotools.org.

[9]

Unified Mapping Platform (JUMP), Vivid Solutions, http://www.jump-project.org.

[10]

JTS Topology Suite, Vivid Solutions http://www.vividsolutions.com/jts/JTSHome.htm.

[11]

Cleopatra, Publishing GML data as interactive SVG maps, Schema Software Inc., http://www.svgopen.org/2003/papers/cleopatra/.

[12]

SAXON XSLT and XQuery Processor, Michael H. Kay http://saxon.sourceforge.net/.

[13]

Java API for XML Processing (JAXP), http://java.sun.com/xml/jaxp/.

[14]

The One Map Project, Gunnar Misund and Knut-Erik Johnsen, http://www.ia.hiof.no/~gunnarmi/omd/gmldev_02.

[15]

H.T. Uitermark, P.J. M. van Oosterom, N. J. I. Mars and M. Molenaar. Ontology-Based Geographic Data Set Integration. In *Proceedings of International Workshop on Spatio-Temporal Database Management*, Springer Verlag, 1999. http://www.gdmc.nl/oosterom/STDBM993.PDF.

[16]

Y. Doytscher, S. Filin and E. Ezra. Transformation of Datasets in a Linear-based Map Conflation Framework. *Surveying and Land Information Systems*, Vol 61, No. 3, 2001.

[17]

A. Sheth and J. Larson. Federated database systems for managing distributed, heterogenous, and autonomius databases. In *ACM Computing Surveys*, 22 (3), 1990.

[18]

Distributed GML Management with SVG Tools, Misund, Kristiansen, Lindh ,http://www.svgopen.org/2003/papers/DistributedGmlManagementWithSVG/

[19]

Schema transformation and generic GML2SVG stylesheets!. http://www.onemap.org/harald/bundle.zip.