

# **Working with Generic GML through Schema Analysis**

**Master of Science Thesis**

**Harald Vålerhaugen**

**Østfold University College, Halden Norway**

---

## Working with Generic GML through Schema Analysis

### Summary

Geography Markup Language (GML) is a markup language used to describe geographic objects. These objects can be represented with location, extent and possibly also other geographic data describing their physical relation to the world. Additionally, they can contain non-geographic information that further describes the purpose of the objects. These objects are typically referred to as features. GML has been described as the foundation of the Geo-Web, because it is an open standard developed to overcome data interchange problems between proprietary systems. It is based on Extensible Markup Language (XML), a widely adopted standard for storing data and interchanging data between vendors and applications especially on the Internet. An XML document is a plain text document where data is described using markup elements. The structure and content of an XML document can be constrained using an XML Schema. A schema is in itself an XML document, built after certain criterias. Some libraries able to parse XML can control if an XML document is adhering to the rules of a related schema, if so the document is described as valid. The GML standard is defined with a set of basic schemas, meant to serve as a foundation for extension. These extensions are called application schemas, and they are literally specializations of these basic schemas, made to fit the profile of one vendors data. In order to store geospatial data using GML it is a requirement that they represented with an application schema. In this thesis you will be presented to a method of both analyzing applications schemas and instance documents that can be used to resolve their datatypes. The information will be accessible through a dictionary containing the data types and their ancestors. Using this dictionary we are much more capable of handling GML documents based on different application schemas in a generic way. The utilization of such a dictionary is exemplified with an XSL Transformation (XSLT)[XSLT1] for transforming GML instance documents into a Scalable Vector Graphics (SVG)[SVG] document.

---



---

## Table of Contents

Foreword .....	ix
1. Introduction .....	1
2. XML software and technologies .....	5
XML .....	5
DTD .....	7
XML Schema .....	7
XSL .....	11
GML .....	14
Web Feature Server (WFS) .....	15
Parsing XML .....	15
Simple API for XML (SAX) .....	16
Document Object Model (DOM) .....	17
Scalable Vector Graphics (SVG) .....	19
3. GML software .....	21
JUMP - Unified Mapping Platform .....	21
GeoTools .....	25
GeoTools DataSource (GMLDataSource) .....	26
GeoTools DataStore .....	28
Cleopatra .....	29
GO Loader .....	31
4. Handling arbitrary GML sources .....	33
Handling arbitrary GML .....	33
GML profiling .....	36
Project OneMap .....	37
Implementation discussion .....	41
Converting application schemas and documents .....	41
Constructing a GML mapping dictionary .....	44
XML Schema API .....	48
Parsing schema with XSLT .....	49
GML design issues .....	50
Cascading GML Analysis .....	55
5. Schema parser and GML viewer .....	66
Parsing schemas .....	66
Mapping dictionary schema .....	66

---

Parsing GML application schemas .....	67
Generic GML Visualization .....	70
6. Conclusions and further work .....	76
Type dictionary .....	76
GML Viewer .....	79
Bibliography .....	81
A. XSL Transformations .....	86
GML Schema to Mapping Dictionary .....	86
Stylesheet for removing identical type maps from mapping dictionary ..	104
Generic GML/Dictionary to SVG transformation .....	105
Stylesheet included into the genericGML2SVG.xslt listed .....	123
B. XML schemas .....	125
Mapping Dictionary Schema .....	125
C. Schema and instance document example .....	127
dens.xsd .....	127
instance.xml .....	128
D. GML schema and instance example .....	131
hbn.xsd .....	131
halden1.xml .....	132

---

## List of Figures

1.1. GML schema design .....	2
2.1. Element visualization .....	6
2.2. Data type visualization .....	10
2.3. DOM Level 2 Architecture (DOM Activity Statement) .....	17
2.4. Simple GML to SVG transformation .....	20
3.1. Technical architecture of JUMP (JUMP Technical Report) .....	21
3.2. JUMP screenshot (JUMP Technical Report) .....	22
3.3. GMLDataSource SAX filters .....	27
3.4. Extending and substituting featureCollection .....	28
3.5. Cleopatra demonstration screenshot .....	30
4.1. Basic GML application schema .....	34
4.2. HaldenByNight application schema .....	35
4.3. OneMap: Gateway screenshot .....	37
4.4. OneMap: Repository .....	38
4.5. Integrated schema hierarchy .....	40
4.6. River fragments constituting complete river .....	41
4.7. Application using converted GML-documents .....	42
4.8. Application utilizing a schema dictionary .....	45
4.9. Retrieve generic GML from repository .....	46
4.10. Ordnance Survey MasterMap schema structure (OSMasterMap User Guide) .....	47
4.11. Interleaved instances and properties .....	51
4.12. Definition of PolygonPropertyType .....	52
4.13. Retrieving additional information about a feature .....	53
4.14. Definition of LinearRingPropertyType .....	54
4.15. Defining a GML vocabulary .....	56
4.16. Halden-by-night vocabulary mapping .....	59
4.17. ContentHandler methods .....	63
4.18. Resolver chains .....	64
5.1. How to traverse schemas .....	67
5.2. Type-mapping of the NightSiteBar-element .....	69
5.3. Utilizing dictionary to parse arbitrary GML .....	70
5.4. Integrated GML transformed to SVG .....	72
5.5. SVG integrated layer visibility .....	73

---

5.6. Feature information window .....	73
5.7. Ordnance survey data with default styling .....	74
6.1. Schema hierarchy search problem .....	77

---

## List of Examples

2.1. Well-formed XML .....	6
2.2. Schema fragment: gambling_machine .....	8
2.3. Schema fragment: slot_machine .....	9
2.4. XSL Transformation example .....	12
2.5. Feature type example .....	14
2.6. Xerces ContentHandler method signatures. ....	17
2.7. parse a document into DOM-structure .....	19
3.1. Template for River-feature .....	23
3.2. Template for Road-feature .....	24
3.3. GMLDataSource recognition of geometry elements. ....	26
3.4. Recognition of features in GMLDataStore .....	27
3.5. Configuring Cleopatra .....	29
4.1. Object model: functional notation .....	54
4.2. Type maps from example data .....	57
4.3. Schema definitions of mapped types .....	58
4.4. Typemap for OS MasterMap type BoundaryLine .....	61
5.1. Simple feature styling .....	72



---

# Foreword

Two persons should have their name printed in gold in this document, but sadly cartridges are expensive enough as they are, so black ink must do. Gunnar Misund presented me to their OneMap-project, and set me off working with GML and GIS the autumn 2003, not exactly my area of expertise up until then. Thanks for patience, interesting challenges and a genuine interest for your students. Of course my live-in girlfriend Kirsti is not forgotten, even though she has been placed second to my work all too many times the last months. Now, finally having this thesis completed I will never again use it as an excuse to avoid social interaction with either her or friends and family.

---

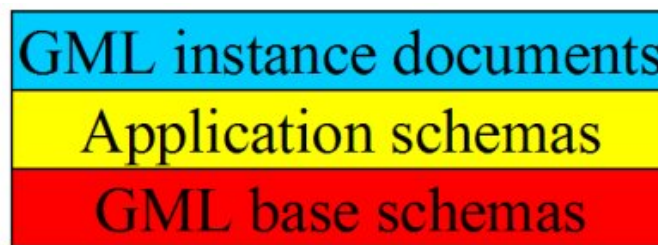
# Chapter 1. Introduction

For a long time, vendors of *Geographical Information Systems* (GIS) did not have common interfaces for interchanging, viewing, editing or querying their geospatial data. Through the *Open Geospatial Consortium* (OGC)[OGC] government agencies, universities and companies participates in a consensus process to develop publicly available interface specifications for just this purpose. Some specifications are already widely adopted, such as *Web Map Service* (WMS)[WMS], while others, such as the *Web Feature Service* (WFS)[WFS] are catching up. A WMS delivers maps based on requests to a web server. Most commonly these maps are delivered as ordinary raster images like jpeg or the transparent png, and consists of layers with homogeneous features. The user decides which layers to retrieve and the sequence of them in addition to the geographical extent and upon this request an image with the named layers in the desired format is delivered. A WFS however, does not serve data as maps, but as XML[XML] representations of features. More specifically the response from a valid WFS query is delivered in a format called *Geography Markup Language*, a specification that has reached version 3.0 as we speak. The specification of GML has advanced from version 1.0 based on Document Type Definitions (DTD)[DTD], to version 2 and 3, heavily relying on *XML Schemas*. XML Schemas encourage an object-oriented and modular design of XML document definition, including important principles like abstract types and derivation. GML defines both abstract and non-abstract types and elements, which forms the foundation for the development of *application schemas*. An application schema form a dialect of GML that is specialized for certain data, like for example data from a company's database. If you want to model some kind of geospatial data model in GML, it is necessary to define one or several schemas capturing the properties and features from the system. It is not sufficient to use data types and elements from the base GML schemas, because many of them are abstract and thus not instantiable.

The endless possibilities when designing GML application schemas, the methods of specifying chains of derived type declarations, together with the use of substitution groups, provides the opportunity to represent a broad range of geospatial data sets. However, as the data gets more complex it gets more difficult to keep track of their origin. Even though the rules of the base GML schemas define the structure of application schemas, their datatypes may change both by name and content, thus making it cumbersome to handle data in a homogenous way.

The different application schemas are created from the three base GML schemas in version 2, while the version 3.0 specification is more than eight times larger. My work is focused on version 2, but the theories are logically transferable to working with GML 3. An application schema represents one dialect of GML, with individual features, properties and geometries. If you only consider the top layer, being the different GML schema dialects, the data are clearly heterogeneous Figure 1.1, “GML schema design”.

**Figure 1.1. GML schema design**



With the non-profit, open source project, *OneMap* [P1M], the main goal is to "provide online public access to a comprehensive and detailed world map". This will be done incrementally and uncoordinated by many submissions. It is an underestimation to call this challenging considering the vast number of formats, covering different parts of the globe with different level of detail. Even though there are enormous amounts of geodata available from various sources, the data must be collected and analyzed, requiring a lot off both human and computational resources.

GML is adopted by a broad range of companies, both profit and non-profit. With this joint effort to develop a common format for geospatial data, interoperability between systems and exchange of data is far less complex than before. When storing geographic data on XML format as GML we can utilize a vast number of software and methods for query, parsing and structural design of schemas. The flexibility of schema design, and the fact that the base GML schemas are meta-language for describing application vocabularies, means that application schemas in most terms can be considered as heterogeneous. As a result most systems working with GML are often designed for one dialect or profile (see the section called “GML profiling”) only. This issue is the foundation of this master thesis, as there are none open-source libraries or methods to handle GML in a generic way. Based upon existing libraries; parsing, analysis and extracting of schema information is possible. By developing a code base to make differ-

---

ent dialects of GML accessible to utilizing applications, data exchange on GML-format will be more encouraged. This information may be provided as a dictionary, where origin of data types can be traced, making it possible for applications to utilize easy accessible meta-information for different GML vocabularies. When different features constructed from arbitrary application schemas can be threatened generically, they can also be mixed into integrating vocabularies, meaning documents that do not define instantiable features in their own namespace, but use feature definitions from other vocabularies.

In Chapter 2, *XML software and technologies* some important standards for working with XML are introduced. Among these are XML generally and the document definition languages DTD and XML Schema. It is important to get a quite profound understanding of XML in order to fully be able make use of some of the other standards presented in this chapter. For altering, parsing and transforming XML there are a number of specifications and implementations. Those presented here is *Document Object Model* (DOM)[DOM], *Simple API for XML* (SAX)[SAX] and *Extensible Stylesheet Language* (XSLT)[XSLT1]. The XML parsers can basically do the same tasks, but the fundamental differences in how a document is parsed makes their working areas somewhat different. Performance does often come in expense of functionality; this is an important point to remember when picking one before the other. It is expected that the reader has a basic understanding of programming, but they do not need to be expert. The introductory chapter can be skipped if you feel comfortable with the XML and the concept of XML parsing.

Chapter 3, *GML software* gives a brief introduction to some of the available software working with arbitrary GML, and some that would greatly benefit from being able to analyze schemas and automate the process of loading GML data sources. There are several methods for working with arbitrary GML, using a manually made mapping file might be the most usual one, one that works excellent when there is only one or a few dialects to be interpreted and imported into a program. Naturally, this could hardly be called support of generic GML data sources, and the amount of work to make such mapping files by hand for tens or hundreds of application schemas, requires some effort.

In Chapter 4, *Handling arbitrary GML sources* you will be given a more thorough introduction to the issue of GML schema analysis. Utilizing arbitrary GML is presented more in detail illustrated with a small example data set. You will also be presented to a cascading method of GML analyzis, a method which is more reliable when working with GML and schemas over the internet in particular, where resources might not al-

---

ways be obtainable.

Two XSL transformations are presented Chapter 5, *Schema parser and GML viewer* . One is used to transform GML schemas into a mapping dictionary and one is for converting GML with a given mapping dictionary into SVG. A description of how I chose to implement them is found in the same chapter, together with some example results.

The last section, presented in Chapter 6, *Conclusions and further work* sums up the work that has been done and some of the problems that arose during implementation. I will try to more thoroughly go through the choices I made regarding implementation strategies.

---

# Chapter 2. XML software and technologies

GML is as mentioned an XML standard, based on another XML standard, namely XML Schema. This chapter gives a brief introduction to XML and some of the libraries and methods developed to work with XML. First and foremost the characteristics of XML in general is described, before moving on to how structure and content of documents can be restricted using DTDs and schemas. These topics can be considered as the basics of XML and are important when it comes to understanding the GML vocabulary, which also is presented in this chapter. The last part treat the art of parsing XML documents, either for conversion to another XML vocabulary or to extract information from them.

## XML

XML is designed to give a flexible, but fairly simple way to store and describe meta data. XML is an abbreviation of the highly complex SGML, the language describing HTML, but also a large range of other more complex languages. By defining this less powerful, but more accessible meta-language interface, it met the requirement for a standard data exchange language on the Internet and between applications.

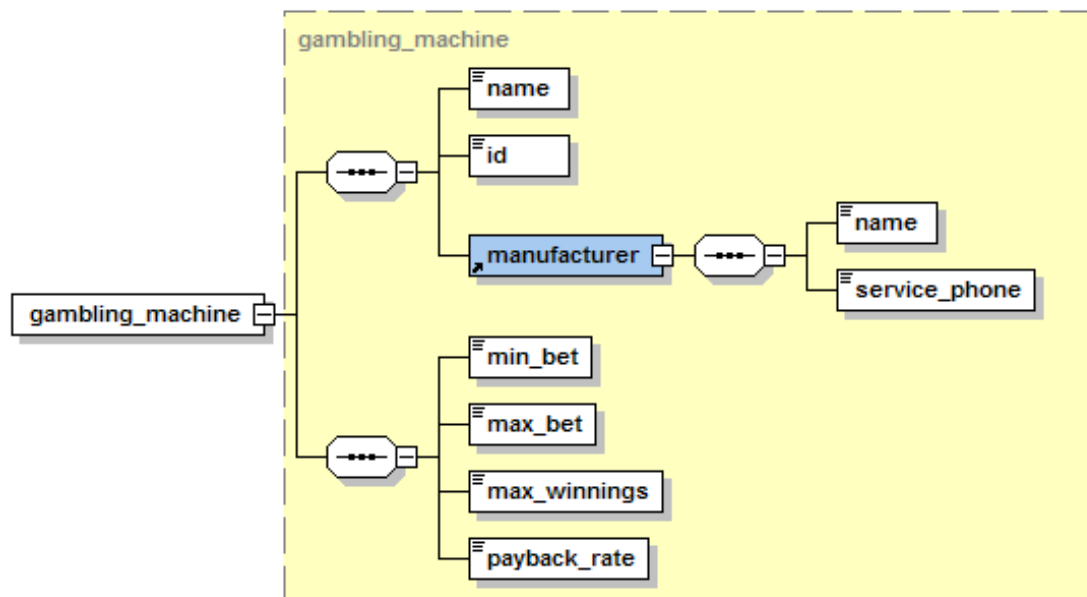
XML is made for describing data, not displaying it like HTML is. HTML has a limited set of elements, all known by web browsers that are able to present the data on basis of these elements. XML however does not have a limited set of elements. They must be *well-formed*, meaning that all tags must be closed or terminated by an end-tag. A document with the tag <description> requires a closing tag </description> well-formed. Alternatively the tag could be an empty tag <description/>. In addition the tags also have to be nested correctly, not allowing closing of other tags than the current tag. Therefore a document can be described as a *tree-structure*, which leaves us the advantage of a relatively clear set of rules regarding the structure of documents and the methods of analysis and traversal. Elements in an XML document should and often are named to describe the content of the document, but the fact that there is no standard set of elements in XML means that it is impossible to make generic XML editors that 'understands' the meaning documents.

The following snippet of an example we shall examine in more detail later is considered well-formed XML. Notice the closing of each element, and the correct nesting. The element visualization (Figure 2.1, “Element visualization”) is a screenshot of a functionality in XMLSPY[SPY], a powerful tool for developers of XML and related technologies. This clearly shows how the nested elements form a hierarchical (tree-like) structure; the *XML fragment* is an instance of this data type.

### Example 2.1. Well-formed XML

```
<...  
<gambling_machine>  
  <name>Pokermania</name>  
  <id>A900-01</id>  
  <manufacturer>  
    <name>Mercury Inc</name>  
    <service_phone>666-234-567</service_phone>  
  </manufacturer>  
  <min_bet>10</min_bet>  
  <max_bet>50</max_bet>  
  <max_winnings>1000</max_winnings>  
  <payback_rate>85</payback_rate>  
</gambling_machine>  
...
```

**Figure 2.1. Element visualization**



---

## DTD

When exchanging data on XML format it is important to be able to describe the content and structure of a document, so that applications can interpret or create documents made for a certain system. A standardized way to define an XML vocabulary is to use a DTD. An instance document can then define what DTD is describing the document, and XML parsers can validate a document against the DTD and report possible divergences. Documents that are in accordance with their DTDs are described as valid. This is an extremely important issue when it comes to exchange of data between systems. A DTD specifies the allowed elements, their allowed content, both type and cardinality.

Often, defining document structure using DTDs are sufficient, but it lacks some fundamental methods for expressing constraints for element and attribute data. Constraining element cardinality is cumbersome to define with a DTD when you for instance want to limit the number of elements to be between e.g. 10 and 20.

A DTD defining the structure of the element visualization (Figure 2.1, “Element visualization”) shown above, could look something like the following:

```
...
<!ELEMENT name (#PCDATA)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT min_bet (#PCDATA)>
<!ELEMENT max_bet (#PCDATA)>
<!ELEMENT max_winnings (#PCDATA)>
<!ELEMENT payback_rate (#PCDATA)>
<!ELEMENT manufacturer (name, service_phone)>
<!ELEMENT gambling_machine ((name, id, manufacturer), (min_bet, max_bet,
max_winnings, payback_rate))>
...
```

## XML Schema

Due to the limitations of DTDs and the fact that some developers desired a less complex way to define the structure of their documents, the work with developing a new standard to define an XML document's structure and legal building blocks started, the result was the XML Schema. The Schema turned out more complex than the DTD, but many of the problems addressed with the DTD was elegantly solved. A schema is in itself a XML document, describing the allowed contents of another XML document, with elements from the *<http://www.w3.org/2001/XMLSchema>* namespace[XMLNS]. On the other hand, the syntax of a DTD is not XML itself, meaning that tools for edit-

---



ing DTDs and validating documents against them, must implement support for one additional syntax. Naturally the same problem arise for developers of DTDs and documents, needed to master both syntaxes.

The *gambling\_machine* complexType in Example 2.1, “Well-formed XML”, is originally defined in a schema along with other elements. Full example listed Appendix C, *Schema and instance document example* . The complete schema *dens.xsd* describes the elements of a document to keep track of gambling dens and slotmachines belonging to them. Interesting data for inspectors of those kinds of businesses. The data type is described in the code block underneath. First, the root of the data type is a *complexType* from the Schema-namespace. This is an example of how we can declare our own complex datatypes; complex meaning that the type consists of other elements nested within, so called *simpleType*.

## Example 2.2. Schema fragment: gambling\_machine

```
<xs:complexType name="gambling_machine">
  <xs:annotation>
    <xs:documentation>Datatype for gambling slot machine,
      ergo machines that pay out prize money in certain
      situations.
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="slot_machine">
      <xs:sequence>
        <xs:element name="min_bet" type="xs:positiveInteger"/>
        <xs:element name="max_bet" type="xs:positiveInteger"/>
        <xs:element name="max_winnings" type="xs:positiveInteger"/>
        <xs:element name="payback_rate">
          <xs:simpleType>
            <xs:restriction base="xs:unsignedShort">
              <xs:maxInclusive value="100"/>
              <xs:minExclusive value="0"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
...
```

If you compare this Schema-snippet to the XML fragment in the section called “XML” [6] , you recognize the structure of elements. However, the schemas give a closer description of what kind of data you are actually dealing with. For instance the *<min\_bet>*-element is of type *positiveInteger*. Take a look at the *<payback\_rate>*, this element is not like the others. It is declared in an element-tag, but there is a *<simpleType>* element nested within. This is actually a method of restricting the al-

lowed values of data type. By a restriction, with an *unsignedShort* as base, we can specify the maximum and minimum allowed value of the unsignedShort-type, resulting in a type that no longer can hold values below zero and above one hundred. If we try to make an instance document violating this rule the validation will fail. Note that the `<payback_rate>` is declared inline and thus not making it reusable in other parts of the Schema. We could have declared an element or data type at the root of the document, then referenced the element or created a new element from the data type inside our `<gambling_machine>`, like this:

```
<xs:element ref="payback_rate"/>
```

if `<payback_rate>` is an element, or like this:

```
<xs:element name="payback_rate" type="payback_rate"/>
```

Even though some elements are missing compared to the XML fragment, the data is still declared in the `<gambling_machine>`, but it might not be easy to spot for an untrained eye at first sight. This data type has a super-type, `<slot_machine>`, where the rest of the content is declared (Example 2.3, “Schema fragment: slot\_machine”). This super-type is made because a gambling machine is a type of slot machine, but so is for instance an arcade game. The concepts of reusability and inheritance are introduced into XML using Schemas. The `<payback_rate>` element was a restriction of the *unsignedShort*, while the `<gambling_machine>` is a *extension* of the `<slot_machine>`, specifying new content and reusing existing.

### Example 2.3. Schema fragment: slot\_machine

```
<xs:complexType name="slot_machine" abstract="true">
  <xs:annotation>
    <xs:documentation>Abstract data type defined to be super-type for
      any type of slot machine in the system.
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="id">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:pattern value="[A,B,C][0-9]{3}[-][0-9]{5}"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

---

```

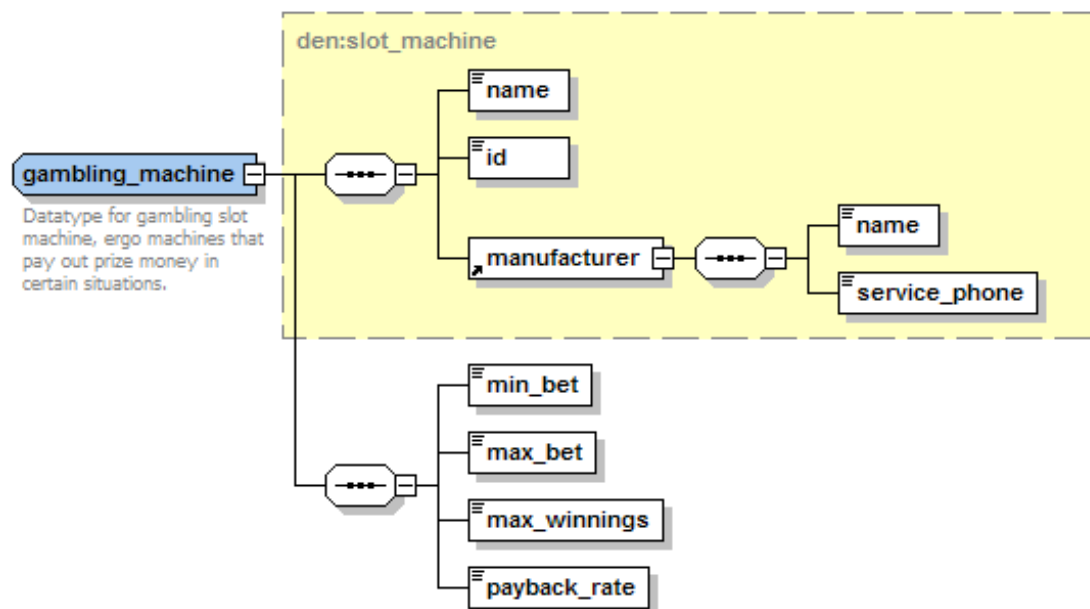
        <xs:element ref="manufacturer"/>
    </xs:sequence>
</xs:complexType>
...

```

The *complexType* declaration above defines an element *id*, showing the use of a *pattern* to restrict string-values. The pattern is a regular-expression that has to match to the element in the instance document in order for it to be valid. The pattern in the above *complexType* is translated into one of the letters A, B or C, followed by three digits from 0-9, a dash, then five more digits. Let us say that this is a registration code that all legal slot machines have to be labeled with within an area. The regular-expression matching ensure that the code is correct according to the rules of registration.

If we take a look at the data type visualization (Figure 2.2, “Data type visualization”) of `<gambling_machine>` in XMLSPY, we can see a more accurate description because the characteristics inherited from `slot_machine` is shown on the yellow background and the rest of the elements on white.

**Figure 2.2. Data type visualization**



There are several other benefits of using schemas, for instance schemas both have a greater number of data types available than DTDs and they provide the opportunity to

---

make your own. The Schema standard allows programmers to take an object-oriented approach to the developing of documents. By deriving other data types, either by restriction or extension, the aspect of *reusability* and tighter control over the allowed element and attribute values, is evident. You can define fundamental properties in abstract data types, and by deriving these and declare *substitution groups*, one element or data type may substitute for another. This method of development is flexible and powerful, but still you can keep tight control over what kind of data is allowed in your instance documents. The example schema describes a data type, *arcade\_game*, with the super-type *slot\_machine*. Even though both of these types in an object-oriented approach could substitute for the *slot\_machine*, we have to specify it in the Schema using the *substitutionGroup* -attribute. Now both *gambling\_machine* and *arcade\_game* can substitute for elements of type *slot\_machine*. Actually, if elements of *slot\_machine* are required, one of these *must* substitute because the *slot\_machine* type is declared abstract, and cannot be instantiated.

```
[...]
<xs:element name="gambling_machine" type="gambling_machine"
substitutionGroup="slot_machine"/>
<xs:element name="arcade_game" type="arcade_game"
substitutionGroup="slot_machine"/>
[...]
```

The complete schema and an examples instance document are located in Appendix C, *Schema and instance document example* .

## XSL

XML will not necessarily ever replace HTML since they basically cover two different purposes, namely markup for describing data and markup for displaying data. There are however technologies under development for displaying XML. As earlier addressed, the HTML-elements are all known to browsers made especially for the purpose to layout the content according to the tagging of a file. Since you define your own elements in XML, browser cannot guess how you want your elements styled and displayed. For this purpose we can use *Extensible Stylesheet Language(XSL)[XSL]*, another W3C specification. XSL is actually a family of three different W3C recommendations, a transformation language called XSL Transformation (XSLT), a language to address and manipulate parts of documents called XML Path Language (XPath)[XP] and a styling language called *XSL Formatting Objects (XSL-FO)[XSL]*.

The purpose of XSLT is to transform a XML document into another document such as

e.g. *Scalable Vector Graphics* (SVG)[SVG], HTML or any other desired format. Parts of the original file are matched against templates in the transformation file, reorganizing the data and placing it on the desired location in the output document. XPath is used to address data from the tree-structured XML document. When people refer to XSL they are often actually talking about XSLT. This is somewhat incorrect considering that XSLT is only one component of the XSL recommendation. The transformation part can however be used independently of the formatting objects and vice versa.

Both structure and content of an XML document can be drastically changed using XSLT. Example 2.4, “XSL Transformation example” presents a short example of how to convert an XML instance document with *gambling\_dens* to a HTML document.

### Example 2.4. XSL Transformation example

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:sl="http://www.dens.com">
  <xsl:output method="html" encoding="UTF-8" indent="yes"/>
  <xsl:template match="/">
    <html>
      <head>
        <title>Registrered slot machines</title>
      </head>
      <body>
        <table border="1">
          <tr>
            <th>Name</th>
            <th>Registration number</th>
            <th>Owner</th>
          </tr>
          <xsl:for-each select="//sl:arcade_game | //sl:gambling_machine">
            <xsl:call-template name="slot_machine"/>
          </xsl:for-each>
        </table>
        <xsl:call-template name="summary"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template name="slot_machine">
    <tr>
      <td>
        <xsl:value-of select="sl:name"/>
      </td>
      <td>
        <xsl:value-of select="sl:id"/>
      </td>
      <td>
        <xsl:value-of select="../../../sl:name"/>
      </td>
    </tr>
  </xsl:template>

  <xsl:template name="summary">
    <xsl:variable name="nrArcade" select="count(//sl:arcade_game)"/>
    <xsl:variable name="nrGambling"
      select="count(//sl:gambling_machine)"/>
    <table>
      <tr>
        <td>Number of arcade games:</td>
        <td><xsl:value-of select="$nrArcade"/></td>
      </tr>
    </table>
  </xsl:template>
</xsl:stylesheet>
```

---

```
<tr>
  <td>Number of gambling machines:</td>
  <td><xsl:value-of select="$nrGambling"/></td>
</tr>
<tr>
  <td>Average number of slot machines per den:</td>
  <td><xsl:value-of select="($nrArcade + $nrGambling) div
count(//sl:gambling_den)"/></td>
</tr> </table>
</xsl:template>
</xsl:stylesheet>
```

As revealed by the document declaration of the example stylesheet, the file is XML itself. The transformation elements used are those represented by the namespace *http://www.w3.org/1999/XSL/Transform*, identified by the abbreviation *xsl* in the document. The elements without any namespace prefix, goes directly to output, these are the HTML-elements that form the layout of the page. A transformation starts with the matching of templates against the document to be converted. To get the conversion 'going', we must use the template to match the XPath '/' identifying the root of the document. When the root is located, the XSLT-engine transforms the data in the instance document according to the order given in the template. A template is either identified with a *match* like the root-element, or with a *name*. If the template has a name, we can call it from other parts of the document. The `<xsl:for-each ...>`-element will traverse the node-sets returned from the XPath-expression found within the select-attribute. As you can see, we combine the xsl-elements with XPath-expressions to get the most out of the transformations. It is important to remember that the `<xsl:for-each ...>`-element is not a ordinary loop like the for-loop found in C/C++/Java. The for-each is used to traverse node-sets and for each iteration the parsers logical position will be inside the current node. If you want some kind of ordinary for-looping, there are no implementations of this in XSLT. You may however imitate a for-loop using recursive templates. In our example you will see that we use the `xsl:call-template-command` for each node. The template outputs a table row, with the name and id of the machine in addition to the name of the place where you can find it. This template can just as well be defined inline in the for-each-element if we do not want to make it reusable to other templates.

The last table row of the stylesheet output sums up the different types of machines. It also divides the number of machines on the total number of dens. Math functionality is crucial if XSLT should be a real competitor to implementations in Java, C++ or any other high level programming language. As you can see, we can store the values in variables like we do in other conventional languages. There is however one important note on variables in XSLT, their value cannot be changed once they are initiated.

---

XSLT is only a specification how transformation stylesheets can be written to convert a document from one XML format to another, not how it should be done. There are quite a few available parsers, implemented to interpret transformation stylesheets and carry out transformations on documents. Most of them only supports the XSLT 1.0 specification, but recently the Saxon project released a parser with some functionality defined in the XSLT 2.0 specification[XSLT20].

## GML

"Geography Markup Language is an XML grammar written in XML Schema for the modeling, transport, and storage of geographic information" [GML30]

The OGC abstract model of geography[AMG] describes the world in terms of geographic entities called features. A feature is a combination of spatial and non-spatial data, properties and geometries. A GML document contains so called *feature collections*, that works as containers for *features*. A feature collection is actually a feature itself, meaning that feature collections can hold other feature collections. There are no limitations on the depth of feature nesting in a GML document. A document may e.g. store information about train stations within an area, it will therefore be composed of many features (stations), each describing the non-spatial properties of stations like the name and other related data, alongside with their geometric properties such as location and boundaries. Example 2.5, "Feature type example" shows a feature, *school*, with properties and geometries.

### Example 2.5. Feature type example

```
...
<Feature fid="142" featureType="school" >
  <Description>Balmoral Middle School</Description>>
  <Property Name="NumFloors" type="Integer" value="3"/>
  <Property Name="NumStudents" type="Integer" value="987"/>
  <Polygon name="extent" srsName="epsg:27354"> <LineString
    name="extent"
    srsName="epsg:27354">
      <CDATA>
        491888.999999459,5458045.99963358
        491904.999999458,5458044.99963358 491908.999999462,5458064.99963358
        491924.999999461,5458064.99963358 491925.999999462,5458079.99963359
        491977.999999466,5458120.9996336 491953.999999466,5458017.99963357
      </CDATA>
    </LineString>
  </Polygon>
</Feature>
...
```

Considering that GML is XML, we can benefit from a broad range of applications and standards to develop, transform and parse the data. From day one there were tools available for these tasks, which must be considered a great argument for choosing GML, not only for exchange of geographic data, but in some cases also storage. GML upholds the principle of separating content from presentation, meaning that presentation of the data is not up to GML.

## Web Feature Server (WFS)

The Web Feature Service Implementation Specification (WFS)[WFS] describes an interface for retrieval of geospatial data encoded in GML. How these data are stored should be opaque to the client utilizing the service. Data sharing between proprietary systems will thus be possible if they are accessible through a WFS. WFS is specified for the HTTP protocol, accepting parameters as key-value-pairs in a GET query.

Through a valid *DescribeFeatureType* request, the client will be served a GML schema describing features available from the WFS interface. The schema returned is designed for the underlying data in particular, meaning that there are no standardized feature schemas meant to fit all features stored behind a WFS interface. Since GML is meant to describe, not present geospatial data, it is up to the client parse and style features in any chosen format. GML served from a WFS is not any different from any ordinary GML document, but a fully functional implementation offers functionality to filter out features based on geographic or non-spatial property values. When wanting to present GML acquired from a WFS, an application must either be able to directly make a graphical presentation of GML, or the GML must be transformed into some kind of presentation format like e.g. SVG.

## Parsing XML

This section introduces some of the methods and libraries available for parsing and working with XML. When working out a method to work with GML in a generic way, one or several parsing libraries like these could be the key elements of success.

We have already seen an example of how to transform XML-structures using XSLT, a powerful XML-based, functional programming language. There are situations when we need APIs to access information in XML documents, for editing or merely reading



---

purposes. You will find two basic approaches to this, *Document Object Model* (DOM) [DOM] and *Simple API for XML* (SAX) [SAX]. They are both powerful and widely adopted standards, but they have fundamentally different approaches on how to parse XML. All programmers working with XML should be acquainted with the differences between them and the situations when you should choose one before the other.

Both APIs are platform- and language-neutral programming interfaces, with dozens of implementations for several different programming languages. All examples provided within this document are written using Xerces2 Java Parser, but should be easily adapted to other programming languages. Programmatically, DOM is probably the most high-level method of the two, while SAX represents the effective one, addressing the fields where DOM lacks in performance. The following sections give a short introduction to the fundamental characteristics and differences between the available XML parsing libraries.

## Simple API for XML (SAX)

SAX was at first a widely adopted API for XML in Java, but is now available for many different programming languages, making it an excellent additional API for parsing XML. W3C are not in charge of the development of SAX, it is under open development as a [SourceForge-project](#)[SRCF].

SAX is an *event-based* API where XML documents are parsed sequentially and events are triggered dependent of document's structure and content. When using DOM, the document has to be parsed into a data structure before the content is reachable by code. SAX parsers parses a document sequentially, meaning that the actual extraction of document information starts instantly from the first byte is read from the a file.

## Programming SAX with Xerces

When we are parsing documents using SAX, the *XMLReader*-object is responsible for the actual parsing. This reader triggers different events to the *ContentHandler*, depending on what kind of data is read at the moment. The code in Example 2.6, “Xerces ContentHandler method signatures.”, shows some of the methods in the interface *ContentHandler*. These methods should be quite self-explanatory, and if we implemented the handler to output the argument values, this would reflect the order of the elements in the document. Remember that you are not able to go 'backwards' in the event-stream when using SAX, if you have interest in the prior elements, the only way to gain access

---

to preceding parts of a document is to store it as a parser reaches it.

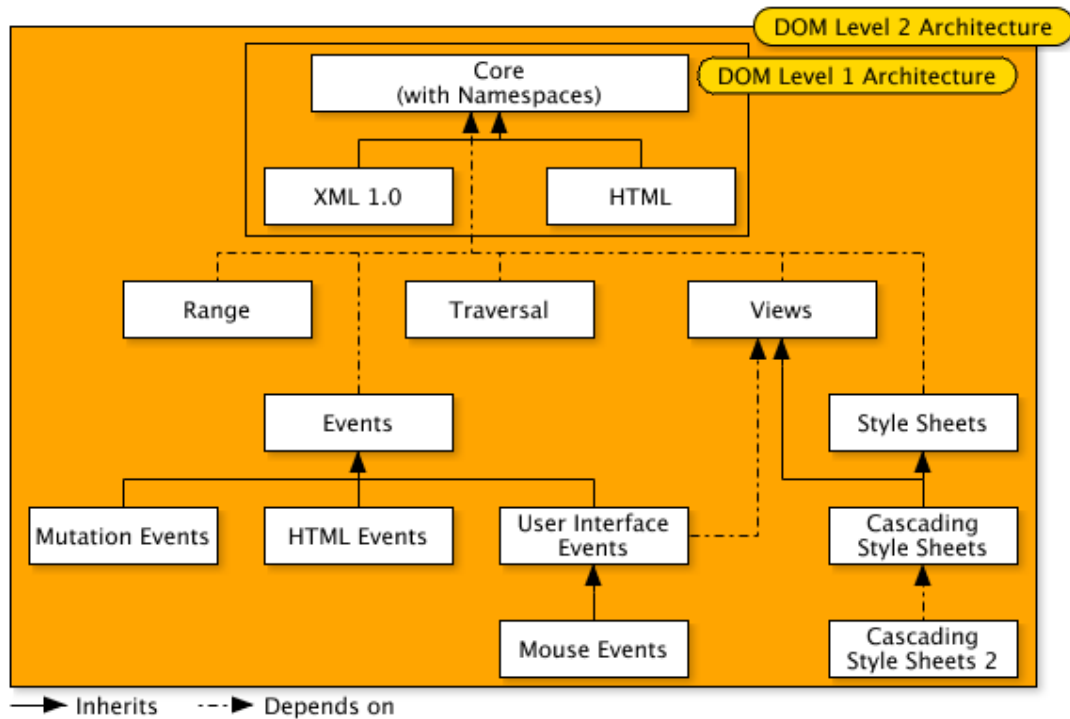
### **Example 2.6. Xerces ContentHandler method signatures.**

```
[...]  
public void characters(char[] ch, int start, int length) throws org.xml.sax.SAXExc  
public void endDocument() throws org.xml.sax.SAXException;  
public void endElement(String namespaceURI, String localName, String qName)  
throws org.xml.sax.SAXException;  
public void startDocument() throws org.xml.sax.SAXException;  
public void startElement(String namespaceURI, String localName, String qName,  
org.xml.sax.Attributes atts) throws org.xml.sax.SAXException;  
[...]
```

## **Document Object Model (DOM)**

The main characteristic of DOM is that it keeps the document in-memory, stored as a tree-structure, making it possible to *access, add, delete, manipulate content in a non-sequential way*. DOM is actually divided into levels, each level providing additional functionality upon the other. So far level 1 and 2 are Recommendations to W3C from the DOM Working Group, which are now working on the level 3 specification. The somewhat altered illustration underneath (Figure 2.3, “DOM Level 2 Architecture (DOM Activity Statement)”), taken from the DOM Activity Statement, show an overview of the functionality offered by DOM level 1 and 2 APIs.

### **Figure 2.3. DOM Level 2 Architecture (DOM Activity Statement)**



Methods provided to navigate the tree and gaining random access to nodes or node-sets, makes this approach an easy pick for many developers. However the advantages of DOM are in some cases considered the disadvantages. Keeping large document in-memory exhausts resources, thus it is important to have a good reason for using DOM. If you just want to traverse a document, possibly to gather data to create object-instances, the overhead of using DOM makes it a bad choice. However if you access the data randomly and often, the time spent to read and make a data structure of the document might be worth the cost of storing it in memory.

## Programming DOM with Xerces

There are dozens of XML parsers supporting DOM and SAX. I have chosen Xerces Java Parser in my work, one of many Java implementations of the interface specification. Some parser do also offer additional functionality that might cover functionality not defined in the interfaces, this might be an important point to remember, because utilizing such functionality will make your software dependent upon one certain type of parser library.

The first step creating a DOM-parser is to instantiate the *org.apache.xerces.parsers.DOMParser*, then pass a String with the document path to

---

the parse-function. The DOMParser will then build an in-memory tree consistent of the information from the file. The content is now accessible from the parser by calling the method `getDocument()` (Example 2.7, “parse a document into DOM-structure”).

### Example 2.7. parse a document into DOM-structure

```
DOMParser parser = new org.apache.xerces.parsers.DOMParser();
try {
    parser.parse("instance.xml");
    org.w3c.dom.Document document = parser.getDocument();
} catch (java.io.IOException e) {
    e.printStackTrace();
    System.exit(1);
}
...
```

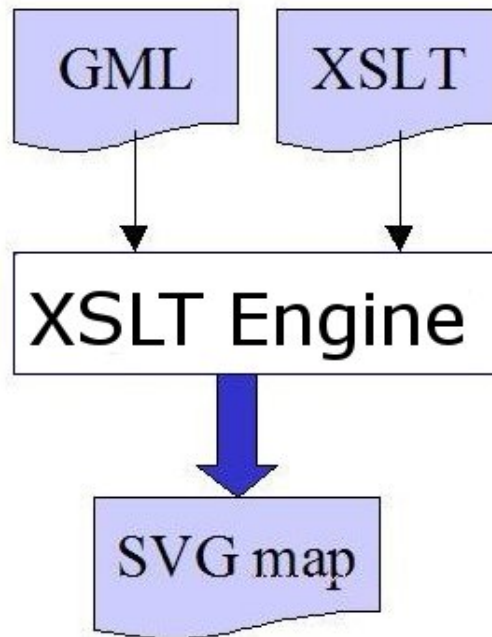
The document-instance represents the whole document, and offers methods for both retrieving and adding data. The document and the sub-nodes all implement the interface *Node*, giving a standard set of methods for traversing the document.[NODE]

## Scalable Vector Graphics (SVG)

GML describes properties and geometries of features, how data should be presented is however not described in a GML document. SVG can be used to present GML content as a map. SVG is another specification based on XML, meant to provide markup for vector graphics. To be able to view SVG documents graphically it is required to have software made specifically for this purpose; Adobe[ADO] has developed the most adopted piece of software for displaying SVG, the *Adobe SVG Viewer*. This provides plugin functionality for web-browsers so that SVG content can be displayed directly from the web. Batik[BAT] is another implementation of the SVG specification, made in Java and available as an open source library made for presentation and altering of SVG. There are a number of libraries and specifications that can be used to convert GML into SVG. One strong candidate is XSLT, stylesheets that can be fed into any XSL-parser together with a GML document to instantly provide the desired output. The simplicity of XSLT stylesheets, makes them an easy choice for XML altering. Most programming languages are equipped with the possibility to run XSL transformations on XML documents, thus making XSL one of the most portable choices (Figure 2.4, “Simple GML to SVG transformation”).

---

**Figure 2.4. Simple GML to SVG transformation**



GML does of course contain geographic data, but in most cases the meta-information contained in each feature is just as important for utilizing software. Meta-data can also be contained in SVG together with the geographical markup. By implementing scripting such as ECMAScript[ECMA] we can achieve dynamic behavior much like in XHTML[XHTML] or other web-standards. The graphical elements in SVG can trigger scripting code for actions such as *mouseover*, *mouseout*, *onload*, *onclick* and so on. Events can also be triggered as a result of the lifecycle of the document in the viewer, these include e.g. *onunload*, *onerror*, *onscroll* and *onzoom*. In addition to this functionality SVG animation is specified through the *Synchronized Multimedia Integration Language 1.0* (SMIL) specification[SMIL].

---

## Chapter 3. GML software

A lot of tools are available to work with GML in some way. These tools have loaders and writers to import GML data into the application. Many of them are *open source* libraries, still under development. The applications covered in this chapter do all load, write or alter GML data, but they handle the issue of different application schemas differently. Some requires additional metadata in their own proprietary format to interpret instances of one particular vendor version. There are also examples of unreliable techniques, like recognizing application elements merely on the basis of their element names. This section attempts to give insight in the solutions implemented by different participants of the GML community.

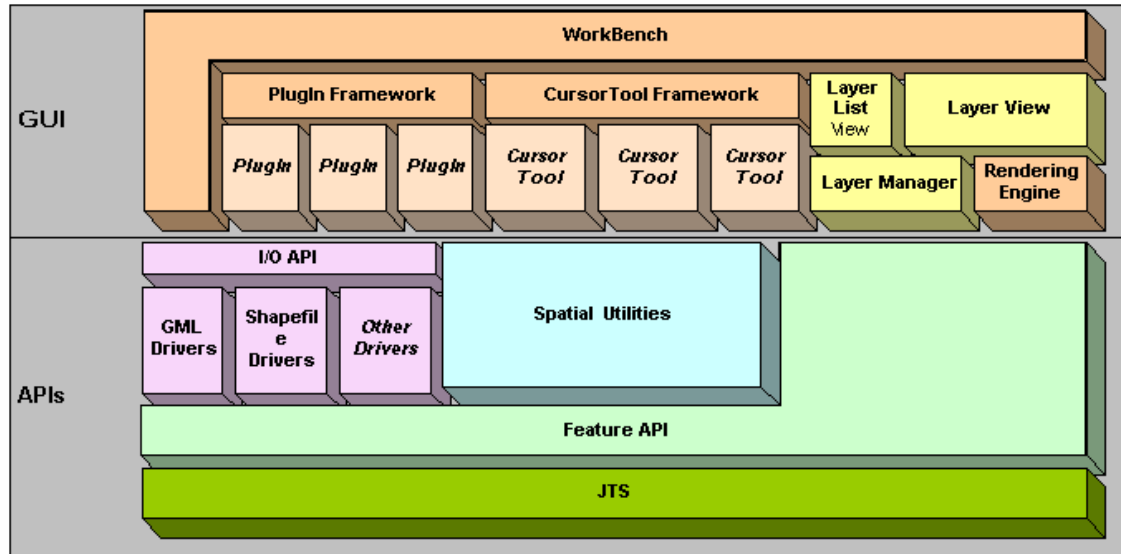
### JUMP - Unified Mapping Platform

JUMP[JUMP] is an open source GUI-based application for viewing, editing and processing spatial data. JUMP utilizes the JTS Topology Suite[JTS], also developed by Vivid Solutions[VIV], to implement the OpenGIS Simple Features Specification[SFS]. The JUMP Workbench is designed for both development of conflation algorithms, invoking of these and as a general-purpose tool for the visualization and edition of spatial data.

To be able to process generic GML, you have to specify a GML Input Template, identifying collections, features, geometry and non-spatial properties. By using an input 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/layer.

JUMP can interpret and write *JUMP GML*, without the need of templates specified by the user. However templates are still used, written and read from the start of the GML-instance. 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.

**Figure 3.1. Technical architecture of JUMP (JUMP Technical Report[JTEC])**



JUMP can also act as a client to OGC *Web Map Service* (WMS)[WMS] servers, providing an interface to create and edit WMS queries Figure 3.2, “JUMP screenshot (JUMP Technical Report)”.

**Figure 3.2. JUMP screenshot (JUMP Technical Report)**





```

    <FeatureElement>River</FeatureElement>
  </GeometryElement>gml:centerLineOf</GeometryElement>
</JCSGMLInputTemplate>

```

If the River-type had any additional non-spatial properties, these could be listed after the GeometryElement. The Road-element, found in the same file has two properties, classification and number. These must also be listed in the template, for the application to be able to read them and for the user to be able to edit and view them in the editor.

### Example 3.2. Template for Road-feature

```

<?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>

```

The type-, value- and valuelocation-elements provide information to the application where to find the property-values, and what kind of values they contain. The classification is a string, and the number is an integer. The values will not be validated against any restrictions made in schemas, and the list of possible types to specify within the template only represent a small subset of values compared to the amount found in XML schema. The file *cambridge.xml* can now be loaded into JUMP. We load the file as two layers, one with the road-template and one with the river-template. Each template provides one layer of information, and can be edited separately.

The idea of specifying input templates is easy understandable and pretty straight forward. As long as users are working with a pretty limited set of features, the time spent creating them manually will probably be quite insignificant. On the other hand, if we want to make use of several different document conforming to different GML2-dialects, this process is not at all ideal.

The plugin functionality of JUMP should make pretty straight forward to extend the program with extra functionality. An automatic template generator could be a very useful plugin, one that could be realized using the solutions presented in this thesis. It is important that the schema analysis can be done in the most portable manner, because it is important to predict what kind of programs that would actually benefit from schema analysis. XSLT libraries are available for most programming languages, while implementing the analyzer in Java or any other programming language that requires compiling will make it less portable.

## GeoTools

The development of GeoTools[GTP] started at The University of Leeds in 1996. The first version was targeted at the applet API, this does now exists as GeoTools-Lite, while the further development of a more broad library continues, taking full advantage of existing Java technologies to develop an open source Java library for development of OpenGIS solutions. GeoTools is divided into separate modules, each implementing different requirements. A subset of these modules will be sufficient for most developers, but as a whole they cover a lot of ground when it comes to development of OpenGIS solutions. The Geotools FAQ states that “The aim of the project is to develop a core set of Java objects in a framework which makes it easy to implement OGC-compliant, server-side services or provide OGC compatibility in standalone applications or applets.”, furthermore they describe the strategy of implementation as “The GeoTools 2 project comprises a core API of interfaces and default implementations of those interfaces”[GFQ]. GeoTools are committed to implementing the standards set by the OGC. This ensures that GeoTools is developed according to OpenGIS specifications, formalized through OGC's structured committee programs and consensus process.

GeoTools strive to support as many geographical data formats as possible, making them accessible for the vast amount of functionality implemented in the GeoTools-suite. Different geospatial formats are transformed into the GeoTools feature representation format through different implementations of a *DataStore* or *DataSource* framework. In order to make proprietary data available to GeoTools, a new implementation must be built upon your data source, following the guidelines of implementation. Among others, GeoTools support *PostGIS*, *GML2.0* and *MySQL data*.

The *DataStore* interface is closely related to the OGC Web Feature Server Specifica-

---

tion, described in the section called “Web Feature Server (WFS)”, where a feature is describes as an atomic unit of geographic information. The `FeatureType` determines the properties of the Feature. In addition each Feature has an unique id.

## GeoTools DataSource (GMLDataSource)

The `GMLDataSource` is an implementation of the `DataSource` interface, meant to handle GML2.0, loading features from GML into the JTS topology suite. The implementation is however pretty "hard coded", the recognition of certain elements from GML is actually done by partial and full string comparison of element names. The following snippet shows how the native GML geometry properties and elements are copied with during SAX-parsing.

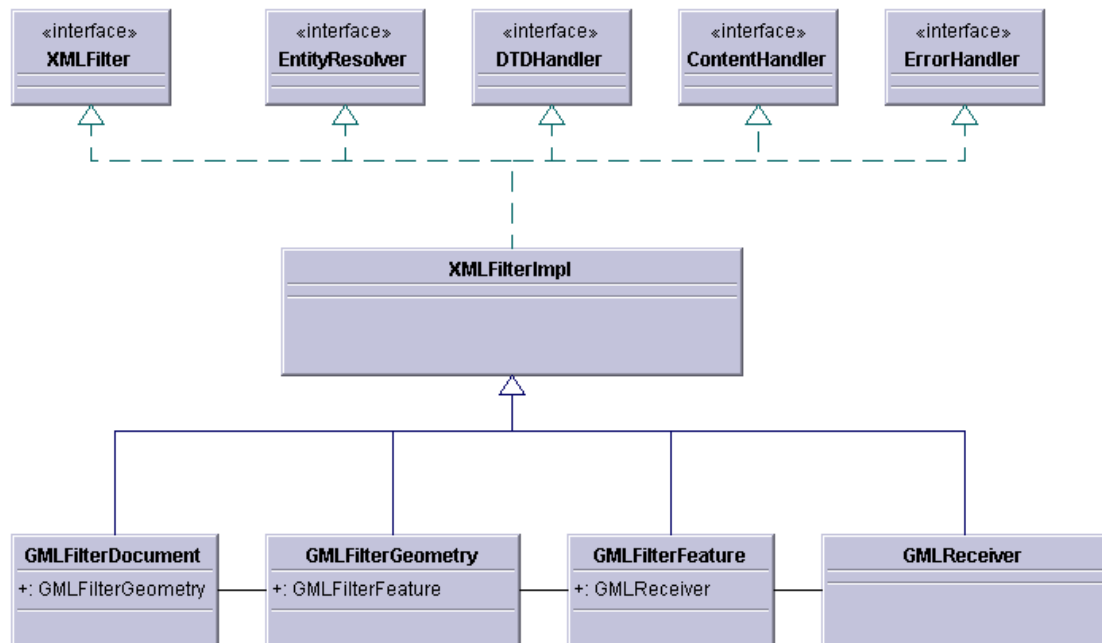
### Example 3.3. GMLDataSource recognition of geometry elements.

```
if (namespaceURI.equals(GML_NAMESPACE)) {
    // if geometry, pass it on down the filter chain
    if (BASE_GEOMETRY_TYPES.contains(localName)) {
        parent.geometryStart(localName, atts)
    } else if (SUB_GEOMETRY_TYPES.contains(localName)) {
        parent.geometrySub(localName);
    } else if (COORDINATES_NAME.equals(localName)) {
        // if coordinate, set one of the internal coordinate methods
        coordinateReader.insideCoordinates(true, atts);
        buffer = new StringBuffer();
    } else if (COORD_NAME.equals(localName)) {
        coordinateReader.insideCoord(true);
        buffer = new StringBuffer();
    } else if (X_NAME.equals(localName)) {
        coordinateReader.insideX(true);
    } else if (Y_NAME.equals(localName)) {
        coordinateReader.insideY(true);
    } else if (Z_NAME.equals(localName)) {
        coordinateReader.insideZ(true);
    } else {
        parent.startElement(namespaceURI, localName, qName, atts);
    }
} else {
    /* all non-GML elements passed on down the filter chain without
    * modification
    */
    parent.startElement(namespaceURI, localName, qName, atts);
}
```

If none of these tests, possibly the first one isn't true, the handling of this *elementStart* is passed directly on to the parent *ContentHandler* in a chain of handlers. Figure 3.3, “GMLDataSource SAX filters” shows the data flow through the provided filters of the `DataSource` implementation, where the top `ContentHandler`; *GMLFilterDocument* controls the flow of data as shown in the program listing above. This fraction of code is actually found in the `startElement`-method in `GMLFilterDocument`.

---

**Figure 3.3. GMLDataSource SAX filters**



The GMLDataSource works perfectly well when working with certain vocabularies, but it does indeed fail on others, because it is assumed that naming is done using a certain convention. Another quick 'hack' is to be found in the class GMLFilterFeature, where featureMember- and featureCollection-elements are recognized merely on the basis of their names (Example 3.4, "Recognition of features in GMLDataStore"). The comments of the author clearly indicate that this solution is not optimal concerning how the elements are identified. As soon as the elements have different names, it is useless to utilize this code to find features!

### Example 3.4. Recognition of features in GMLDataStore

```

[...]  

public void startElement(String namespaceURI, String localName,  

    String qName, Attributes atts) throws SAXException {  

    if (localName.endsWith("Collection")) {  

        // if we scan the schema this can be done better.  

        NAMESPACE = namespaceURI;  

        //_log.debug("starting a collection with namespace " + NAMESPACE + " and  

        Name " + localName);  

        return;  

    }  

    // if it ends with Member we'll assume it's a feature for the time being
    
```

```
// nasty hack to fix members of multi lines and polygons
if (localName.endsWith("Member") &&
    !localName.endsWith("StringMember")
    && !localName.endsWith("polygonMember")) {
    [...]
}
{ [...]
}
```

By defining a schema where the features and feature collections are defined with names not ending with 'Collection' or 'Member', the features of the instance document will no longer be available to the GMLDataSource. Figure 3.4, “Extending and substituting featureCollection” shows a perfectly legal way to extend a FeatureCollection, and at the same renaming it. If we want the GMLDataSource implementation to be able to parse documents where this collection is present, some code altering is necessary.

**Figure 3.4. Extending and substituting featureCollection**

```
[...]
<xs:element name="HaldenByNight" type="HaldenByNightType" substitutionGroup="gml:_FeatureCollection"/>
[...]
<xs:complexType name="HaldenByNightType">
  <xs:complexContent>
    <xs:extension base="gml:AbstractFeatureCollectionType">
      <xs:attribute name="lastupdated" type="xs:dateTime" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
[...]
```

The primary drawback of implementing access to your data using DataSource-interfaces, is that one DataSource only provides access to one feature type. There are also some issues regarding performance, as all the features are loaded into memory. The implementation is therefore best suited for small data sets. Access to subsets of features is possible by implementations of *Filter* or *Query*.

## GeoTools DataStore

DataStore supersedes DataSource as interface for data access. It provides all the basic functionality found in DataSource, along with many improvements. The most obvious improvement when it comes to functionality, is *support for multiple feature types* for

---

each DataStore. This makes it possible to read multi-feature documents using one DataStore.

DataStores also improves performance when working with a big data sets. Features can be loaded and manipulated one by one, not exhausting limited memory resources. It is still possible to gain access to feature collections, as in-memory structures if this is needed.

For working with features, GeoTools provides two interface specifications. *Expression* and *Filter*. Expression-classes are implemented to perform calculations on features, returning a generic object. Expressions are usually composed of other Expressions. Filters are implemented to be able to extract features that satisfy certain criteria. Filters can perform tests on attributes and geometries of features, and reports back whether a feature satisfies the filter condition or not. A Filter can be wrapped in a *Query*, to provide more complex conditions. David Zwiers has started the development of a *GML DataStore*, the project was recently added as a branch of the GeoTools project. The GMLDataStore is intended to be OGC GML 2.1 compliant. Most likely the DataStore implementation will make use of schema parsing in order to interpret document content, but the details are still unknown.

## 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[CLEO]. 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 fields to expose. Example 3.5, “Configuring Cleopatra” shows a small fraction of the configuration file for Cleopatra, where XPathS to specific parts of the document are provided. For each application schema and document, this configuration file must be present for Cleopatra to parse the data correctly. By pointing to external *Cascading Style Sheets* (CSS), the features' geometries are styled for viewing.

### Example 3.5. Configuring Cleopatra

```
[...]
<!-- this has various GML application Schema specific xpathS -->
```

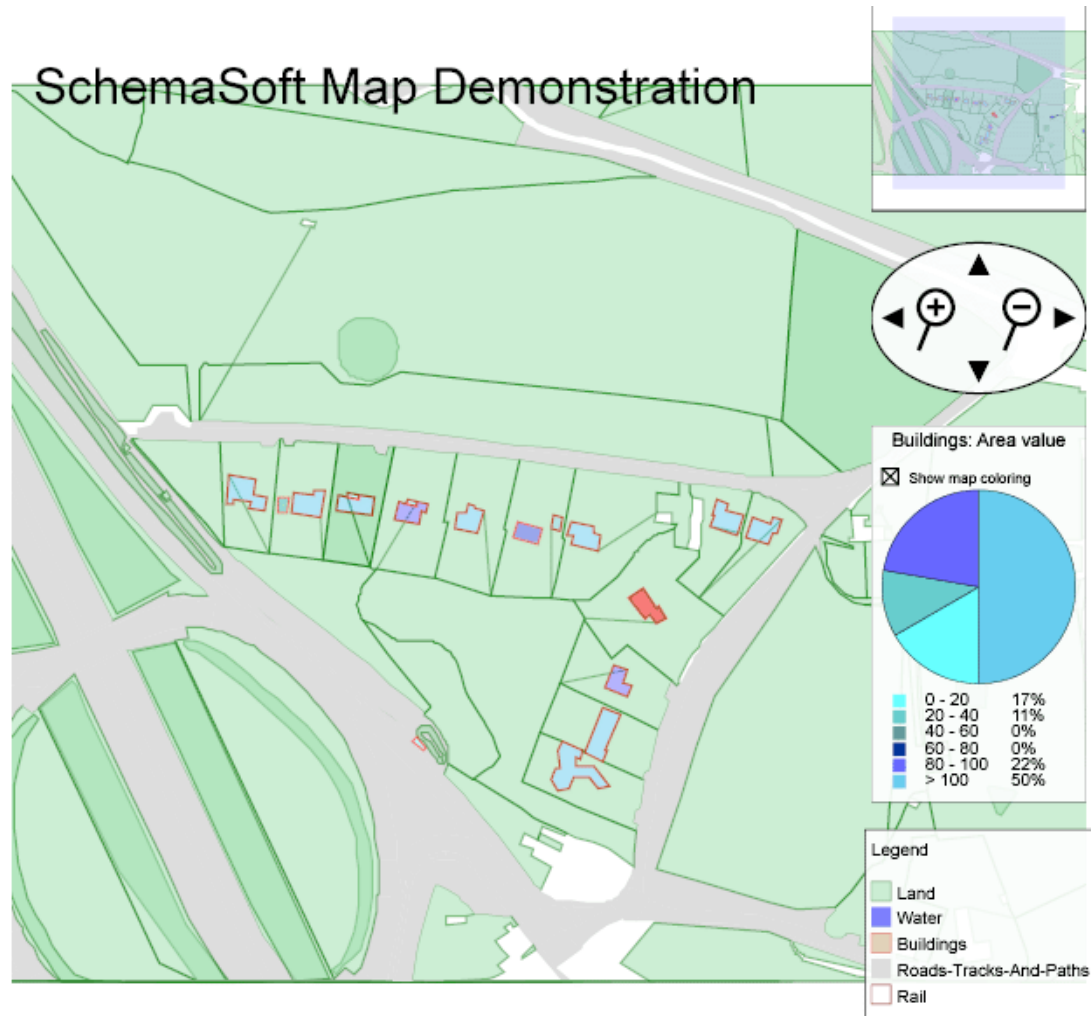
---

```

<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>
[...]
```

The configuration files for Cleopatra are pretty thorough and complicated, which also makes the viewer very customizable. There is no tool provided for creation of such files, so they have to be made manually. A schema parser could provide support for recognition of generic feature types and properties so that the creation of these files would be easier. Treating feature- and property types based on their ancestors could also be possible in applications like Cleopatra. One example is styling, where every feature, which is descendant of a certain GML feature type, should be styled in a certain way. This relationship is only possible to recognize through a schema analysis. This is a Figure 3.5, “Cleopatra demonstration screenshot” is a screenshot from a demonstration found at *Schemasoft's* homepage.

**Figure 3.5. Cleopatra demonstration screenshot**



## GO Loader

Snowflake Software Ltd.[SFL] has developed specialized software for working with GML data. They offer a free *Ordnance Survey (OS) MasterMap* [OS] viewer that is made specifically to work with data based on the OS application schemas. Ordnance Survey is a government department and executive agency, providing a broad range of products and services. Surveying and topographic mapping of Great Britain is the basis for their activities, one being digitally storing the geospatial data and offer it as OS MasterMap GML. Even though the free viewer only supports OS MasterMap, they offer a viewer for generic GML, but not as freeware.

The GO Loader is developed to serve the purpose of "modeling, loading and maintenance of content delivered in GML into an Oracle Spatial / Locator database.". The



---

most exciting fact about this loader, is that it *analyzes the application schemas* for the instances to be loaded, and therefore it does not require any additional meta data, to be able to parse and load the data into the database. Since this is commercial software, we can only speculate how the schema parsing is done, but the process is most likely done by utilizing some implementations of XML parsing libraries, like the lightweight SAX or the more complex DOM. An alternative, possibly combined with the mentioned ones, is of course XSLT.

As correctly pointed out on Snowflake's website, there are numerous benefits of this approach. The ability to read new sources of GML without writing any kind of translation software or input/output templates, might be the most significant advantage over other similar tools. Especially if you are frequently loading GML data, based on unknown application schemas, you will benefit over and over again. Just as a growing number of other geospatial software, GO Loader is written in Java, making it possible to execute it on a number of platforms.

---

---

# Chapter 4. Handling arbitrary GML sources

The preceding chapter hopefully gave insight to some of the applications somehow dependent on utilizing GML for various purposes. They are all exposed to the problems related to the handling of generic GML. The implementers have all chosen different strategies for the task, some more reliable than others. This chapter covers and discusses the problems that arise when you programmatically want to work with on beforehand unknown vocabularies of GML. Additionally, it covers profiling of GML, a well-known strategy to restrict the loosely defined GML rules regarding both allowed structure and content of documents.

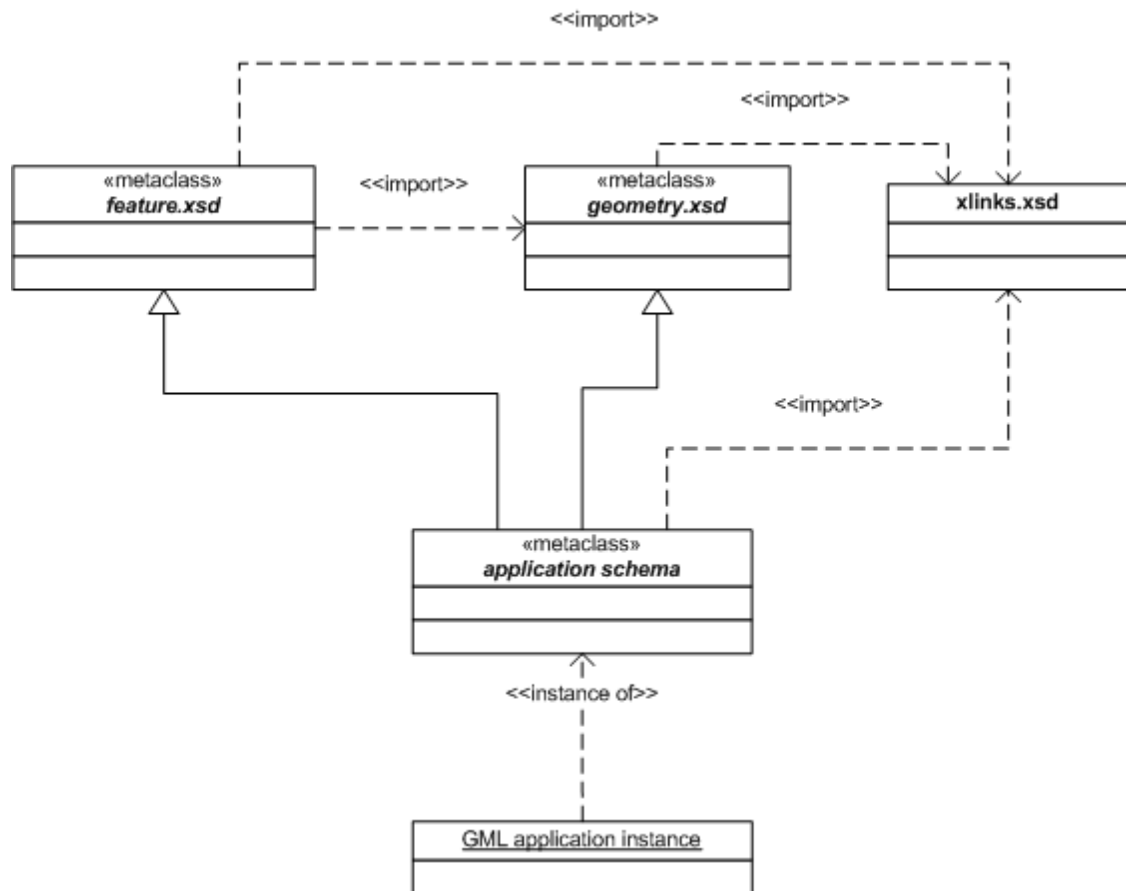
## Handling arbitrary GML

GML application schemas are developed from the base GML schemas, replacing abstract types and declaring substitution groups. The development is often done in layers, leaving the final application schemas with datatypes extending or restricting base GML types indirectly through other application specific types. This naturally increases the complexity in finding back to the source, even though it is clearly feasible when considering for example schema validators. These make sure that instance documents conform to their schemas, meaning that they validate the grammar against the base data types.

Developers of applications and toolkits such as GeoTools and JUMP strive to make their software support as many data formats as possible. GML2 is difficult to deal with in a generic way, because the elements defined in the schemas *feature.xsd* and *geometry.xsd*, are not sufficient and not intended to serve as the only schemas for instance documents. They define a meta-language, to be used as a basis for other another meta-language. Elements like *\_Feature* and *\_FeatureCollection* and their corresponding datatypes are *abstract*. This means that it is a requirement for application schemas to define their own datatypes, deriving and substituting for these. Handling arbitrary GML sources in a generic way is not a trivial task, nevertheless, it should be a sought after functionality in many programs, considering e.g. some of the applications presented in Chapter 3, *GML software* . Most systems does handle one GML vocabulary only, nevertheless, this does not necessarily mean that the data format is not interesting

in some other context than that served by this specific application. Open source toolkits for analyzing GML schemas could provide some assistance for application developers to make software less vendor specific. Figure 4.1, “Basic GML application schema” illustrates how a simple GML 2.0 application schema is designed. Through utilization of the base GML data types, found in the two schemas *feature.xsd* and *geometry.xsd*.

**Figure 4.1. Basic GML application schema**

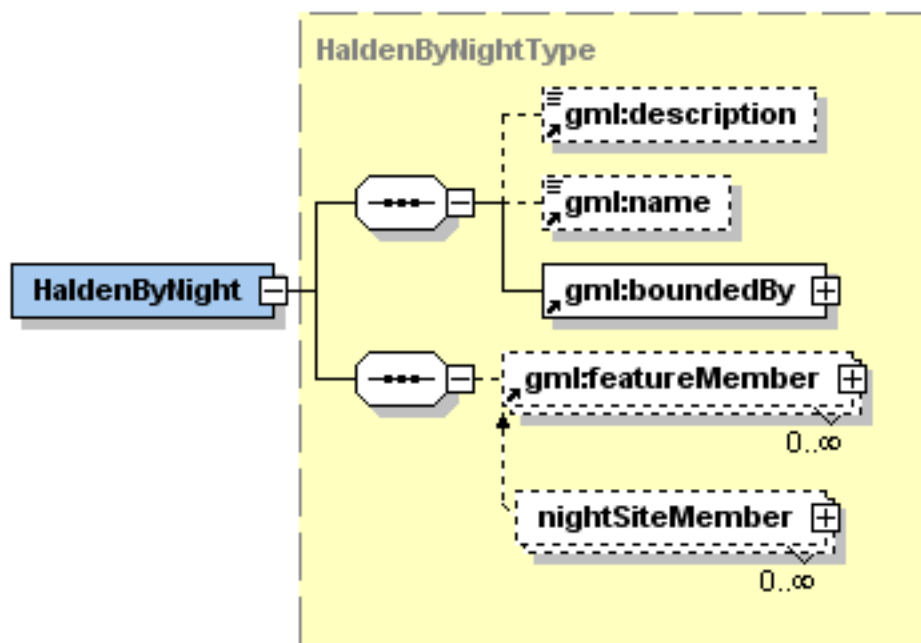


A simple GML application schema is illustrated in Figure 4.2, “HaldenByNight application schema”. This schema is created to model the places to go when day turns into night in the small town of Halden. The element *HaldenByNight* is the root element of application documents, making it an instantiation of a *FeatureCollection* complex type. Figure 3.4, “Extending and substituting featureCollection” shows how this type is declared in the schema, namely as an extension of *gml:AbstractFeatureCollection*, where *GML* is the namespace abbreviation for the GML-namespaces. The element is declared to substitute for the *gml:\_FeatureCollection*, in accordance to the GML 2.1.2 specific-

---

ation. Feature elements and indeed other FeatureCollection elements are enclosed within the non-abstract *gml:featureMember* element or possibly a specialized feature member type, deriving *FeatureAssociationType*. The fact that a feature collection legally can be nested inside another featureCollection, makes it clear that schemas and thereby documents can be built in a recursive manner. This makes GML very flexible considering how the geospatial data can be modeled, simplifying grouping of related element in an intuitive manner.

**Figure 4.2. HaldenByNight application schema**



Instances of the HaldenByNight-schema is valid GML2, and is therefore also valid data for the Cleopatra viewer, the JUMP application presented in Chapter 3, *GML software*, or any other application somehow able to work with arbitrary GML. However, these applications rely on configuration files to be able to handle the generic GML.

Handling arbitrary GML should be straight forward when considering the close relation between various application schemas, but it turns out not to be. Significant GIS vendors have addressed the problem with different solutions to the problems. Profiling, as discussed in section the section called “GML profiling” is one of them, particularly addressed by *Environmental Systems Research Institute* (ESRI)[GPR] one of the world leading vendors of geographic information systems. As earlier mentioned, documents can be defined in a recursive manner, meaning that a feature collection may be nested

---

deep inside a hierarchy of other feature collections. This loose restriction on structure, provides the freedom to indeed model your documents recursively. One of the advantages of this doing this, is that the features and feature collections, can be grouped in accordance with their relationship and their role within the entire structure. It has been pointed out that a great deal of flexibility, restricts and makes it complicated to work out guidelines for interoperability between GML sources.

## GML profiling

A profile of GML is a restriction of the basic descriptive capability of GML. These profiles may either be defined by construct of additional schemas, or as procedural agreements within an information community[GML20]. Additionally, a GML profile may not be defined so that it goes beyond the constraints of the GML specification. ESRI has taken the initiative to make a common profile for GML. Through meetings with members of the OGC, a software extension, *The OGC Interoperability Add-on for ArcGIS*, has been released to extends their desktop suite, ArcGIS, to act as a client for OGC WMS-and WFS-based services. By using this extension, GML data fitting a certain profile can be exchanged between different vendors. Among restrictions coerced through the profile is[GPR]:

- A FeatureCollection should not include endless levels of other FeatureCollection elements
- A FeatureCollection should include a homogeneous set of features.
- Features should contain well-defined data types.

There is no doubt that by profiling GML, and thereby avoiding unnecessarily complicated application schemas, implementing tools for interoperability between systems is less complicated. Profiling is however restriction of the abilities of GML, and thereby the application schemas. Pitfalls may of course be avoided if developers adhere to a GML profile. A profile defined by the creation of one or several schemas "on top of" the base GML schemas, can be viewed as a meta-dialect of GML. To describe a system as generic because it accepts GML in accordance to one or several specialized GML profiles, is of course not correct, even if the profile is widely agreed up on by vendors. A generic parser must be able to read and possibly write arbitrary sources, as long as they are valid against the base schemas. Nevertheless, widely accepted profiles, both the physical and normative, like the ones agreed up on by OGC, have been made

on basis of field experience and addressed problem issues. Therefore, when designing application schemas, developers should at least keep topical profiles in mind as guideline to reduce complexity and encourage interoperability.

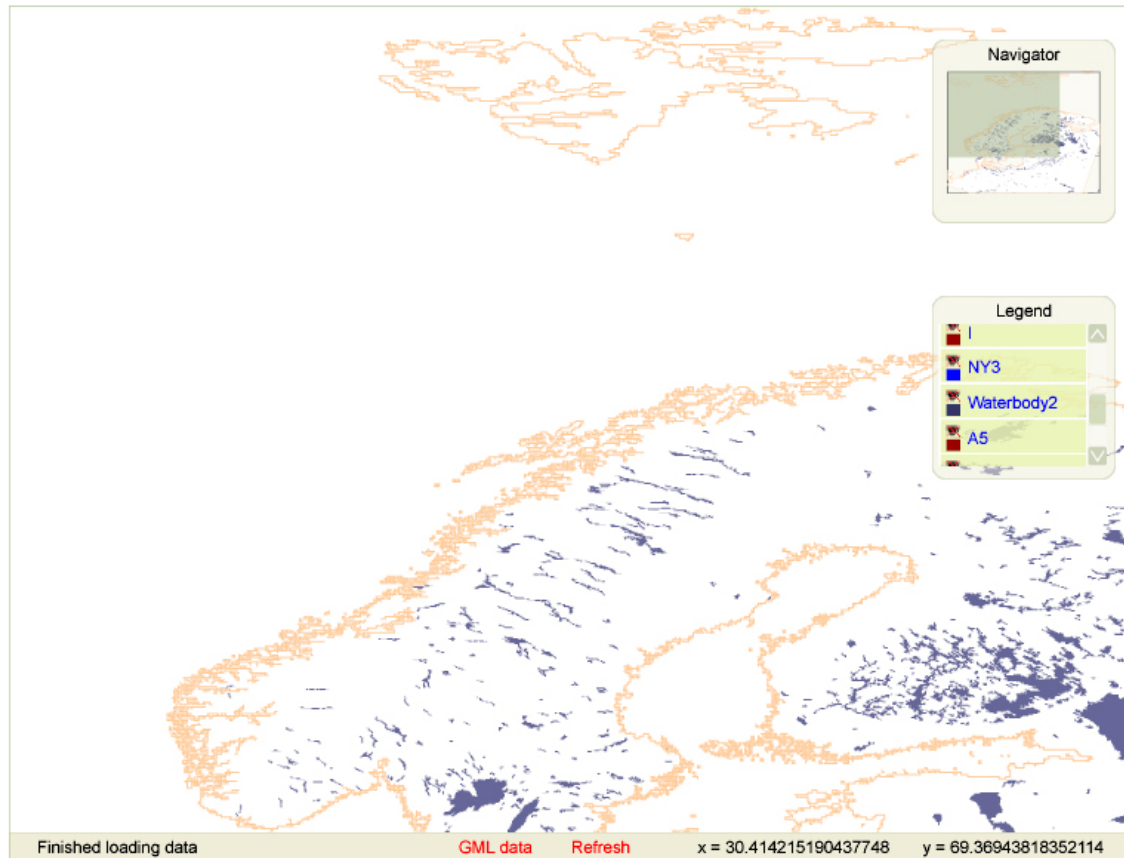
Specialized GML profiles could successfully be introduced for particular fields, where certain modeling features of GML is futile. By such standardization, interoperability and homogeneity, is aided. At the same time business specific structures and modeling rules could be introduced into a profile, to make it more specialized towards the field of interest. For instance transport planning profiles or environmental computing profiles, can provide specialized functionality commonly needed for utilizing systems. On the other hand, developing a generic GML profile plunders some freedom and flexibility from GML. As long as there are no such restrictions introduced into the GML specification itself, generic tools must be able to interpret all profiles of GML.

## Project OneMap

Project OneMap is hosted by and coordinated from *Østfold University College*, Faculty of Computer Science, Halden, Norway. By combining efforts from several contributing parties, we hope to collect, manage, process and provide global, comprehensive and detailed geospatial data, free of charge. The project is designed with three main infrastructure components[ONE].

The *Gateway* is the main entry point for users of the service. *"The Gateway is a browser based user interface, for retrieval of OneMap data. Currently this imply a SVG/JavaScript implementation, which provides simple but sufficient navigation and query possibilities."* [GED], The Gateway is based on the same principles presented the OneMap GML Editor[OME]. Using the SVG editor, changes can be carried out on the SVG model before they are submitted to the central server for update of the source data.

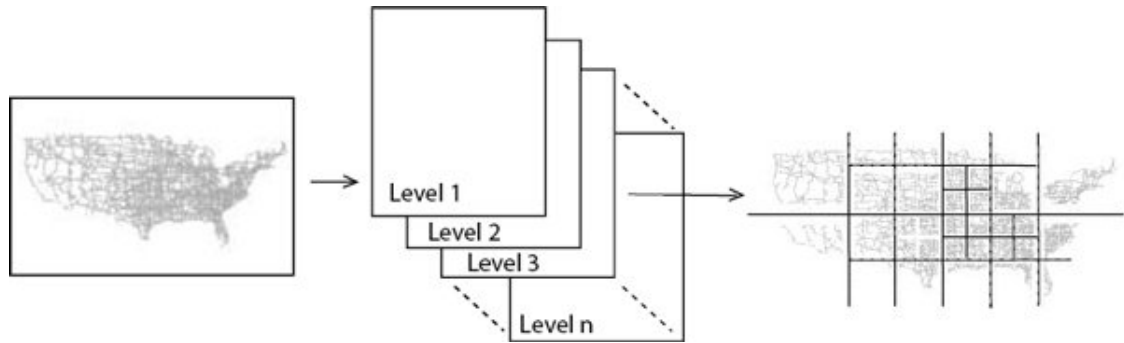
### Figure 4.3. OneMap: Gateway screenshot



The *Clearinghouse* represents the modules related to data submission. Data is collected through contributions from a wide variety of parties. Clearinghouse focuses on tasks concerning building, updating and revising the OneMap geodata. The main objective, namely to accumulate enormous amounts of geodata, covering the entire globe, is done in an uncoordinated, consensus driven manner, based on the principles of *peer review* [ONE].

Finally, the heart of OneMap, the *Repository*. This is a distributed storage structure, where a huge set of XML files are stored and managed to efficiently support retrieval and updating the geodata comprising the world map. Each feature type, e.g. *roads* and *buildings*, are presented in a global logically consistent layer. These layers are preprocessed into a level-of-detail hierarchy, subsequently they are tiled by *adaptive quad-tree subdivision*, so that the size of each tile is below a certain threshold.

**Figure 4.4. OneMap: Repository**



Obviously, the choice of using GML as OneMap data exchange and storage format, is based on the fact that GML is a widely adopted, open standard specifically designed for the task. The gathering of data from several different sources would have been impossible if the system required data on some sort of proprietary format. Even a request for data in conformity with a certain GML profile, could imply that submitted data from important contributors, would require time-consuming transformation to be valid. In such a system, the best solution could be to store all GML data as close as possible to its original format as possible. Dependent on in what extent the system is able to handle arbitrary GML, this can be an option or not. This would require efficient and reliable processing and analysis of the data, a very difficult task compared to working against one GML application schema only. When grouping data in layers, e.g. *roads*, *buildings*, *rivers* etc., like done in the OneMap system, human interference is required to recognize these features/feature collections. In such a system, a GML dictionary creator can be just the tool needed for the treating GML generically.

## Lazy 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 are found in[OBI]. The geometric integration corresponds to the problem often referred to as map conflation[CON]. In the following we assume that each submitted feature or feature collection may be classified according to the OneMap Feature Catalog.



The 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 feature collection corresponds to a traditional map layer. Further, we want to restrict a given OneMap integrating feature collection 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 feature collection, 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 it should be easy to include a new external feature type in a given integrating feature collection.

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

The theory of data integration is simple, namely to include features and GML types into OneMap system. A feature is supposed to be included into an instance document as is, meaning that the system has to be able to handle all features generically. The schema standard and namespaces does of course allow us to import as many namespaces into our application schemas as desired, so the focus is on integrating the data in a way so that the system can make use of it.

The strategy for structuring the data in the system is as layers of related features. These features are not homogeneous in terms of their GML definition, nevertheless they are heterogeneous representations of the same real world objects.

## **Figure 4.5. Integrated schema hierarchy**

Figure 4.5, “Integrated schema hierarchy” 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.

As an example, consider Figure 4.6, “River fragments constituting complete river” where a river is represented as a feature collection, consisting of features being fragments of the complete river. The membership of a river is restricted through the integration schemas, where features qualifying as a fragment of a collection are registered. When creating instance documents we can now integrate features from other documents directly into our own document. This can be done by copying them or possibly by allowing the use of linking to other documents in the integrating schema.

### **Figure 4.6. River fragments constituting complete river**

This integration of features can give an extremely rich feature set. Schema analysis is an important tool for any such application. The next section discusses how to implement the schema analysis library to most efficiently be able to utilize it in applications where this kind of functionality is needed.

## **Implementation discussion**

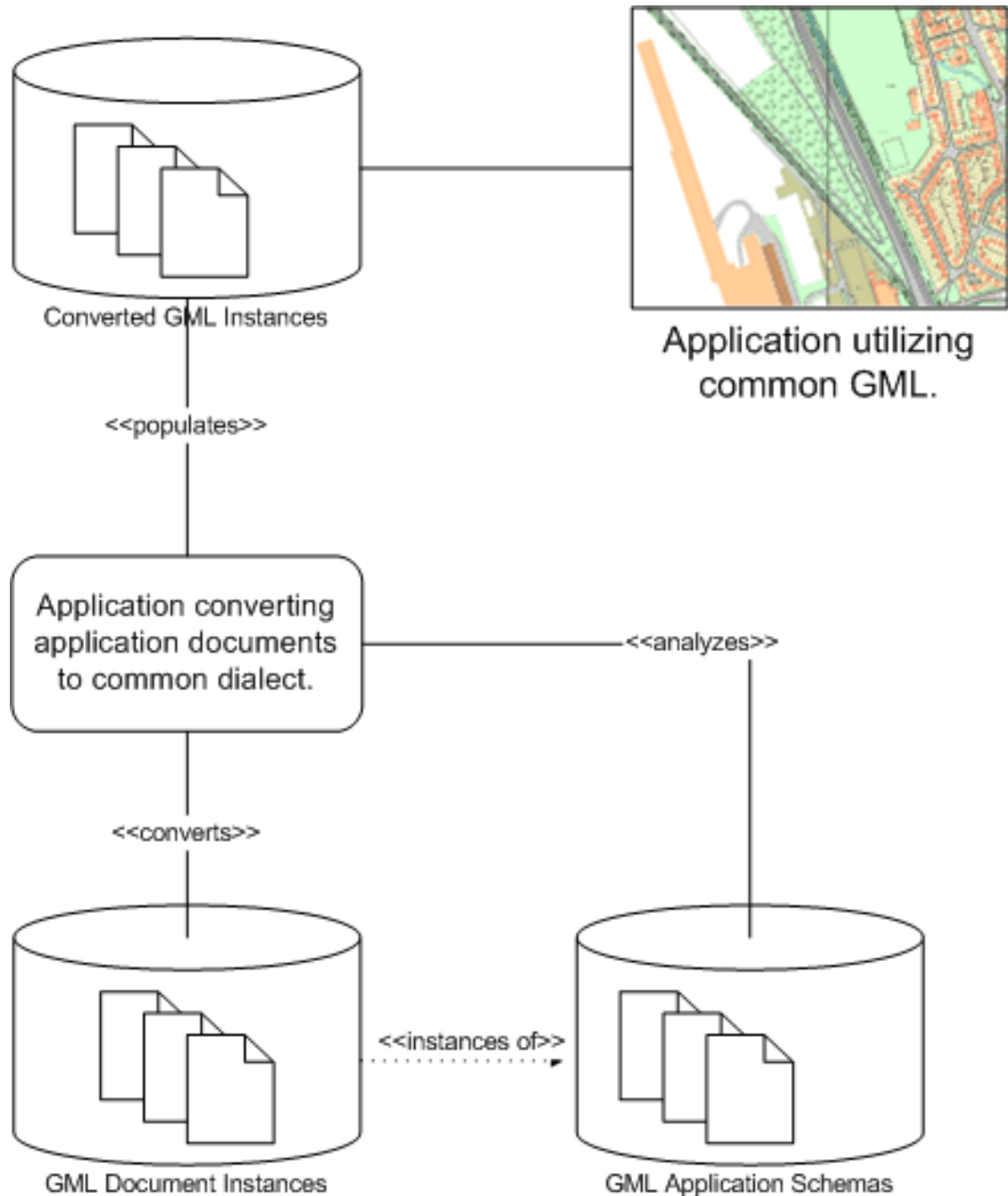
Being able to analyze and handle arbitrary GML could bring a good application to the next level. Some applications are preconfigured to read a set of known vocabularies, and often this is sufficient. However, many applications could make great benefits from being able to interpret new application schemas and thereby being able to work with their instance documents. Such functionality could be made available in a number of ways, including e.g. a Web Service or implemented in a code library. No matter the final solution, it should be kept in mind that the implementation should be portable and usable in any type of application.

## **Converting application schemas and documents**

---

By defining and implementing a method to convert and transform application schemas and instance documents to a common format, we can store the converted instance document in our own proprietary format. We can identify restrictions, extensions, substitution groups and other relevant information, together with the transformed version of the documents. This way applications can be programmed to access the data directly. Now applications can utilize "any" GML, through the converted documents. Figure 4.7, "Application using converted GML-documents".

**Figure 4.7. Application using converted GML-documents**



XML Schemas provides a great deal of flexibility speaking of structure and content in the instance documents. We can build schemas able to store the semantics within a range of different document. These can be designed loose, meaning that it is possible for users and applications to add almost any valid XML content, still keeping the entire document valid. The drawback of such design, is the fact that it proves complicated to transform or interpret the data, when the content of a final document is unpredictable

---

to parsers or other utilizing applications.

With this approach, the conversion process would be minimal, leaving the interpreting to a parser or interpreter, thus thereby making the coding of these more complex. Much of the original data from the application schemas could be kept as found, merely adding information-elements in the document, describing the origin of datatypes.

The option to alter instance documents directly, is not optimal. First and foremost, changing schemas to make room for meta information inside the documents, is like denying the existence of available meta information in the schemas. This method "copies" the structure information into the document, duplicating it, but possibly making it more accessible and processed for utilization. This requires that each instance document will have to provide this information, even though they descend from the same application schemas as an unlimited amount of other documents. Small changes in the schemas, will also require the reprocessing of every document, to make them up to date. Finally, you also have to change the application schemas, to make room for a meta data section, if you want the documents to validate.

As a solution meant to act as a foundation for applications, to able to deal with generic GML, it is a poor. Converting original data to a certain proprietary format, means that any such "generic" parser, must be able to deal one profile of GML only, and to characterize it as generic, would be somewhat incorrect. In addition, the conversion of all data going into a such parser, will take up an unacceptable amount of resources both speaking of storage and processing.

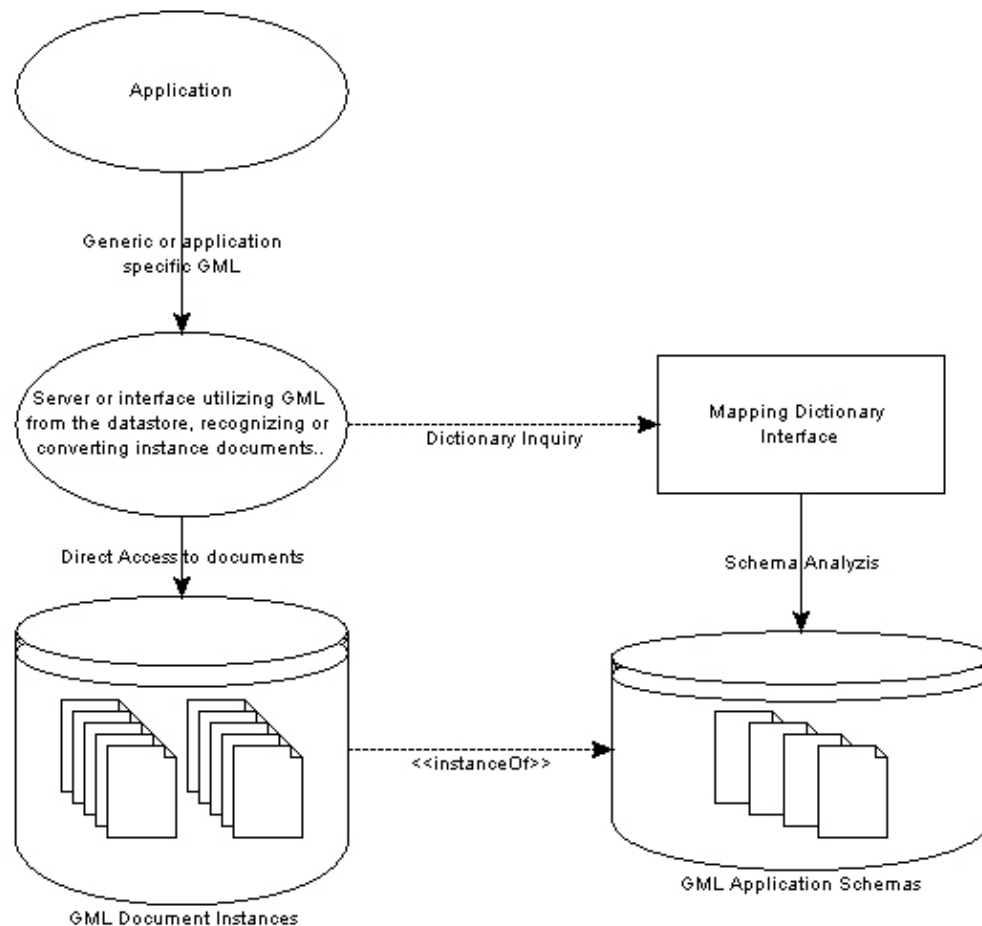
## Constructing a GML mapping dictionary

By analyzing the application schemas, the structure of the instance documents can be stored. As long as the schemas is not altered in any way, the instances are bound to the structure and rules of the basic GML2 schemas, specialized by the application schemas. If provided a code library for this purpose, applications can access unknown dialects of GML, treating application specific data types with knowledge of their origin. A solution may be to implement a *query-interface*, for applications to utilize if origin of datatypes is of any importance to the application Figure 4.8, "Application utilizing a schema dictionary". This approach is much less critical than a conversion of schemas and documents. A dictionary can be accessed when needed, providing the desired information about the application GML types. No conversion is necessary, just direct or

---

queryable access to the a mapping dictionary.

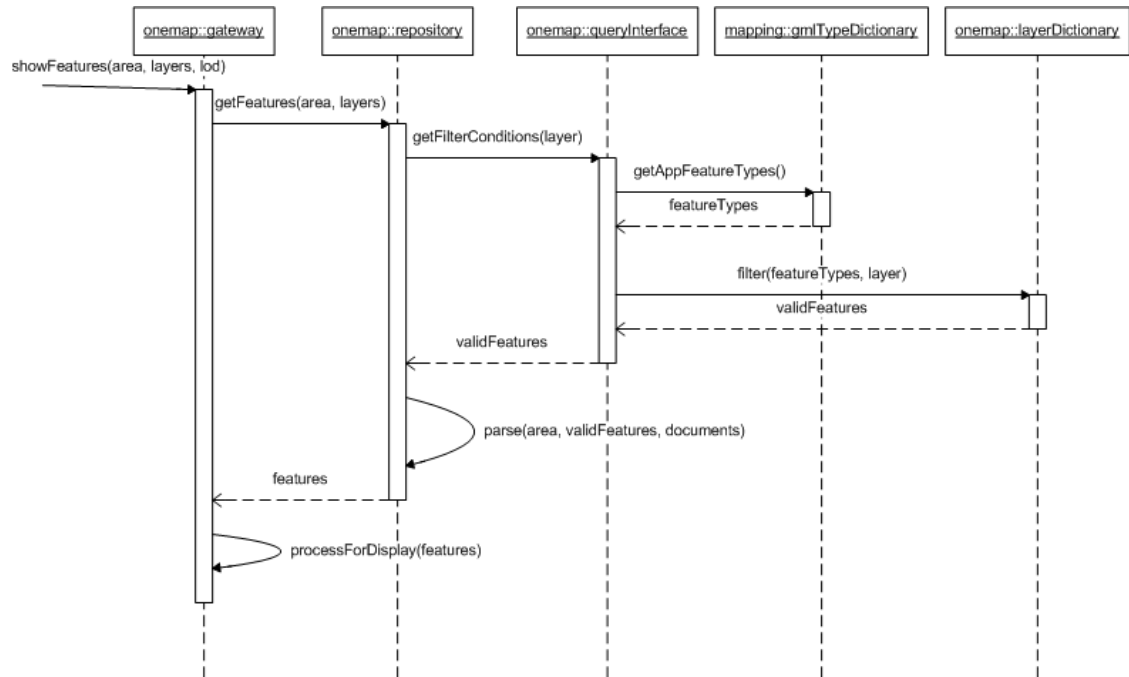
**Figure 4.8. Application utilizing a schema dictionary**



We can try to see this in connection with the OneMap gateway, if this one time in the future would utilize generic GML from the repository (see Figure 4.9, “Retrieve generic GML from repository”). This design is somewhat different than the current one, where layers of roads, coastlines etc. are found in separate files. Here we can imagine the files from different contributors stored as provided, possibly preprocessed into smaller files because of the support for level-of-detail display. When the gateway requests the coastline layer from the repository, the repository first inquires a *mapping dictionary*, for type-mappings. A *layer dictionary*, have to be made manually, by selecting the features or feature collections (traced by mapping) belonging to each specific layer. When this information is gathered, the instance document can be filtered, be-

fore the layer features inside the desired area, are returned. Access to the mapping dictionary, will probably also be necessary for the viewer, to sort out what kind of geometry types, features have.

**Figure 4.9. Retrieve generic GML from repository**

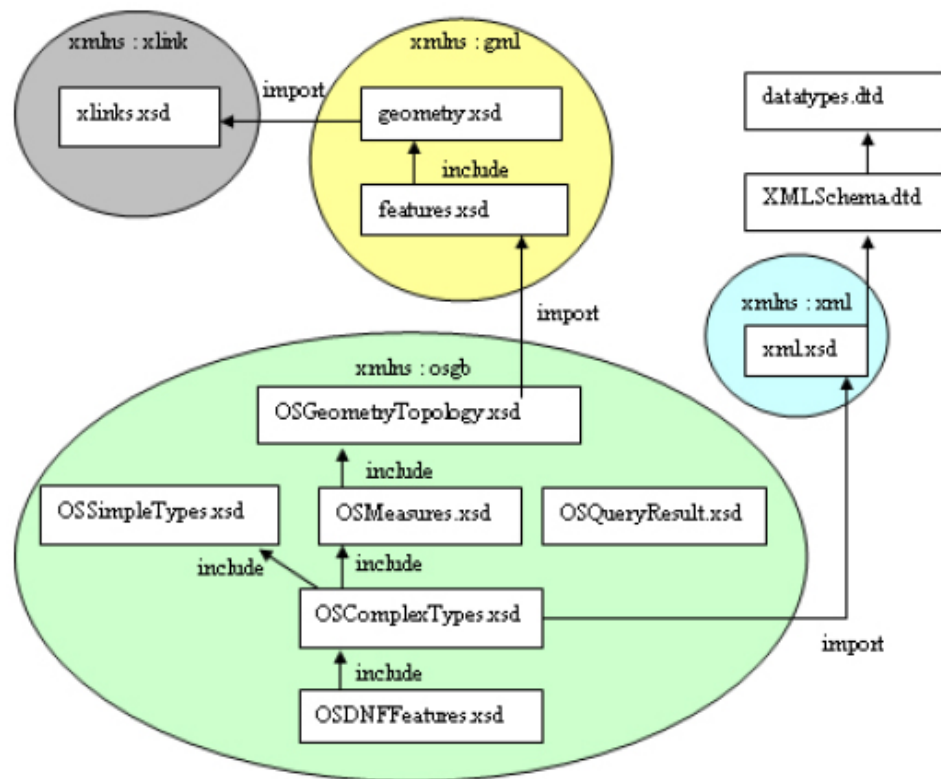


The primary source for document information, are of course the application schemas. Provided that all schemas, both the ones describing the application namespace, and those imported into the target namespace, are available, we can parse the schemas to get all the information we need. An XML application, can consist of one or several schemas, all with a common target namespace. Several schemas, define the Ordnance Survey MasterMap namespace, with the *OSDNFFeatures.xsd* as root, including directly or indirectly all the additional ones. Figure 4.10, “Ordnance Survey MasterMap schema structure (OSMasterMap User Guide)” [MMUG]. Dividing the target namespace definition into several schemas is merely a method to organize the schemas in a more structural way. Related types and elements, e.g. those constituting an abstract, application specific layer, can be found within one file. Alternatively, different structures, like *simpleType* and *complexType*s, can be split into individual files. This is naturally one of the facts that we have to be aware of, especially considering that even though the root schema is available, other included schemas might be inaccessible. We might therefore have a situation where we have a partially available vocabulary.

---

**Figure 4.10. Ordnance Survey MasterMap schema structure (OSMasterMap User Guide)**

### Schema structure



### XML namespaces

xlink – <http://www.w3.org/1999/xlink>

gml – <http://www.opengis.net/gml>

osgb – <http://www.ordnancesurvey.co.uk/xml/namespaces/osgb>

xml – <http://www.w3.org/XML/1998/namespace>

There are a number of GML vocabularies available (e.g. OS MasterMap, Top10NL etc.), in some cases defining, more or less, the same type of features and geometries. Considering for instance that a *geometry property*, defined in another GML vocabulary fits your needs, you can benefit from the *import*-mechanism. By importing a namespace, the constructs in it, will be made available to the importing schema. This is

---



the only way to make use of a schema with a different target namespace than the one you are defining. If we want to make a complete parsing of a vocabulary, it is also necessary to parse the imported namespaces, to find out what kind constructs therein is utilized by the importing schema.

A GML vocabulary is fully defined through the it's referenced schema. This schema may, as mentioned, have *import* and *include*-statements, to utilize constructs from other namespaces (import), or possibly include other files from same namespace (include). For parsing and working with the schemas, we can choose any technology suited for XML document access. However, it is likely that we need to access the schema constructs in a random manner, meaning that ordinary SAX-parsing isn't the right tool for this job. DOM however, could prove quite perfect for the job, considering that we will benefit from the advantage of an in memory structure of the schemas. In addition, you will probably not find a schema or schema hierarchy so extensive, that this will fail due to lack of computer resources. Having said that, there are already available code libraries to access and explorer schema vocabulary.

## XML Schema API

The *XML Information Set* [PSVI], often referred to as the *InfoSet*, is describes as "a set of definitions for use in other specifications that need to refer to the information in an XML document". The infoset describes what kind information from an XML document that should be reported from a parser. A parser reporting the constructs described in the infoset, is 'in conformity' with the infoset. In addition to a *document information item*, revealing information about the document as a whole, the infoset consist of *element information items* and *attribute information items*, one for each and every element and attribute in the document. There are also a number of other items, describing other possible constructs within a document. The infoset is available through *DOM Level 3 Core*, ergo a DOM3-parser can provide information about the document, described in the XML infoset[XIS].

XML parsers, supporting schemas, will upon request, validate schema-based documents. This results in an *extended infoset*, capturing validation results and type information based on how elements and other constructs are defined through the schema. This augmented infoset is called the *post-schema-validation infoset (PSVI)* [PSVI], but even though the information in this infoset is useful for many computing tasks, there was until recently, no common interface specification for accessing it. In December 2003, *IBM* and *X-Hive* submitted the *XML Schema API* [SAPI], a specification that defines

an interface to dynamic access and query of the PSVI. In addition it defines an interface for loading XML schema documents. The XML Schema API is platform -and language-neutral, like DOM and SAX, and is already implemented in the *Apache Xerces2 Java Parser* and *Apache Xerces C++ Parser*. The information found in a schema, is of course crucial information for 'schema aware' applications, e.g. advanced XML editors, schema editors or any other thinkable application in need of XML metadata information. Using the XML Schema API to access either schemas directly, or through a document being parsed, we can build schema aware applications without the requirement of building our own schema parser.

By using the XML Schema API, it is possible to build base-type-aware libraries for GML parsing. For efficiency, data being of importance to the application can be stored and accessed on demand. Using Xerces2 Java Parser to parse XML and access the PSVI, we have a very powerful framework for working with GML documents, opposed to for example using a 'lighter' implementation where we analyze the schemas using XSLT. Ideally, we will see a very flexible framework for analyzing and working with arbitrary GML. The drawback is that the specification is rather complex, which will make it very time consuming to program against the specification. Example code is also rare, and there is no question that implementations of this API is still pretty undocumented and untested.

It should be noted that this API is for accessing XML Schemas in general, and offers no functionality specific for GML or any other profile of XML. Using this an implementation of this API, will serve as a foundation for the schema specific constructs, while all GML logic must be implemented from scratch. *Galdos Systems* has developed a Java-based API, *GML4J* [G4J], to facilitate working with GML. This project is open source, with a beta release available for download. It seems like the project is in hibernation, considering that this beta is dated April 2, 2002. In addition, documentation is scarce, likewise open source implementations utilizing the API. Considering these facts, it was decided not to delve deeper into using this project, even though it surely could offer great functionality.

## Parsing schema with XSLT

The most lightweight method to parse the GML applications schemas, is to use XSLT, transforming the information in one or several schemas, into more accessible meta-information encapsulated in one mapping file. Thereafter, additional XSL transformations can be applied to an instance document, to provide desired output. If this second

---

transformation is made for generic GML, is up to the programmer, but a mapping file will provide up-to-date information about the schemas. A probable use-case would be a transformation made for one certain vocabulary of GML. By parsing the schemas, and transforming the documents using the mapping file, small changes and additional constructs extending the original schemas, could appear 'transparent' to the transformation because it can threat new types according to what kind of parent type it derives. Mapping files can easily also be parsed into data structures and utilized in GML-applications, no matter the implementing language.

The *Last Call Working Draft of XSLT 2.0*??? was released February 15, 2004, and according to the document, the working group is planning to advance the specification to become a *Candidate Recommendation*. This version represents significant increases in capability of the language, also considering that *XPath 2.0* [XP2] is developed alongside XSLT 2.0, and will be a part of XSLT 2.0 functionality. Perhaps one of the most significant changes, considering XSLT for our purpose is that while XSLT 1.0 completely ignored all element information, obtainable from a DTD or Schema, XSLT 2.0 documents takes into account such information.

## GML design issues

So far the gateway only utilizes one type of GML, OneMap GML. You would probably never see a totally generic viewer for GML, the reason is simply that it is too easy to design GML in a 'proprietary' way, not considered for common purposes. For instance, best practice guidelines for GML application design, recommends that application types derive as specialized GML base types as possible. Designers are however not bound to this guideline, meaning that they e.g. can build their own *LineString*-type extending the general *AbstractGeometryType*, instead of using or deriving the provided *gml:LineStringType*. A schema aware parser, will then be able to tell that it is dealing with a geometry type, but not be able to tell how such a type should be dealt with. This might prove as the main obstacle, making a generic viewer for GML. In some cases the necessary base geometry types are not available, thus requiring designers to build their own types. Drawing such types without any human interference, will probably not be possible. However, they can be identified by a GML generic application, making the schema parsing valuable, in spite of the fact that it is not totally generic.

When designing GML, properties and instances are interleaved, meaning that "a feature instance contains feature properties, each as an XML element whose name is the property name". Furthermore, "these properties contains another element, whose name

---

is the property value or instance; this produces a 'layered' syntax in which properties and instances are interleaved". To distinguish properties from instances, instances of GML classes starts with uppercase letter, while properties start with lower case. Figure 4.11, "Interleaved instances and properties", shows how the root element *HaldenByNight* is written with uppercase first letter, because this is an instance element of a *FeatureCollection*. Further, the element has some properties, one being a *gml:featureMember*, holding another element instance, namely a application specific *FeatureCollection*, *Surroundings*, with additional properties and instances.

**Figure 4.11. Interleaved instances and properties**

```
<?xml version="1.0" encoding="UTF-8"?>
<HaldenByNight [..]>
  [..]
  <gml:featureMember>
    <Surroundings>
      <gml:boundedBy>
        <gml:Box>
          <gml:coord>
            <gml:X>0.5</gml:X>
            <gml:Y>0.5</gml:Y>
          </gml:coord>
          <gml:coord>
            <gml:X>44</gml:X>
            <gml:Y>36</gml:Y>
          </gml:coord>
        </gml:Box>
      </gml:boundedBy>
      <gml:featureMember>
        <River>
          <gml:name>Tista</gml:name>
          <gml:centerLineOf>
            <gml:LineString>
              <gml:coordinates>8.0 0.5, 18.0 8.0, 19.0 14.0, 24.0 20.0, 30.0 22.0, 32.0 26.0, 34.0 36.0</gml:coordinates>
            </gml:LineString>
          </gml:centerLineOf>
        </River>
      </gml:featureMember>
    </gml:featureMember>
  </gml:featureMember>
  [..]
</HaldenByNight>
```

What kind of information is significant for a schema parser? First and foremost, for an application wanting to utilize the GML, to e.g. build a SVG document, recognizing geometric instances is crucial. This can of course be hard-coded in proprietary software, but when dealing arbitrary GML, the element names vary, so does the type names. *Association*-types can represents properties in GML; in GML2, we find *FeatureAssociation*- and *GeometryAssociation*-types. The *featureMember*-element is the only represented FeatureAssociation, while there are a number of GeometryAssociation-elements. Examples are *pointProperty*, *polygonProperty*, *lineStringProperty*, and some

more descriptive, substituting for these; *centerOf*, *extentOf* and *centerLineOf*. Figure 4.12, “Definition of PolygonPropertyType” shows how the *PolygonPropertyType* is defined, restricting *GeometryAssociationType*, dictating that an instantiation either encapsulates a *gml:Polygon* or points to one, using a *simpleLink*, defined in the XLinks-schema. Through associations, property values can be restricted and controlled, for example by only allowing certain feature members inside specific feature collections. Lack of document knowledge, will of course make it nearly impossible to threat such GML.

### Figure 4.12. Definition of PolygonPropertyType

```
<element name="polygonProperty" type="gml:PolygonPropertyType" substitutionGroup="gml:_geometryProperty"/>

[...]

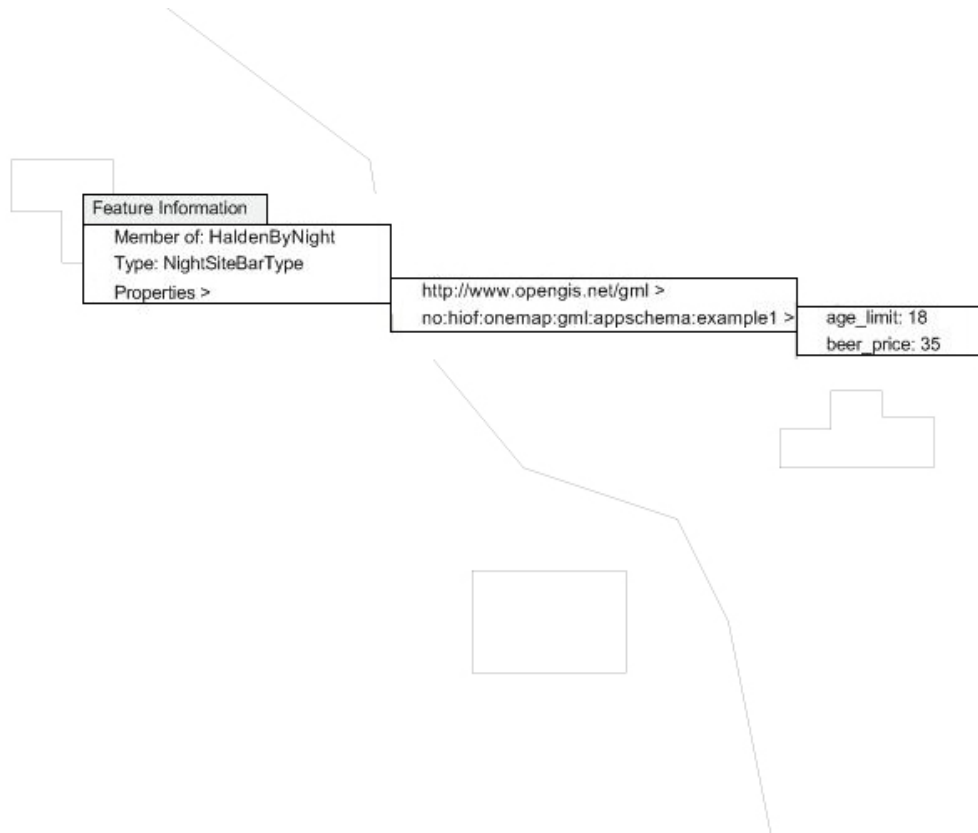
<complexType name="PolygonPropertyType">
  <annotation>
    <documentation>
      Encapsulates a single polygon to represent coverage or extentOf
      properties.
    </documentation>
  </annotation>
  <complexContent>
    <restriction base="gml:GeometryAssociationType">
      <sequence minOccurs="0">
        <element ref="gml:Polygon"/>
      </sequence>
      <attributeGroup ref="xlink:simpleLink"/>
      <attribute ref="gml:remoteSchema" use="optional"/>
    </restriction>
  </complexContent>
</complexType>
```

We do acknowledge that the geometric instances are very important when wanting to do calculations or wanting to transform GML into e.g. SVG; without these it is impossible to do any kind of mapping onto a coordinate system. We do probably also agree that subtypes of *AbstractFeatureType* and *AbstractFeatureCollectionType* are important to identify, to make it possible to view content related to the geometric properties of features and feature collections. Furthermore, when all geometries are in place, and the map is drawn, the non-spatial properties are important. GML instances can be very rich on non-spatial content, related to different features or feature collections within the document. These must of course be available, read-only or not. Figure 4.13, “Retrieving additional information about a feature” shows an example of how non-spatial properties can be retrieved, through accessing a feature, drawn out using one or several geometric properties. Considering that project OneMap, has developed a

---

GML Editor, able to handle GML2 compliant documents, it would be very interesting putting effort into making this editor able to handle generic GML.

**Figure 4.13. Retrieving additional information about a feature**



The number of predefined geometric types and properties in GML2, is very limited. In many cases, the types provided are sufficient to model the features, but this still leaves room for defining custom property names, to further enhance the relation between instances. In the *HaldenByNight*-example, a complex type, *LinearRingPropertyType*, is defined, deriving *gml:geometryAssociationType*, encapsulating a *gml:LinearRing* (see Figure 4.14, “Definition of *LinearRingPropertyType*”). Several similar geometric property types are defined in the base schema *features.xsd*, however none encapsulating a *LinearRing*-element. By instantiating these geometric types, we define properties like *centerLineOf*, *location*, *coverage* etc., ergo *role names* describing the relation between features or feature collections, and their properties. The roles do not necessarily hold a geometric property; the GML implementation specification describes an alternate view of this object model, the *functional view*.

---

## Example 4.1. Object model: functional notation

```
extentOf(House) = Polygon  
address(House) = String
```

**Figure 4.14. Definition of LinearRingPropertyType**

```
[...]  
<xs:complexType name="LinearRingPropertyType">  
  <xs:annotation>  
    <xs:documentation>  
      Encapsulates a LinearRing, to be used as a geometric property  
    </xs:documentation>  
  </xs:annotation>  
  <xs:complexContent>  
    <xs:restriction base="gml:GeometryAssociationType">  
      <xs:sequence minOccurs="0">  
        <xs:element ref="gml:LinearRing" minOccurs="1" maxOccurs="1"/>  
      </xs:sequence>  
    </xs:restriction>  
  </xs:complexContent>  
</xs:complexType>  
[...]
```

Maybe the functional view, is more intuitive, when discussing the importance of the properties. As shown, property names vary, depending on what information they actually describe represent inside an object. A *school*-feature, can e.g. hold two geometric properties, *schoolYardExtent* and *pupilAreaCoverage*, both encapsulating a Polygon. If we want to transform the document into a SVG map, the best solution would probably be to have a different style on the two polygons, maybe as a dotted line for the pupil-AreaCoverage and as a filled solid polygon for the schoolYardExtent. A set of *OS MasterMap style definitions*, is found in the user guide???. These definitions are default styles for presentation of data within OS MasterMap. All definitions are presented using SVG, and are can be used as reference for customers implementing their own viewers. To sum up, it will be nearly impossible to style GML automatically, because there are no way to know how the authors want to represent the features styling-wise. Some OS MasterMap features do not have a styling, so some will not be drawn when

the styling is applied. The reasons vary, there might be features that are more valuable as structural data, than viewable data, for example.

Undoubtedly, converting GML to a graphical format, without any other styling than a default one, will not serve use as a very attractive view of the data. Styling data, having knowledge of it, will give a more correct and intuitive view of features, presenting them in their correct role. At the same time, constructing a graphical view, maybe even one where it is possible to edit the data, will be sufficient for many purposes. This can be done by merely identifying the subtypes of the base GML types, mapping the geometric ones to a coordinate system, and making it possible to access the data within a features, through its graphical representation.

## 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.

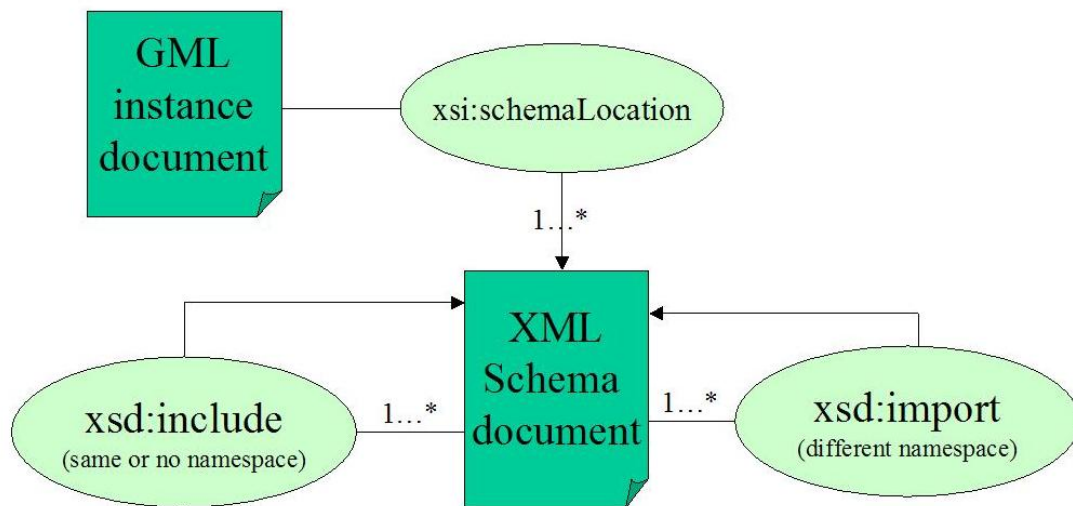


---

## Schema analysis

Schema analysis is a pretty straightforward 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 have 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. Figure 4.15, “Defining a GML vocabulary” shows how the file components are related when working with XML schemas.

**Figure 4.15. Defining a GML vocabulary**



xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsd = "http://www.w3.org/2001/XMLSchema"

---

When analyzing schemas, the main objective is to find out how elements relate to other elements, and possibly if they are 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.

## Example 4.2. Type maps from example data

```
<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>
```

### Example 4.3. Schema definitions of mapped types

```
[...]
<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[XSLT1] 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 classified 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[SX]. Using some of the functionality from XSLT 2.0 to simplify the implementation, we have developed a schema parsing XSLT stylesheet that parses one or several vocabularies into a mapping file. The stylesheet can either follow the *schemaLocation* attribute value from an instance file, or alternatively a specified 'root' schema or a manually 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[CODE]. 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 as it will be used in Project OneMap.

First, our small 'hello world' application schema and instance document. It models fea-

---

tures 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 4.16, “Halden-by-night vocabulary 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 4.16. Halden-by-night vocabulary mapping**

```

<?xml version="1.0" encoding="UTF-8" ?>
- <bm:MappingDictionary xmlns:xsd="http://www.w3.org/2001/XMLSchema" xml
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:bm="no:h
  stud.hiof.no/~haraldva/schema/MappingInterface.xsd">
+ <documentNamespaces>
- <typeMaps>
  - <TypeMap id="d1e7">
    - <appElement>
      <localname>HaldenByNight</localname>
      <namespace>no:hiof:onemap:gml:appschema:example1</namespace>
    </appElement>
    - <instanceOf>
      <localname>HaldenByNightType</localname>
      <namespace>no:hiof:onemap:gml:appschema:example1</namespace>
    </instanceOf>
    - <gmlDerivedType>
      <localname>AbstractFeatureCollectionType</localname>
      <namespace>http://www.opengis.net/gml</namespace>
    </gmlDerivedType>
    - <substitutesFor>
      <localname>_FeatureCollection</localname>
      <namespace>http://www.opengis.net/gml</namespace>
    </substitutesFor>
  </TypeMap>
+ <TypeMap id="d1e9">
+ <TypeMap id="d1e11">
+ <TypeMap id="d1e13">
+ <TypeMap id="d1e15">
+ <TypeMap id="d1e17">
+ <TypeMap id="d1e19">
+ <TypeMap id="d1e23">
+ <TypeMap id="d1e25">
+ <TypeMap id="d1e78">
+ <TypeMap id="d1e80">
+ <TypeMap id="d1e105">
</typeMaps>
</bm:MappingDictionary>

```

So, let's 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 national 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 striven 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 (see Figure 4.10, “Ordnance Survey MasterMap schema structure (OSMasterMap User Guide)”).

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 guideline, (indirectly) deriving from *AbstractFeatureType* and (indirectly) substituting for *\_Feature*.

#### Example 4.4. Typemap for OS MasterMap type BoundaryLine

```
[...]
<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 in included or imported schemas are invalid, the information from these will of course not be mapped. This means that there will be missing vocabulary type-maps for elements 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. When using SAXON as XSLT engine, an error message will be produced if the document is unavailable, however this will not inflict on the further parsing of the vocabulary.

## Structural analysis

---

A schema analyzer 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):

- The root element must be directly or transitively descended from *gml:AbstractFeatureCollectionType*.
- 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 nesting 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 analyzer 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.

## 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 foolproof method. There might occur situations where there is a question whether an element is one of two possible, or maybe there

---

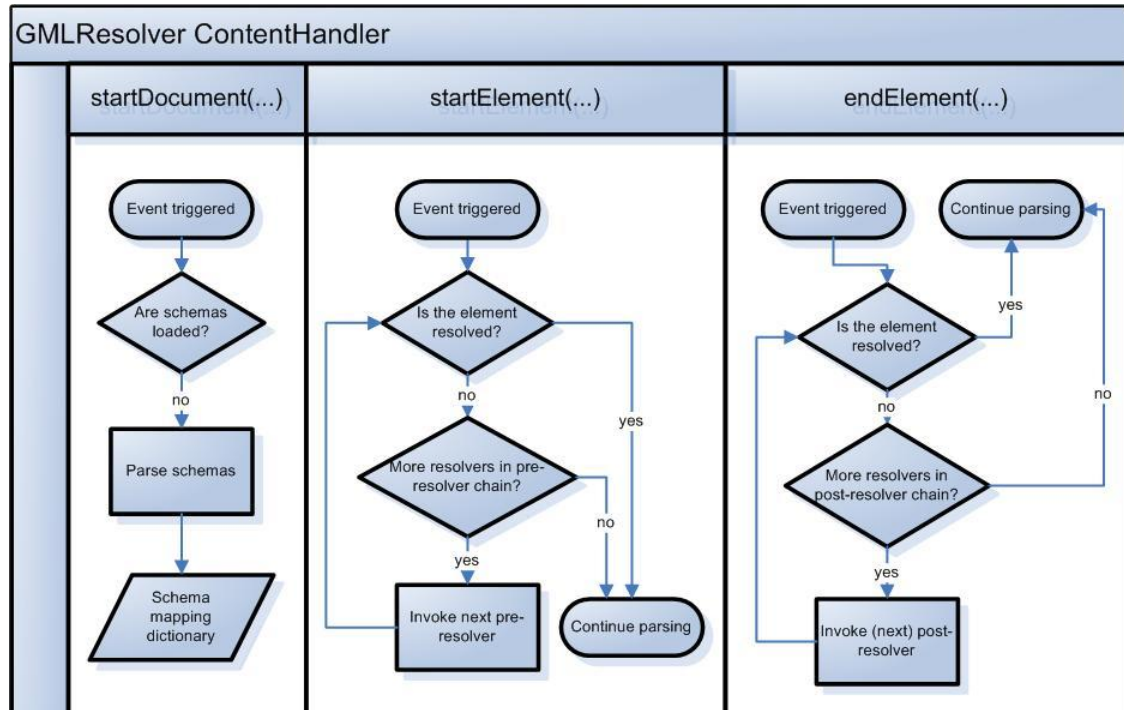
aren't any presented options. This is where the software surrenders, and we should present to the user the unidentified elements, trusting that he will fill out the missing pieces.

## 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[JAXP]. SAX parsing is event based, meaning that the parser generates events 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 Figure 4.17, “ContentHandler methods”). The outlining of structure and partial implementation of this framework was done by Gunnar Misund, as an example of how a dictionary type resolver can be used in a broader context when analysis GML document content.

**Figure 4.17. ContentHandler methods**

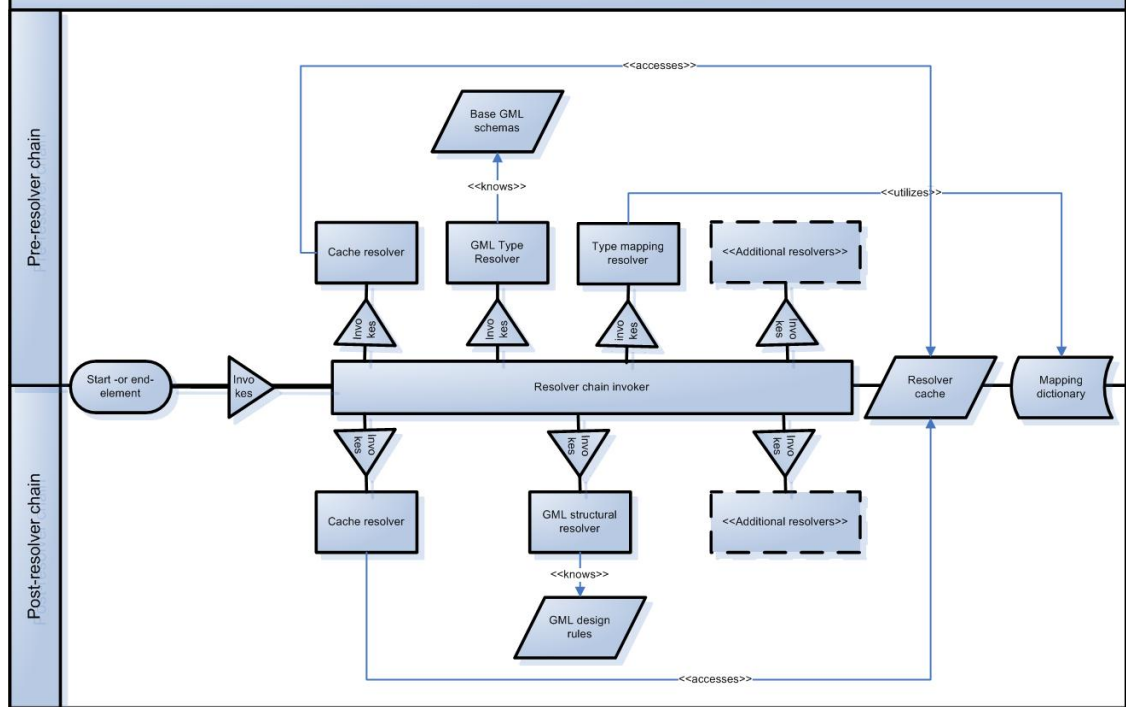




In addition to the element being added to the internal tree-representation, the resolvers are invoked here. A chain of pre-resolvers attempts to identify the element in the *startElement* method, while all non-resolved elements are attempted resolved by a chain of post-resolvers when reaching the *endElement* method (see Figure 4.18, “Resolver chains”). 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 4.18. Resolver chains**

# GMLResolver resolver chains



---

# Chapter 5. Schema parser and GML viewer

The primary objective of my work is develop a framework to use for the handling of arbitrary GML. The result will be a foundation to use in different software, working with GML in some way. This chapter presents a solution where GML vocabularies are transformed into dictionaries, where type information for each element is stored in an easy accessible XML structure. These dictionaries does not contain information not found in the schemas, but considering that type information for XML vocabularies can be scattered across an almost unlimited amount of files, these dictionaries are very helpful. You will also be presented to a proof-of-concept GML viewer; a GML to SVG stylesheet able to present arbitrary GML. Through the generated SVG and the connected scripts, the features can be viewed and both type and element information can be accessed through their geometry.

## Parsing schemas

I chose to use XSLT to parse the schemas, making a mapping dictionary in XML format. As mentioned, this is the most lightweight method, still very powerful when dealing with transformation of XML. First it was important to recognize what kind of information which should be taken from the schemas into a mapping file. It is possible to build entire derivation hierarchies, identifying all super-types of a type. However, the most crucial information, is if a vocabulary's type is deriving from a base GML type, and if so, what type is it deriving. I do acknowledge that more detailed information can be important for some applications, but to visualize the ideas presented in former sections of this document, I consider the GML base types, and possible the substitution groups as the most important type information found in GML application schemas.

## Mapping dictionary schema

It is important to keep in mind why we want to parse the GML schemas, namely to find out how the elements are related to the base GML types. The mapping file will include the following information:

- The target namespace, in other words the vocabulary being described.
- Related namespaces, being those present in the applications schemas.
- Type mappings, one for each element declared in the application schemas, locally and globally.

Each type map will contain:

- The element's name and namespace (grouped using a `complexType`)
- If the element's type is GML derived, either a *gmlType*- or *gmlBaseType*-element, depending on the element type being a direct instantiation of a GML type or just an instantiation of a derived GML type.
- If the element substitutes for a 'proprietary' element, a *substitutesFor*-element, naming the *substitutionGroup*-value.
- If the element indirectly substitutes for a GML-element, a *baseSubstitutionGroup*-element, with the name of the GML type as value.

These data will hopefully provide a sufficient amount of information to an application or other stylesheet, utilizing arbitrary GML. Examples from a mapping file will be presented together with the schema transformation. The schema defining the mapping dictionary is found in the section called “GML Schema to Mapping Dictionary”.

## Parsing GML application schemas

If we want to map elements in an instance document, we have to look at the *element-declaration*-tags in an XML schema. These can either be *globally* defined, or declared *inline* a global type or element. Figure 5.1, “How to traverse schemas” shows a some constructs found in the HaldenByNight-schema, which will hopefully shed some light on how element-types can be traced.

**Figure 5.1. How to traverse schemas**

---

```

[...]
<xs:element name="NightSiteBar" type="NightSiteBarType" substitutionGroup="_NightSiteFeature"/> (a)
<xs:element name="buildingOutline" type="LinearRingPropertyType" substitutionGroup="gml:_geometryProperty"/> (b)

<xs:complexType name="NightSiteBarType"> (c)
  <xs:complexContent>
    <xs:extension base="NightSiteType">
      <xs:sequence>
        <xs:element name="age_limit" type="xs:nonNegativeInteger"/>
        <xs:element name="beer_price">
          <xs:simpleType>
            <xs:restriction base="xs:double">
              <xs:minExclusive value="0"/>
              <xs:maxInclusive value="100"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="NightSiteType" abstract="true"> (d)
  <xs:complexContent>
    <xs:extension base="gml:AbstractFeatureType">
      <xs:sequence>
        <xs:element ref="buildingOutline"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="LinearRingPropertyType"> (e)
  <xs:annotation>
    <xs:documentation>
      Encapsulates a LinearRing, to be used as a geometric property
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:restriction base="gml:GeometryAssociationType">
      <xs:sequence minOccurs="0">
        <xs:element ref="gml:LinearRing"/>
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
[...]
```

- a. This element is an instantiation of *NightSiteBarType*(c). Elements can occur in instance documents of the schema, and the type therefore has to be mapped.
  - b. This elements represents a property, a role to be used between a *feature* and *geometry*. The *LinearRingPropertyType*, which this element is an instantiation of, is not a GML type, and we will therefore track this type too.
  - c. This is a type-declaration, and if no element is directly or indirectly instantiation from this type, it will not be mapped. However, the *NightSiteBar* is, and therefore
-

a trace will show that this type is derived from the *NightSiteType* (d); further tracing is required. This type does however have to inline elements, one an instantiation of the base XML schema data type *nonNegativeInteger*, and one a restriction of another schema-type, *double*. Further tracing will not be necessary for these types, now knowing that they do not descend from the GML schemas.

- d. This is the *NightSiteType*, from which, among others, the *NightSiteBarType*(c) derives. When tracing the *NightSiteBars* type, we will eventually find this type. This is also where the tracing is completed, and we can come to the conclusion that *NightSiteBar* is derived from a base GML type, namely the *AbstractFeatureType*.
- e. The type-definition of the *LinearRingPropertyType*, deriving from *gml:GeometryAssociationType*. Tracing elements to this type clarifies that the property is a geometric one, not what kind of role the geometry has to the feature.

**Figure 5.2. Type-mapping of the NightSiteBar-element**

```
[...]
<TypeMap>
  <appElement>
    <localname>NightSiteBar</localname>
    <namespace>no:hiof:onemap:gml:appschema:example1</namespace>
  </appElement>
  <gmlDerivedType derivedBy="extension">
    <localname>AbstractFeatureType</localname>
    <namespace>http://www.opengis.net/gml</namespace>
  </gmlDerivedType>
  <substitutesFor>
    <localname>_NightSiteFeature</localname>
    <namespace>no:hiof:onemap:gml:appschema:example1</namespace>
  </substitutesFor>
  <baseSubstitutionGroup>
    <localname>_Feature</localname>
    <namespace>http://www.opengis.net/gml</namespace>
  </baseSubstitutionGroup>
</TypeMap>
[...]
```

- a. The *appElement* contains the name and namespace of the element being mapped.
- b. This element is derived from a GML type; the top GML element is stated with

name and namespace. The attribute *derivedBy*, states whether the direct derivation from the GML type is by restriction or extension, not necessarily whether the element derives its parent by restriction or extension.

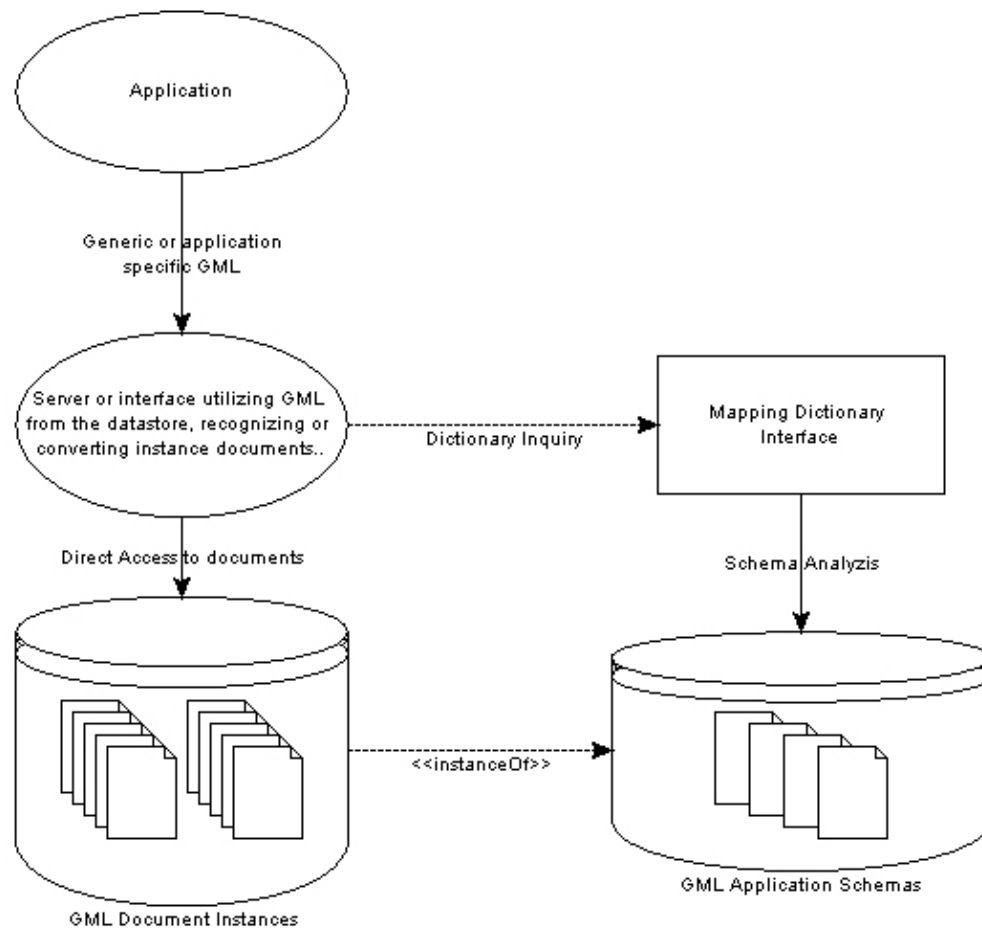
- c. If the element declaration has a *substitutionGroup*-attribute, the element name and namespace for which it substitutes is stated here.
- d. If the element substitutes directly or indirectly for a base GML type, the element name and namespace is contained in the *baseSubstitutionGroup*-element.

## Generic GML Visualization

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 access to the non-geometric properties of the features. Transformations 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 Figure 5.3, “Utilizing dictionary to parse arbitrary GML”). It is however required that the cascading analysis 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 5.3. 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[GED], 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 possibilities. 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 utilizing a specifically created GML format.

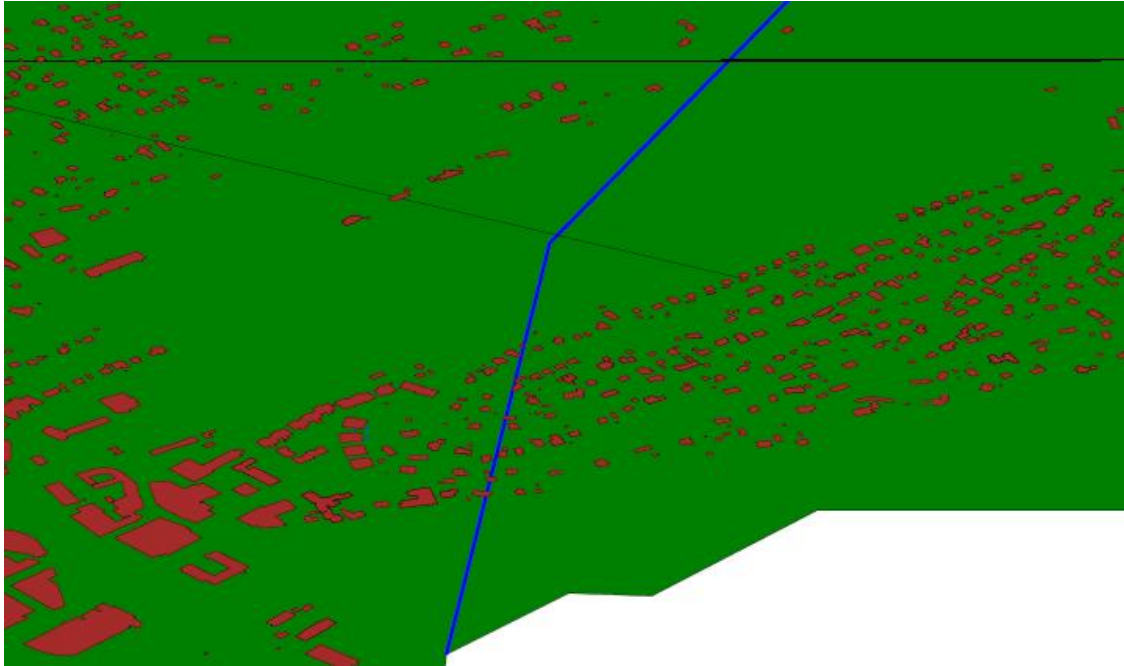
The integration example from the preceding section has integrated features from *Ordnance Survey*, *GML2 spec example*, *Norkart* and *OneMap*. Applying the SVG transformation on these data results in a map containing all integrated feature ( see Fig-

---



ure 5.4, “Integrated GML transformed to SVG”).

**Figure 5.4. Integrated GML transformed to SVG**



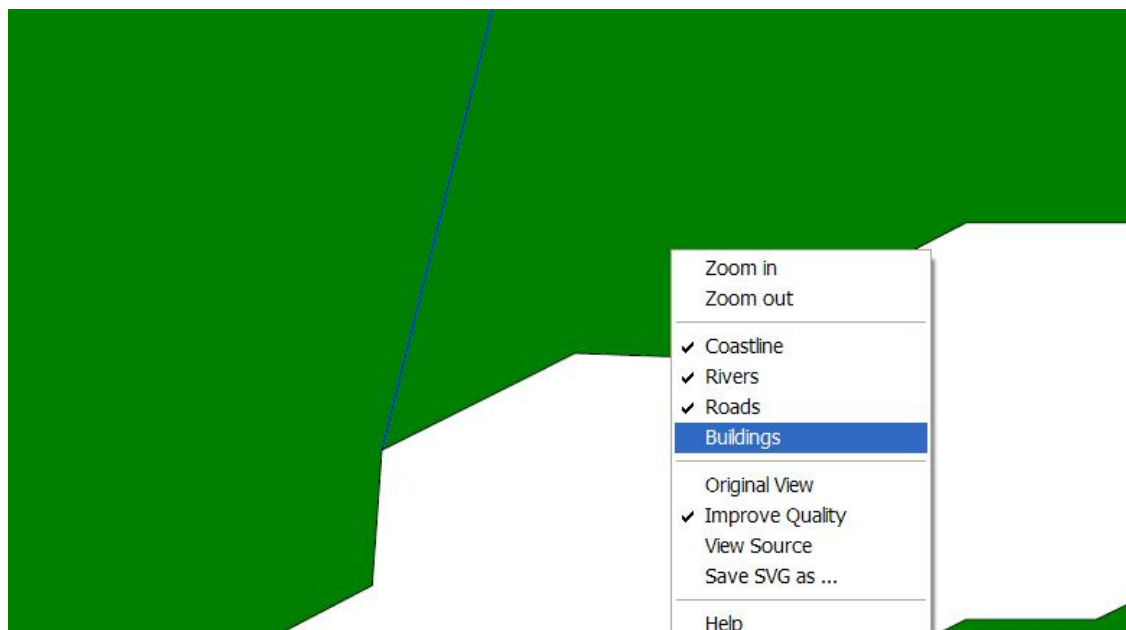
The styling is as simple as possible, allowing users to specify custom styles for each namespace present. This file is the specified when converting. All namespaces, that has not been applied a user style, will get a default style. The style to specify is identical to the value of the SVG *style attribute*, 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, and then the filename is passed to the transformation as a command line argument.

### Example 5.1. Simple feature styling

```
<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 provided a mapping file is available, the integration namespace has introduced an attribute that can be used on a feature collections representing a feature layer, e.g. *roads* or *rivers* (Figure 5.5, “SVG integrated layer visibility”).

**Figure 5.5. 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, together with the type information can be accessed (Figure 5.6, “Feature information window”). 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 5.6. 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 5.7, “Ordnance survey data with default styling”.

**Figure 5.7. Ordnance survey data with default styling**



**Feature Information - Microsoft Internet Explorer**

Feature Information	
Name:	AdministratiefGebied
Namespace:	http://www.gdmc.nl/tdn
Data type information	
Instance of:	Name: AdministratiefGebiedType
	Namespace: http://www.gdmc.nl/tdn
Properties	
Name:	top10_id
Namespace:	http://www.gdmc.nl/tdn
Value:	8400004
Instance of:	Name: integer
	Namespace: http://www.w3.org/2001/XMLSchema
Name:	object_begindatum
Namespace:	http://www.gdmc.nl/tdn
Value:	2001-12-17T13:24:10+02:00
Instance of:	Name: dateTime
	Namespace: http://www.w3.org/2001/XMLSchema

---

# Chapter 6. Conclusions and further work

GML type dictionaries can, as shown Chapter 5, *Schema parser and GML viewer*, be helpful when working with arbitrary GML. They encapsulate important element and type information that can be used to threat documents in a generic way. Using XSLT as extensive as I did when developing the solutions presented in this thesis, does of course have both pros and cons. Many tasks can be cumbersome to do using XSLT, and performance is a very critical issue when choosing a strategy. In this chapter I will conclude my work and discuss some of the implementation choices that were of importance for the results achieved. Finally, I will try to sum up to what extent I feel I succeeded and, equally important, where I regard my solutions as unfinished or inadequate.

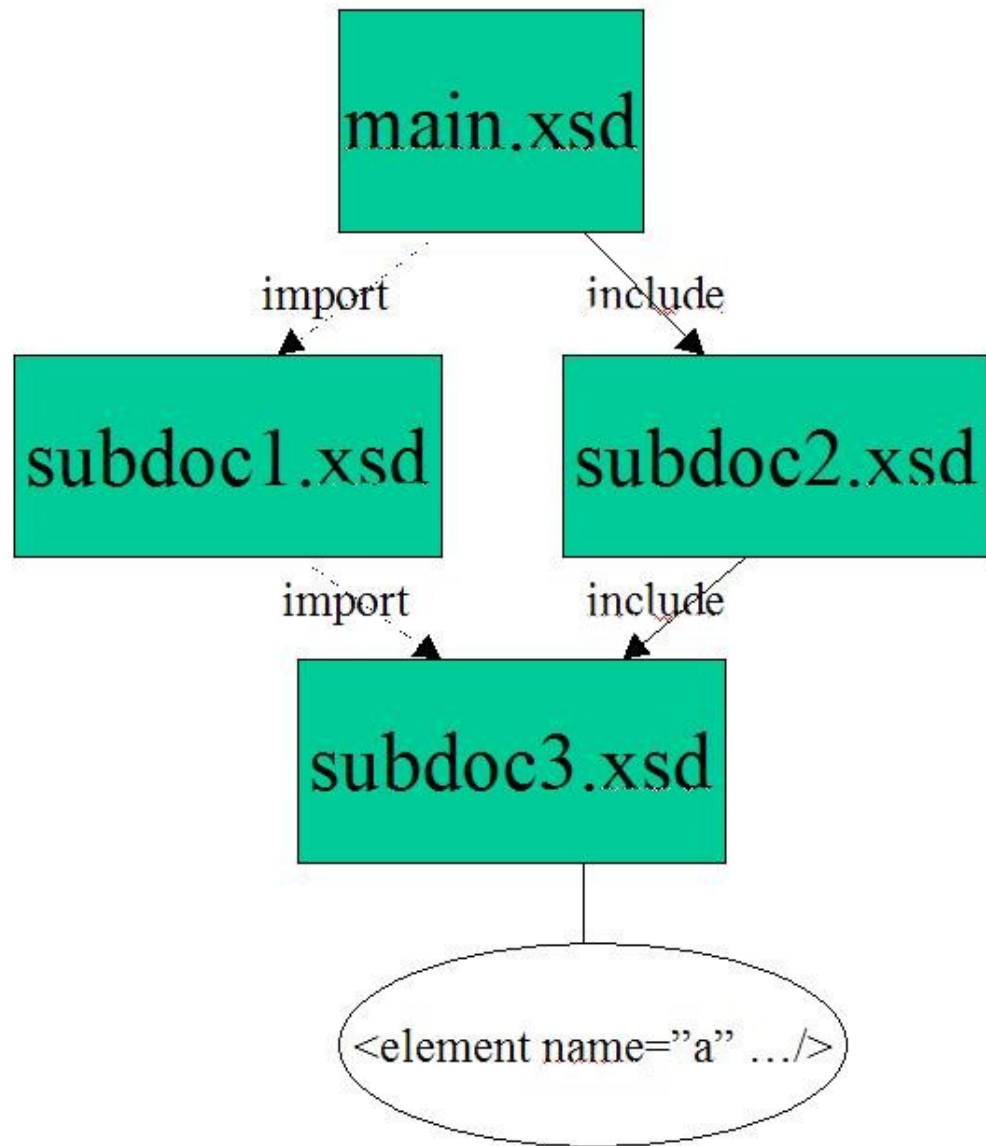
## Type dictionary

The stylesheets for generating dictionaries from GML applications schemas was successfully finished and tested on a range of GML vocabularies. Entire vocabularies were mapped tracing import and include elements inside the schemas. The nature of XSLT sometimes makes it difficult to do trivial tasks, like error handling, navigating documents and in particular debugging. Only in the latter stages of development, did I try to make use of some of the XSLT 2.0 functionality, because the implementation and documentation of this standard still is on an early stage, and I did not want to get into undocumented bugs, considering that some of the stylesheets grew quite heavy. Only when I got to the stage of tidying my stylesheets, did I use some XSLT 2.0 functionality like functions. This was only for the purpose of readability, not functionality. There are no fancy environments programming XSLT and XPath. There is XSLT 1.0 debugging and XPath visualizing capabilities in XmlSpy and some other tools, but they are not always able to cope with complex documents and their value is therefore limited. I stuck to mostly using XmlSpy during development, because of the XSLT debugger, but in the end when I introduced XSLT 2.0 in my stylesheets it was merely the coloration of XML elements that was of any assistance. This situation can get quite frustrating, and small problems tend to take a lot of time searching through Internet and newsgroups for answers, to trivial problems because poor quality of documentation. Therefore I would like to mention the possibility that there are flaws in the code,

even though this shouldn't really be needed to point out when we are talking about software.

I did not get the chance to test the stylesheets on big schema hierarchies, meaning in the range of from about hundred to several thousand files. Most applications and users would probably never require mapping of such structures, but nevertheless, I take it that transforming would be quite resource extensive and slow. The reason is simply that the traversing of documents and searching for elements and data types, are the most exhaustive operation done in the stylesheets. The search is done recursively depth-first, and because of the nature of not being able to have global parameters in XSLT, all paths to all linked documents are traversed before the search ends. If the design of the vocabulary is done in a way, where the same file hierarchy is included in several related files, this hierarchy and its sub-hierarchies could all be traversed a number of times even after the data type being searched for is found. Figure 6.1, “Schema hierarchy search problem” tries to explain this problem in more detail. These problems would not even be an issue using e.g. Java, as I planned from the beginning. Nevertheless, XSLT is very powerful mechanisms when converting from one XML format to another.

### **Figure 6.1. Schema hierarchy search problem**



When parsing the file *main.xsd*, the stylesheet might parse an element substituting for the element *a* defined in the file *subdoc3.xsd*. This element will thus be searched for, so that the stylesheet can search for the origin of this element. The search will be done by following the import and include elements in the schema file. By traversing to *subdoc3.xsd* following two import-statements, the element is identified, and this element's type-information is mapped before the recursive search nest back to the *main.xsd*. Now, there is no information available for the parser to know that this element is actually already mapped. It will therefore also follow the include-statement through *sub-*

---

*doc2.xsd* to *subdoc3.xsd*, and record the information one additional time. Because of this problem the output dictionary get repetitive occurrences of type mappings, and we get a performance problem. Interrupting this is however difficult, and the searching algorithm may possibly be rewritten to avoid this issue.

The mapping stylesheets was implemented for use with GML2, but will work just as well for GML3 schemas. The reason is that they simply identify types and elements based on their namespace. Applying the mapping stylesheet on GML3 schemas, should therefore be just as successful as on GML2 schemas. The dictionary creation was tested on the *U.S. Census Bureau's* TIGER/GML schemas[TIG], which are based on GML3. The viewer however is dependent on recognition of base spatial and non-spatial types, and will require some programming to adapt to GML3 documents.

## GML Viewer

The generic GML viewer presented in the section called “Generic GML Visualization” was implemented in SVG, using ECMAScript for browsing functionality. As proof of concept this implementation captured the essence of generic GML handling. However, each document has to be transformed manually from GML to SVG, thus not making the solution very scalable. If intended for use when visualizing several GML sources, it will be necessary to implement some functionality e.g. through servlets or CGI for loading and managing of GML resources. If blessed with more development hours it would have been interesting to build the generic GML viewer as a client of WFS servers. WFSs normally serve their data as GML, and do also have functionality to get GML schema descriptions of all the individual feature types the server contains. The transformation is successfully tested on a number of different GML sources with success. It could be a very helpful tool for easy, graphical access to GML data, for development purposes.

A simple styling mechanism (illustrated in Example 5.1, “Simple feature styling”) was just partially implemented, and is used to define styling for features from different namespaces. When we integrate several features from the same namespace, we end up with only one type of style for all features, even though it is probable that they should be styled differently. Sadly I did not find time to implement any smart styling for the generic GML to SVG transformation. The *Styled Layer Descriptor* (SLD)[SLD] specification is particularly interesting when it comes to styling individual features. SLD is used to individually style layers retrieved from WMS[WMS] servers, posting an XML document to the server, including among other variables the styles which certain



features should be displayed with. Typically, not all layers or features are stylable, but e.g. a layer *buildings* may be stylable, making it possible to provide certain rules for certain buildings. The styles can be applied to all buildings in one layer or some buildings fitting a given profile; a *filter*. A somewhat similar method could be used for styling GML features in SVG documents. A feature type or super type of several features could be addressed in an SLD-like document, and all types or sub-types could then be styled with the style given. This document could e.g. provide certain styles depending on the generic GML type of the object, like polygon or point. Apart from a somewhat imperfect styling configuration, the generic GML viewer stylesheets are very illustrative examples of how a type dictionary could be utilized for accessing arbitrary GML, and does provide instant access to all non-spatial properties.

The GML to SVG transformation was done using XSLT. The XML dictionary is accessed directly and small modifications on the structure of the dictionary would require reconstruction of the GML to SVG transformation stylesheet. If I could start from scratch, I might have chosen to implement a framework for the dictionary in a higher level language, using SAX to load the structure into a framework that would enable easier access to data analysis and higher level access to them. This would allow creation of an interface specification, where changes to the dictionary structure could be transparent to utilizing software. Considering that a fully functional GML viewer needs an operating environment, e.g. through a web application or an ordinary application, such an implementation could prove useful. Having loaded the dictionary into a more programmatically accessible data structure, there is a shorter way to providing an environment for generic GML handling, through different applications both for analysis, transformation and styling and viewing of GML or possibly also altering and analysis of the dictionaries.

---

# Bibliography

[ADO] *Adobe Systems Incorporated* [<http://www.adobe.com/>] .

[AMG]                      OGC                      Abstract                      Specifications  
[<http://www.opengeospatial.org/specs/?page=abstract/>]

[BAT] *Batik, Apache XML project* [<http://xml.apache.org/batik/>] .

[CLEO] *Cleopatra, Publishing GML data as interactive SVG maps*  
[<http://www.svgopen.org/2003/papers/cleopatra/>] . Alison meynert.

[CODE] *Schema transformation and generic GML2SVG stylesheets*  
[<http://www.onemap.org/harald/bundle.zip>] .

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

[DOM] *Document Object Model* [<http://www.w3.org/DOM/>] .

[DTD] *Document Type Definitiion* [<http://www.w3.org/TR/REC-xml/#dt-doctype>]  
(described in *XML 1.0 Recommendation*).

[ECMA]                      *ECMAScript*                      *Language*                      *Specification*  
[<http://www.ecma-international.org/publications/standards/Ecma-262.htm>] .

[EDI]                      *Distributed GML Management with SVG Tools*  
[<http://www.svgopen.org/2003/papers/DistributedGmlManagementWithSVG/>] .  
Gunnar Misund, Henning Kristiansen, and Mats Lindh.

[FED] *Federated database systems for managing distributed, heterogenous, and auto-  
nomius databases*. . A. Sheth and J. Larson. *ACM Computing Surveys*, 22 (3),  
1990..

[FO] *XML Path Language (XPath) Version 1.0* [<http://www.w3.org/TR/xpath/>] . W3C  
Recommondation 16 November 1999.

[G4J] *GML4J, SourceForge project page* [<https://sourceforge.net/projects/gml4j/>] .  
W3C Recommendation 16 November 1999.

- [GED] *Distributed GML Management with SVG Tools*. Gunnar Misund, Henning Kristiansen, and Mats Lindh.
- [GEX] *GML Experiences from the Field*  
[<http://www.safe.com/solutions/whitepapers/pdfs/GML%20-%20Experiences%20From%20the%20Field1.pdf>] . Don Murray and Juan Chu Chow.
- [GML20] *Geography Markup Language (GML) 2.0 Implementation Specification*  
[<http://www.opengeospatial.org/docs/01.029.pdf>] .
- [GML30] *OpenGIS Geography Markup Language (GML) Implementation Specification*  
[<http://www.opengeospatial.org/docs/02-023r4.pdf>] .
- [GPR] *GML Profiling: Why It's Important for Interoperability*  
[<http://www.esri.com/news/arcuser/0403/special-section/gml-profiling.pdf>] .  
ArcUser April-June 2003 ([www.esri.com](http://www.esri.com)).
- [GTP] *GeoTools project* [<http://geotools.org>] .
- [GFQ] *Geotools project FAQ* [<http://www.geotools.org/FAQ>] .
- [JAXP] *Java API for XML Processing (JAXP)* [<http://java.sun.com/xml/jaxp/>] .
- [JUMP] *Unified Mapping Platform (JUMP)* [<http://www.jump-project.org>], Vivid Solutions .
- [JTEC] *JUMP Unified Mapping Platform, Technical Report*  
[[http://www.jump-project.org/inc/JUMP/assets/JUMP\\_Technical\\_Report.pdf](http://www.jump-project.org/inc/JUMP/assets/JUMP_Technical_Report.pdf)] .  
Martin Davis.
- [JTS] *JTS Topology Suite* [<http://www.vividsolutions.com/jts/JTSHome.htm>], Vivid Solutions .
- [MMUG] *Ordnance Survey MasterMap User Guide, part 2*  
[<http://www.ordnancesurvey.co.uk/oswebsite/products/osmastermap/guides/useguide.html>] .
- [NODE] *Document Object Model Core, Interface Node*  
[<http://www.w3.org/TR/DOM-Level-2-Core/core.html#ID-1950641247>] .
- [P1M] *Project OneMap homepage* [<http://www.onemap.org>], Østfold University Col-

---

*lege* .

[PSVI] XML Schema, Post-schema-validation info set  
[[http://www.w3.org/TR/xmlschema-1/#PSVI\\_contributions](http://www.w3.org/TR/xmlschema-1/#PSVI_contributions)] .

[SAX] Simple API for XML [<http://www.saxproject.org>] .

[SFS] Simple Features Specification for SQL  
[<http://www.opengis.org/docs/99-049.pdf>], Open GIS Consortium, Inc..

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

[OGC] Open Geospatial Consortium [<http://www.opengis.org>] .

[OME] OneMap SVG Map Editor [<http://globus.hiof.no/editor/editor.html>] . GML Days 2003.

[ONE] The One Map Project [[http://www.ia.hiof.no/~gunnarmi/omd/gmldev\\_02](http://www.ia.hiof.no/~gunnarmi/omd/gmldev_02)] . Gunnar Misund and Knut-Erik Johnsen.

[OS] OS MasterMap  
[<http://www.ordnancesurvey.co.uk/oswebsite/products/osmastermap/>], Ordnance Survey.

[OSUG] OS MasterMap User Guide, part 2  
[<http://www.ordnancesurvey.co.uk/products/osmastermap/pdf/userguidepart2.pdf>] .

[RDS] XML Schema: Reconciling Diversity with Standardisation  
[<http://www.snowflakesoft.co.uk/news/papers/xmlSchema.pdf>] . Eddie Curtis.

[SAPI] XML Schema API, W3C Member Submission  
[<http://www.w3.org/Submission/xmlschema-api/>] .

[SFL] Snowflake Software homepage [<http://www.snowflakesoft.co.uk/>] . Alison meynert.

[SLD] Styled Layer Descriptor Implementation Specification  
[<http://www.geoconnections.org/architecture/technical/specifications/sld/styled>]

---

*\_layer\_descriptor\_1\_0.pdf]* .

[SMIL] *Synchronized Multimedia Integration Language (SMIL 2.0)*, W3C Recommendation [<http://www.w3.org/TR/smil20/>] .

[SPY] *Altova XML Spy* [<http://www.xmlspy.com>], XML application .

[SRCF] *Sourceforge, Open Source development website* [<http://sourceforge.net>] .

[SVG] *Scalable Vector Graphics (SVG) 1.1 Specification* [<http://www.w3.org/TR/SVG/>] . W3C Recommendation 14 January 2003.

[SX] *SAXON XSLT and XQuery Processor* [<http://saxon.sourceforge.net/>] . Michael H. Kay.

[TIG] *U.S. Census Bureau, TIGER/GML Schemas* [<http://aries.geo.census.gov/WebTIGER/CensusTIGERGMLSchemas.html>] .

[VIV] *Vivid Solutions Inc.* [<http://www.vividsolutions.com>] .

[WFS] *Web Feature Service (WFS) Implementation Specification* [[http://www.geoconnections.org/architecture/technical/specifications/filter\\_encoding/filter\\_encoding\\_1\\_0.pdf](http://www.geoconnections.org/architecture/technical/specifications/filter_encoding/filter_encoding_1_0.pdf)] .

[WMS] *Web Map Service Implementation Specification 1.1.1* [<http://www.opengis.org/techno/specs/01-068r3.pdf>] .

[XHTML] *XHTML 1.0 The Extensible HyperText Markup Language (Second Edition)* [<http://www.w3.org/TR/xhtml1/>] .

[XIS] *Document Object Model (DOM) Level 3 Core Specification, Appendix C: Infoset mapping* [<http://www.w3.org/TR/2003/WD-DOM-Level-3-Core-20030609/infoset-mapping.html>] .

[XML] *Extensible Markup Language (XML) 1.0 (Third Edition)* [<http://www.w3.org/TR/REC-xml/>] .

[XMLNS] *Namespaces in XML, W3 document* [<http://www.w3.org/TR/REC-xml-names/>] .

[XP] *XML Path Language (XPath) Version 1.0* [<http://www.w3.org/TR/xpath/>] . W3C

Recommondation 16 November 1999.

[XP2] *XML Path Language (XPath) Version 2.0* [<http://www.w3.org/TR/xpath20/>] .  
W3C Recommendation 16 November 1999.

[XSC] *XML Schema Part 0: Primer, W3C Proposed Recommendation* , 30 March 2001 [<http://www.w3.org/TR/2001/PR-xmlschema-0-20010330/>] .

[XSL] *Extensible Stylesheet Language (XSL) Version 1.0* [<http://www.w3.org/TR/xsl/>] .  
W3C Recommendation 15 October 2001.

[XSLT1] *XSL Transformations (XSLT) Version 1.0* [<http://www.w3.org/TR/xslt>] .

[XSLT20] *XSL Transformations (XSLT) Version 2.0, W3C Working Draft* [<http://www.w3.org/TR/xslt20/>] .

---

# Appendix A. XSL Transformations

## GML Schema to Mapping Dictionary

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
This stylesheet is written for version 2.0 of xslt. At this time only
experimentally supported by Saxon.
-->
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsdold="http://www.w3.org/2000/10/XMLSchema"
  xmlns:app="no:hiof:osgb:appschema"
  xmlns="no:hiof:basemapper"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:bm="no:hiof:basemapper"
  xmlns:map="no:hiof:basemapper:functions"
  xsi:schemaLocation="no:hiof:basemapper
http://www.ia-stud.hiof.no/~haraldva/schema/MappingInterface.xsd">
  <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>

  <!--
  There are four parameteres that can be passed from console to the stylesheet.
  Parameters:
    $basefile - filename of a schema being analyzed. This parameter is only
    accessed when the transformation is done directly on a schema.
    It is recommended if there are circular includes or imports in the
    schema, because it is used to prevent just this.
    $baseloc - A url defining the directory where the schemas are located. This
    is used by the transformation when schemas are pointed to
    with relative paths. Use full url's, including preceding slash.
    E.g. file:///c:/schemas/myvocabulary or
    http://www.mysite.org/schemas/
    $instance - If specified with value 'yes', the file sent to transformations
    should be an instance file of a vocabulary. The stylesheet will then access
    the schemaLocation-attribute of the file, and parse the schemas
    pointed to there. If the location is relative to the instance
    file, the
    $baseloc-argument must be specified. If the instance file is
    large, it is recommended to instead pass the $schemaLocation-
    argument
    to the stylesheet.
    $schemalocation - This argument overrides the $basefile and $instance-
    arguments, traversing the schemas specified in the string. The syntax is
    thee
    same as with an ordinary xsi:schemaLocation-attribute. The
    $baseloc should be specified if the url's to the schemas
    to the current directory.
    .
  -->
  <xsl:param name="basefile"/>
  <xsl:param name="baseloc" select="''"/>
  <xsl:param name="instance" select="'no'"/>
  <xsl:param name="schemalocation" select="''"/> <!-- this param overrides the
  schemaLocation-attribute found in an instance-file. -->
  <!--
  This variable is meant to be used throughout the stylesheet, where there is
  output meant for debugging.
  -->
  <xsl:variable name="debug" select="boolean('')"/>

  <!--
  A few "constants".
  -->
  <xsl:variable name="gml_full" select="'http://www.opengis.net/gml'"/>
  <xsl:variable name="xlink_full" select="'http://www.w3.org/1999/xlink'"/>
  <xsl:variable name="xml_full" select="'http://www.w3.org/XML/1998/namespace
  '"/>
```

```
<!--
The template to start the transformation.
-->
<xsl:template match="/">
  <xsl:if test="$debug">
    <xsl:message terminate="no"><xsl:value-of select="$basefile"/>
  </xsl:if>

  <!--
  This element is the root of the mapping dictionary.
  -->
  <bm:MappingDictionary>
    <xsl:attribute name="schemaLocation" namespace="
      http://www.w3.org/2001/XMLSchema-instance">
      <xsl:value-of select="'no:hiof:basemapper
        http://www.ia-stud.hiof.no/~haraldva/schema/MappingInterface.xsd'"/>
    </xsl:attribute>

    <!--
    Here we map a list of all the namespaces declared in the first file. This
    list isn't necessarily complete, if other
    imported or included files declare other namespaces.
    -->
    <xsl:element name="documentNamespaces">
      <xsl:choose>
        <!--
        If the global param schemalocation has been passed to the stylesheet,
        the files pointed to by this string will
        be parsed
        -->
        <xsl:when test="$schemalocation != ''">
          <xsl:call-template name="TraverseSchemaLocations">
            <xsl:with-param name="baselocation" select="$baseloc"/>
            <xsl:with-param name="task" select="'namespaces'"/>
            <xsl:with-param name="schemaLocString" select="$schemalocation"/>
          </xsl:call-template>
        </xsl:when>
        <!--
        If the instance-param is set to 'yes', the file being transformed by
        the stylesheet is an instance file. Therefore
        the schemaLocation-attribute of the file is acquired, then passed to
        the TraverseSchemaLocations-template.
        -->
        <xsl:when test="$instance = 'yes'">
          <xsl:variable name="schemas" select="normalize-space(child::*[1]
            /@xsi:schemaLocation)"/>
          <xsl:choose>
            <xsl:when test="$schemas">
              <xsl:call-template name="TraverseSchemaLocations">
                <xsl:with-param name="baselocation" select="$baseloc"/>
                <xsl:with-param name="task" select="'namespaces'"/>
                <xsl:with-param name="schemaLocString" select="$schemas"/>
              </xsl:call-template>
            </xsl:when>
            <xsl:otherwise>
              <xsl:message terminate="yes">
                The parameter 'instance' was passed to this stylesheet with
                value 'yes', but there is no xsi:schemaLocation-attribute
                specified in the provided instance file.
                The transformation requires this to be able to find the
                schemas related with the vocabulary.
              </xsl:message>
            </xsl:otherwise>
          </xsl:choose>
        </xsl:when>
        <xsl:otherwise>
          <!--
          If the instance-parameter is not passed to the stylesheet, the file
          being transformed is actually a schema.
          The WriteNamespaces-template outputs the namespaces defined in the
          schema to the mapping dictionary.
          -->
          <xsl:call-template name="WriteNamespaces">
            <xsl:with-param name="root" select="current()"/>
          </xsl:call-template>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:element>
  </bm:MappingDictionary>
</xsl:template>
```



---

```

</xsl:element>

<!--
The typeMaps-elements, is the root for all the typemaps in the mapping.
The template iterateSchema takes care
of iterating one physical file.
-->
<typeMaps>
  <xsl:choose>
    <!--
    If the schemalocation-param is passed to the stylesheet, we call the
    TraverseSchemaLocations, just as we did
    when writing out the related namespaces. This time, the task-argument
    is however set to 'typeMaps', meaning it
    will traverse the schemaLocation-string in the same manner, but this
    time do another task with each file.
    -->
    <xsl:when test="$schemalocation != ''">
      <xsl:call-template name="TraverseSchemaLocations">
        <xsl:with-param name="baselocation" select="$baseloc"/>
        <xsl:with-param name="task" select="'typeMaps'"/>
        <xsl:with-param name="schemaLocString" select="$schemalocation"/>
      </xsl:call-template>
    </xsl:when>
    <!--
    We are dealing with an instance file.
    -->
    <xsl:when test="$instance = 'yes'">
      <xsl:variable name="schemas" select="normalize-space(child::*[1]
/@xsi:schemaLocation)"/>

      <xsl:call-template name="TraverseSchemaLocations">
        <xsl:with-param name="baselocation" select="$baseloc"/>
        <xsl:with-param name="schemaLocString" select="$schemas"/>
        <xsl:with-param name="task" select="'typeMaps'"/>
      </xsl:call-template>
    </xsl:when>
    <!--
    The file being transformed is a schema, we can therefore iterate it
    directly without having to traverse a schemaLocation-string.
    -->
    <xsl:otherwise>
      <!--
      first a few values to be used in the mapping. These values will be
      passed about in the stylesheet,
      but may eventually change, when imports and includes are followed
      and mapped.
      -->
      <xsl:variable name="tns_full" select="string(child::*[1]
/@targetNamespace)"/>
      <xsl:variable name="tns_prefix" select="name(child::*[1]
/namespace::*[string(.)=$tns_full])"/>
      <xsl:variable name="gml_prefix" select="name(child::*[1]
/namespace::*[string(.)=$gml_full])"/>

      <xsl:call-template name="IterateSchema">
        <xsl:with-param name="locationRootPath"
select="map:GetFileRoot($basefile)"/>
        <xsl:with-param name="tns_full" select="$tns_full"/>
        <xsl:with-param name="tns_prefix" select="$tns_prefix"/>
        <xsl:with-param name="gml_prefix" select="$gml_prefix"/>
      </xsl:call-template>
    </xsl:otherwise>
  </xsl:choose>
</typeMaps>
</bm:MappingDictionary>
</xsl:template>

<!--
This function takes a full path to a file, and returns the location of the
file, without the filename at the end.
It is dependent upon the function map:lastIndexOf to find the last occurrence
of the folder delimiter.
-->
<xsl:function name="map:GetFileRoot">
  <xsl:param name="fullname"/>

  <xsl:choose>
    <xsl:when test="contains($fullname, '/')">

```

---

```

        <!--<xsl:variable name="lastIndex" select="index-of($fullname, '/')
[last()]" />-->
        <xsl:variable name="lastIndex" select="map:lastIndexOf($fullname, '/',
string-length($fullname))" />
        <!--<xsl:message terminate="no"><xsl:value-of
select="substring($fullname, 1, $lastIndex)" /></xsl:message>-->
        <xsl:value-of select="substring($fullname, 1, $lastIndex )" />
    </xsl:when>
    <xsl:when test="contains($fullname, '\\')">
        <xsl:variable name="lastIndex" select="map:lastIndexOf($fullname, '\\',
string-length($fullname))" />
        <xsl:value-of select="substring($fullname, 1, $lastIndex )" />
    </xsl:when>
    <xsl:otherwise>
        <xsl:value-of select="'" />
    </xsl:otherwise>
</xsl:choose>
</xsl:function>

<!--
This function return the last index of the $char-argument inside the $string
argument. -1 if the $char is not within the $string.
-->
<xsl:function name="map:lastIndexOf">
    <xsl:param name="string" />
    <xsl:param name="char" />
    <xsl:param name="currentIndex" />

    <!--<xsl:message terminate="no"><xsl:value-of select="$char" />, <xsl:value-
of select="$string" />, <xsl:value-of select="$currentIndex" />
</xsl:message>-->

    <xsl:if test="not($string)">
        <xsl:value-of select="number(-1)" />
    </xsl:if>
    <xsl:if test="string-length($char) != 1">
        <xsl:message terminate="yes">Invalid argument passed to
map:lastIndexOf($string, $char, $currentIndex). Argument $char should be
only one character.</xsl:message>
    </xsl:if>
    <xsl:if test="$currentIndex > string-length($string) or $currentIndex
< 1">
        <xsl:message terminate="yes">Invalid argument passed to
map:lastIndexOf($string, $char, $currentIndex). Argument $currentIndex
have value between [1, string-length($string)]</xsl:message>
    </xsl:if>

    <xsl:variable name="lastChar" select="substring($string, $currentIndex,
1)" />
    <xsl:choose>
        <xsl:when test="$lastChar = $char">
            <xsl:value-of select="$currentIndex" />
        </xsl:when>
        <xsl:when test="$currentIndex = 1">
            <xsl:value-of select="number(-1)" />
        </xsl:when>
        <xsl:otherwise>
            <xsl:value-of select="map:lastIndexOf(substring($string, 1,
$currentIndex - 1), $char, $currentIndex - 1)" />
        </xsl:otherwise>
    </xsl:choose>
</xsl:function>

<!--
This function is used to determine the location of a file, f.ex. being
included by a schema. It takes the argument $currentLocation, that holds the
location of the file currently being parsed. E.g. if the file
http://www.mysite.org/schemas/schemal.xsd is being parsed, the
$currentLocation (which is passed
throughout the stylesheet), holds the value http://www.mysite.org/schemas/. If
a new file is linked to by this stylesheet, the link is either absolute or
relative to this location.
This function calls the function IsAbsolutePath, to determine if the $newLink-
param should be concatenated with the $currentLocation, or if the location is
fully
defined in the $newLink-value.
-->
<xsl:function name="map:GetLocation">
    <xsl:param name="currentLocation" />
    <xsl:param name="newLink" />

```

```

<!--<xsl:message terminate="no"><xsl:value-of
select="concat($currentLocation, ' ', $newLink)"/></xsl:message>-->
<xsl:choose>
  <xsl:when test="map:IsAbsolutePath($newLink) = true()">
    <!--<xsl:message terminate="no">Return: <xsl:value-of
      select="$newLink"/></xsl:message>-->
    <xsl:value-of select="$newLink"/>
  </xsl:when>

  <xsl:otherwise>

    <xsl:variable name="returnvalue" select="concat($currentLocation,
      $newLink)"/>

    <!--<xsl:message terminate="no">joined: <xsl:value-of
      select="$returnvalue"/></xsl:message>-->

    <xsl:value-of select="$returnvalue"/>
  </xsl:otherwise>
</xsl:choose>
</xsl:function>

<!--
This takes an argument $url, and tells if this is an absolute path or not
(relative one).
-->
<xsl:function name="map:IsAbsolutePath">
  <xsl:param name="url"/>

  <xsl:choose>
    <xsl:when test="contains($url, 'http://') or contains($url, 'file://') or
      contains($url, ':')"> <!-- the last test might indicate a file-path like
      e.g. c:\ or d:\ etc.-->
      <xsl:value-of select="true()"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="false()"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:function>

<!--
This template provides the functionality to traverse a schemaLocation-string,
either passed as argument to the stylesheet, or pointed to by
an instance document.
-->
<xsl:template name="TraverseSchemaLocations">
  <xsl:param name="baselocation"/>
  <xsl:param name="schemaLocString"/>
  <xsl:param name="last" select="''"/>
  <xsl:param name="task"/>
  <xsl:param name="parsed" select="''"/>

  <!--
  The schemalocation-string holds pair-values, namespace/location. We
  therefore "parse" two and two. The nextToken-function is used to
  acquire the string located after a given one.
  -->
  <xsl:variable name="ns" select="normalize-
    space(map:nextToken($schemaLocString, $last))"/>
  <xsl:variable name="location" select="normalize-
    space(map:nextToken($schemaLocString, $ns))"/>

  <!--
  If we successfully got both the $ns and $location of a namespace, we either
  write the namespace-strings of this vocabulary or write the
  typeMaps, dependent upon the value of the $task-param.
  -->
  <xsl:if test="$ns != '' and $location != ''">

    <!--<xsl:message terminate="no"><xsl:value-of select="$ns"/>
    </xsl:message>-->
    <xsl:choose>
      <xsl:when test="$task = 'namespaces'">
        <!--<xsl:message terminate="no">GetLocation() - <xsl:value-of
          select="map:GetLocation($baselocation, $location)"/></xsl:message>-->

        <xsl:call-template name="WriteNamespaces">

```

---

```

        <xsl:with-param name="root"
        select="document(map:GetLocation($baselocation, $location))"/>
    </xsl:call-template>
</xsl:when>
<xsl:when test="$task = 'typeMaps'">
    <!--<xsl:message terminate="no">TraverseSchemaLocations - task=
    typeMaps - call IterateSchema with nodes of <xsl:value-of
    select="map:GetLocation($baselocation, $location)"/></xsl:message>-->

    <xsl:call-template name="IterateSchema">
        <xsl:with-param name="locationRootPath"
        select="map:GetFileRoot(map:GetLocation($baselocation,
        $location))"/>
        <xsl:with-param name="nodes"
        select="document(map:GetLocation($baselocation, $location))"/>
        <xsl:with-param name="tns_full" select="$ns"/>
        <xsl:with-param name="scanned" select="$parsed"/>
        <xsl:with-param name="current" select="$location"/>
        <!--<xsl:with-param name="tns_prefix" select="$tns_prefix"/>-->
        <!--<xsl:with-param name="gml_prefix" select="$gml_prefix"/>-->
    </xsl:call-template>
    </xsl:when>
</xsl:choose>

    <xsl:call-template name="TraverseSchemaLocations">
        <xsl:with-param name="baselocation" select="$baselocation"/>
        <xsl:with-param name="schemaLocString" select="$schemaLocString"/>
        <xsl:with-param name="last" select="$location"/>
        <xsl:with-param name="task" select="$task"/>
        <xsl:with-param name="parsed" select="concat($parsed, $location)"/>
    </xsl:call-template>
</xsl:if>
</xsl:template>

<!--
This function returns the next token in a string, after a given $lastVal. E.g.
the call map:nextToken('a b c', 'b') returns 'c'

ISSUE: If one namespace or file, is identical to a part of another namespace
or file, the nextToken might return the wrong value, because it
searches for the occurrence of one string within another, not
checking if there are several occurrences.
-->
<xsl:function name="map:nextToken">
    <xsl:param name="fullString"/>
    <xsl:param name="lastVal"/>

    <xsl:variable name="delimiter" select="' '"/>

    <xsl:choose>
        <xsl:when test="$lastVal != ''">
            <xsl:variable name="temp" select="normalize-space(substring-
            after($fullString, $lastVal))"/>
            <xsl:choose>
                <xsl:when test="contains($temp, $delimiter)">
                    <xsl:value-of select="substring-before($temp, $delimiter)"/>
                </xsl:when>
                <xsl:otherwise>
                    <xsl:value-of select="$temp"/>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:when>
        <xsl:otherwise>
            <xsl:value-of select="normalize-space(substring-before($fullString,
            $delimiter))"/>
        </xsl:otherwise>
    </xsl:choose>
</xsl:function>

<!--
This function outputs the namespaces defined in a file, according to the
format specified in the mapping dictionary schema.
-->
<xsl:template name="WriteNamespaces">
    <xsl:param name="root"/>

    <xsl:element name="targetns"><xsl:value-of select="$root/child::*[1]
    /@targetNamespace"/></xsl:element>

```

---

```

        <xsl:for-each select="$root/child::*[1]/namespace::*">
          <xsl:element name="namespace"><xsl:value-of select="."/></xsl:element>
        </xsl:for-each>
      </xsl:template>

```

```

<!--
***** template IterateSchema *****

```

This templates takes care of the mapping of all elements declared as descendants of the \$nodes parameter passed to the template. This is throughout this file the root of a file.

Parameter list:

```

$locationRootPath -> The location of the file being parsed. This string is
used to determine the location of
                    files being pointed to by url's relative to this files
                    location.
$nodes            -> predecessor of element declarations. Usually root of
document.
$scanned          -> a string consistent of a concatenation of files already
traversed by this template. This is used to
                    avoid eternal loops of imports or includes.
$current          -> name of the file currently being analyzed.
$tns_full         -> target namespace of file being analyzed.
$tns_prefix       -> the prefix of the target namespace used in this
particular file.
$gml_prefix       -> the GML prefix used in this file.

```

notes:

```

  problems may arise if:
    -> One file uses more than one prefix for it's target namespace. This
        option is not fully tested.

```

```

-->
<xsl:template name="IterateSchema">
  <xsl:param name="locationRootPath"/>
  <xsl:param name="nodes" select="."/>
  <xsl:param name="scanned" select="''"/>
  <xsl:param name="current" select="$basefile"/>
  <xsl:param name="tns_full" select="string($nodes/child::*[1]
/@targetNamespace)"/>
  <xsl:param name="tns_prefix" select="name($nodes/child::*[1]
/namespace::*[string(.)=$tns_full])"/>
  <xsl:param name="gml_prefix" select="map:full2prefixFunc($gml_full,
$nodes)"/>

  <xsl:message terminate="no">Scanning schema: <xsl:value-of select="concat(
$locationRootPath, $current)"/></xsl:message>
  <!--<xsl:message terminate="no">TNS: <xsl:value-of select="$tns_full"/>
Prefix: <xsl:value-of select="$tns_prefix"/></xsl:message>-->
  <!--
  If the current isn't contained in the scanned-parameter, this file isn't
  already analyzed.
  -->
  <xsl:if test="not(contains($scanned,$current))">
    <!--
    For each element declaration in this file, we do a typemap, calling the
    template BuildTypeMaps.
    -->
    <xsl:for-each select="$nodes//(xsd:element | xsdold:element)">
      <xsl:call-template name="BuildTypeMaps">
        <xsl:with-param name="locationRootPath" select="$locationRootPath"/>
        <xsl:with-param name="root" select="$nodes"/>
        <xsl:with-param name="element" select="."/>
        <xsl:with-param name="current" select="$current"/>
        <xsl:with-param name="tns_full" select="$tns_full"/>
        <xsl:with-param name="tns_prefix" select="$tns_prefix"/>
        <xsl:with-param name="gml_prefix" select="$gml_prefix"/>
      </xsl:call-template>
    </xsl:for-each>

    <!--
    Then, for each include and import-statement in the schema, we call this
    template recursively, to make sure that all relevant data types are
    mapped.
    -->
    <xsl:for-each select="$nodes//(xsd:include | xsdold:include)">
      <xsl:call-template name="IterateSchema">
        <xsl:with-param name="locationRootPath"
select="map:GetFileRoot(map:GetLocation($locationRootPath,
@schemaLocation))"/>

```

```

        <xsl:with-param name="nodes"
        select="document(map:GetLocation($locationRootPath,
        @schemaLocation))"/>
        <xsl:with-param name="scanned" select="concat($scanned, $current)"/>
        <xsl:with-param name="current" select="@schemaLocation"/>
        <xsl:with-param name="tns_full" select="$tns_full"/>
    </xsl:call-template>
</xsl:for-each>
<xsl:for-each select="$nodes //(xsd:import | xsdold:import)">
    <!--
    If this import is pointing to a "known" namespace, we do not want to
    analyze them, since they are not relevant for the mapping.
    -->
    <xsl:if test="@namespace != $gml_full and @namespace != $xlink_full and
    @namespace != $xml_full">
        <xsl:if test="$debug"><xsl:message terminate="no">Schema import:
        <xsl:value-of select="@namespace"/> location: <xsl:value-of
        select="@schemaLocation"/></xsl:message></xsl:if>

        <xsl:call-template name="IterateSchema">
            <xsl:with-param name="locationRootPath"
            select="map:GetFileRoot(map:GetLocation($locationRootPath,
            @schemaLocation))"/>
            <xsl:with-param name="nodes"
            select="document(map:GetLocation($locationRootPath,
            @schemaLocation))"/>
            <xsl:with-param name="tns_full" select="string(@namespace)"/>
            <xsl:with-param name="current" select="@schemaLocation"/>
        </xsl:call-template>
    </xsl:if>
</xsl:for-each>
</xsl:if>
</xsl:template>

```

```

<!--
***** template: BuildTypeMaps *****

```

This template builds typemap for the provided <element>-descendant.

Parameter list:

\$locationRootPath -> The location of the file being parsed. This string is used to determine the location of files being pointed to by url's relative to this files location.

\$element -> node containing the current element being mapped.

\$root -> root being the predecessor of element-element. This parameter is passed down through the template-calls, and are used for some purposes.

\$tns\_full -> target namespace

\$tns\_prefix -> prefix used for the target namespace in this file.

\$gml\_prefix -> the prefix used for the GML namespace in the current file.

```

-->
<xsl:template name="BuildTypeMaps">
    <xsl:param name="locationRootPath"/>
    <xsl:param name="root" select="."/>
    <xsl:param name="tns_full"/>
    <xsl:param name="tns_prefix"/>
    <xsl:param name="current" select="$basefile"/>
    <xsl:param name="element" select="//(xsd:element | xsdold:element)"/>
    <xsl:param name="gml_prefix" select="map:full2prefixFunc($gml_full,
    $root)"/>

    <xsl:if test="$debug">
        <xsl:message terminate="no">BuildTypeMaps: <xsl:value-of
        select="$tns_full"/></xsl:message>
    </xsl:if>

    <!--
    If the element has a name (and therefore is not a ref to another element, we
    map it directly.
    -->
    <xsl:if test="$element/@name">
        <TypeMap>
            <!--
            An id is made for the TypeMap. As you may experience; we might get
            several identical typemaps in the final file. These are results
            of circular imports or includes in the schema-files which the vocabulary

```

---

```

is consistant of. These may be removed by applying the stylesheet
wash.xslt.
-->
<xsl:attribute name="id" select="generate-id()"/>
<appElement>
  <localname><xsl:value-of select="$element/@name"/></localname>
  <namespace><xsl:value-of select="$tns_full"/></namespace>
</appElement>

<!--
First we record what type this element is. If it is an instantiation of
a complexType or simpleType, the type is recorded, and possibly traced
if it's not a GML type.
Otherwise it's a "native" type.
-->
<xsl:choose>
  <xsl:when test="$element/@type">
    <!--
    First the element instanceOf is instantiated, using the type-
    attribute of the element. The type may possible have a prefix. If
    so, it's stripped away, and used for
    finding the full namespace, using the function prefix2full, provided
    in this stylesheet.
    -->
    <xsl:element name="instanceOf">
      <xsl:element name="localname">
        <xsl:choose>
          <xsl:when test="contains($element/@type, ':')">
            <xsl:value-of select="substring-after($element/@type,
            ':'')"/>
          </xsl:when>
          <xsl:otherwise>
            <xsl:value-of select="$element/@type"/>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:element>
      <xsl:element name="namespace">
        <xsl:variable name="fullns">
          <xsl:choose>
            <xsl:when test="contains($element/@type, ':')">
              <xsl:value-of select="map:prefix2fullFunc(substring-
              before($element/@type, ':'), $root)"/>
            </xsl:when>
            <xsl:otherwise>
              <xsl:value-of select="map:prefix2fullFunc('', $root)"/>
            </xsl:otherwise>
          </xsl:choose>
        </xsl:variable>
        <xsl:value-of select="$fullns"/>
      <!--
      <xsl:choose>
        <xsl:when test="$fullns != ''">
          <xsl:value-of select="$fullns"/>
        </xsl:when>
        <xsl:otherwise>
          <xsl:message terminate="no">Error parsing schema. No valid
          namespace specified for element <xsl:value-of
          select="$element/@type"/>.</xsl:message>
          <xsl:value-of select="'SCHEMA PARSING ERROR'"/>
        </xsl:otherwise>
      </xsl:choose>-->
    </xsl:element>
  </xsl:element>

  <xsl:if test="$debug">
    <xsl:message terminate="no">BuildTypeMaps: #1b</xsl:message>
  </xsl:if>

  <!--If the instanceOf-element is not a GML type (or some other known
  type), we must trace the type.-->
  <xsl:variable name="fullns">
    <xsl:choose>
      <xsl:when test="contains($element/@type, ':')">
        <xsl:value-of select="map:prefix2fullFunc(substring-
        before($element/@type, ':'), $root)"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="map:prefix2fullFunc('', $root)"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:variable>

```

---

```

        </xsl:otherwise>
    </xsl:choose>
</xsl:variable>
<xsl:if test="$tns_full != $gml_full and $tns_full != $xlink_full
and $tns_full != $xml_full">
    <xsl:if test="$debug">
        <xsl:message terminate="no">BuildTypeMaps: #2</xsl:message>
    </xsl:if>

    <!--
    The TrackGMLBaseType-template is called, with the localname and
    namespace of the element we want to
    trace the origin of. We are still situated inside the same file,
    so we can pass on the value of the prefixes
    target namespace (tns) we've already got.
    -->
    <xsl:call-template name="TrackGMLBaseType">
        <xsl:with-param name="localname">
            <xsl:choose>
                <xsl:when test="contains($element/@type, ':')">
                    <xsl:value-of select="substring-after($element/@type,
                    ':')"/>
                </xsl:when>
                <xsl:otherwise>
                    <xsl:value-of select="$element/@type"/>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:with-param>
        <xsl:with-param name="namespace">
            <xsl:choose>
                <xsl:when test="contains($element/@type, ':')">
                    <xsl:value-of select="map:prefix2fullFunc(substring-
                    before($element/@type, ':'), $root)"/>
                </xsl:when>
                <xsl:otherwise>
                    <xsl:value-of select="map:prefix2fullFunc('', $root)"/>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:with-param>
        <xsl:with-param name="locationRootPath"
        select="$locationRootPath"/>
        <xsl:with-param name="current" select="$current"/>
        <xsl:with-param name="tns_full" select="$tns_full"/>
        <xsl:with-param name="tns_prefix" select="$tns_prefix"/>
        <xsl:with-param name="gml_prefix" select="$gml_prefix"/>
    </xsl:call-template>
</xsl:if>
</xsl:when>
<!--
The element did not have a type-attribute, meaning that it can
possibly be fully or partially defined inline.
If the element is and extension or restriction of another type, we
handle that here.
-->
<xsl:when test="$element/(xsd:complexContent | xsd:simpleContent |
xsd:simpleType | xsd:complexType | xsdold:complexContent |
xsdold:simpleContent | xsdold:simpleType | xsdold:complexType)
/(xsd:restriction | xsd:extension | xsdold:restriction |
xsdold:extension)">
    <xsl:variable name="typedef" select="$element/(xsd:complexContent |
xsd:simpleContent | xsd:simpleType | xsd:complexType |
xsdold:complexContent | xsdold:simpleContent | xsdold:simpleType |
xsdold:complexType)/(xsd:restriction | xsd:extension |
xsdold:restriction | xsdold:extension)"/>
    <xsl:if test="$debug">
        <xsl:message terminate="no">BuildTypeMaps: #4</xsl:message>
    </xsl:if>

    <xsl:element name="instanceOf">
        <xsl:element name="localname">
            <xsl:choose>
                <xsl:when test="contains($typedef/@base, ':')">
                    <xsl:value-of select="substring-after($typedef/@base,
                    ':')"/>
                </xsl:when>
                <xsl:otherwise>
                    <xsl:value-of select="$typedef/@base"/>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:element>

```



Now, two options are left. The element is either a reference to another type, meaning that this type will be dealt with when reached, or that the element is fully defined inline. Meaning that it is not an instantiation of another type, and certainly not indirectly derived from another type.

---

```

        </namespace>
    </substitutesFor>

    <!--
    Then we call the template to track the subgroup.
    -->
    <xsl:call-template name="trackGMLSubstitutionGroup">
        <xsl:with-param name="locationRootPath"
            select="$locationRootPath"/>
        <xsl:with-param name="gml_prefix" select="$gml_prefix"/>
        <xsl:with-param name="tns_full" select="$tns_full"/>
        <xsl:with-param name="current" select="$current"/>
        <xsl:with-param name="localname">
            <xsl:choose>
                <xsl:when test="contains(@substitutionGroup, ':')">
                    <xsl:value-of select="substring-after(@substitutionGroup,
                        ':')"/>
                </xsl:when>
                <xsl:otherwise>
                    <xsl:value-of select="@substitutionGroup"/>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:with-param>
        <xsl:with-param name="namespace">
            <xsl:choose>
                <xsl:when test="contains(@substitutionGroup, ':')">
                    <xsl:value-of select="map:prefix2fullFunc(substring-
                        before(@substitutionGroup, ':'), $root)"/>
                </xsl:when>
                <xsl:otherwise>
                    <xsl:value-of select="map:prefix2fullFunc('', $root)"/>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:with-param>
    </xsl:call-template>
    </xsl:when>
</xsl:choose>
</TypeMap>
</xsl:if>
</xsl:template>

```

```

<!--
*****   template: TrackGMLBaseType   *****

```

This template traces the type of a native type, to find out if it directly or indirectly derives from a GML-type.

Parameter list:

```

$locationRootPath -> The location of the file being parsed. This string is
used to determine the location of
                    files being pointed to by url's relative to this files
                    location.
$root              -> root of the tree where the subgroup will be searched for.
This is the root of a document throughout this file.
$localname         -> localname of the element we are currently looking for.
$namespace         -> namespace of the element we are currently looking for.
$scanned           -> concatenated string, containing the names of all files that
has already been searched for this element. Used to avoid eternal recursion.
$tns_full          -> target namespace
$tns_prefix        -> prefix used for target namespace in this file.
$typeTrack         -> this is an important element, used to avoid eternal
recursion when element names and types are equal.
$current           -> the file currently being parsed.
$gml_prefix        -> the prefix used for the GML namespace in this file. Passed
on within same file, not passed on when moving to another file.

```

```

-->
<xsl:template name="TrackGMLBaseType">
    <xsl:param name="locationRootPath"/>
    <xsl:param name="root" select="/."/>
    <xsl:param name="localname"/>
    <xsl:param name="namespace"/>
    <!--<xsl:param name="recurse" select="boolean('true')"/>      -->
    <xsl:param name="scanned"/>
    <xsl:param name="current" select="$basefile"/>
    <!--<xsl:param name="depth" select="number(0)"/>      -->
    <xsl:param name="tns_full" select="string($root/child::*[1]
/@targetNamespace)"/>
    <xsl:param name="tns_prefix" select="name($root/child::*[1]
/namespace::*[string(.)=$tns_full])"/>

```

---

```

<xsl:param name="typeTrack" select="boolean('false')"/>
<xsl:param name="gml_prefix" select="map:full2prefixFunc($gml_full,
$root)"/>

<xsl:if test="$debug">
  <xsl:message terminate="no">TrackGMLBaseType: <xsl:value-of
    select="$localname"/></xsl:message>
</xsl:if>

<xsl:variable name="element" select="$root//(<xsd:element | xsdold:element>
[@name=string($localname)])"/>
<xsl:variable name="derivedelement" select="$root//(<xsd:complexType |
xsd:simpleType | xsdold:complexType | xsdold:simpleType>)[@name=
string($localname)]"/>

<!--
We first have to check if we are scanning an application schema that is
already scanned. The scanned schemas are
concatenated in the parameter scanned, the current schema uri is stored in
the parameter current.
-->
<xsl:if test="not(contains($scanned,$current))">
  <!--<xsl:message terminate="no"><xsl:value-of select="$localname"/>
  </xsl:message>-->
  <xsl:choose>
    <!--
    The element is defined with a complexType-element. However, we still do
    not know whether this element is
    derived from a gml base type. If the derivedElement has a
    complexContent-element child, we know that it is a derived
    type, possibly from gml. If not, we must recursively find the gml base
    type.
    -->
    <xsl:when test="$derivedelement">

      <xsl:variable name="gmlDerived">
        <xsl:choose>
          <xsl:when test="$gml_prefix != ''">
            <xsl:value-of select="substring-
              after($derivedelement/(<xsd:complexContent | xsd:simpleContent |
              xsdold:complexContent | xsdold:simpleContent>)/(<xsd:restriction |
              xsd:extension | xsdold:restriction | xsdold:extension>)/@base,
              concat($gml_prefix, ':'))"/>
          </xsl:when>
          <xsl:otherwise>
            <xsl:value-of select="$derivedelement/(<xsd:complexContent |
              xsd:simpleContent | xsdold:complexContent |
              xsdold:simpleContent>)/(<xsd:restriction | xsd:extension |
              xsdold:restriction | xsdold:extension>)/@base"/>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:variable>

      <xsl:choose>
        <!--
        If the element substitutes for an element from the gml-namespace, we
        have found what we are looking for.
        -->
        <xsl:when test="$gmlDerived != ''">

          <gmlDerivedType>
            <localname><xsl:value-of select="$gmlDerived"/></localname>
            <namespace><xsl:value-of select="$gml_full"/></namespace>
          </gmlDerivedType>
        </xsl:when>
        <!--
        The element didn't substitute for a gml type, therefore we call the
        template again, this time one step closer to
        a possible gml-origin.
        -->
        <xsl:otherwise>

          <xsl:variable name="trackname">
            <xsl:choose>
              <xsl:when test="contains($derivedelement/(<xsd:complexContent |
              xsd:simpleContent | xsdold:complexContent |

```

```

xsdold:simpleContent)/(xsd:restriction | xsd:extension |
xsdold:restriction | xsdold:extension)/@base, ':'))">
  <xsl:value-of select="substring-
after($derivedelement/(xsd:complexContent |
xsd:simpleContent | xsdold:complexContent |
xsdold:simpleContent)/(xsd:restriction | xsd:extension |
xsdold:restriction | xsdold:extension)/@base, ':'))"/>
</xsl:when>
<xsl:when test="$derivedelement/(xsd:complexContent |
xsd:simpleContent | xsdold:complexContent |
xsdold:simpleContent)/(xsd:restriction | xsd:extension |
xsdold:restriction | xsdold:extension)/@base">
  <xsl:value-of select="$derivedelement/(xsd:complexContent |
xsd:simpleContent | xsdold:complexContent |
xsdold:simpleContent)/(xsd:restriction | xsd:extension |
xsdold:restriction | xsdold:extension)/@base"/>
</xsl:when>
</xsl:choose>
</xsl:variable>
<xsl:variable name="tracknamespace">
  <xsl:choose>
    <xsl:when test="contains($derivedelement/(xsd:complexContent |
xsd:simpleContent | xsdold:complexContent |
xsdold:simpleContent)/(xsd:restriction | xsd:extension |
xsdold:restriction | xsdold:extension)/@base, ':'))">
      <xsl:value-of select="map:prefix2fullFunc(substring-
before($derivedelement/(xsd:complexContent |
xsd:simpleContent | xsdold:complexContent |
xsdold:simpleContent)/(xsd:restriction | xsd:extension |
xsdold:restriction | xsdold:extension)/@base, ':'),
$root)"/>
    </xsl:when>
    <xsl:when test="$derivedelement/(xsd:complexContent |
xsd:simpleContent | xsdold:complexContent |
xsdold:simpleContent)/(xsd:restriction | xsd:extension |
xsdold:restriction | xsdold:extension)/@base">
      <xsl:value-of select="map:prefix2fullFunc(' ', $root)"/>
    </xsl:when>
  </xsl:choose>
</xsl:variable>

<xsl:call-template name="TrackGMLBaseType">
  <xsl:with-param name="locationRootPath"
select="$locationRootPath"/>
  <xsl:with-param name="root" select="$root"/>
  <xsl:with-param name="localname">
    <xsl:choose>
      <xsl:when test="$trackname">
        <xsl:value-of select="$trackname"/>
      </xsl:when>

      <xsl:otherwise>
        <xsl:message terminate="yes">Error: Tracing GMLBaseType
failed. Element localname: <xsl:value-of
select="$localname"/></xsl:message>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:with-param>
  <xsl:with-param name="namespace">
    <xsl:choose>
      <xsl:when test="$tracknamespace">
        <xsl:value-of select="$tracknamespace"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:message terminate="yes">Error: No namespace found.
Tracing GMLBaseType failed. Element localname:
<xsl:value-of select="$localname"/>.</xsl:message>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:with-param>
  <xsl:with-param name="current" select="$current"/>
  <xsl:with-param name="tns_full" select="$tns_full"/>
  <xsl:with-param name="tns_prefix" select="$tns_prefix"/>
  <xsl:with-param name="typeTrack" select="boolean('true')"/>
  <xsl:with-param name="gml_prefix" select="$gml_prefix"/>
</xsl:call-template>
</xsl:otherwise>

```

---

```

    </xsl:choose>
</xsl:when>

<!--
If typeTrack=true, this means that the element-declaration is already
found, and we are now
searching for this elements type. If this test->true, it means we are
still searching for
the element declaration.
-->
<xsl:when test="$element and not($typeTrack)">
  <xsl:choose>
    <!--
    If the element substitutes for an element from the gml-namespace, we
    have found what we are looking for.

    todo: handle a situation where there is an empty gml_prefix.
    -->
    <xsl:when test="starts-with($element/@type, concat($gml_prefix,
    ':''))">
      <gmlDerivedType>
        <localname><xsl:value-of select="substring-after($element/@type,
        ':'')"/></localname>
        <namespace><xsl:value-of select="$gml_full"/></namespace>
      </gmlDerivedType>

      <!--<xsl:message terminate="no">Element <xsl:value-of
      select="$localname"/> traced to: <xsl:value-of
      select="$element/@type"/></xsl:message>      -->
    </xsl:when>
    <!--
    The element didn't substitute for a gml type, therefore we call the
    template again, this time one step closer to
    a possible gml-origin.
    -->
    <xsl:otherwise>
      <xsl:variable name="tracename">
        <xsl:choose>
          <xsl:when test="contains($element/@type, ':'')">
            <xsl:value-of select="substring-after($element/@type,
            ':'')"/>
          </xsl:when>
          <xsl:otherwise>
            <xsl:value-of select="$element/@type"/>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:variable>
      <xsl:variable name="tracenamespace">
        <xsl:choose>
          <xsl:when test="contains($element/@type, ':'')">
            <xsl:value-of select="map:prefix2fullFunc(substring-
            before($element/@type, ':''), $root)"/>
          </xsl:when>
          <xsl:otherwise>
            <xsl:value-of select="map:prefix2fullFunc('', $root)"/>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:variable>

      <xsl:if test="not(tracename) or not(tracenamespace)">
        <xsl:message terminate="yes">Error code:
        TrackGMLBaseType#01</xsl:message>
      </xsl:if>

      <xsl:call-template name="TrackGMLBaseType">
        <xsl:with-param name="locationRootPath"
        select="$locationRootPath"/>
        <xsl:with-param name="localname" select="$tracename"/>
        <xsl:with-param name="namespace" select="$tracenamespace"/>
        <xsl:with-param name="tns_full" select="$tns_full"/>
        <xsl:with-param name="tns_prefix" select="$tns_prefix"/>
        <xsl:with-param name="typeTrack" select="boolean('true')"/>
      </xsl:call-template>
    </xsl:otherwise>
  </xsl:choose>
</xsl:when>
<xsl:otherwise>
  <xsl:for-each select="$root/(xsd:include | xsdold:include)">
    <xsl:call-template name="TrackGMLBaseType">
      <xsl:with-param name="locationRootPath"

```

---

```

        select="map:GetFileRoot(map:GetLocation($locationRootPath,
        @schemaLocation))"/>
        <xsl:with-param name="root"
        select="document(map:GetLocation($locationRootPath,
        @schemaLocation))"/>
        <xsl:with-param name="localname" select="string($localname)"/>
        <xsl:with-param name="namespace" select="string($namespace)"/>
        <xsl:with-param name="scanned" select="concat(string($scanned),
        string($current))"/>
        <xsl:with-param name="current" select="@schemaLocation"/>
        <!--<xsl:with-param name="depth" select="number($depth) + 1"/>-->
        <xsl:with-param name="tns_full" select="$tns_full"/>
        <!--<xsl:with-param name="tns_prefix" select="$tns_prefix"/>-->
        <xsl:with-param name="typeTrack" select="$typeTrack"/>
    </xsl:call-template>
</xsl:for-each>
<xsl:for-each select="$root/(xsd:import | xsdold:import)">
    <xsl:if test="@namespace!=$gml_full and @namespace=$xlink_full and
    @namespace=$xml_full">
        <xsl:call-template name="TrackGMLBaseType">
            <xsl:with-param name="locationRootPath"
            select="map:GetFileRoot(map:GetLocation($locationRootPath,
            @schemaLocation))"/>
            <xsl:with-param name="root"
            select="document(map:GetLocation($locationRootPath,
            @schemaLocation))"/>
            <xsl:with-param name="localname" select="string($localname)"/>
            <xsl:with-param name="namespace" select="string($namespace)"/>
            <xsl:with-param name="scanned" select="concat(string($scanned),
            string($current))"/>
            <xsl:with-param name="current" select="@schemaLocation"/>
            <!--<xsl:with-param name="depth" select="number($depth) +
            1"/>-->
            <!--<xsl:with-param name="tns_full" select="$tns_full"/>-->
            <!--<xsl:with-param name="tns_prefix" select="$tns_prefix"/>-->
            <xsl:with-param name="typeTrack" select="$typeTrack"/>
        </xsl:call-template>
    </xsl:if>
</xsl:for-each>
</xsl:otherwise>
</xsl:choose>
</xsl:if>
</xsl:template>

```

```

<!--
*****   template: TrackGMLSubstitutionGroup   *****

```

This template traces a substitutionGroup-value, to find out if the substitution is indirectly for an GML type.

Parameter list:

```

$locationRootPath -> The location of the file being parsed. This string is
used to determine the location of
                        files being pointed to by url's relative to this files
                        location.
$root              -> root of the tree where the subgroup will be searched for.
This is the root of a document throughout this file.
$localname         -> localname of the element we are currently looking for.
$namespace         -> namespace of the element we are currently looking for.
$scanned           -> concatenated string, containing the names of all files that
has already been searched for this element. Used to avoid eternal recursion.
$tns_full          -> target namespace
$tns_prefix        -> prefix used for target namespace in this file.
$current           -> the filename of the file currently being parsed.
-->

```

```

<xsl:template name="trackGMLSubstitutionGroup">
    <xsl:param name="locationRootPath"/>
    <xsl:param name="root" select="/."/>
    <xsl:param name="tns_full" select="string($root/child::*[1]
    /@targetNamespace)"/>
    <xsl:param name="gml_prefix" select="map:full2prefixFunc($gml_full,
    $root)"/>
    <xsl:param name="localname"/>
    <xsl:param name="namespace"/>
    <xsl:param name="scanned"/>
    <xsl:param name="current" select="$basefile"/>

    <xsl:variable name="element" select="$root/(xsd:element | xsdold:element)
    [@name=string($localname)]"/>

```

```
<xsl:if test="$debug">
  <xsl:message terminate="no">trackGMLSubstitutionGroup: <xsl:value-of
    select="$localname"/></xsl:message>
</xsl:if>

<!--
We first have to check if we are scanning an application schema that is
already scanned. The scanned schemas are
concatenated in the parameter scanned, the current schema uri is stored in
the parameter current.
-->
<xsl:if test="not(contains(string($scanned),string($current)))">
  <xsl:choose>
    <!-- if the element is found (the element which is substituted for
    another place). -->
    <xsl:when test="$element">
      <xsl:choose>
        <!--
        If the element substitutes for an element from the gml-namespace, we
        have found what we are looking for.
        -->
        <xsl:when test="starts-with($element/@substitutionGroup,
          concat($gml_prefix, ':'))">
          <baseSubstitutesFor>
            <localname><xsl:value-of select="substring-
              after($element/@substitutionGroup, ':')"/></localname>
            <namespace><xsl:value-of select="$gml_full"/></namespace>
          </baseSubstitutesFor>
        </xsl:when>
        <!--
        If this element isn't substituting for another, we have traced the
        substitutionGroup-attribute as long as possible.
        The type it substituted for was not a GML type, but one from another
        namespace.
        -->
        <xsl:when test="not($element/@substitutionGroup)">
          <baseSubstitutesFor>
            <localname><xsl:value-of select="$element/@name"/></localname>
            <namespace><xsl:value-of select="$tns_full"/></namespace>
          </baseSubstitutesFor>
        </xsl:when>
        <!--
        The element didn't substitute for a gml type, therefore we call the
        template again, this time one step closer to
        a possible gml-origin.
        -->
        <xsl:otherwise>
          <xsl:call-template name="trackGMLSubstitutionGroup">
            <xsl:with-param name="locationRootPath"
              select="$locationRootPath"/>
            <xsl:with-param name="localname">
              <xsl:choose>
                <xsl:when test="contains($element/@substitutionGroup, ':')">
                  <xsl:value-of select="substring-
                    after($element/@substitutionGroup, ':')"/>
                </xsl:when>
                <xsl:otherwise>
                  <xsl:value-of select="$element/@substitutionGroup"/>
                </xsl:otherwise>
              </xsl:choose>
            </xsl:with-param>

            <xsl:with-param name="namespace">
              <xsl:choose>
                <xsl:when test="contains($element/@substitutionGroup, ':')">
                  <xsl:value-of select="map:prefix2fullFunc(substring-
                    before($element/@substitutionGroup, ':'), $root)"/>
                </xsl:when>
                <xsl:otherwise>
                  <xsl:value-of select="map:prefix2fullFunc('', $root)"/>
                </xsl:otherwise>
              </xsl:choose>
            </xsl:with-param>
          </xsl:call-template>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:when>
  </xsl:choose>
</xsl:if>
```

---

```

<!--
The element being substituted for some place, was not found in this
file. We therefore have to trace the substitutionGroup-attribute into
the files included and imported.
-->
<xsl:otherwise>
  <xsl:for-each select="$root/(xsd:include | xsdold:include)">
    <xsl:call-template name="trackGMLSubstitutionGroup">
      <xsl:with-param name="locationRootPath"
        select="map:GetFileRoot(map:GetLocation($locationRootPath,
          @schemaLocation))"/>
      <xsl:with-param name="root"
        select="document(map:GetLocation($locationRootPath,
          @schemaLocation))"/>
      <xsl:with-param name="tns_full" select="$tns_full"/>
      <xsl:with-param name="localname" select="string($localname)"/>
      <xsl:with-param name="namespace" select="string($namespace)"/>
      <xsl:with-param name="scanned" select="concat(string($scanned),
        string($current))"/>
      <xsl:with-param name="current" select="@schemaLocation"/>
      <!--<xsl:with-param name="depth" select="number($depth) + 1"/>-->
    </xsl:call-template>
  </xsl:for-each>
  <xsl:for-each select="$root/(xsd:import | xsdold:import)">
    <xsl:if test="@namespace!=$gml_full and @namespace=$xlink_full and
      @namespace=$xml_full">
      <xsl:call-template name="trackGMLSubstitutionGroup">
        <xsl:with-param name="locationRootPath"
          select="map:GetFileRoot(map:GetLocation($locationRootPath,
            @schemaLocation))"/>
        <xsl:with-param name="root"
          select="document(map:GetLocation($locationRootPath,
            @schemaLocation))"/>
        <xsl:with-param name="tns_full" select="@namespace"/>
        <xsl:with-param name="localname" select="string($localname)"/>
        <xsl:with-param name="namespace" select="string($namespace)"/>
        <xsl:with-param name="scanned" select="concat(string($scanned),
          string($current))"/>
        <xsl:with-param name="current" select="@schemaLocation"/>
        <!--<xsl:with-param name="depth" select="number($depth) +
          1"/>-->
      </xsl:call-template>
    </xsl:if>
  </xsl:for-each>
</xsl:otherwise>
</xsl:choose>
</xsl:if>
</xsl:template>

<!--
This function determines the full namespace, given the abbreviation. $root
holds the root of a schema file.
-->
<xsl:function name="map:prefix2fullFunc">
  <xsl:param name="abbr"/>
  <xsl:param name="root"/>

  <xsl:variable name="temp">
    <xsl:value-of select="string($root/child::*[1]/namespace::*[name()=$abbr]
      /.)"/>
  </xsl:variable>

  <xsl:choose>
    <xsl:when test="$temp != ''">
      <xsl:value-of select="$temp"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:message terminate="no">Invalid namespace prefix. No namespace
        (default or prefixed) found.</xsl:message>
      <xsl:value-of select="'INVALID NS PREFIX'"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:function>

<!--
This function determines the prefix used for a given namespace. $root holds
the root of a schema file.
-->
<xsl:function name="map:full2prefixFunc">
  <xsl:param name="full"/>

```

---



```

<xsl:param name="root"/>

<xsl:variable name="isPresent" select="boolean($root/child::*[1]
/namespace::*[string(.)=$full])"/>

<xsl:choose>
  <xsl:when test="$isPresent">
    <xsl:value-of select="string(name($root/child::*[1]
/namespace::*[string(.)=$full])"/>
  </xsl:when>
  <xsl:otherwise>
    <xsl:value-of select="'_NPIF_'"/> <!-- namespace is not present i file
-->
  </xsl:otherwise>
</xsl:choose>

</xsl:function>
</xsl:stylesheet>

```

## Stylesheet for removing identical type maps from mapping dictionary

This transformation converts any GML 2.1.2 data into SVG, given a correct mapping dictionary, created with the mapElements.xsd listed the section called “GML Schema to Mapping Dictionary”.

```

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:bm="no:hiof:basemapper">
  <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>

  <xsl:template match="/">
    <bm:MappingDictionary>
      <xsl:copy-of select="/bm:MappingDictionary/bm:documentNamespaces"/>

      <bm:typeMaps>
        <xsl:call-template name="washTypeMaps"/>
      </bm:typeMaps>
    </bm:MappingDictionary>
  </xsl:template>

  <xsl:template name="washTypeMaps">
    <xsl:param name="transferred" select="''"/>
    <xsl:param name="current" select="//bm:TypeMap[1]"/>

    <xsl:choose>
      <xsl:when test="not(contains($transferred, concat('_', $current/@id,
        '_')))">
        <xsl:copy-of select="$current"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:message terminate="no">Duplicate TypeMap filtered: <xsl:value-of
          select="$current/@id"/></xsl:message>
      </xsl:otherwise>
    </xsl:choose>

    <xsl:variable name="following" select="$current/following-sibling::*[1]"/>
    <xsl:if test="$following">
      <xsl:call-template name="washTypeMaps">
        <xsl:with-param name="transferred" select="concat('_', $transferred,

```

```
        ' ' $current/@id, ' ')" />
        <xsl:with-param name="current" select="$following" />
    </xsl:call-template>
</xsl:if>
</xsl:template>
</xsl:stylesheet>
```

## Generic GML/Dictionary to SVG transformation

This stylesheet converts a GML 2.1.2 compliant file to SVG, given a correct mapping dictionary in accordance with the schema listed the section called “Mapping Dictionary Schema”. This file can be created using the transformation listed the section called “GML Schema to Mapping Dictionary”.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
This transformation stylesheet is made as a proof of concept software, for the
GML mapping dictionary. It utilizes arbitrary
GML files, transforming them to SVG, using the information stored in a mapping
dictionary.
-->
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:gml="http://www.opengis.net/gml"
  xmlns:c="no:hiof:basemapper:constants"
  xmlns:g2s="no:hiof:basemapper:gmlsvg"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:bm="no:hiof:basemapper"
  xmlns:meta="no:hiof:onemap:gml:metainfo"
  xmlns:one="http://onemap.org"
  xmlns:style="userstyle">

  <xsl:output method="xml" indent="yes"
    doctype-public="-//W3C//DTD SVG 20010904//EN"
    doctype-system="http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd"
    cdata-section-elements="script cdata style"/>
  <xsl:output method="html" name="htmldoc"/>

  <!--
  constants.xslt holds some of the base GML names.
  -->
  <xsl:include href="./constants.xslt"/>

  <!-- To be able to retrieve a command line param, we must declare the param
  here-->
  <xsl:param name="mapfile"/>
  <xsl:param name="viewport" select="''"/>
  <xsl:param name="styling"/>
  <xsl:param name="flat" select="'no'"/>

  <xsl:variable name="userStyles">
    <xsl:choose>
      <xsl:when test="$styling">
        <!--<xsl:message terminate="no">Styling: <xsl:value-of
        select="$styling"/></xsl:message>
        <xsl:value-of select="document($styling)" />-->
        <style:styles xmlns:style="userstyle" xmlns="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.gisline.no</style:namespace>
        <style:stylestring>stroke:black; stroke-width: 0.05%; fill:brown;
        fill-opacity:1</style:stylestring>
      </style:style>
      <style:style>
        <style:namespace>http://www.onemap.net</style:namespace>
        <style:stylestring>stroke:black; stroke-width: 0.05%;
        fill:#1BE439</style:stylestring>
      </style:style>
      <style:style>
        <style:namespace>http://www.ordnancesurvey.co.uk/xml/namespaces/osgb
        </style:namespace>
        <style:stylestring>stroke:black; stroke-width: 0.05%; fill:black;
        fill-opacity:0.0</style:stylestring>
      </style:style>
      <style:style>
        <style:namespace>http://www.opengis.net/examples</style:namespace>
        <style:stylestring>stroke:blue; stroke-width: 0.05%; fill:black;
        fill-opacity:0.0</style:stylestring>
      </style:style>
    </style:styles>
  </xsl:when>
  <xsl:otherwise>
    <style:styles>
      <style:style>
        <style:namespace>default</style:namespace>
        <style:stylestring>stroke:black; stroke-width: 0.05%;
        fill:none;</style:stylestring>
      </style:style>
    </style:styles>
  </xsl:otherwise>
</xsl:choose>
</xsl:variable>

<xsl:variable name="INFORMATIVE_MSG" select="boolean('')"/>
<xsl:variable name="WARNING_MSG" select="boolean('true')"/>
<xsl:variable name="DRAW_FLAT" select="boolean('true')"/>

<xsl:variable name="feature_style" select="'stroke:black; stroke-width: 0.03%;
stroke-color: black; fill:none'"/>
<xsl:variable name="featureCollectionStyle" select="''"/>

<!-- The variable DICT_ROOT contains the childs of the outermost element -->
<xsl:variable
  name="gDictRoot"
  select="document(string($mapfile))/bm:MappingDictionary"/>

<!-- We might need access to the geometry schema of GML2 for recognizing
geometric properties. -->
<xsl:variable
  name="gGeometrySchema"
  select="document('./geometry.xsd')/xs:schema"/>

<xsl:variable
  name="gFeatureSchema"
  select="document('./feature.xsd')/xs:schema"/>

<!-- If there are application specific properties, these the base substitution
group will be gml:_geometryProperty. This relationship is to be found in the
dictionary -->
<!--<xsl:variable
  name="gAppGeometries"
  select="$gDictRoot/bm:typeMaps/bm:TypeMap[
    (bm:baseSubstitutesFor/bm:localname=$GEOMETRY_PROPERTY_BASE and
    bm:baseSubstitutesFor/bm:namespace=$GML_NAMESPACE) or
    (bm:instanceOf/bm:localname=$GEOMETRY_PROPERTY_TYPE and
    bm:instanceOf/bm:namespace=$GML_NAMESPACE) or
    (bm:gmlDerivedType/bm:localname=$GEOMETRY_PROPERTY_TYPE and
    bm:gmlDerivedType/bm:namespace=$GML_NAMESPACE) or
    (bm:baseSubstitutesFor/bm:localname=$GEOMETRY_PROPERTY_TYPE and
    bm:baseSubstitutesFor/bm:namespace=$GML_NAMESPACE)]

```

---

```

/bm:appElement/bm:localname"/>-->

<!--<xsl:variable
  name="gGeometryTypes"
  select="$gGeometrySchema/xs:element[@substitutionGroup=concat('gml:',
    $GEOMETRY_BASE)]/@name"/> -->

  <!-- The variable gFeatureCollections holds the names of all the elements
  identified as descendants of gml:AbstractFeatureCollectionType -->

<xsl:variable
  name="gFeatureCollections"
  select="$gDictRoot/bm:typeMaps/bm:TypeMap/(bm:gmlDerivedType |
    bm:instanceOf)[
      bm:localname=$FEATURE_COLLECTION_TYPE and bm:namespace=$GML_NAMESPACE]
  /../bm:appElement"/>

<!-- The variable gFeatures holds the names of all the elements recognized as
types derived from AbstractFeatureType -->

<xsl:variable
  name="gFeatures"
  select="$gDictRoot/bm:typeMaps/bm:TypeMap/(bm:instanceOf |
    bm:gmlDerivedType)[
      bm:localname=$FEATURE_TYPE and bm:namespace=$GML_NAMESPACE]
  /../bm:appElement"/>

<!--
All localnames recognized as linestring-types, are stored in this variable.
It can be noted that this and the other geometry variables only store the
localname of an element, and there might be some mix ups if there are several
elements with
same name, but in different namespace. However this is a proof of concept, and
further enhancement is required to make it fully reliable.
-->
<xsl:variable
  name="gAppLineStrings"
  select="$gDictRoot/bm:typeMaps/bm:TypeMap[
    (bm:instanceOf/bm:localname=$GEOMETRY_LINESTRING_TYPE and
      bm:instanceOf/bm:namespace=$GML_NAMESPACE) or
    (bm:gmlDerivedType/bm:localname=$GEOMETRY_LINESTRING_TYPE and
      bm:gmlDerivedType/bm:namespace=$GML_NAMESPACE)]
  /bm:appElement/bm:localname"/>

<xsl:variable
  name="gAppLinearRings"
  select="$gDictRoot/bm:typeMaps/bm:TypeMap[
    (bm:instanceOf/bm:localname=$GEOMETRY_LINEARRING_TYPE and
      bm:instanceOf/bm:namespace=$GML_NAMESPACE) or
    (bm:gmlDerivedType/bm:localname=$GEOMETRY_LINEARRING_TYPE and
      bm:gmlDerivedType/bm:namespace=$GML_NAMESPACE)]
  /bm:appElement/bm:localname"/>

<xsl:variable
  name="gAppPolygons"
  select="$gDictRoot/bm:typeMaps/bm:TypeMap[(bm:instanceOf/bm:localname=
    $GEOMETRY_POLYGON_TYPE and bm:instanceOf/bm:namespace=$GML_NAMESPACE) or
    (bm:gmlDerivedType/bm:localname=$GEOMETRY_POLYGON_TYPE and
      bm:gmlDerivedType/bm:namespace=$GML_NAMESPACE)]
  /bm:appElement/bm:localname"/>

<xsl:variable
  name="gAppPoints"
  select="$gDictRoot/bm:typeMaps/bm:TypeMap[
    (bm:instanceOf/bm:localname=$GEOMETRY_POINT_TYPE and
      bm:instanceOf/bm:namespace=$GML_NAMESPACE) or
    (bm:gmlDerivedType/bm:localname=$GEOMETRY_POINT_TYPE and
      bm:gmlDerivedType/bm:namespace=$GML_NAMESPACE)]
  /bm:appElement/bm:localname"/>

<xsl:variable
  name="gAppCoords"
  select="$gDictRoot/bm:typeMaps/bm:TypeMap[
    (bm:instanceOf/bm:localname=$COORD_TYPE and bm:instanceOf/bm:namespace=
      $GML_NAMESPACE) or
    (bm:gmlDerivedType/bm:localname=$COORD_TYPE and
      bm:gmlDerivedType/bm:namespace=$GML_NAMESPACE)]
  /bm:appElement/bm:localname"/>

<xsl:variable

```

```

name="gAppCoordinates"
select="$gDictRoot/bm:typeMaps/bm:TypeMap[
  (bm:instanceOf/bm:localname=$COORDINATES_TYPE and
   bm:instanceOf/bm:namespace=$GML_NAMESPACE) or
  (bm:gmlDerivedType/bm:localname=$COORDINATES_TYPE and
   bm:gmlDerivedType/bm:namespace=$GML_NAMESPACE)]
/bm:appElement/bm:localname"/>

<!-- Now let's find the featureMember-types. The featureMember-element from
feature.xsd can be used directly in instance document,
or application schemas can restrict the FeatureAssociationType and substitute
for featureMember, to restrict membership inside the featureMember-element -->
<!--<xsl:variable
name="gAppFeatureMembers"
select="$gDictRoot/bm:typeMaps/bm:TypeMap[bm:gmlDerivedType/bm:localname=
$FEATURE_MEMBER_TYPE or
  (bm:instanceOf/bm:localname=$FEATURE_MEMBER_TYPE and
   bm:instanceOf/bm:namespace=$GML_NAMESPACE) or
  (bm:baseSubstitutesFor/bm:localname=$FEATURE_MEMBER_ELEMENT and
   bm:baseSubstitutesFor/bm:namespace=$GML_NAMESPACE) or
  (bm:substitutesFor/bm:localname=$FEATURE_MEMBER_ELEMENT and
   bm:substitutesFor/bm:namespace=$GML_NAMESPACE)]
/bm:appElement/bm:localname"/>-->

<xsl:variable
name="gAppFeatureMembers"
select="$gDictRoot/bm:typeMaps/bm:TypeMap/(bm:instanceOf | bm:gmlDerivedType
| bm:baseSubstitutesFor | bm:substitutesFor)[
  (bm:localname=$FEATURE_MEMBER_TYPE and bm:namespace=$GML_NAMESPACE) or
  (bm:localname=$FEATURE_MEMBER_ELEMENT and bm:namespace=$GML_NAMESPACE)]
/../../bm:appElement"/>

<!--
This variable holds the names of all the GML types, being descendants of a
geometry association type, and
-->
<!--<xsl:variable name="gGeometryPropertyTypes" select="($gFeatureSchema |
$gGeometrySchema)/xs:complexType[xs:complexContent/xs:restriction/@base=
concat('gml:', $GEOMETRY_ASSOCIATION_TYPE)]/@name"/>-->

<!--
The two following variables stores the geometrymember-elements in defined in
the featureschema. E.g. centerLineOf, centerOf etc. These are not abstract
elements, and can be used directly in an instance document.
-->
<xsl:variable
name="gBaseGeometryMembers"
select="$gFeatureSchema/xs:element[@substitutionGroup=concat('gml:',
$GEOMETRY_PROPERTY_BASE) or (@type=concat('gml:',
$GEOMETRY_ASSOCIATION_TYPE) and not(@abstract='true'))]"/>

<xsl:variable
name="gBaseGeometryAliases"
select="$gFeatureSchema/xs:element[$gBaseGeometryMembers/@name=substring-
after(@substitutionGroup, ':')]">

<!--
The variable gAppGeometryProperties is however correct. Storing the whole
appElement-element from the mapping dictionary. This can be exhaustive for
resources
if done for all variables. I combine this with a function called
isElementInAppElement, to search through the appElement-nodes.
-->

<xsl:variable name="gAppGeometryProperties"
select="$gDictRoot/bm:typeMaps/bm:TypeMap/(bm:instanceOf |
bm:gmlDerivedType)[
  (concat('gml:', bm:localname)=$gBaseGeometryMembers/@type and
   bm:namespace=$GML_NAMESPACE) or
  (bm:localname=$GEOMETRY_ASSOCIATION_TYPE and bm:namespace=$GML_NAMESPACE)
or
  (bm:localname=$GEOMETRY_PROPERTY_TYPE and bm:namespace=$GML_NAMESPACE)]
/../../bm:appElement"/>

<xsl:template match="/">
<xsl:message terminate="no">Features in mapping file: <xsl:value-of
select="count($gFeatures)"/></xsl:message>
<xsl:message terminate="no">FeatureCollections in mapping file: <xsl:value-

```

---

```

of select="count($gFeatureCollections)"/></xsl:message>
<xsl:message terminate="no">FeatureMembers in mapping file: <xsl:value-of
select="count($gAppFeatureMembers)"/></xsl:message>

  <!--<xsl:for-each select="$userStyles">
    <xsl:message terminate="no"><xsl:value-of select="namespace-uri(.)"/>
  </xsl:for-each>
  <xsl:message terminate="no"><xsl:value-of select="count($userStyles)"/>
  <xsl:value-of select="local-name($userStyles)"/></xsl:message>-->

<xsl:variable name="viewBox">
  <xsl:choose>
    <xsl:when test="not($viewbox = '')">
      <xsl:value-of select="g2s:flip($viewbox)"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:call-template name="viewBox">
        <xsl:with-param name="box" select="child::* / gml:boundedBy"/>
      </xsl:call-template>
    </xsl:otherwise>
  </xsl:choose>
</xsl:variable>

<xsl:element name="svg">
  <xsl:attribute name="width">100%</xsl:attribute>
  <xsl:attribute name="height">100%</xsl:attribute>
  <xsl:attribute name="viewBox">
    <xsl:choose>
      <xsl:when test="not($viewBox)">
        <xsl:message terminate="no">Bounding box not specified. Please
          specify viewBox as stylesheet parameter.</xsl:message>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="$viewBox"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:attribute>

  <xsl:attribute name="onload">
    <xsl:value-of select="'init(evt);'"/>
  </xsl:attribute>

  <xsl:variable name="rootLayers" select="//one:layerDescription"/>
  <defs>
    <xsl:call-template name="addMenuDef">
      <xsl:with-param name="rootLayers" select="$rootLayers"/>
    </xsl:call-template>
    <xsl:call-template name="defineStyles"/>
  </defs>

  <xsl:call-template name="addScript"/>
  <xsl:call-template name="writeInfoWindow"/>
  <!--
considering that SVG coordinates originates from the top left corner, and the
GML coordinates originates from the bottom left. We specify a matrix
transformation here.
In addition, the viewBox has to be changed, so that the view of the data doesn't
disappear.
-->
  <g transform="matrix(1,0,0,-1,0,0)">
    <xsl:choose>
      <xsl:when test="$flat = 'yes'">
        <xsl:apply-templates select="//*[local-name()=
          $gFeatures/bm:localname and namespace-uri()=
          $gFeatures/bm:namespace]"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:apply-templates select="child::*[local-name()=
          $gFeatureCollections/bm:localname and namespace-uri()=
          $gFeatureCollections/bm:namespace]"/>
      </xsl:otherwise>
    </xsl:choose>
  </g>

  <xsl:copy-of select="document($mapfile)"/>
</xsl:element>

```

---

```

</xsl:template>

<xsl:template name="viewBox">
  <xsl:param name="box"/>

  <xsl:variable name="coord" select="$box//gml:coord | $box//*[local-name()=
$gAppCoords]"/>
  <xsl:variable name="coordinates" select="$box//gml:coordinates |
$box//*[local-name()=$gAppCoordinates]"/>

  <xsl:choose>
    <xsl:when test="$coord">
      <xsl:variable name="x" select="$coord[1]/gml:X"/>
      <xsl:variable name="y" select="$coord[1]/gml:Y"/>
      <xsl:variable name="width" select="number($coord[2]/gml:X) -
number($x)"/>
      <xsl:variable name="height" select="number($coord[2]/gml:Y) -
number($y)"/>

      <!-- to flip the coordinate system, we make the y negative and subtracts
the height-->
      <xsl:variable name="strVB" select="concat(concat($x, ' ', $y), concat('
', $width), concat(' ', $height))"/>
      <xsl:value-of select="g2s:flip($strVB)"/>
    </xsl:when>
    <xsl:when test="$coordinates">
      <xsl:variable name="separator" select="g2s:getSeparator($coordinates)"/>

      <xsl:variable name="x" select="normalize-space(substring-
before($coordinates, $separator))"/>
      <xsl:variable name="y" select="normalize-space(substring-
before(substring-after($coordinates, $separator), ' '))"/>
      <xsl:variable name="x2" select="normalize-space(substring-
before(substring-after(substring-after($coordinates, $separator), ' '),
$separator))"/>
      <xsl:variable name="y2" select="normalize-space(substring-
after(substring-after(substring-after($coordinates, $separator), ' '),
$separator))"/>

      <xsl:variable name="width" select="number($x2) - number($x)"/>
      <xsl:variable name="height" select="number($y2) - number($y)"/>

      <xsl:variable name="strVB" select="normalize-space(concat(concat($x, '
', $y), concat(' ', $width), concat(' ', $height)))/>

      <xsl:value-of select="g2s:flip($strVB)"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="'0 -2000 2000 2000'"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<!--
This function takes a viewBox-string, makes the min-y negative and subtracts
the height of the
box. This should be done in relation with transform="matrix(1,0,0,-1,0,0)",
which turns the drawing upside-down.
Reason: svg coordinates has 0,0 as top left corner, while GIS and GML has a
y-axis with positive up and negative down.
-->
<xsl:function name="g2s:flip">
  <xsl:param name="strViewBox"/>

  <xsl:variable name="x" select="substring-before(normalize-
space($strViewBox), ' ')/>
  <xsl:variable name="temp" select="substring-after(normalize-
space($strViewBox), ' ')/>
  <xsl:variable name="y" select="substring-before(normalize-space($temp), '
')"/>
  <xsl:variable name="temp2" select="substring-after(normalize-space($temp), '
')"/>
  <xsl:variable name="width" select="substring-before(normalize-space($temp2),
' ')/>
  <xsl:variable name="height" select="substring-after(normalize-space($temp2),
' ')/>

  <xsl:value-of select="concat(concat($x, ' ', (-1 * number($y))-
number($height)), concat(' ', $width, ' ', $height))"/>

```

```
</xsl:function>

<xsl:function name="g2s:getSeparator">
  <xsl:param name="coordinates"/>

  <xsl:choose>
    <xsl:when test="$coordinates/@cs">
      <xsl:value-of select="$coordinates/@cs"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="', '"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:function>

<!--Draw polygons or descendant-->
<xsl:template match="*[node() and (local-name()=$GEOMETRY_POLYGON_ELEMENT and
namespace-uri()=$GML_NAMESPACE) or
  local-name()=$gAppPolygons]">

  <xsl:variable name="hasCoordinates" select="g2s:hasCoordinates(.)"/>

  <xsl:choose>
    <xsl:when test="$hasCoordinates">
      <xsl:element name="polygon">

        <xsl:attribute name="points">
          <xsl:call-template name="gmlCoordinateString">
            <xsl:with-param name="shape" select="."/>
          </xsl:call-template>
        </xsl:attribute>
        <!--<xsl:attribute name="style">
          <xsl:text>stroke: black; fill: none;</xsl:text>
        </xsl:attribute-->
      </xsl:element>
    </xsl:when>
    <xsl:otherwise>
      <xsl:if test="$WARNING_MSG"><xsl:message terminate="no">Drawing
cancelled: <xsl:value-of select="local-name(.)"/> no coordinates
recognized.</xsl:message></xsl:if>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<!--Draw linestrings or descendant-->
<xsl:template match="*[node() and (local-name()=$GEOMETRY_LINESTRING_ELEMENT
and namespace-uri()=$GML_NAMESPACE) or
  local-name()=$gAppLineStrings]">

  <xsl:variable name="hasCoordinates" select="g2s:hasCoordinates(.)"/>

  <xsl:choose>
    <xsl:when test="$hasCoordinates">
      <xsl:element name="polyline">

        <xsl:attribute name="points">
          <xsl:call-template name="gmlCoordinateString">
            <xsl:with-param name="shape" select="."/>
          </xsl:call-template>
        </xsl:attribute>
        <!--<xsl:attribute name="style">
          <xsl:text>stroke: black; fill: none;</xsl:text>
        </xsl:attribute-->
      </xsl:element>
    </xsl:when>
    <xsl:otherwise>
      <xsl:if test="$WARNING_MSG"><xsl:message terminate="no">Drawing
cancelled: <xsl:value-of select="local-name(.)"/> no coordinates
recognized.</xsl:message></xsl:if>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<!--Draw linestrings or descendant-->
<xsl:template match="*[node() and (local-name()=$GEOMETRY_LINEARRING_ELEMENT
and namespace-uri()=$GML_NAMESPACE) or
  local-name()=$gAppLinearRings]">

  <xsl:variable name="hasCoordinates" select="g2s:hasCoordinates(.)"/>
```



---

```

<xsl:choose>
  <xsl:when test="$hasCoordinates">
    <xsl:element name="polyline">
      <xsl:attribute name="points">
        <xsl:call-template name="gmlCoordinateString">
          <xsl:with-param name="shape" select="."/>
        </xsl:call-template>
      </xsl:attribute>
      <!--<xsl:attribute name="style">
        <xsl:text>stroke: black; fill: none;</xsl:text>
      </xsl:attribute-->
    </xsl:element>
  </xsl:when>
  <xsl:otherwise>
    <xsl:if test="$WARNING_MSG"><xsl:message terminate="no">Drawing
      cancelled: <xsl:value-of select="local-name(.)"/> no coordinates
      recognized.</xsl:message></xsl:if>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<xsl:template match="*[node() and (local-name()=$GEOMETRY_POINT_ELEMENT and
namespace-uri()=$GML_NAMESPACE) or
local-name()=gAppPoints]">
  <xsl:variable name="point" select="."/>

  <xsl:variable name="coord" select="$point//gml:coord | $point//*[local-
name()=$gAppCoords]"/>
  <xsl:variable name="coordinates" select="$point//gml:coordinates |
$point//*[local-name()=$gAppCoordinates]"/>
  <xsl:variable name="cs" select="$coordinates/@cs"/>

  <xsl:variable name="cx">
    <xsl:choose>
      <xsl:when test="$coordinates">
        <!--<xsl:call-template name="Trim">
          <xsl:with-param name="strInput">-->
          <xsl:choose>
            <xsl:when test="$cs">
              <xsl:value-of select="normalize-space(substring-
before($coordinates, $cs))"/>
            </xsl:when>
            <xsl:otherwise>
              <xsl:value-of select="normalize-space(substring-
before($coordinates, ','))"/>
            </xsl:otherwise>
          </xsl:choose>
        <!--</xsl:with-param>
      </xsl:call-template-->
      </xsl:when>
      <xsl:when test="$coord">
        <xsl:value-of select="$coord/gml:X"/>
      </xsl:when>
    </xsl:choose>
  </xsl:variable>

  <xsl:variable name="cy">
    <xsl:choose>
      <xsl:when test="$coordinates">
        <!--<xsl:call-template name="Trim">
          <xsl:with-param name="strInput">-->
          <xsl:choose>
            <xsl:when test="$cs">
              <xsl:value-of select="normalize-space(substring-
before($coordinates, $cs))"/>
            </xsl:when>
            <xsl:otherwise>
              <xsl:value-of select="normalize-space(substring-
after($coordinates, ','))"/>
            </xsl:otherwise>
          </xsl:choose>
        <!--</xsl:with-param>
      </xsl:call-template-->
      </xsl:when>
      <xsl:when test="$coord">
        <xsl:value-of select="$coord/gml:Y"/>
      </xsl:when>
    </xsl:choose>
  </xsl:variable>

```

---

```

</xsl:variable>
<xsl:element name="circle">
  <xsl:attribute name="cx">
    <xsl:value-of select="$cx"/>
  </xsl:attribute>
  <xsl:attribute name="cy">
    <xsl:value-of select="$cy"/>
  </xsl:attribute>
  <xsl:attribute name="r">0.30%</xsl:attribute>
</xsl:element>
</xsl:template>

<!-- patterns may not contain variable or parameters, therefore the test
whether the element is a featureCollection, is done inside the template, so
that variables may be referenced. -->

<xsl:template match="//*[local-name()=$gFeatureCollections/bm:localname and
namespace-uri()=$gFeatureCollections/bm:namespace]">
<!--<xsl:template name="handleFeatureCollections">-->
  <!--<xsl:param name="collections" select=""/>-->

  <xsl:if test="$INFORMATIVE_MSG"><xsl:message terminate="no">
featureCollection: <xsl:value-of select="local-name()"/></xsl:message>
</xsl:if>
<g>
  <xsl:if test="one:layerDescription != ''">
    <xsl:attribute name="id"><xsl:value-of select="one:layerDescription"/>
    </xsl:attribute>

    <xsl:attribute name="title">
      <xsl:value-of select="one:layerDescription"/>
    </xsl:attribute>

    <xsl:attribute name="visibility">visible</xsl:attribute>
  </xsl:if>

  <xsl:apply-templates select="child::*[(local-name()=
$FEATURE_MEMBER_ELEMENT and namespace-uri()=$GML_NAMESPACE) or (local-
name()=$gAppFeatureMembers/bm:localname and namespace-uri()=
$gAppFeatureMembers/bm:namespace)]"/>
</g>

</xsl:template>

<xsl:template match="//*[(local-name()=$FEATURE_MEMBER_ELEMENT and namespace-
uri()=$GML_NAMESPACE) or (local-name()=$gAppFeatureMembers/bm:localname and
namespace-uri()=$gAppFeatureMembers/bm:namespace)]">
  <xsl:if test="$INFORMATIVE_MSG"><xsl:message terminate="no">featureMember:
<xsl:value-of select="local-name()"/></xsl:message></xsl:if>

  <g>

    <xsl:apply-templates select="child::*[(local-name()=
$gFeatures/bm:localname and namespace-uri()=$gFeatures/bm:namespace) or
(local-name()=$gFeatureCollections/bm:localname
and namespace-uri()=
$gFeatureCollections/bm:namespace)]"/>

  </g>
</xsl:template>

<xsl:template match="//*[(local-name()=$gFeatures/bm:localname and namespace-
uri()=$gFeatures/bm:namespace)]">
  <xsl:variable name="current" select="current()"/>
  <xsl:if test="$INFORMATIVE_MSG"><xsl:message terminate="no">feature:
<xsl:value-of select="local-name()"/></xsl:message></xsl:if>
  <xsl:element name="g">
    <!--<xsl:attribute name="class">
      <xsl:value-of select="'default'"/>
    </xsl:attribute-->
    <xsl:variable name="feature_id">
      <xsl:choose>
        <xsl:when test="@fid">
          <xsl:value-of select="@fid"/>
        </xsl:when>
        <xsl:otherwise>

```

---

```

        <xsl:value-of select="concat('feature_', generate-id())"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:variable>
  <xsl:attribute name="id">
    <xsl:value-of select="$feature_id"/>
  </xsl:attribute>
  <!--<xsl:attribute name="style">
    <xsl:value-of select="$feature_style"/>
  </xsl:attribute-->
  <xsl:attribute name="class">
    <xsl:choose>
      <xsl:when test="$userStyles/style:styles/style:style/style:namespace =
        namespace-uri()">
        <xsl:for-each select="$userStyles/style:styles/style:style">
          <xsl:if test="style:namespace = namespace-uri($current)">
            <xsl:value-of select="concat('fstyle' , position())"/>
          </xsl:if>
        </xsl:for-each>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="'default'"/>
      </xsl:otherwise>
    </xsl:choose>

  </xsl:attribute>
  <xsl:attribute name="onactivate">
    <xsl:value-of select="concat(concat('showFeatureData('', $feature_id),
    '''))"/>
  </xsl:attribute>
  <xsl:attribute name="onfocusin">
    <xsl:value-of select="concat(concat('setStrokeWidth('', $feature_id),
    ''', '0.20%'))"/>
  </xsl:attribute>
  <xsl:attribute name="onfocusout">
    <xsl:value-of select="concat(concat('setStrokeWidth('', $feature_id),
    ''', '0.05%'))"/>
  </xsl:attribute>

  <xsl:element name="defs">
    <xsl:element name="meta:MetaInformation">
      <xsl:element name="meta:TypeInfoInformation">
        <!--
        Need to store the namespace and localname in temporary variables in
        order to get comparison to work..
        Strange problem, posted it to the saxon-help-list.
        -->
        <xsl:variable name="tmpName" select="local-name()"/>
        <xsl:variable name="tmpNamespace" select="namespace-uri()"/>
        <xsl:variable name="mapId"
        select="$gDictRoot/bm:typeMaps/bm:TypeMap[bm:appElement/bm:localname
        =$tmpName and bm:appElement/bm:namespace=$tmpNamespace]/@id"/>
        <xsl:element name="TypeMap" namespace="no:hiof:basemapper">
          <xsl:attribute name="xlink:type"><xsl:value-of
          select="'simple'"/></xsl:attribute>
          <xsl:attribute name="xlink:href"><xsl:value-of
          select="concat('#', string($mapId))"/></xsl:attribute>

        </xsl:element>
        <!--<xsl:copy-of
        select="$gDictRoot/bm:typeMaps/bm:TypeMap[bm:appElement/bm:localname
        =$tmpName and bm:appElement/bm:namespace=$tmpNamespace]"/>-->

        <!--<xsl:message terminate="no"><xsl:value-of select="local-
        name()"/></xsl:message-->
        <xsl:message terminate="no"><xsl:value-of select="namespace-
        uri()"/></xsl:message-->

      </xsl:element>
      <xsl:element name="meta:properties">
        <xsl:for-each select="child::*[node()]">
          <xsl:variable name="isAppGeometryProp"
          select="g2s:isElementInAppElement(current())"/>
          <xsl:choose>
            <xsl:when test="not($isAppGeometryProp)">
              <xsl:call-template name="writePropertyMeta">

```

---

---

```

        <xsl:with-param name="property" select="current()"/>
    </xsl:call-template>
</xsl:when>
<xsl:otherwise>
    <xsl:call-template name="writePropertyMeta">
        <xsl:with-param name="property" select="current()"/>
        <xsl:with-param name="isGeometry" select="boolean('true')"/>
    </xsl:call-template>
</xsl:otherwise>
</xsl:choose>
</xsl:for-each>
</xsl:element>
</xsl:element>
</xsl:element>
    <xsl:apply-templates select="child::*"/>
</xsl:element>
</xsl:template>

<xsl:function name="g2s:isElementInAppElement">
    <xsl:param name="element"/>

    <xsl:if test="local-name($element)=$gBaseGeometryMembers/@name and
namespace-uri($element)=$GML_NAMESPACE">
        <xsl:value-of select="boolean('true')"/>
    </xsl:if>

    <xsl:if test="local-name($element)=$gBaseGeometryAliases/@name and
namespace-uri($element)=$GML_NAMESPACE">
        <xsl:value-of select="boolean('true')"/>
    </xsl:if>

    <xsl:for-each select="$gAppGeometryProperties">
        <xsl:if test="bm:localname=local-name($element) and bm:namespace=
namespace-uri($element)">
            <xsl:value-of select="boolean('true')"/>
        </xsl:if>
    </xsl:for-each>

    <xsl:value-of select="''"/>
</xsl:function> <!-- end function: g2s:isElementInAppElement -->

<xsl:template name="writePropertyMeta">
    <xsl:param name="property"/>
    <xsl:param name="isGeometry" select="boolean('')"/>

    <xsl:choose>
        <xsl:when test="$property">
            <xsl:element name="meta:Property" xml:space="preserve">
                <xsl:variable name="mapid"
select="$gDictRoot/bm:typeMaps/bm:TypeMap[bm:appElement/bm:localname=
local-name($property) and bm:appElement/bm:namespace=namespace-
uri($property)]/@id"/>
                <xsl:choose>
                    <xsl:when test="$mapid">
                        <xsl:element name="TypeMap" namespace="no:hiof:basemapper">
                            <xsl:attribute name="xlink:type"><xsl:value-of
select="'simple'"/></xsl:attribute>
                            <xsl:attribute name="xlink:href"><xsl:value-of
select="concat('#', string($mapid))"/></xsl:attribute>
                        </xsl:element>
                    </xsl:when>
                    <xsl:otherwise>
                        <xsl:element name="meta:name-ns">
                            <xsl:element name="meta:localname">
                                <xsl:value-of select="local-name($property)"/>
                            </xsl:element>
                            <xsl:element name="meta:namespace">
                                <xsl:value-of select="namespace-uri($property)"/>
                            </xsl:element>
                        </xsl:element>
                    </xsl:otherwise>
                </xsl:choose>
            </xsl:element>
            <xsl:element name="meta:elementValue">
                <xsl:choose>
                    <xsl:when test="$isGeometry">

```

---

```

        <xsl:comment>GEOMETRY PROPERTY</xsl:comment>
      </xsl:when>
      <xsl:otherwise>
        <xsl:text disable-output-escaping="yes">&lt;![CDATA[</xsl:text>
        <xsl:copy-of select="."/>
        <xsl:text disable-output-escaping="yes">]]&gt;</xsl:text>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:element>
</xsl:element>
</xsl:when>
<xsl:otherwise>
  <xsl:if test="$WARNING_MSG">
    <xsl:message terminate="no">Empty property specification passed to
    template writePropertyMeta</xsl:message>
  </xsl:if>
</xsl:otherwise>
</xsl:choose>
</xsl:template> <!-- end template: writePropertyMeta -->

<xsl:function name="g2s:hasCoordinates">
  <xsl:param name="shape"/>

  <xsl:value-of select="boolean($shape//gml:coord | $shape//*[local-name()=
  $gAppCoords] | $shape//gml:coordinates | $shape//*[local-name()=
  $gAppCoordinates])"/>
</xsl:function>

<xsl:template name="gmlCoordinateString">
  <xsl:param name="shape"/>

  <xsl:variable name="coords" select="$shape//gml:coord | $shape//*[local-
  name()=$gAppCoords]"/>
  <xsl:variable name="coordinates" select="$shape//gml:coordinates |
  $shape//*[local-name()=$gAppCoordinates]"/>

  <xsl:choose>
    <xsl:when test="$coords">
      <xsl:for-each select="$coords">
        <xsl:value-of select="gml:X"/>
        <xsl:text>,</xsl:text>
        <xsl:value-of select="gml:Y"/>
        <xsl:if test="position() != count($coords)">
          <xsl:text> </xsl:text>
        </xsl:if>
      </xsl:for-each>
    </xsl:when>
    <xsl:when test="$coordinates">
      <!--<xsl:call-template name="Trim">-->
      <xsl:value-of select="normalize-space($coordinates)"/>
      <!--</xsl:call-template>-->
    </xsl:when>
    <xsl:otherwise>
      <xsl:if test="$WARNING_MSG"><xsl:message terminate="no">No coordinates
      recognized for element: <xsl:value-of select="local-name($shape)"/>
      </xsl:message> </xsl:if>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<!--
This template makes sure that all elements that are not handled by any of the
other defined in this document, is "silent"
-->
<xsl:template match="text()" />

<xsl:template name="addMenuDef">
  <xsl:param name="rootLayers"/>

  <menu id="layerMnu" xmlns="http://www.onemap.org/svgmenu"
  onload="GetPosition(evt);">
    <header>Action menu</header>
    <item action="ZoomIn">Zoom &in</item>
    <item action="ZoomOut">Zoom &out</item>

    <separator />

```

```
<xsl:for-each select="$rootLayers">
  <xsl:if test="$INFORMATIVE_MSG = true()"><xsl:message terminate="no">
    <xsl:value-of select="current()"/></xsl:message></xsl:if>
    <item onactivate="javascript:toggleVisibility('{current()}')"
      checked="yes">
      <xsl:attribute name="id">
        <xsl:value-of select="concat('mnu_', current())"/>
      </xsl:attribute>

      <xsl:value-of select="current()"/>
    </item>
  </xsl:for-each>

  <separator />

  <item action="OriginalView">&Original View</item>
  <item action="Quality">Improve &Quality</item>
  <item action="ViewSource">&View Source</item>
  <item action="SaveSnapshotAs">&Save SVG as ...</item>
  <separator />
  <item action="Help">&Help</item>
  <item action="About">&About 'SVG Viewer'...</item>
</menu>
</xsl:template>

<xsl:template name="defineStyles">
  <style type="text/css">
    <xsl:for-each select="$userStyles/style:styles/style:style">
      <!--<xsl:message terminate="no"><xsl:value-of select="style:namespace"/>
      </xsl:message>-->
      <xsl:choose>
        <xsl:when test="style:namespace = 'default'">
          <xsl:text>.default{</xsl:text><xsl:value-of
            select="style:stylestring"/><xsl:text>} </xsl:text>
          </xsl:when>
          <xsl:otherwise>
            <xsl:value-of select="concat('fstyle', string(position()), '{')"/>
            <xsl:value-of select="style:stylestring"/><xsl:text>} </xsl:text>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:for-each>
    </style>
  </xsl:template>

  <!--
This template writes the javascript into the SVG-document as a CDATA section.
The script-element is defined as a
CDATA-element, so that the CDATA is transferred to output.
-->
  <xsl:template name="addScript">
    <script type="text/ecmascript">
      <REPLACE_THIS_WITH_CDATA>
        var svgdoc;
        var htmlDoc;
        var GML_NAMESPACE = 'http://www.opengis.net/gml';
        //var popup;

        /*
        This toggles the visibility on and of for layers where the layerId-is
        specified.
        (optional attribute for integratedLayer, in
        http://www.onemap.org/integration-namespace)
        */
        function toggleVisibility(id) {
          var gElement = svgdoc.getElementById(id);

          if(gElement == null)
            alert('Unknown layer id.');
```

---

```

var mnuItem = svgdoc.getElementById('mnu_' + id);
if( visibility == 'hidden' ) {
    mnuItem.setAttribute("checked", "no");
}
else
{
    mnuItem.setAttribute("checked", "yes");
}

var newMenuRoot = parseXML( printNode( document.getElementById(
    'layerMnu' ) ), contextMenu );
contextMenu.replaceChild( newMenuRoot.firstChild,
    contextMenu.firstChild );
}

function getFeatureInformation(element) {
    var map = element.getElementsByTagNameNS('no:hiof:onemap:gml:metainfo',
        'TypeInformation');

    if(map.length != 0) {
        var typemap = map.item(0).getElementsByTagNameNS('no:hiof:basemapper',
            'TypeMap').item(0);
        var id = typemap.getAttribute('xlink:href');

        var current = svgdoc.getElementById(id.substring(1));

        if(current == null)
            return '';

        var info = '';

        var local =
            current.getElementsByTagNameNS('no:hiof:basemapper', 'appElement').item
            (0);
        var localname =
            local.getElementsByTagNameNS('no:hiof:basemapper', 'localname').item(0)
            .firstChild.nodeValue; //works
        var namespace =
            local.getElementsByTagNameNS('no:hiof:basemapper', 'namespace').item(0)
            .firstChild.nodeValue; //works
        info += '<table width="100%">';
        info += '    <tr>';
        info += '        <td colspan="2" class="tdcaption">Feature
        Information</td>';
        info += '    </tr>';
        info += '    <tr>';
        info += '        <td class="tdsmallcaption">Name:</td>';
        info += '        <td class="tdinfo">' + localname + '</td>';
        info += '    </tr>';
        info += '    <tr>';
        info += '        <td class="tdsmallcaption">Namespace:</td>';
        info += '        <td class="tdinfo">' + namespace + '</td>';
        info += '    </tr>';
        info += '</table>';

        var instance =
            current.getElementsByTagNameNS('no:hiof:basemapper', 'instanceOf').item
            (0);
        var instanceName =
            instance.getElementsByTagNameNS('no:hiof:basemapper', 'localname').item
            (0).firstChild.nodeValue; //works
        var instanceNS =
            instance.getElementsByTagNameNS('no:hiof:basemapper', 'namespace').item
            (0).firstChild.nodeValue; //works
        info += '<table width="100%">';
        info += '    <tr>';
        info += '        <td colspan="3" class="tdcaption">Data type information
        </td>';
        info += '    </tr>';
        info += '    <tr>';
        info += '        <td class="tdsmallcaption">Instance of: </td>';
        info += '        <td class="tdsmallcaption">Name:</td>';
        info += '        <td class="tdinfo">' + instanceName + '</td>';
        info += '    </tr>';
        info += '    <tr>';
        info += '        <td>&nbsp;</td>';
        info += '        <td class="tdsmallcaption">Namespace:</td>';
        info += '        <td class="tdinfo">' + instanceNS + '</td>';
    
```

---

```

        info += ' </tr> ';
        info += '</table>';

        return info;
    } else {
        return '';
    }
} //end getFeatureInformation(element)

function indent(astring, depth) {
    for(var i=0; i<depth; i++) {
        astring = ' ' + astring;
    }

    return astring;
} //end indent(astring, depth)

function prettyPrint(string) {
    var depth = 0;
    var newstring = '';
    firstElement = true;
    flagTextSearch = false;
    flagElement = false;
    insideStartElement = false;
    insideEndElement = false;
    insideText = true;

    for( var i=0; i<string.length; i++) {
        var char = string.charAt(i);

        if(char == '\"')
            char = '&quot;';

        if( char == '<' ) {
            flagElement = true;
            insideText = false;

            if(!firstElement) {
                newstring += '<br/>';
            } else {
                firstElement = false;
            }

            if(string.charAt(i+1) == '/') { //We need a peek, to check figure
                out the indenting
                newstring += indent( '&lt; ', depth );
            } else {
                newstring += indent( '&lt; ', depth+1 );
            }
        } else if( char == '>' ) {
            insideText = true;
            flagTextSearch = true; //This is true until first non-space char is
            read.

            if(insideEndElement)
                depth++;
            else
                depth--;

            insideStartElement = false;
            insideEndElement = false;

            newstring += '&gt;<br/>';
        } else if( char == '/' && flagElement ) {
            flagElement = false;
            insideEndElement = true;

            newstring += char;
        } else if( !insideEndElement && flagElement ) {
            insideStartElement = true;

            newstring += char;
        } else if( insideText ) {
            if(!flagTextSearch) { //If we have already-found non-space
                characters, the character is passed.
                newstring += char;
            } else {
                if(char != ' ') {

```



```
        flagTextSearch = false;
        newstring +=char;
    } else {

        //Do nothing. We don't want preciding spaces transferred to
        output.
    }
}
} else if( insideStartElement) {
    newstring += char;
} else if( insideEndElement) {
    newstring += char;
}
}

return newstring;
} //end prettyPrintString(string, depth)

function getProperties(element) {
    var props =
    element.getElementsByTagNameNS('no:hiof:onemap:gml:metainfo','Property')
    ;

    if(props == null && props.length == 0)
        return '';

    var info = '';
    info += '<table width="100%">';
    info += '    <tr>';
    info += '        <td class="tdcaption" colspan="3">Properties</td>';
    info += '    </tr>';

    //alert(props.length);
    for(var i=0; i<props.length; i++) {
        var name, namespace;
        var typemap =
        props.item(i).getElementsByTagNameNS('no:hiof:basemapper', 'TypeMap');
        var pointedTo;

        if( typemap.length != 0 ) {
            var id = typemap.item(0).getAttribute('xlink:href');
            pointedTo = svgdoc.getElementById(id.substring(1));

            name=
            pointedTo.getElementsByTagName('localname').item(0).firstChild.nodeValue;
            namespace=
            pointedTo.getElementsByTagName('namespace').item(0).firstChild.nodeValue;
        } else {
            name =
            props.item(i).getElementsByTagNameNS('no:hiof:onemap:gml:metainfo',
            'localname').item(0).firstChild.nodeValue;
            namespace =
            props.item(i).getElementsByTagNameNS('no:hiof:onemap:gml:metainfo',
            'namespace').item(0).firstChild.nodeValue;
        }

        var value =
        props.item(i).getElementsByTagNameNS('no:hiof:onemap:gml:metainfo',
        'elementValue').item(0).firstChild.nodeValue;
        value = prettyPrint(value);

        var stripStart = value.indexOf('&gt;');
        var stripEnd = value.lastIndexOf('&lt;');

        //The following block removes the first and last tag from the string,
        including the <br/>-tag. It's a bit hairy..
        {
            var stripStart = value.indexOf('<br/>');
            var lastBRloc = value.lastIndexOf('<br/>');
            if(lastBRloc != -1)
                value = value.substring(0, value.lastIndexOf('<br/>'));
            var stripEnd = value.lastIndexOf('&lt;');
        }
    }
}
```

---

```

        if(stripStart != -1) {
            value = value.substring(stripStart + 5);
        }
        if(stripEnd != -1) {
            value = value.substring(0, stripEnd);
            /*
             Strange thing. The first lastIndexOf returns a high number,
             indicating that the string contains the &lt;
            */
            stripEnd = value.lastIndexOf('&lt;');
            if(stripEnd != -1)
                value = value.substring(0, stripEnd);
        }
    }
}

/*
value = value.replace('<', '&lt;');
value = value.replace('>', '&gt;');
value = value.replace('"', '&quot;');
*/

info += '<tr>';
info += '    <td colspan="2" class="tdsmallcaption">Name:</td>';
info += '    <td class="tdinfo">' + name + '</td>';
info += '</tr>';
info += '<tr>';
info += '    <td colspan="2" class="tdsmallcaption">Namespace:</td>';
info += '    <td class="tdinfo">' + namespace + '</td>';
info += '</tr>';

info += '<tr>';
info += '    <td colspan="2" class="tdsmallcaption">Value:</td>';
info += '    <td class="tdinfo">' + value + '</td>';
info += '</tr>';

var typemap;
var gmlDerivedTypeName, instanceOfName, instanceOfNS;

if( pointedTo != null) {
    var instanceOf =
        pointedTo.getElementsByTagNameNS('no:hiof:basemapper',
            'instanceOf');

    if(instanceOf) {
        instanceOfName =
            instanceOf.item(0).getElementsByTagNameNS('no:hiof:basemapper',
                'localname').item(0).firstChild.nodeValue;
        instanceOfNS =
            instanceOf.item(0).getElementsByTagNameNS('no:hiof:basemapper',
                'namespace').item(0).firstChild.nodeValue;

        info += '    <tr>';
        info += '        <td class="tdsmallcaption">Instance of: </td>';
        info += '        <td class="tdsmallcaption">Name:</td>';
        info += '        <td class="tdinfo">' + instanceOfName + '</td>';
        info += '    </tr>';
        info += '    <tr>';
        info += '        <td>&nbsp;</td>';
        info += '        <td class="tdsmallcaption">Namespace:</td>';
        info += '        <td class="tdinfo">' + instanceOfNS + '</td>';
        info += '    </tr>';
    }
}

var gmlDerivedType =
    pointedTo.getElementsByTagNameNS('no:hiof:basemapper',
        'gmlDerivedType');
if( gmlDerivedType.length != 0 ) {
    gmlDerivedTypeName =
        gmlDerivedType.item(0).getElementsByTagNameNS('no:hiof:basemapper',
            'localname').item(0).firstChild.nodeValue;
    //gmlDerivedTypeNS =
        gmlDerivedType.item(0).getElementsByTagNameNS('no:hiof:basemapper',
            'namespace').item(0).firstChild.nodeValue;

    info += '    <tr>';
    info += '        <td colspan="2" class="tdsmallcaption">GML base

```

---

```

        type: </td>      '
        info += '        <td class="tdinfo">' + gmlDerivedTypeName +
        '</td>';
        info += '      </tr>  '
    }
}

    info += '<tr><td colspan="3">&nbsp;</td></tr>';
}

info += '</table>';

return info;
} //end getProperties(element)

function showFeatureData(id) {
    var defElement = svgdoc.getElementById(id);
    var body = getFeatureInformation(defElement);
    body += getProperties(defElement);

    if(body!='') {
        //window.parent.showModalDialog('featurewindow.html', body,
        'dialogHeight:500pt;dialogWidth:450pt;status:no;resizable:yes;help=
        no;scroll=auto');
        var oNewDoc = window.parent.open("featurewindow.html", "replace",
        'height=600,width=650,menubar=no,resizable=yes,scrollbars=
        yes,titlebar=no');
        oNewDoc.document.getElementsByTagName("p").item(0).innerHTML = body;
    } else {
        alert('No feature information available');
    }
} //end showFeatureData(id)

function setStrokeWidth(id, sw) {
    var element = svgdoc.getElementById(id);
    var legendStyle = element.getStyle();

    legendStyle.setProperty("stroke-width",sw); //set line thickness
} //end setStrokeWidth(id, sw)

    function init(evt) {
        svgdoc = evt.getTarget().getOwnerDocument();

        var newMenuRoot = parseXML( printNode( document.getElementById(
        'layerMnu' ) ), contextMenu );
        contextMenu.replaceChild( newMenuRoot.firstChild,
        contextMenu.firstChild );

        //popup = svgdoc.getElementById('aboutFeature');
    }

</script>
</REPLACE_THIS_WITH_CDATA>

</xsl:template>

<!--
This template automatically generates the html-file, used by the javascript to
display feature information.
It is dependent upon a XSLT 2.0 parser, because of the xsl:result-document -
call. This enables output
to several different files.

-->
<xsl:template name="writeInfoWindow">
    <xsl:result-document href="featurewindow.html" format="html doc">
    <xsl:text disable-output-escaping="yes">
    <!DOCTYPE HTML PUBLIC &quot;-//W3C//DTD HTML 4.01
    Transitional//EN&quot;
    &quot;http://www.w3.org/TR/html4/loose.dtd&quot;&gt;
    &lt;html&gt;
    &lt;head&gt;
    &lt;meta http-equiv=&quot;Content-Type&quot; content=&quot;text/html;
    charset=iso-8859-1&quot;&gt;

```

```
<!--title-->Feature Information</title-->
<!--script language="javascript"-->
  function setMessage()
  {
    /*var body = document.getElementsByTagName("p").item(0);
    body.innerHTML = window.dialogArguments; */
  }
</script-->

<!--style type="text/css"-->
<!--
body {
  background-color: #66CCFF;
}

.tdsmallcaption {
  background-color: #6633FF;
  font-weight:600;
  font:Arial, Helvetica, sans-serif;
  color: #FFFFFF;
  vertical-align:top;
}

.tdcaption {
  background-color: #000000;
  font-weight:600;
  font:Arial, Helvetica, sans-serif;
  font-size:16px;
  color: #FFFFFF;
  vertical-align:top;
}

.tdinfo {
  background-color:#FFFFFF;
  width:65%;
}
-->
</style-->
</head-->

<!--body onload="setMessage()"-->
  <p-->
</body-->
</html-->
</xsl:text-->
</xsl:result-document-->
</xsl:template-->
</xsl:stylesheet-->
```

## **Stylesheet included into the genericGML2SVG.xslt listed the section called “Generic GML/Dictionary to SVG transformation”**

This stylesheet defines some constant mappings to the base GML types as defined in the base GML2 schemas.

---

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:variable name="GML_NAMESPACE" select="'http://www.opengis.net/gml'"/>
  <xsl:variable name="FEATURE_BASE" select="'Feature'"/>
  <xsl:variable name="FEATURE_TYPE" select="'AbstractFeatureType'"/>
  <xsl:variable name="FEATURE_COLLECTION_BASE" select="'_FeatureCollection'"/>
  <xsl:variable name="FEATURE_COLLECTION_TYPE"
select="'AbstractFeatureCollectionType'"/>
  <xsl:variable name="FEATURE_MEMBER_TYPE" select="'FeatureAssociationType'"/>
  <xsl:variable name="FEATURE_MEMBER_ELEMENT" select="'featureMember'"/>
  <xsl:variable name="GEOMETRY_BASE" select="'_Geometry'"/>
  <xsl:variable name="GEOMETRY_PROPERTY_BASE" select="'_geometryProperty'"/>
  <xsl:variable name="GEOMETRY_PROPERTY_BASE_ELEMENT"
select="'geometryProperty'"/>
  <xsl:variable name="COORDINATES_BASE" select="'coordinates'"/>
  <xsl:variable name="COORDS_BASE" select="'coords'"/>
  <xsl:variable name="COORD_TYPE" select="'CoordType'"/>
  <xsl:variable name="COORDINATES_TYPE" select="'CoordinatesType'"/>

  <!-- Those elements substituting for _Geometry-->

  <xsl:variable name="GEOMETRY_POINT" select="'Point'"/>

  <xsl:variable name="GEOMETRY_LINEARRING_ELEMENT" select="'LinearRing'"/>
  <xsl:variable name="GEOMETRY_LINEARRING_TYPE" select="'LinearRingType'"/>

  <xsl:variable name="GEOMETRY_POINT_ELEMENT" select="'Point'"/>
  <xsl:variable name="GEOMETRY_POINT_TYPE" select="'PointType'"/>

  <xsl:variable name="GEOMETRY_POLYGON_ELEMENT" select="'Polygon'"/>
  <xsl:variable name="GEOMETRY_POLYGON_TYPE" select="'PolygonType'"/>

  <xsl:variable name="GEOMETRY_LINESTRING_ELEMENT" select="'LineString'"/>
  <xsl:variable name="GEOMETRY_LINESTRING_TYPE" select="'LineStringType'"/>

  <xsl:variable name="GEOMETRY_ASSOCIATION_TYPE"
select="'GeometryAssociationType'"/>
  <xsl:variable name="GEOMETRY_PROPERTY_TYPE" select="'GeometryPropertyType'"/>
  <xsl:variable name="GEOMETRY_PROPERTY_POINT" select="'pointProperty'"/>
  <xsl:variable name="GEOMETRY_PROPERTY_POLYGON" select="'polygonProperty'"/>
  <xsl:variable name="GEOMETRY_PROPERTY_LINESTRING"
select="'lineStringProperty'"/>
  <xsl:variable name="GEOMETRY_PROPERTY_MULTIPPOINT"
select="'multiPointProperty'"/>
  <xsl:variable name="GEOMETRY_PROPERTY_MULTILINESTRING"
select="'multiLineStringProperty'"/>
  <xsl:variable name="GEOMETRY_PROPERTY_MULTIPOLYGON"
select="'multiPolygonProperty'"/>
  <xsl:variable name="GEOMETRY_PROPERTY_MULTIGEOMETRY"
select="'multiGeometryProperty'"/>
</xsl:stylesheet>
```

---

---

# Appendix B. XML schemas

## Mapping Dictionary Schema

This schema defines the structure of a Mapping Dictionary, as it is created by the transformation mapElements.xslt found in the section called “GML Schema to Mapping Dictionary”, given a valid GML 2.1.2 Schema as input.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="no:hiof:basemapper" elementFormDefault="qualified"
attributeFormDefault="unqualified" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="no:hiof:basemapper">
  <xs:element name="MappingDictionary">
    <xs:annotation>
      <xs:documentation>The root of the mapping dictionary.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="documentNamespaces" type="Document_Namespaces"/>
        <xs:element ref="typeMaps"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="typeMaps">
    <xs:annotation>
      <xs:documentation>This element is child of the MappingDictionary-element,
and contains all the element-mappings for a schema.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="TypeMap" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="TypeMap" type="TypeMap_Type"/>
  <xs:complexType name="Document_Namespaces">
    <xs:annotation>
      <xs:documentation>A type that represents the targetNamespace and related
namespaces defined by an XML schema.</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="targetns" type="xs:uri" maxOccurs="unbounded"/>
      <xs:element name="namespace" type="xs:uri" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="TypeMap_Type">
    <xs:annotation>
      <xs:documentation>
        Type for storing a schema element with:
        - the elements name (1)
        - either: a gmlType, meaning this element is a direct instantiation of a
        GML type
        or a gmlDerivedType, meaning that this elements type is derived
        directly or indirectly from a GML type.
        - what element this element can substitute for (0/1)
        - if the substitutionGroup element, directly or indirectly substitutes
        for a GML type, the GML element (0/1)
      </xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="appElement" type="Basic_Type"/>
      <xs:element name="instanceOf" type="Basic_Type" minOccurs="0"/>
      <xs:element name="gmlDerivedType" type="Derived_Type" minOccurs="0"/>
      <xs:element name="substitutesFor" type="Basic_Type" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

```

```
    <xs:element name="baseSubstitutesFor" type="Basic_Type" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:id"/>
  <!--<xs:attribute name="mapId" use="optional" type="xs:id"/>-->
</xs:complexType>
<xs:complexType name="Basic_Type">
  <xs:annotation>
    <xs:documentation>Type to represent a basic xml type, with localname and
      namespace.</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="localname" type="xs:string"/>
    <xs:element name="namespace" type="xs:string"/>
  </xs:sequence>
  <xs:attribute name="typeId" type="xs:string"/>
</xs:complexType>
<xs:complexType name="Derived_Type">
  <xs:annotation>
    <xs:documentation>A specialization of the Basic_Type complexType, adding
      an attribute telling if this type is derived by extension or
      restriction</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="Basic_Type">
      <xs:attribute name="derivedBy" use="optional">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="extension"/>
            <xs:enumeration value="restriction"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
</xs:schema>
```

---

# Appendix C. Schema and instance document example

This example consists of a simple schema and instance document.

## dens.xsd

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xs:schema targetNamespace="http://www.dens.com" elementFormDefault="qualified"
attributeFormDefault="unqualified" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="http://www.dens.com" xmlns:den="http://www.dens.com">
  <xs:element name="gambling_dens">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="gambling_den" type="gambling_den" minOccurs="0"
maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
    <xs:key name="PK_Den">
      <xs:selector xpath="den:gambling_den"/>
      <xs:field xpath="@den_id"/>
    </xs:key>
  </xs:element>
  <xs:complexType name="gambling_den">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:any minOccurs="0"/>
      <xs:element name="machines">
        <xs:complexType>
          <xs:sequence>
            <xs:element ref="slot_machine" minOccurs="0" maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="den_id" type="xs:positiveInteger" use="required"/>
  </xs:complexType>
  <xs:complexType name="slot_machine" abstract="true">
    <xs:annotation>
      <xs:documentation>Abstract datatype defined to be super-type for any type
of slot machine in the system.</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="id">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:pattern value="[A,B,C][0-9]{3}[-][0-9]{5}"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element ref="manufacturer"/>
      <xs:any minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="no_id_slotmachine">
    <xs:annotation>
      <xs:documentation>Denne datatypen er laget for å illustrere bruk av
restriction på complexType datatyper.</xs:documentation>
    </xs:annotation>
    <xs:complexContent>
      <xs:restriction base="slot_machine">
        <xs:sequence>
          <xs:element name="name" type="xs:string"/>
          <xs:element ref="manufacturer"/>
        </xs:sequence>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>

```



```

        </xs:sequence>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="gambling_machine">
    <xs:annotation>
      <xs:documentation>Datatype for gambling slot machine, ergo machines that
        pay out prize money in certain situations.</xs:documentation>
    </xs:annotation>
    <xs:complexContent>
      <xs:extension base="slot_machine">
        <xs:sequence>
          <xs:element name="min_bet" type="xs:positiveInteger"/>
          <xs:element name="max_bet" type="xs:positiveInteger"/>
          <xs:element name="max_winnings" type="xs:positiveInteger"/>
          <xs:element name="payback_rate">
            <xs:simpleType>
              <xs:restriction base="xs:unsignedShort">
                <xs:maxInclusive value="100"/>
                <xs:minExclusive value="0"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="credit_price">
    <xs:simpleContent>
      <xs:extension base="xs:nonNegativeInteger">
        <xs:attribute name="currency" use="optional" default="USD">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="NOK"/>
              <xs:enumeration value="USD"/>
              <xs:enumeration value="EUR"/>
              <xs:enumeration value="GBP"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="arcade_game">
    <xs:complexContent>
      <xs:extension base="slot_machine">
        <xs:sequence>
          <xs:element name="credit_price" type="credit_price"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="slot_machine" type="slot_machine" abstract="true">
    <xs:key name="PK">
      <xs:selector xpath="den:slotmachine"/>
      <xs:field xpath="den:id"/>
    </xs:key>
  </xs:element>
  <xs:element name="gambling_machine" type="gambling_machine"
    substitutionGroup="slot_machine"/>
  <xs:element name="arcade_game" type="arcade_game"
    substitutionGroup="slot_machine"/>
  <xs:element name="manufacturer">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="service_phone" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>

```

# instance.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<gambling_dens xmlns="http://www.dens.com" xmlns:xsi="
http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="
http://www.dens.com dens.xsd">
  <gambling_den den_id="1">
    <name>Arcade Paradise</name>
    <machines>
      <arcade_game>
        <name>World Rally Experience</name>
        <id>A999-01000</id>
        <manufacturer>
          <name>Arcade Workshop Inc</name>
          <service_phone>111-HELP-ME</service_phone>
        </manufacturer>
        <credit_price>10</credit_price>
      </arcade_game>
      <arcade_game>
        <name>Street Fighter 2000</name>
        <id>A911-12112</id>
        <manufacturer>
          <name>Arcade Workshop Inc</name>
          <service_phone>111-HELP-ME</service_phone>
        </manufacturer>
        <credit_price>10</credit_price>
      </arcade_game>
      <arcade_game>
        <name>Space Monkey Shootout</name>
        <id>C112-11123</id>
        <manufacturer>
          <name>Abandoned Games Manufacturers</name>
          <service_phone>910-111-9721</service_phone>
        </manufacturer>
        <credit_price currency="NOK">5</credit_price>
      </arcade_game>
      <arcade_game>
        <name>Galaxy</name>
        <id>A001-09090</id>
        <manufacturer>
          <name>Montys</name>
          <service_phone>123-PAYTHEPRICE</service_phone>
        </manufacturer>
        <credit_price>10</credit_price>
      </arcade_game>
      <gambling_machine>
        <name>RipOff</name>
        <id>B119-12456</id>
        <manufacturer>
          <name>Pickpocket Pros</name>
          <service_phone>555-666-777</service_phone>
        </manufacturer>
        <min_bet>5</min_bet>
        <max_bet>50</max_bet>
        <max_winnings>300</max_winnings>
        <payback_rate>30</payback_rate>
      </gambling_machine>
    </machines>
  </gambling_den>
  <gambling_den den_id="2">
    <name>Critical Corner Casino</name>
    <machines>
      <gambling_machine>
        <name>Titanic</name>
        <id>C119-01333</id>
        <manufacturer>
          <name>Jackpot Systems</name>
          <service_phone>333-LONGDISTANCE</service_phone>
        </manufacturer>
        <min_bet>1</min_bet>
        <max_bet>10</max_bet>
        <max_winnings>2000</max_winnings>
        <payback_rate>25</payback_rate>
      </gambling_machine>
    </machines>
  </gambling_den>
</gambling_dens>
```

---

---

```
</gambling_machine>
<gambling_machine>
  <name>Motherinlaw</name>
  <id>B234-99444</id>
  <manufacturer>
    <name>Moneymakers</name>
    <service_phone>987-MACHINE</service_phone>
  </manufacturer>
  <min_bet>10</min_bet>
  <max_bet>100</max_bet>
  <max_winnings>20000</max_winnings>
  <payback_rate>10</payback_rate>
</gambling_machine>
<gambling_machine>
  <name>Pokermania</name>
  <id>A900-01555</id>
  <manufacturer>
    <name>Mercury Inc</name>
    <service_phone>666-234-567</service_phone>
  </manufacturer>
  <min_bet>10</min_bet>
  <max_bet>50</max_bet>
  <max_winnings>1000</max_winnings>
  <payback_rate>85</payback_rate>
</gambling_machine>
</machines>
</gambling_den>
</gambling_dens>
```

---

# Appendix D. GML schema and instance example

## hbn.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="no:hiof:onemap:gml:appschema:hbn" xmlns:xlink="
http://www.w3.org/1999/xlink" xmlns:gml="http://www.opengis.net/gml" xmlns:xs="
http://www.w3.org/2001/XMLSchema" xmlns="no:hiof:onemap:gml:appschema:hbn"
elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:import namespace="http://www.opengis.net/gml"
    schemaLocation="feature.xsd"/>
  <xs:import namespace="http://www.w3.org/1999/xlink"
    schemaLocation="xlinks.xsd"/>
  <xs:element name="HaldenByNight" type="HaldenByNightType"
    substitutionGroup="gml:_FeatureCollection"/>
  <xs:element name="Surroundings" type="gml:AbstractFeatureCollectionType"
    substitutionGroup="gml:_FeatureCollection"/>
  <xs:element name="nightSiteMember" type="NightSiteMemberType"
    substitutionGroup="gml:featureMember"/>
  <xs:element name="NightSiteBar" type="NightSiteBarType"
    substitutionGroup="_NightSiteFeature"/>
  <xs:element name="NightSiteKebabStore" type="NightSiteKebabStoreType"
    substitutionGroup="_NightSiteFeature"/>
  <xs:element name="_NightSiteFeature" type="gml:AbstractFeatureType"
    abstract="true" substitutionGroup="gml:_Feature"/>
  <xs:element name="buildingOutline" type="LinearRingPropertyType"
    substitutionGroup="gml:_geometryProperty"/>
  <!--
the element myCoordinates are representet here, to prove the point that you
are free to define aliases for basic gml element.
This is by far not a recommended thing to do, but it's still possible, and in
some
-->
  <xs:element name="myCoordinates" type="gml:CoordinatesType"
    substitutionGroup="gml:coordinates"/>
  <xs:element name="River" type="RiverType" substitutionGroup="gml:_Feature"/>
  <xs:complexType name="HaldenByNightType">
    <xs:complexContent>
      <xs:extension base="gml:AbstractFeatureCollectionType">
        <xs:attribute name="lastupdated" type="xs:dateTime" use="optional"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="NightSiteMemberType">
    <xs:complexContent>
      <xs:restriction base="gml:FeatureAssociationType">
        <xs:sequence minOccurs="0">
          <xs:element ref="_NightSiteFeature"/>
        </xs:sequence>
        <xs:attributeGroup ref="xlink:simpleLink"/>
        <xs:attribute ref="gml:remoteSchema" use="optional"/>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="NightSiteType" abstract="true">
    <xs:complexContent>
      <xs:extension base="gml:AbstractFeatureType">
        <xs:sequence>
          <xs:element ref="buildingOutline"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="NightSiteBarType">
    <xs:complexContent>
      <xs:extension base="NightSiteType">
```

```
<xs:sequence>
  <xs:element name="age_limit" type="xs:nonNegativeInteger"/>
  <xs:element name="beer_price">
    <xs:simpleType>
      <xs:restriction base="xs:double">
        <xs:minExclusive value="0"/>
        <xs:maxInclusive value="100"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>
<xs:complexType name="NightSiteKebabStoreType">
  <xs:complexContent>
    <xs:extension base="NightSiteType">
      <xs:sequence>
        <xs:element name="kebabprice" type="xs:int"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="RiverType">
  <xs:complexContent>
    <xs:extension base="gml:AbstractFeatureType">
      <xs:sequence>
        <xs:element ref="gml:centerLineOf"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="Street">
  <xs:complexContent>
    <xs:extension base="gml:AbstractFeatureType">
      <xs:sequence>
        <xs:element ref="gml:centerLineOf"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="LinearRingPropertyType">
  <xs:annotation>
    <xs:documentation>
      Encapsulates a LinearRing, to be used as a geometric property
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:restriction base="gml:GeometryAssociationType">
      <xs:sequence minOccurs="0">
        <xs:element ref="gml:LinearRing"/>
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
</xs:schema>
```

## halden1.xml

Instance document of application schema hbn.xsd listed above.

```
<?xml version="1.0" encoding="UTF-8"?>
<HaldenByNight xmlns="no:hiof:onemap:gml:appschema:hbn"
  xsi:schemaLocation="no:hiof:onemap:gml:appschema:hbn hbn.xsd" xmlns:gml="
  http://www.opengis.net/gml" xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <gml:boundedBy>
```

---

```
<gml:Box>
  <gml:coord>
    <gml:X>0.5</gml:X>
    <gml:Y>0.5</gml:Y>
  </gml:coord>
  <gml:coord>
    <gml:X>44</gml:X>
    <gml:Y>36</gml:Y>
  </gml:coord>
</gml:Box>
</gml:boundedBy>
<gml:featureMember>
  <Surroundings>
    <gml:boundedBy>
      <gml:Box>
        <gml:coord>
          <gml:X>0.5</gml:X>
          <gml:Y>0.5</gml:Y>
        </gml:coord>
        <gml:coord>
          <gml:X>44</gml:X>
          <gml:Y>36</gml:Y>
        </gml:coord>
      </gml:Box>
    </gml:boundedBy>
  </gml:featureMember>
  <River>
    <gml:name>Tista</gml:name>
    <gml:centerLineOf>
      <gml:LineString>
        <myCoordinates>8.0 0.5, 18.0 8.0, 19.0 14.0, 24.0 20.0, 30.0 22.0,
          32.0 26.0, 34.0 36.0</myCoordinates>
      </gml:LineString>
    </gml:centerLineOf>
  </River>
</gml:featureMember>
</Surroundings>
</gml:featureMember>
<nightSiteMember>
  <NightSiteBar>
    <gml:name>Hr. Dietz</gml:name>
    <buildingOutline>
      <gml:LinearRing>
        <gml:coord>
          <gml:X>4.0</gml:X>
          <gml:Y>8.0</gml:Y>
        </gml:coord>
        <gml:coord>
          <gml:X>4.0</gml:X>
          <gml:Y>10.0</gml:Y>
        </gml:coord>
        <gml:coord>
          <gml:X>6.0</gml:X>
          <gml:Y>10.0</gml:Y>
        </gml:coord>
        <gml:coord>
          <gml:X>6.0</gml:X>
          <gml:Y>12.0</gml:Y>
        </gml:coord>
        <gml:coord>
          <gml:X>8.0</gml:X>
          <gml:Y>12.0</gml:Y>
        </gml:coord>
        <gml:coord>
          <gml:X>8.0</gml:X>
          <gml:Y>8.0</gml:Y>
        </gml:coord>
        <gml:coord>
          <gml:X>4.0</gml:X>
          <gml:Y>8.0</gml:Y>
        </gml:coord>
      </gml:LinearRing>
    </buildingOutline>
    <age_limit>18</age_limit>
    <beer_price>35</beer_price>
  </NightSiteBar>
</nightSiteMember>
<nightSiteMember>
  <NightSiteBar>
```

---

```

    <gml:name>Gamle Krogs</gml:name>
    <buildingOutline>
      <gml:LinearRing>
        <gml:coordinates>22.0,24.0 28.0,24.0 28.0,28.0 22.0,28.0
        22.0,24.0</gml:coordinates>
      </gml:LinearRing>
    </buildingOutline>
    <age_limit>18</age_limit>
    <beer_price>35</beer_price>
  </NightSiteBar>
</nightSiteMember>
<nightSiteMember>
  <NightSiteKebabStore>
    <gml:name>Lunchbaren</gml:name>
    <buildingOutline>
      <gml:LinearRing>
        <gml:coord>
          <gml:X>34.0</gml:X>
          <gml:Y>18.5</gml:Y>
        </gml:coord>
        <gml:coord>
          <gml:X>34.0</gml:X>
          <gml:Y>20</gml:Y>
        </gml:coord>
        <gml:coord>
          <gml:X>40.0</gml:X>
          <gml:Y>20.0</gml:Y>
        </gml:coord>
        <gml:coord>
          <gml:X>40.0</gml:X>
          <gml:Y>18.0</gml:Y>
        </gml:coord>
        <gml:coord>
          <gml:X>38.0</gml:X>
          <gml:Y>18.0</gml:Y>
        </gml:coord>
        <gml:coord>
          <gml:X>38.0</gml:X>
          <gml:Y>17.0</gml:Y>
        </gml:coord>
        <gml:coord>
          <gml:X>36.0</gml:X>
          <gml:Y>17.0</gml:Y>
        </gml:coord>
        <gml:coord>
          <gml:X>36.0</gml:X>
          <gml:Y>18.5</gml:Y>
        </gml:coord>
        <gml:coord>
          <gml:X>34.0</gml:X>
          <gml:Y>18.5</gml:Y>
        </gml:coord>
      </gml:LinearRing>
    </buildingOutline>
    <kebabprice>50</kebabprice>
  </NightSiteKebabStore>
</nightSiteMember>
</HaldenByNight>

```