Critical Errors in OGC's GML 3.1.0

Preface

This page was added prior to the release of GML version 3.1.1. At this point, the deficiencies described herein should be taken to warn unwary users away from any GML3 prior to 3.1.1. The last comment on this page is an early evaluation of 3.1.1 with a set of validating parsers. It appears that the DOM style parsers fare better than the SAX style parsers, but GML 3.1.1 is indeed expressed in correct XML schema. In any case, due to the facts mentioned below, one should be extremely skeptical of any product, person, or label which claims compatibility with a version of GML3 prior to 3.1.1. As far as usability is concerned, 3.1.1 is the *first* (and the *only*) GML3.

GML3 Release Status as of October 13, 2005

GML 3.1.1 has been released. This occurred sometime between August 19, 2005 and now. However, there seems to be a "housekeeping error" with respect to the link on the specification page. The zipfile which is downloaded by the link on the OGC Webpage still contains the old documentation for 3.1.0. I have not compared the schemas to see if they have been updated. However, the schemas for 3.1.1 are hosted on the OGC schemas website.

Introduction

If my understanding is correct, this document presents fatal flaws in GML 3.1.0 (and probably 3.0.0). These flaws manifest themselves in the Implementation Specification as invalid XML Schema. As the schema are invalid, it is of course impossible to produce a "technically valid" GML 3.1.0 instance document. Worse, the errors are of a "nonsensical" nature, meaning that the actual schema attempts to express concepts which just do not make sense. If I am correct, the fix to these errors requires a human to read the schema in conjunction with the specification document to produce valid XML Schema which conforms to the "perceived intent" of the specification.

This document is structured as follows:

- a presentation of the relevant portions of XML Schema. This section summarizes my understanding of the W3C specification on specific topics relevant to this discussion. If my understanding in this area is correct, the problems with GML are very real. If my understanding is incorrect, GML may still have a chance of being usable.
- an overview of the structure of GML 3. This section presents the components of GML3 and the particular collection of modules associated with the coordinateReferenceSystem entry point to the schema. This is where I found the problems. I cannot make any statements about problems in other areas of the specification as I haven't looked anywhere else.
- a presentation of the problems exhibited by the GML implementation specification. There are two ubiquitous errors reported by the schema parsers which seem to be caused by one conceptual problem on the part of the schema author. An example of this conceptual error is given.

XML Schema Overview

NOTE: As used in this document, "XML Schema" refers to the W3C standard.

This section summarizes my understanding of the W3C specification on specific topics relevant to this discussion. In particular, the XML Schema mechanisms for the following concepts are explored:

- relationship of types to elements
- type inheritance

- element substitution groups

It is good to remind the reader at this point that XML is a *markup* and not a *programming* language. By itself, a well formed XML document does not need to conform to any schema. In order to be well formed, the XML document must have a single root element and each element must have matching opening and closing tags. XML Schema is an XML application to define legal combinations of elements and legal places for elements with certain names to appear. XML Schema also provides ways to specify legal values for elements. An XML application (e.g., GML) expressed in XML Schema defines a framework that XML documents can use to store information. These XML documents can be validated against the grammar and constraints specified by the XML application (e.g., GML). Data containing documents can then be said to be conformant or noncompliant with respect to the XML application (e.g., GML). It is entirely possible for an XML document to be well-formed but noncompliant with respect to a particular application.

This is the crux of the problem: when an XML Application like GML is expressed with an invalid XML Schema, it is impossible to construct an XML document which conforms to the standard. Since XML Schema is all about specifying which tags are legal where (and what they are allowed to contain when they do appear), an invalid XML application might as well not exist. The whole point of GML is to encourage interoperability by providing one unique expression of spatial concepts, permitting vendor neutral access to data. Invalid XML Schema in the standard is more than just inconvenient or difficult to work with: it defeats the purpose of having the standard.

Types and Elements

Unlike the programming languages of which I am aware, XML Schema decouples types from the elements which possess type attributes. On the other hand, XML Schema is not a dynamically typed language (like Python) either. It is nearly correct to state that an XML Schema type is like a *typedef* in C, but this also falls short of capturing the concept.

XML Schema is a statically typed language. The concept which has been giving me fits is that the objects possessing type information ("elements") are not capable of containing data. The very closest analog I can draw is the C *typedef* may be considered the association of an XML schema type with an XML schema element. Nothing resembling a "variable" is present in XML schema. "Variables" are always instantiated and initialized in one atomic operation by the "instance" XML document which conforms to the schema. Furthermore, "variables" in XML instance documents are always anonymous no-name-having things. (sidebar: One could possibly consider the concept of a "key" to be analgous to a "variable name", but that's not relevant to the discussion.)

Having satisfied the need to identify what composes the traditional notion of a "Type" and what constitutes a "variable", a discussion of inheritance can ensue. It turns out that in addition to being a statically typed language, XML Schema is a semi-object-oriented language. The prefix "semi" is due to the fact that the traditional notion of a "Type" is composed of two things (an XML Schema type and an XML Schema element) and these two things behave differently. This is the topic of the next section.

Type inheritance and Element Substitution Groups

An XML Schema type possesses inheritance properties, but an XML schema element does not. Because these two objects, taken together, are what allow the expression and storage of data in an XML instance document, and because these two objects have different properties, XML schema possess semi-object-orientated concepts.

I will remind the reader once again (because I find myself continually falling into a programming language mindset) that XML is a markup language and not a programming language. In this context, inheritance means inheritance of markup and not inheritance of methods to operate on data. This is XML Schema's mechanism to re-use collections, sequences and choices of tags in multiple contexts.

XML Schema type inheritance

XML Schema types possess a knowledge of inheritance. Unlike Java and C++, child types are derived from parent types by *either* extension or restriction. Extension works much as you would expect: any markup in the child type is appended to the parent type. Restriction works by any or all of the following mechanisms:

- the child may restrict the values permitted in tags allowed by the parent
- the child may restrict the number of times that tags permitted by the parent may appear

In all cases, the child may only specify constructs which are permitted by the parent.

An important property of little-t type inheritance in XML Schema is that little-t child types may be used in place of little-t parent types within the schema definition itself.

Substitution Groups

XML Schema elements possess no knowledge of inheritance. They do, however possess a non-heirarchical notion of interchangability. This concept is known as a "substitution group". An element is allowed to specify another element with which it aspires to be interchangable. When many elements aspire to be interchangable with the same element (known as the *head*), this collection of elements is known as a substitution group.

Note the parallel structure of the two components of a Traditional Type:

- little-t types may specify a parent
- elements may specify a "head" to which it aspires to be interchangable

XML Schema imposes constraints on an element definition such that the element's little-t type and substitution group properties must be consistent. If element LEAF declares element HEAD in it's substitution group, then element LEAF must either possess the same little-t type as HEAD, or it must possess a child of HEAD's little-t type.

An important property of element substitution groups is that they are *not* used within the schema definition itself (it's only *declared* there). The substitution group is exercised by the instance document and *not* the schema definition.

Semi-object-oriented-ness

Programmers using statically typed object oriented programming languages like C++ or Java expect to be able to use subclasses wherever a superclass is required. This is because the subclass is guaranteed to have at least the functionality of the superclass. (This is where it is important to remember that XML Schema is not a programming language.) An XML Schema definition does *not* permit this substitution. The substitution is allowed to occur in the XML "instance" document which contains the data, not in the schema document which contains the definition.

Let's take the programmers view of XML Schema to see where it goes wrong. Say we have an element named LEAF which is a legal member of the substition group HEAD. Let's say these are building blocks for a larger vocabulary.

Using good object-oriented design, we want to make a general purpose BASE element which contains HEAD (among other items). We also want to make a DERIVED element, "subclassed" from BASE, which address a more specific concern. In the following, remember these facts:

- BASE and DERIVED are little-t types
- HEAD and LEAF are elements
- HEAD and LEAF have valid declarations which are not shown.
- LEAF is a member of the HEAD substitution group

Here is the definition of BASE:

Definition of BASE

```
<xsd:complexType name="BASE">
  <xsd:sequence>
    ... other elements ...
    <xsd:element ref="HEAD">
    ... other elements ...
  </xsd:sequence>
</xsd:complexType>
```

A programmer might be tempted to "subclass" BASE like this:

Incorrect definition of DERIVED

```
<xsd:complexType name="DERIVED">
  <xsd:restriction base="BASE">
    <xsd:sequence>
      ... other elements ...
      <xsd:element ref="LEAF">
      ... other elements ...
    </xsd:sequence>
  </xsd:restriction>
</xsd:complexType>
```

The important line to note is the substitution of the LEAF element for the HEAD element. This is *probably* illegal. The reason I say probably is that I cannot find anything in the W3C spec which allows (or forbids) it, but it doesn't seem to pass the schema validators I've tried. The two paragraphs devoted to XML Schema Substitution Groups in *XML in a Nutshell* specifically mention that their use is in instance documents.

GML 3 Overview

GML 3 adds significant capability to GML 2. Whereas GML 2 was able to express simple features, GML 3 is capable of expressing most (if not all) of the concepts embodied in the OGC specifications, including coordinate reference systems and all the associated components. GML 3 has seven top-level entry points which do not depend on the

other six entry points. This was done in order to reduce the size of the schema for applications which are only concerned with specific subtopics. The organization is depicted by the following illustration, which is a screen-capture of a page out of the specification:

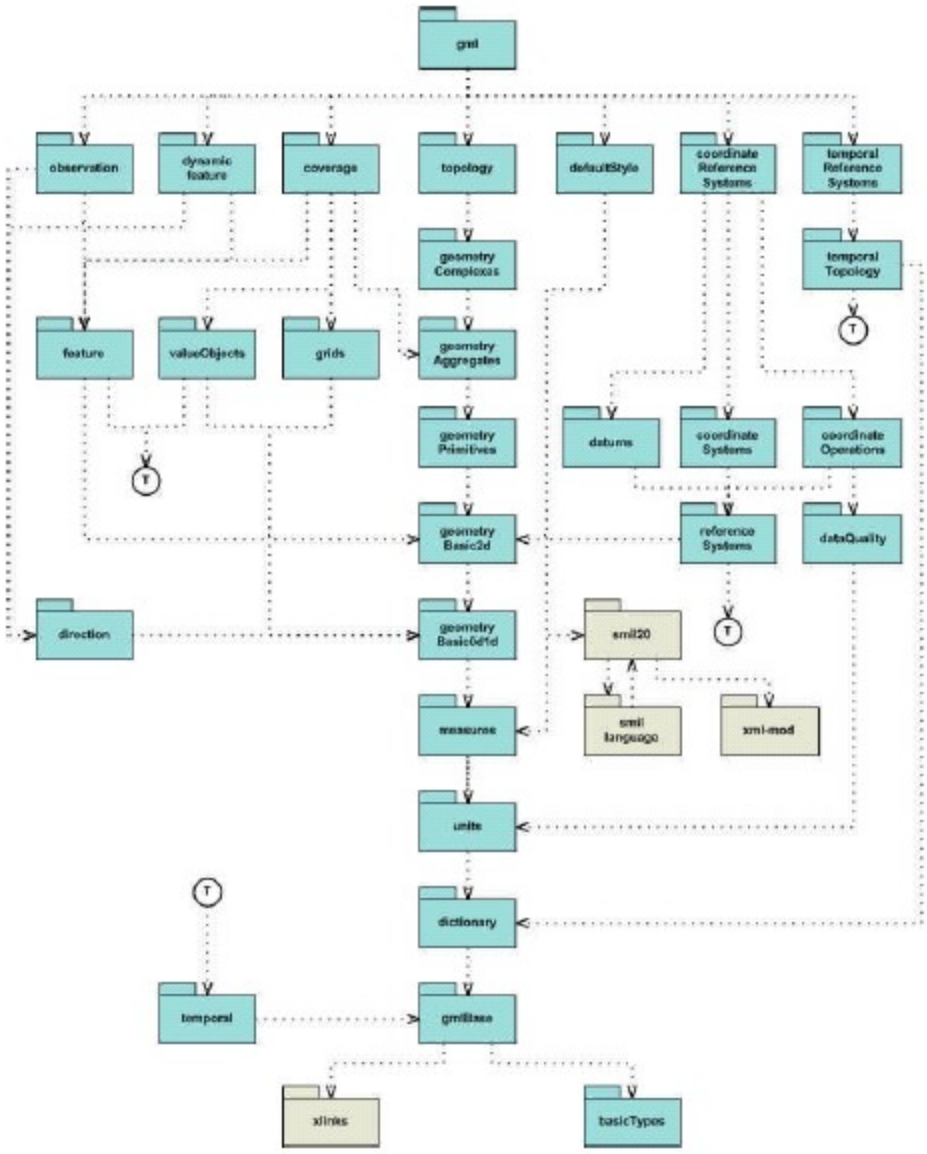

Figure 11 — Schema Dependencies

This discussion concerns only those elements in the path which begins at the coordinate reference system top level element. The schema and the PDF document are available for download [from the OGC website](#).

GML 3 Problem Areas

Feeding a top level object to an XML schema validator typically yields a suspiciously round number of errors indicative of an artificial limit to the number of errors reported. To determine the root cause of the errors, I began at the base of the heirarchy (basicTypes.xsd) and traversed the tree towards coordinateReferenceSystem.xsd, fixing the errors as I went. I did not make it past referenceSystem.xsd.

In spite of the fact that there are a lot of errors, they seemed to be grouped primarily in two types of error:

Error 1

NameAndTypeOK: The Element name/uri in restriction does not match that of corresponding base element

and

Error 2

Recurse: There is not a complete functional mapping between the particles

There is also the occasional "Unique Particle Attribution" (UPA) error. The UPA error can indicate the situation where the same markup is called for more than once and it would not be clear, reading the document, which rule called for it.

Error 1 and Error 2 are both spawned by incorrect usage of type inheritance. Frequently, the type inheritance is incorrect because it does not express a concept which makes sense. An example of this will be given in a section to follow.

One easy fix

In the navigation of the schema tree, the first error encountered is in gmlBase.xsd:

MetaDataPropertyType

```
<complexType name="MetaDataPropertyType">
  <annotation>
   <documentation>Base type for complex
metadata property types.</documentation>
  </annotation>
  <choice minOccurs="0">
   <element ref="gml:_MetaData"/>
   <any processContents="lax"/>
  </choice>
  <attributeGroup
ref="gml:AssociationAttributeGroup"/>
  <attribute name="about" type="anyURI"
use="optional"/>
 </complexType>
```

This is a UPA error and the error is in the "any" tag. A validating parser will not be able to tell what rule to use to accept a gml:_MetaData element. This is important because the "any" tag may have different constraints than the element tag. This is easy to fix because the intent of the schema authors is obvious and it's just a matter of fixing syntax.

Fix this one by adding

> *namespace="##other"*

as a parameter to the "any" tag.

Example of bad inheritance

Aside from this one fix in gmlBase.xsd, one can make it all the way up the tree to referenceSystem.xsd without incident. Unfortunately, this is where it gets hard. The problems start with AbstractReferenceSystemBaseType. They then infect the entire heirarchy based on this type.

In the following, I present the parent type for AbstractReferenceSystemBaseType (DefinitionType), as well as the incorrect attempt to subclass DefinitionType. The first thing to note is that the AbstractReferenceSystemBaseType is derived by restriction. This means, among other things, that if an element in the parent is not present in the child, then that element is not allowed. It also means that if an element is required by the parent, then it must also be present in the child.

This error type 2 occurs because the child does not include the "name" property.

AbstractReferenceSystemBaseType

```
<complexType name="DefinitionType">
  <annotation>
     <documentation>...</documentation>
  </annotation>
  <complexContent>
    <restriction base="gml:AbstractGMLType">
    <sequence>
      <element ref="gml:metaDataProperty"
minOccurs="0" maxOccurs="unbounded"/>
      <element ref="gml:description"
minOccurs="0"/>
      <element ref="gml:name"
maxOccurs="unbounded"/>
    </sequence>
      <attribute ref="gml:id"
use="required"/>
    </restriction>
  </complexContent>
</complexType>

...
<complexType
name="AbstractReferenceSystemBaseType"
abstract="true">
  <annotation>
    <documentation>...</documentation>
  </annotation>
```

```
  <complexContent>
    <restriction base="gml:DefinitionType">
      <sequence>
        <element ref="gml:metaDataProperty"
minOccurs="0" maxOccurs="unbounded"/>
        <element ref="gml:remarks"
minOccurs="0">
          <annotation>
            <documentation>Comments on or
information about this reference system,
including source information.
</documentation>
          </annotation>
        </element>
        <element ref="gml:srsName"/>
      </sequence>
      <attribute ref="gml:id"
use="required"/>
```

```
    </restriction>
  </complexContent>
</complexType>
```

Actually, there are two elements of the parent which are not referred to in the child: description and name. Name is the only one which causes the error because the parent requires the presence of a name, but does not require the presence of a description. To top it all off, this specification attempts to *add* two elements which are not present in the parent. Stop and think about that. When deriving from the parent by "restriction", lets add markup that the parent doesn't have.

Further investigation reveals the following facts:

- gml:srsName is part of the substitution group with gml:name at the head.
- gml:remarks is part of the substitution group with gml:description at the head.

The schema author is obviously a programmer. They're trying to use the subclass in the place where the class goes. The problem is twofold: XML is not a programming language; and being a member of a substitution group is not a direct analog to being a subclass.

The schema compiler is not going to let this fly for the same reason that this problem is not easy to fix: it amounts to a replacement of the parent's markup with the child's markup. The schema author wants to change the name of the description tag to "remarks", and change the name of the "name" tag to "srsName". Since this can be broken down into two steps: removal of the old tags and insertion of the new tags, it cannot logically happen under the auspices of "restriction". One cannot implement this renaming of attributes in less than two steps (a restriction followed by an extension). Even if one were to do this two step procedure, the parent will not permit the child to eliminate the "name" element.

Much the same mechanism is responsible for the type 1 errors, and the difference between the two seems to be whether the child is eliminating a required parent element or not. Presumably, an error analgous to this example is responsible for all the type 1 and type 2 errors regurgitated in the validation process.

How to proceed?

GML, as an XML Schema application, attempts to make extensive use of object-oriented concepts like re-use by inheritance. Unfortunately, XML Schema is only semi-object-oriented: the conforming instance documents can make use of the inheritance tree, but the schema definition itself cannot. In particular, one cannot "override" parent elements by child elements in the schema definition.

In general, between an analysis of the schema, reference to the specification, and some conservative guesswork, it should be possible to divine the intent of the specification authors. In the example provided, I posit that the author intends to rename "description" to "remarks" and "name" to "srsName". In the instances where it is not possible for a human to intuit the intent, reasonable decisions could be made about how to represent the data.

It is certainly possible to make a working XML schema for GML, but who should do so remains an open question. The OGC supports open standards, and releases the standards under an open-source-like model, but does not have an open development model. This may preclude the user community from fixing the schema and submitting it as a "patch" to the OGC.

Additionally, the rumor is that 3.1.1 is going to be released "real soon now" (note these rumors started in Nov. 2004.) and the scuttlebutt is that 3.1.1 schemas actually validate cleanly.

There are a number of options for going forward:

1. Wait for the OGC to release 3.1.1. See if errors are still there.
2. Fix the schemas ourselves, submit to OGC, and call it "FixedGML" until it's accepted.
3. Volunteer to be GML beta testers for release 3.1.1.

Please add comments to this page with your thoughts.