NeXML: automating generation of an XML schema and EMF Java bindings from the XML templates of the NeXus neutron, x-ray, and muon science data format

*A **webel.com.au** Scientific IT Consultancy technical report*

Presented to the NeXus International Advisory Committee (NIAC), Sydney, Australia, 30 Oct 2008

Dr Darren R. C. Kelly,
webel.com.au,
PO Box 1816, Bondi Junction, NSW 1355, Australia
darren@webel.com.au

Abstract

It is demonstrated that the XML base class templates (a.k.a. “meta-DTDs”) of the NeXus neutron and x-ray science data format can - with the aid of the Java XSD API of the Eclipse Foundation XSD project - be transformed into an XML Schema, thus introducing convenient validation of NeXus XML instance files, with bindings to Java EObjects through the XML schema import mechanism of the Eclipse Modelling Framework (EMF), which also provides a generative, validating, schema-aware XML model instance editor.

The approach is limited by known inconsistencies in the current NeXus templates and by some aspects of the NeXus class design. The transformation process affords powerful diagnosis of and identification of inconsistencies in the NeXus XML templates. Recommendations are made here for changes to the NeXus template design and format to support this generative approach, so that the NeXus International Advisory Committee (NIAC) could in principle continue to maintain adapted forms of the NeXus XML templates, rather than moving the base class definition effort completely to the XML Schema approach, given that the XML Schema could then be generated after the method shown here. The NeXML approach optionally leverages the prototype NeXus.xsd base schema, and some recommendations are also made here for changes to that base schema to admit more robust automated generation of NeXus XML Schema classes, and stronger validation criteria.

It is shown that instruments and experiments can be represented graphically in the Unified Modelling Language (UML) as UML StructuredClassifiers and as UML InstanceSpecifications, using reverse engineered XML Schema types as UML Classes with structural UML Properties, UML Stereotypes and XML Schema annotations.

---

14 Nov 2008: Postscript to NIAC2008

At the NIAC2008 it was decided to evolve the NeXus XML templates from the Meta-DTD form to an XML-schema friendlier form that can be validated against a meta-schema to be called the NeXus Definition Language (NXDL). The highly automated, generative NeXML approach presented in this technical report can - with minor adaption - be equally applied to the NXDL format XML templates to create a consistent XML schema for NeXus, EObject Java bindings, and a validating EMF model instance editor for the next generation of NeXus.

Table of Contents

1) Introduction

The definition of the base classes of the NeXus neutron and x-ray science data format [1] is currently recorded as XML templates (a.k.a. “Meta-DTDs”) that are supposed to conform to the stated design rules of NeXus, although no truly automated digital validation of XML instances against the format as a schema is currently possible.

From 2005-2007 Kelly and Hauser [3] developed a prototype Java engine for transforming digested NeXus XML templates into forward-engineered Java classes meeting the JavaBeans[15] pattern, which were denoted alpha and beta *NeXusBeans* and promoted through the online NeXML[4] child project site, which is dedicated to providing improved XML Schema[7], Java, and UML[5] tools supporting the parent NeXus project. The generated NeXusBeans classes were reverse-engineered into Unified Modelling Language (UML)[5] models and diagrams to produce the first graphical schema models for NeXus, and also the first graphical NeXus representations of instances of neutron beam instruments, using UML StructuredClassifiers with UML Properties in composite structure diagrams and also UML InstanceSpecifications in object diagrams.

However, it was found that automating transformation of the UML-based schema to an XML Schema compatible with NeXus, and that representing multiplicities and complex dimensionality in Plain Old Java Objects (POJOs), was not trivial, due in part to the use of Java5 generics[10] in the data items, but primarily due to the intrinsic complexity of the NeXus template dimensionality indicators. Kelly demonstrated in 2006 the transformation of UML representations of the structural aspects of Java NeXusBeans to a UML representation of an XML Schema, which was then forward-engineered to the first attempt at an inheritance-aware XML Schema for NeXus, however it did not satisfactorily capture the complex data content. Kelly and Rayner[1] also demonstrated[2] automated, schema-aware serialisation and persistence of XML instances using the XStream Java API[8], and thus the first true validation system for structural aspects of the NeXus format in XML, although this required substantial intervention into the XStream serialization mechanism, esp. concerning multi-dimensional data and multiplicities of elements.

Eclipse Modelling Framework (EMF) Eobjects [6], in combination with the Eclipse XSD API Error: Reference source not found provide a promising alternative strategy, whereby the XML Schema is generated first, and then Java bindings as EObjects are generated by EMF, which bindings already cater for multiplicities through EMF annotations, and for serialization to XML subject to the loaded XML Schema through an automated schema mapping and EMF “genmodel”, thus alleviating much of the customisation employed in the prototype versions of Java NeXusBeans. The EMF system is able to create a validating editor along with the Java bindings for a wide range of XML Schema, and this is demonstrated successfully below for the NeXML schema generated from transformation of NeXus XML templates with the Eclipse XSD API, including restrictions on units and types per data item.

The automatically generated XML Schema introduced here shall be denoted the *NeXML Schema*. It optionally imports and uses selected aspects of Akeroyd's hand-written `NeXus.xsd` base schema [9], although NeXML is able - in principle - to function completely independently.

Non-Java users are free to interact directly with the generated NeXML Schema using any desired language or XML tool[3]; Java users can take advantage of the EMF-based NeXusBeans bindings.

In all cases, UML graphical engineering proved a powerful complementing technology, and some UML diagrams will be given, reverse engineered from both the XML Schema directly and from the automatically generated EObject Java bindings - with a focus on the XML Schema in this report.

While it is hoped that through this automated transformation recipe an adapted form of the NeXus XML templates can be used in the future in combination with the NeXML system, it will be seen that the approach is still compromised by many inconsistencies and problems in the NeXus design

1 Hugh Rayner, Year in Industry Student 2006, NBI Computing and Electronics, ANSTO, OPAL Facility

2 At NIAC2006 and during visits by various NIAC members to the ANSTO OPAL facility 2005-2007.

3 Indeed much of the XML Schema work presented here was done in the Netbeans IDE XML Schema editor.

and templates, and recommendations are given to improve the templates to admit better transformation to an XML Schema. The NeXML transformation strategy proves a powerful diagnostic of the robustness of the NeXus class design and of problems in the XML class templates.

1.1) Notation

The following namespaces and prefixes are used:

xsd	= http://www.w3.org/2001/XMLSchema	
nx	= http://definition.nexusformat.org/schema/3.0	(`NeXus.xsd`)
wca	= http://www.webel.com.au/nexml	(`NeXML.xsd`, generated)

(Another possible candidate for the NeXML schema is 'nxl', to distinguish it from 'nx' for NeXus.)

Generated XSD types are explicitly suffixed with 'Type', however this verbose policy may change.

1.2) Overview of the NeXML transformation process

The NeXML transformation process involves the following main steps (simplified):

1. Reading and parsing of all NeXus XML templates (either from filesystem or streamed directly from the NeXus SVN repository).
2. Generation of XSD definitions for abstract base classes specific to the NeXML Schema.
3. Generation of XSD import for the `NeXus.xsd` (leveraged in selected cases).
4. Looping over all XML parsings of the XML templates to generate XSD complex types corresponding to the NeXus class of each XML template.
5. Looping over XML representations of all child elements of each NeXus class:
    1. Extraction and interpretation of multiplicity indicators and enumerations from embedded documentation.
    2. Extraction and interpretation of types and (in the case of data items) dimension indicators.
    3. Generation of group reference elements.
    4. Generation of data item reference elements, if required with reference to dedicated per-data-item types to encapsulate limits on value types, dimensions and units.

Currently the generator writes to a single file `NeXML.xsd`, however individual files within the same namespace could also be written.

1.3) The NeXML transformer leveraging the Eclipse XSD API

The prototype Java transformer for NeXML will not be presented in detail here; future versions of the code, UML diagrams, and the Java API will be provided via the NeXML web site[4] . A very brief overview in UML is provided;

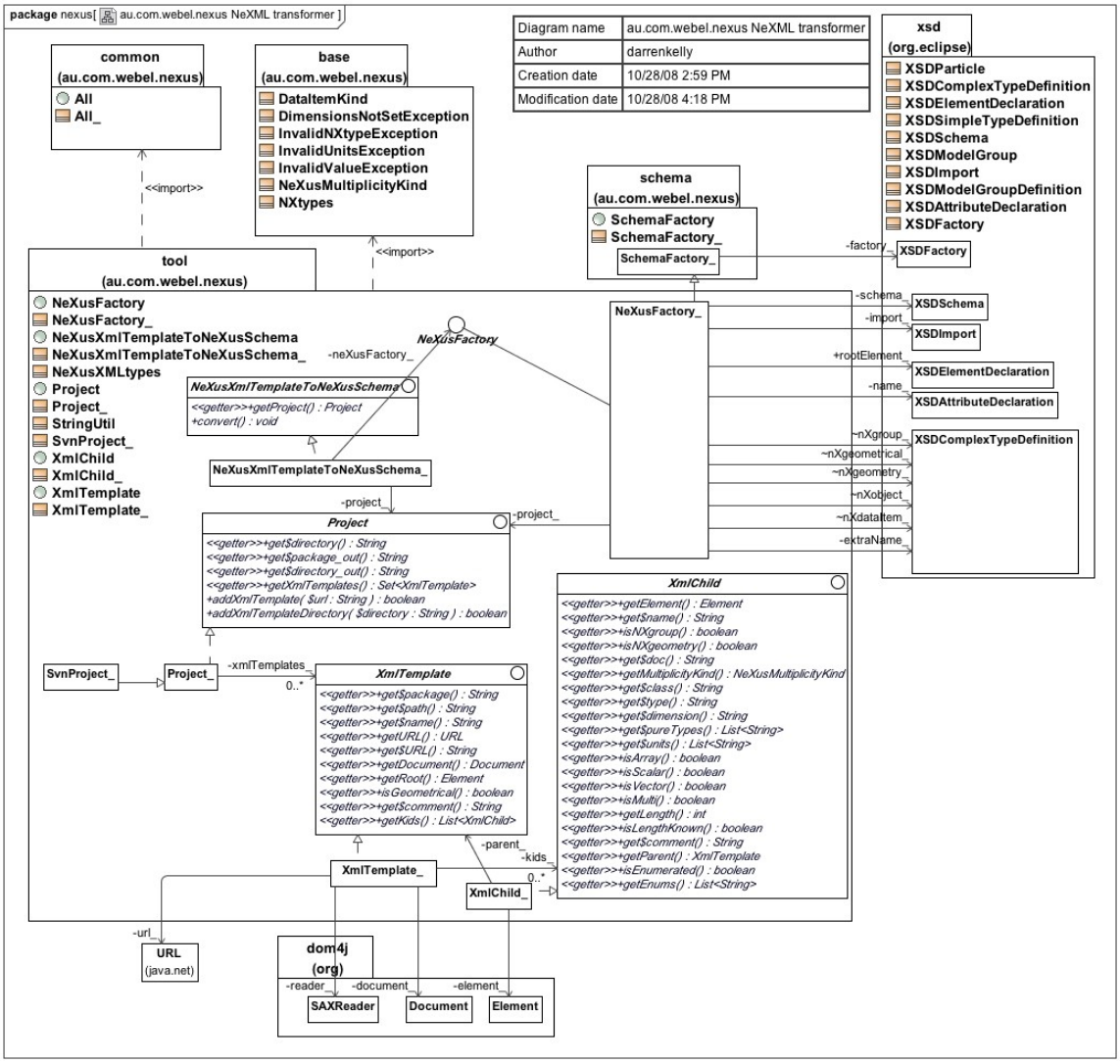


Illustration 1: Reverse-engineered UML overview of the NeXML transform Java classes

1.4) The NeXML base classes

Directly following on from the approach already introduced successfully in the prototype Java NeXusBeans[4], the NeXML generator provides for a number of base abstractions to promote reuse through inheritance and improved organisation of the type hierarchy[5]. However, no way was found to combine the strategies of per-data-item restrictions on 'units' and data list lengths and value enumerations (introduced below) with inheritance of a `DataItemType` base.

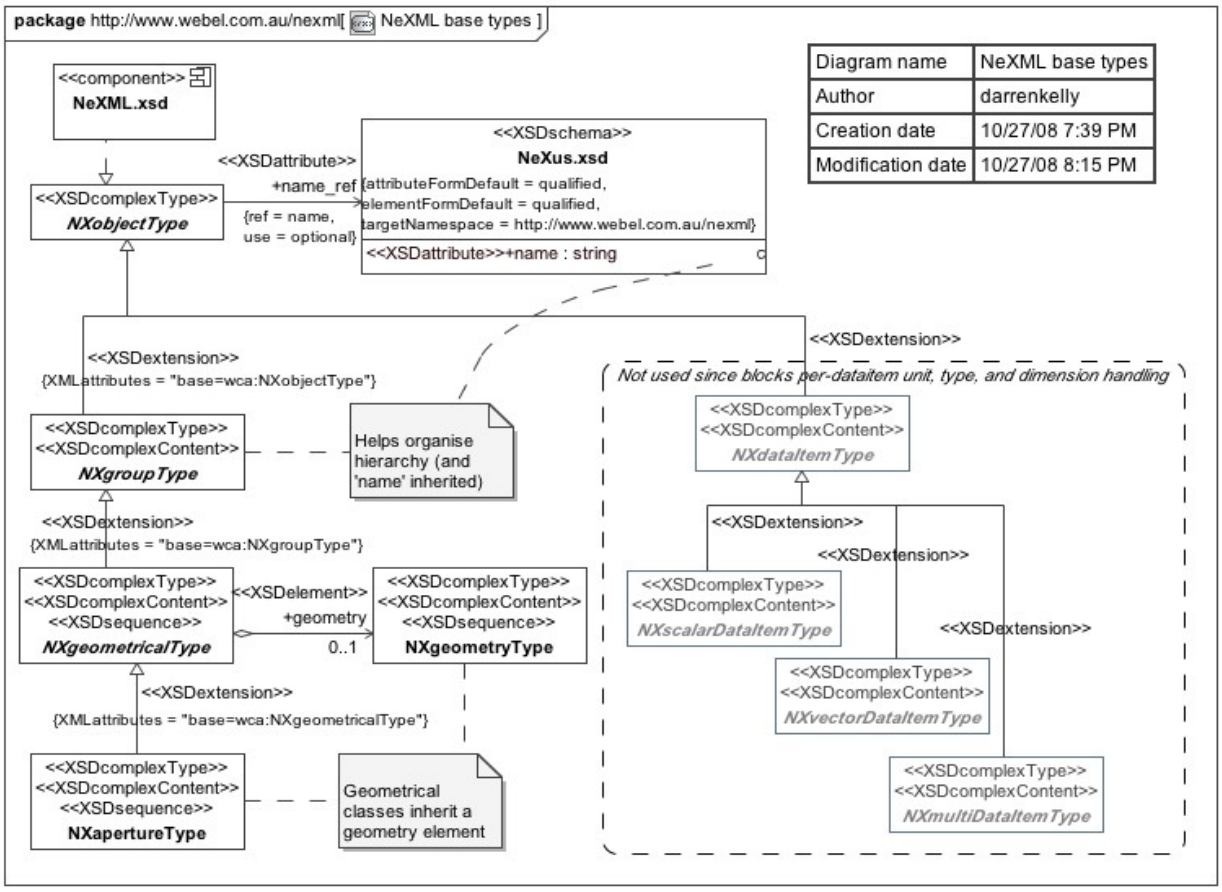


Illustration 2: Reverse engineered UML overview of the NeXML base types

2) Transformation of the NeXus XML class templates to NeXML Schema complex types

Each NeXus XML class template has a single root class definition element (which must be a NeXus ”group” class) and documentation, and some child elements which refer either to other class definitions or to local “data item” definitions[6]:

4 These inheritance and base class concepts were presented by Kelly as Java and UML examples in a presentation on NeXusBeans and the need for an XML Schema at NIAC2006 at ILL in Grenoble, France.

5 Some of these shared base class concepts now find analogy in the `NeXus.xsd` complex types

6 The data item and group child elements will be discussed in detail later in dedicated sections.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
URL:     http://www.nexus.anl.gov/classes/xml/NXentry.xml
Editor:  NIAC
$Id$

Template of the top-level NeXus group which contains all the data and
..
-->
<NXentry name="{Entry Name}">
      <title>{Extended title for entry}?</title>
..
      <duration type="NX_INT" units="seconds">
            {Duration of measurement}?
      </duration>
..
      <NXsample name="{sample}">
            ?
      </NXsample>
      <NXinstrument name="{Name of instrument}">
            ?
      </NXinstrument>
      <NXmonitor name="{Name of monitor}">
            *
      </NXmonitor>
      <NXdata name="{Name of data block}">
            *
      </NXdata>
..
</NXentry>
```

Table 1: Extract from the NeXus `NXentry.xml` template (a.k.a."Meta DTD")

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
URL:     http://www.nexus.anl.gov/classes/xml/NXinstrument.xml
Editor:  NIAC
$Id$

Template of instrument descriptions comprising various beamline
..
valid for both reactor and pulsed instrumentation.

-->
<NXinstrument name="{Name of instrument}">
      <name short_name="{abbreviated name of instrument}">
            {Name of instrument}?
      </name>
      <NXsource name="{Name of facility}">
            *
      </NXsource>
..
</NXinstrument>
```

Table 2: Extract from the `NXinstrument.xml` template

Each class is transformed into a named XSD complex type in the NeXML schema, with its top-level documentation included and the URL of the original XML template recorded. All concrete generated classes are considered to extend a special abstract `NXgroup` complex type (and through it also the universal complex `NXobject`), either directly or indirectly:

```
 <xsd:complexType abstract="false" name="NXentryType">
    <xsd:annotation>
      <xsd:appinfo source="...NXentry.xml"/>
      <xsd:documentation source="...NXentry.xml">
URL:     http://www.nexus.anl.gov/classes/xml/NXentry.xml
Editor:  NIAC
$Id$
Template of the top-level NeXus group which contains all the data and
..
</xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="wca:NXgroupType">
...
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
```

Table 3: Extract from NeXML.xsd showing part of the transformed NXentry complex type

Some classes known to have an `<NXgeometry>` child element are transformed (as was likewise already done in the prototype Java NeXusBeans) to extend an intermediate `NXgeometricalType` (comparable to the `NXcomponentType` in `NeXus.xsd`), and thus inherit a geometrical element:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
URL:     http://www.nexus.anl.gov/classes/xml/NXaperture.xml
Editor:  NIAC
$Id$
Template of a beamline aperture.
-->
<NXaperture name="{Name of aperture}">
      <NXgeometry name="">
      {location and shape of aperture}?
      </NXgeometry>
      <material type="NX_CHAR">
      {Absorbing material of the aperture}?
      </material>
      <description type="NX_CHAR">
      {Description of aperture}?
      </description>
</NXaperture>
```

Table 4: The `NXaperture.xml` template with `NXgeometry` child element

```
<xsd:complexType abstract="false" name="NXapertureType">
..
    <xsd:complexContent>
      <xsd:extension base="wca:NXgeometricalType">
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
```

Table 5: Extract from `NeXML.xsd` showing part of a transformed geometrical complex type

```
  <xsd:complexType abstract="true" name="NXgeometricalType">
    <xsd:complexContent>
      <xsd:extension base="wca:NXgroupType">
        <xsd:sequence>
          <xsd:element maxOccurs="1" minOccurs="0" name="geometry"
type="wca:NXgeometryType"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
```

Table 6: Extract from `NeXML.xsd` showing the abstract `NXgeometricalType` with geometry.

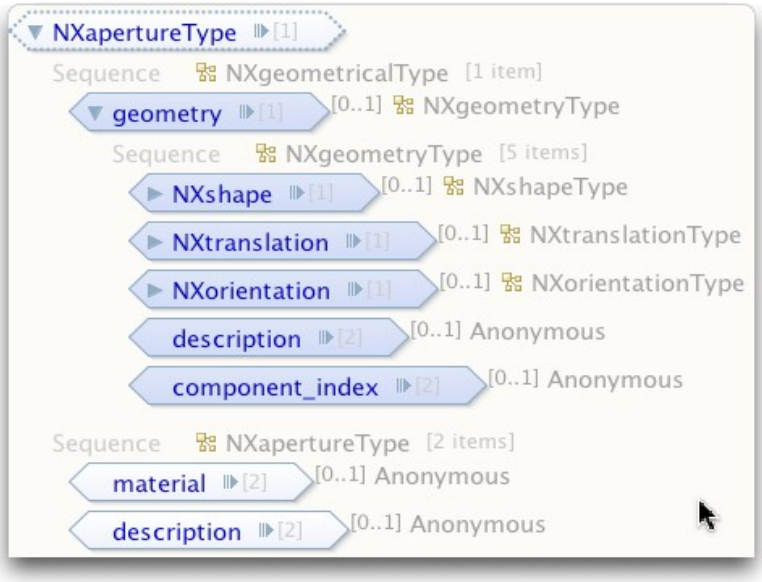


Illustration 3: Netbeans IDE XML Schema design view of inheritance of geometry into `NXapertureType`

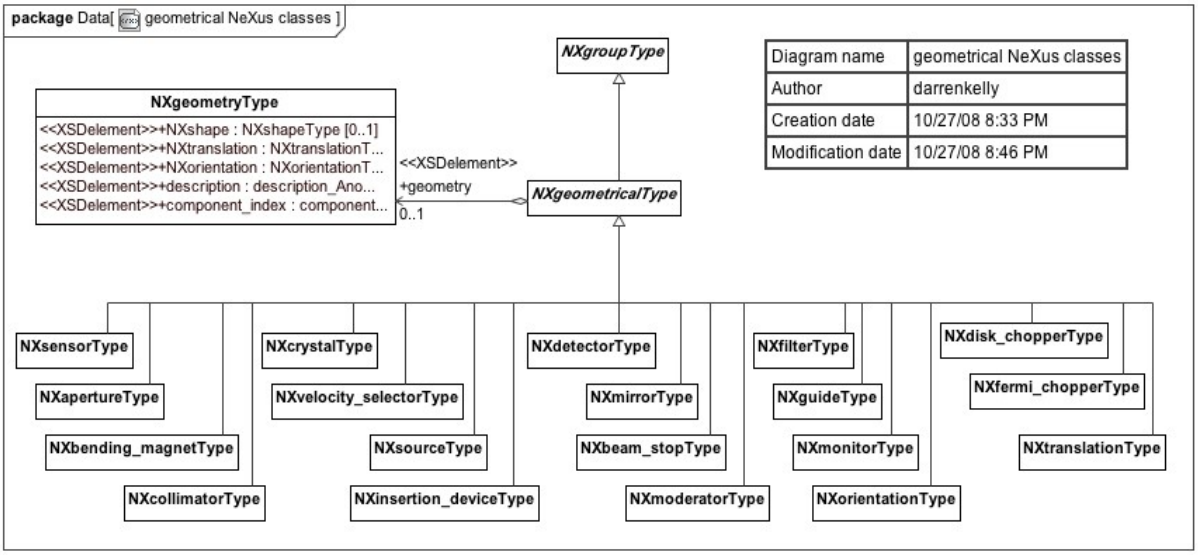


Illustration 4: UML reverse engineering of the NeXML schema showing geometrical types

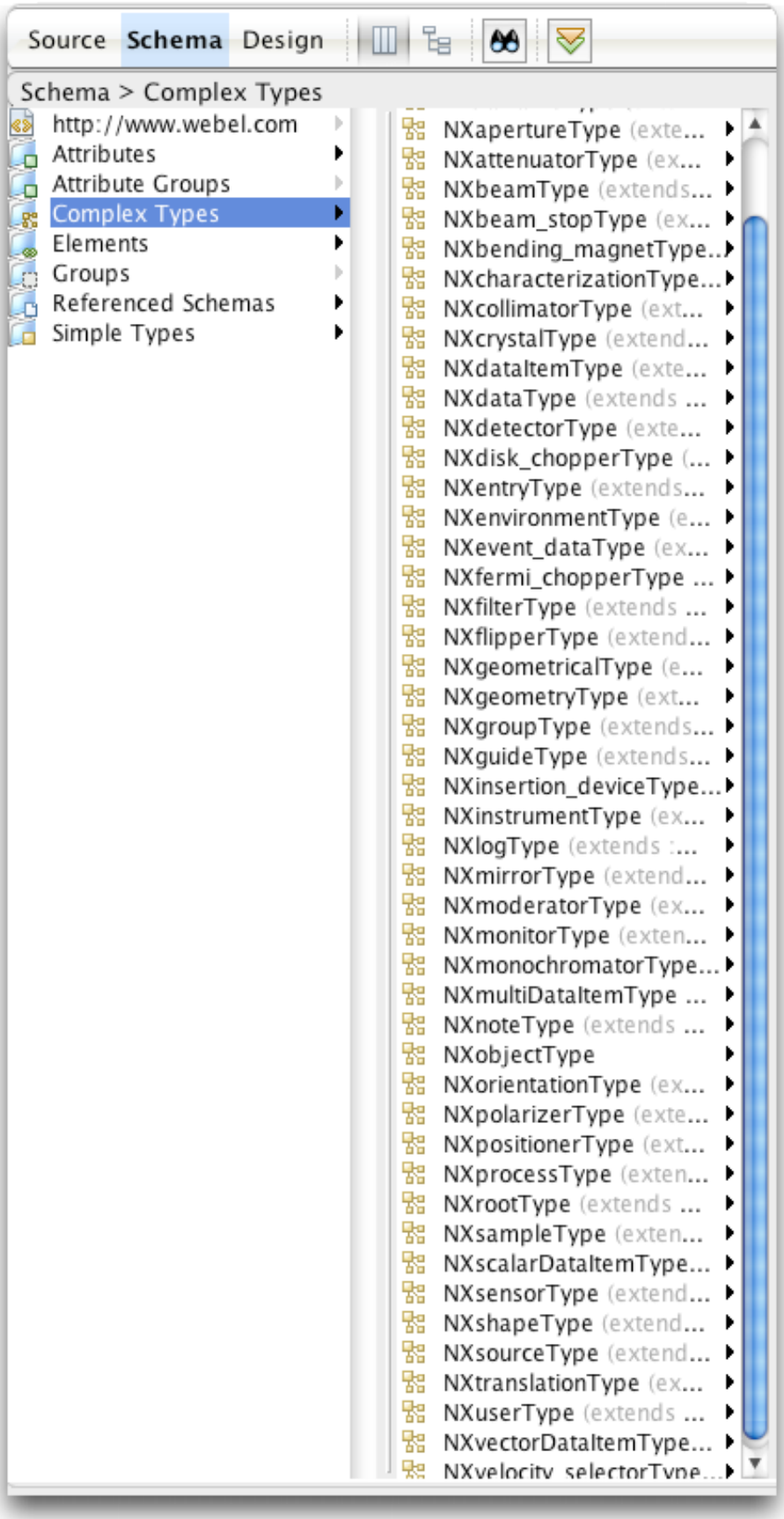


Illustration 5: All transformed NeXus group types and the supporting custom generated abstract base group NeXML types are shown in a Netbeans IDE XML Schema editor view of the NeXML schema.

2.1) NeXus child elements and multiplicities

The child element of the root class definition element of a NeXus XML template may indicate either a NeXus group element or a NeXus data item element, both of which may carry additional XML attributes:

```
<NXentry name="{Entry Name}">
..
     <title>{Extended title for entry}?</title>
     <start_time type="ISO8601">
          {Starting time of measurement}?
     </start_time>
..   <duration type="NX_INT" units="seconds">
          {Duration of measurement}?
     </duration>
..
     <NXcharacterizations>
          ?
     </NXcharacterizations>
..
     <NXuser name="{user}">
          *
     </NXuser>
..
     <NXmonitor name="{Name of monitor}">
          *
     </NXmonitor>
..
</NXentry>
```

Table 7: Selected data item and group children of the `NXentry` class XML template

```
<NXroot ..>
        <NXentry name="{entry name}">
            +
        </NXentry>
</NXroot>
```

Table 8: There must be at least 1 `NXentry` child in an `NXroot`, as indicated by a '+'.

```
<NXattenuator name="{Name of attenuator}">
     <distance type="NX_FLOAT" units="m">
          {Distance from sample}
     </distance>
..
```

Table 9: In some NeXus XML templates there are child elements with no multiplicity indicator

The permitted multiplicity may be indicated by one of the following characters in the contained documentation text for the child element, usually as the last character of the documentation:

'?' = May occur 0 or 1 times

'*' = May occur 0 or more times

'+' = Must occur at least once

If the multiplicity indicator is missing the element must occur exactly once.

These indicators map well in XML Schema to the XSD 'minOccurs' and 'maxOccurs' attributes, in UML to MultiplicityElement and MultiplicityKind, and in EMF to multiplicity annnotations.

2.2) Transformation of NeXus group child elements

The child elements of the root element of a NeXus XML template may indicate allowed children typed by NeXus groups. Simple examples include `<NXsource>`, `<NXdisk_chopper>` and in `NXinstrument.xml`:

```
<NXinstrument name="{Name of instrument}">
..
     <NXsource name="{Name of facility}">
     *
     </NXsource>
     <NXdisk_chopper name="{Name of chopper}">
     *
     </NXdisk_chopper>
..
</NXinstrument>
```

Table 10: Extract from `NXinstrument.xml` showing selected NeXus group child elements

After extraction and interpretation of the multiplicity indicator the generation of a corresponding XML element typed by the complex type of the matching group is straightforward. However the choice of containing the children within a strict XSD 'sequence' rather than an XSD 'all' presents a complication. Unfortunately, the 'all' grouping can't be used in general for all child elements of all NeXus XML templates because:

> "All the elements in the group may appear once or not at all, and they may appear in any order. The all group .. is limited to the top-level of any content model. Moreover, the group's children must all be individual elements (no groups), and no element in the content model may appear more than once, i.e. the permissible values of minOccurs and maxOccurs are 0 and 1." [11]

Therefore for ease and consistency the NeXML transformer collects all NeXus data item and group child elements in strict sequences in the same order they are encountered in the NeXus XML templates. (It could however be adapted to use 'all' grouping only in those cases where the multiplicity of all child elements does not exceed 1.)

```
 <xsd:complexType abstract="false" name="NXinstrumentType">
 ..
   <xsd:complexContent>
      <xsd:extension base="wca:NXgroupType">
        <xsd:sequence>
..
          <xsd:element maxOccurs="unbounded" minOccurs="0"
name="NXsource" type="wca:NXsourceType"/>
          <xsd:element maxOccurs="unbounded" minOccurs="0"
name="NXdisk_chopper" type="wca:NXdisk_chopperType"/>
..
```

Table 11: Extract from `NeXML.xsd` showing some child elements of the `NXinstrumentType` typed by other complex types for NeXus groups and with automatically generated multiplicity (occurrence) ranges.

```
  <xsd:complexType abstract="false" name="NXrootType">
    <xsd:complexContent>
      <xsd:extension base="wca:NXgroupType">
        <xsd:sequence>
          <xsd:element maxOccurs="unbounded" minOccurs="1"
name="NXentry" type="wca:NXentryType"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
```

Table 12: Extract from `NeXML.xsd` showing the single child element of the `NXrootType` typed by `NXentryType`, which must occur at least once and may occur many times.

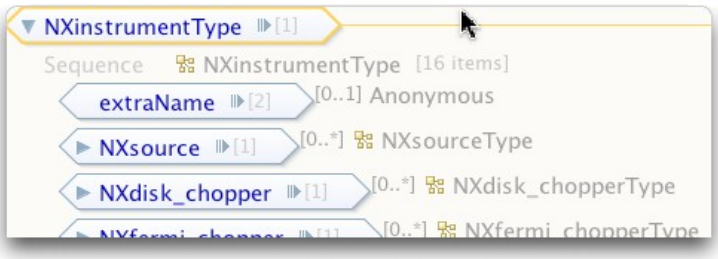


Illustration 6: Some NeXus group children in the Netbeans IDE XML Schema design view of `NeXML.xsd`

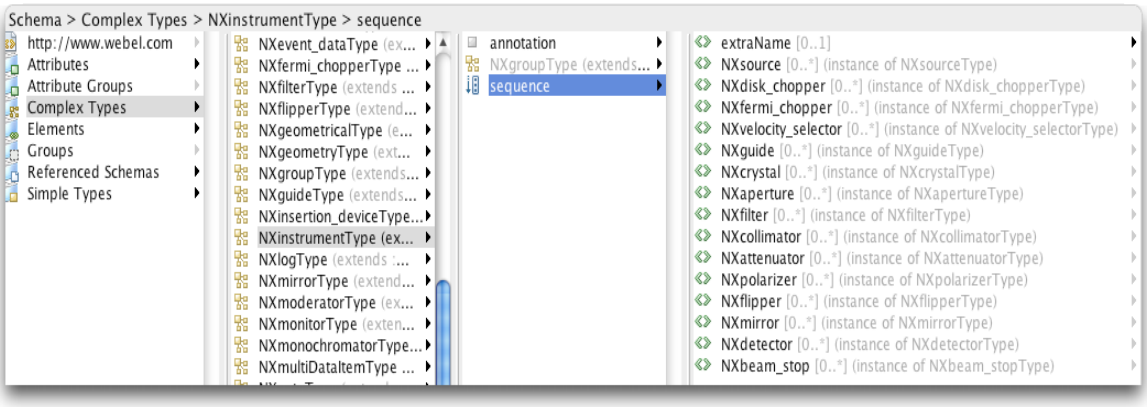


Illustration 7: All NeXus group child elements for `NXinstrumentType` in the Netbeans XML Schema view of `NeXML.xsd` (together with a special data item child `extraName`).

2.3) Encapsulating and restricting types in data items

The valid type for data content of data items in the NeXus XML templates is indicated through *NeXus API types* (known as a *NAPIType*) given as strings in a ‘type’ attribute.

2.3.1) Case: scalar data value of unique type

In the simplest case a single NAPIType for a scalar data value is given:

NXaperture.xml:	<material type="**NX_CHAR**"
NXattenuator.xml:	<distance type="**NX_FLOAT**"
NXdisk_chopper.xml:	<slits type="**NX_INT**"
NXdetector.xml:	<calibration_date type="**ISO8601**"
NXmoderator.xml:	<coupled type="**NX_BOOLEAN**"

In such cases the data item child element can be encapsulated in XML as a complex type extending (either directly or via the `NeXus.xsd` simple types) a simple XSD type suited to the NAPItype. The 'type' attribute from the XML template can also be propagated as an inline XML Schema enumeration literal, to enforce correct supporting metadata on the type in NeXus XML instances:

```
  <xsd:complexType abstract="false" name="NXattenuatorType">
..
  <xsd:complexContent>
      <xsd:extension base="wca:NXgroupType">
        <xsd:sequence>
          <xsd:element maxOccurs="unbounded" name="distance">
            <xsd:complexType>
              <xsd:simpleContent>
                <xsd:extension base="nx:float32">
                  <xsd:attribute name="type" use="required">
                    <xsd:simpleType>
                      <xsd:restriction base="xsd:string">
                        <xsd:enumeration value="NX_FLOAT"/>
                      </xsd:restriction>
                    </xsd:simpleType>
                  </xsd:attribute>
```

Table 13: Extract from `NeXML.xsd` showing complex extension of a simple `nx:float32` data type and an enforced NX_FLOAT metadata enumeration literal for the 'type' attribute

The NeXML transformer deliberately does not extend or leverage the `NeXus.xsd` list complex DataTypes with their reused data attribute mixed in, because this would exclude many possibilities, including the generation of per-data-item 'units' restrictions, length facets, and correct handling of scalars. Instead, the NeXML generates per-data-item complex types with correct 'units', 'type' and list lengths, either anonymous or named as required.

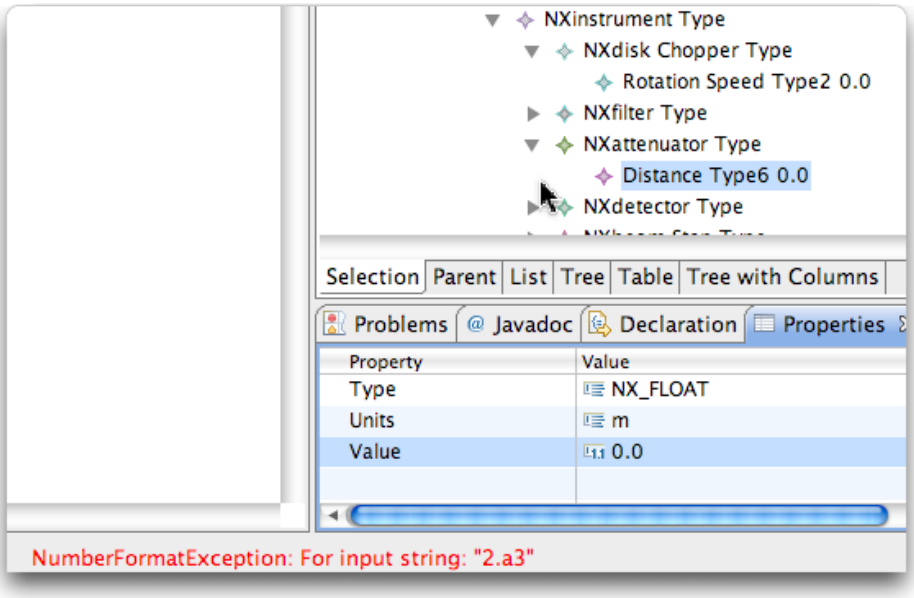


Illustration 8: The validating EMF editor is able to prevent incorrect entry of scalar data item values of known type, in this case a float.

2.3.2) Case: non-scalar data value of unique type

In some cases the pure NAPI type indicator is combined with a simple vector or complex multi-dimension indicator. The NeXML transformer separates this dimension indicator from the pure NAPI-type indicator in the XML template's 'type' attribute and processes it separately. There are a number of challenging cases which are presented in turn.

2.3.2.1) Case: the length of every dimension is known

`NXcrystal.xml:`	`<reflection type="NX_INT[3]">`
`NXcrystal.xml:`	`<orientation_matrix type="NX_FLOAT[3,3]">`
`NXdisk_chopper.xml:`	`<wavelength_range type="NX_FLOAT[2]">`

In this case and XSD length facet can be used to restrict the total length of a dedicated, named, data list simple type of restricted length which can then be referenced by a complex data item type, which extends the simple type with 'units' and 'type' attributes as well as a known 'dimension' as enumeration literal. Although this approach is quite verbose and increases the complexity of the schema, it is necessary as a complex type can't carry a length facet, while a simple type can't carry additional attribute definitions (required for 'units', 'type', and other metadata):

```
<xsd:simpleType name="NXcrystal.reflectionFiniteDataType">
  <xsd:restriction>
    <xsd:simpleType>
      <xsd:list itemType="nx:int32"/>
    </xsd:simpleType>
    <xsd:length value="3"/>
  </xsd:restriction>
</xsd:simpleType>
```

Table 14: Extract from `NeXus.xsd` showing a dedicated data item simple type of restricted length

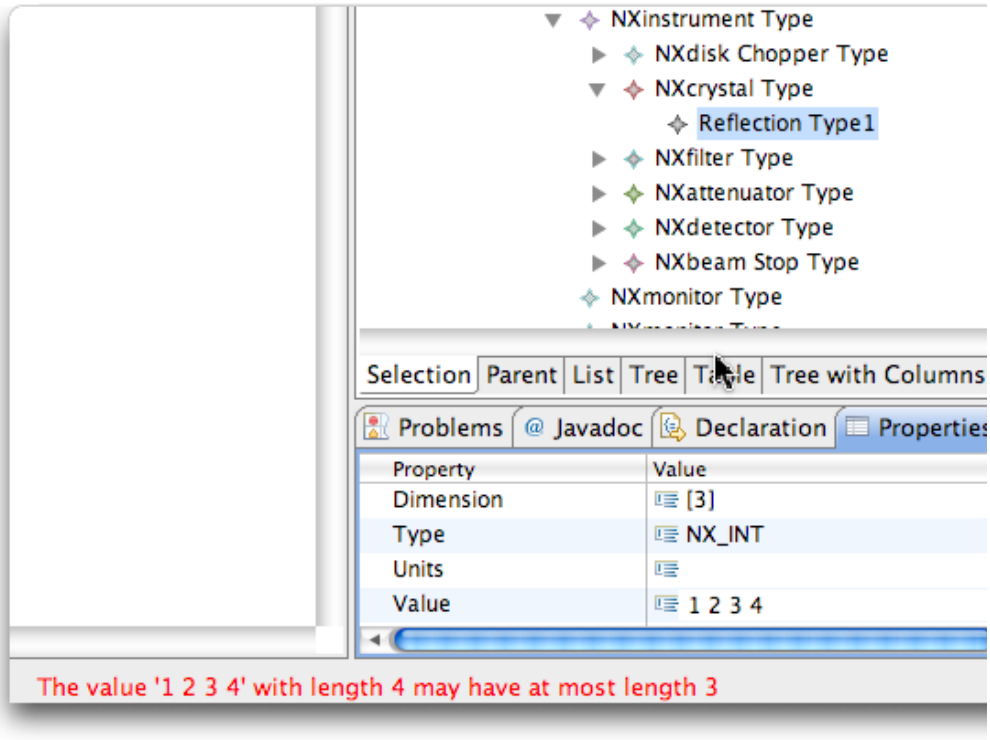


Illustration 9: The validating EMF instance editor is able to prevent entry of data item lists of incorrect length (in the case where the length is known).

```
 <xsd:complexType abstract="false" name="NXcrystalType">
  ..
  <xsd:complexContent>
      <xsd:extension base="wca:NXgeometricalType">
        <xsd:sequence>
..
          <xsd:element maxOccurs="1" minOccurs="0" name="reflection">
            <xsd:complexType>
              <xsd:simpleContent>
                <xsd:extension base="wca:NXcrystal.reflectionFinite-
DataType">
                  <xsd:attribute name="type" use="required">
                    <xsd:simpleType>
                      <xsd:restriction base="xsd:string">
                        <xsd:enumeration value="NX_INT"/>
                      </xsd:restriction>
                    </xsd:simpleType>
                  </xsd:attribute>
                  <xsd:attribute name="dimension" use="required">
                    <xsd:simpleType>
                      <xsd:restriction base="xsd:string">
                        <xsd:enumeration value="[3]"/>
                      </xsd:restriction>
                    </xsd:simpleType>
                  </xsd:attribute>
..
                </xsd:extension>
              </xsd:simpleContent>
            </xsd:complexType>
          </xsd:element>
```

Table 15: Extract from `NeXus.xsd` showing a dedicated simple data item of known length reused in a complex data item element, with supporting 'dimension' attribute with enumeration literal.

2.3.2.2) Case: dimensions known and the length of every dimension is unbounded

NXbeam.xml:	<incident_energy type="**NX_FLOAT[:]**"
NXentry.xml:	<experiment_identifier type="**NX_CHAR[]**"
NXentry.xml:	<run_cycle type="**NX_CHAR[]**"
NXmonochromator.xml:	<wavelength type="**NX_FLOAT[]**"

Table 16: NeXus XML template data items with fixed number of dimensions of unbound length.

In these cases a trivial dedicated named list simple type is still required for each data item element, as a target for extension by an anonymous complex type required to mix in specific 'units' etc. attributes. (The complex `DataType` definitions of the `NeXus.xsd` can't be used for this purpose, because they already mix in very general 'units' attributes etc. and thwart the union of types needed in some special cases.)

```
 <xsd:simpleType name="NXbeam.incident_energyDataType">
    <xsd:list itemType="nx:float32"/>
  </xsd:simpleType>
```

Table 17: Extract from `NeXML.xsd` showing simple list type of unbounded length

```
  <xsd:complexType abstract="false" name="NXbeamType">
..
  <xsd:complexContent>
      <xsd:extension base="wca:NXgroupType">
        <xsd:sequence>
..
          <xsd:element maxOccurs="1" minOccurs="0"
name="incident_energy">
            <xsd:complexType>
              <xsd:simpleContent>
                <xsd:extension base="wca:NXbeam.incident_energy-
DataType">
                  <xsd:attribute name="type" use="required">
                    <xsd:simpleType>
                      <xsd:restriction base="xsd:string">
                        <xsd:enumeration value="NX_FLOAT"/>
                      </xsd:restriction>
                    </xsd:simpleType>
                  </xsd:attribute>
                  <xsd:attribute name="dimension" use="required">
                    <xsd:simpleType>
                      <xsd:restriction base="xsd:string">
                        <xsd:enumeration value="[:]"/>
                      </xsd:restriction>
                    </xsd:simpleType>
                  </xsd:attribute>
                  <xsd:attribute name="units" use="required">
                    <xsd:simpleType>
                      <xsd:restriction base="xsd:string">
                        <xsd:enumeration value="meV"/>
                      </xsd:restriction>
                    </xsd:simpleType>
                  </xsd:attribute>
                </xsd:extension>
              </xsd:simpleContent>
            </xsd:complexType>
          </xsd:element>
```

Table 18: Extract from `NeXML.xsd` showing a simple list data item type extended by an anonymous complex type of a data item element to include 'units', 'type' indicator and 'dimension' attributes.

2.3.3) Case: length of dimensions partly known

In some cases the length of at least one dimension is known, so that the total length of the data list must be a multiple of the known dimensions, yet possibly unbounded:

NXbeam.xml:	<incident_beam_divergence type="**NX_FLOAT[2,:]**"

The simple data list type in the XML schema would have to be unbounded yet constrained to be a multiple of the known dimensions, which constraint is not yet offered by the NeXML transformer.

2.3.4) Case: length of dimensions depends on a variable

In some cases the length of at least one dimension depends on an implied variable of the system:

NXevent_data.xml:	<pixel_number type="**NX_INT[i]**">
NXevent_data.xml:	<pulse_time type="**NX_INT[j]**"
NXmonitor.xml:	<efficiency type="**NX_FLOAT[i]**">
NXdetector.xml:	<raw_time_of_flight type="**NX_INT[tof+1]**"
NXfilter.xml:	<unit_cell type="**NX_FLOAT[n_comp,6]**)">
NXfilter.xml:	<unit_cell_volume type="**NX_FLOAT[n_comp]**”
NXfilter.xml:	<orientation_matrix type="**NX_FLOAT[n_comp,3,3]**">
NXfilter.xml:	<coating_roughness type="**NX_FLOAT[nsurf]**">
NXorientation.xml:	<value type="**NX_FLOAT[numobj,6]**">
NXpositioner.xml:	<value type="**NX_FLOAT[n]**"
NXsample.xml:	<unit_cell type="**NX_FLOAT[n_comp,6])**">
NXsample.xml:	<unit_cell_volume type="**NX_FLOAT[n_comp]**"
NXshape.xml:	<size type="NX_FLOAT**[numobj,nshapepar]**"
NXtranslation.xml:	<distances type=**"NX_FLOAT[numobj,3]"**

As there is no formal mechanism in the XML templates design to define such implied variables these cases are very difficult to transform to XML Schema, and are not yet handled by NeXML.

2.3.5) Case: optional dimensions indicated by ‘?’

NXdetector.xml:	<detector_number type="**NX_INT[i?,j?]**">
NXdetector.xml:	<distance type="**NX_FLOAT[np?,i?,j?]**">
NXdetector.xml:	<x_pixel_offset type="**NX_FLOAT[i?]**"
NXdetector.xml:	<efficiency type="**NX_FLOAT[i?,j?,k?]**"
NXevent_data.xml:	<pulse_height type="**FLOAT[i,k?]**"

These cases are not yet specifically handled by the NeXML transformer (an unbounded list is provided).

2.3.6) Case: total dimensionality unknown

NXdata.xml:	<errors type="**NX_FLOAT[:...]**">

These cases are not yet specifically handled by the NeXML transformer (an unbounded list is provided).

2.3.7) Case: data type given as options

NXlog.xml:	<value type="**NX_FLOAT|NX_INT**">

NXdata.xml:	<variable_errors type="**NX_FLOAT[:]|NX_INT[:]**">
NXdata.xml:	<data type="**NX_FLOAT[:...]|NX_INT[:...]**"
NXdetector.xml:	<data type="**NX_FLOAT[np?,i?,j?,tof?]|NX_INT[np?,i?,j?,tof?]**"
NXdetector.xml:	<solid_angle type="**NX_FLOAT[i?,j?]**"
NXdetector.xml:	<count_time type="**NX_INT[np?]|NX_FLOAT[np?]**"
NXmonitor.xml:	<data type="**NX_INT[i]|NX_FLOAT[i]**"
NXpositioner.xml:	<raw_value type="**NX_FLOAT[n]|NX_INT[n]**"

Currently the NeXML transformer takes the first possible simple type indicated in the NeXus XML templates, and extends that to create a compromised data item, while correctly restricting the 'type' metadata for the NAPItype by enumeration literals. An XML Schema 'union' might resolve this situation, in the case where the dimensionality is known.

```
<xsd:element name="data_error">
            <xsd:complexType>
              <xsd:annotation/>
              <xsd:complexContent>
                <xsd:extension base="nx:float32DataType">
                  <xsd:attribute name="type" use="required">
                    <xsd:simpleType>
                      <xsd:restriction base="xsd:string">
                        <xsd:enumeration value="NX_FLOAT"/>
                        <xsd:enumeration value="NX_INT"/>
                      </xsd:restriction>
                    </xsd:simpleType>
                  </xsd:attribute>
..
                </xsd:extension>
              </xsd:complexContent>
            </xsd:complexType>
          </xsd:element>
```

Table 19: Extract from `NeXML.xsd` showing enumerated 'type' options of a scalar data item, in a complex data item type that extends a single simple data type.

2.3.8) Case: NeXus group as data item type

In some cases the type is not given as a pure NAPIType, but instead a NeXus group type is given:

NXbending_magnet.xml:	<spectrum type="**NXdata**">
NXcrystal.xml:	<temperature_log type="**NXlog**">
NXcrystal.xml:	<reflectivity type="**NXdata**">
NXdetector.xml:	<calibration_method type="**NXnote**">
NXenvironment.xml:	<position type="**NXgeometry**">
NXfilter.xml:	<sensor_type type="**NXsensor**">
NXsample.xml:	<beam type="**NXbeam**">
NXsample.xml:	<temperature_env type="**NXenvironment**">
NXsensor.xml:	<external_field_full type="**NXorientation**">

These cases are not handled yet by the NeXML transformer[7]. Kelly has suggested a redesign of the NeXus templates to avoid such hybrid data items (although the prototype NeXusBeans did provide a type-correct handling in Java).

7 The **NeXusBeans** transformer did provide automated mappings for group types used as data item types.

2.4) Encapsulating restricted units in data items

For many data items in the NeXus XML templates, scientific units are indicated as freestyle strings (NX_CHAR) in a 'units' attribute, and sometimes a choice of units is indicated. The reader should inspect the variation in 'units' attribute in these selected extracts from the current XML templates:

NXattenuator.xml:	<distance units="m"
NXattenuator.xml:	<thickness units="cm"
NXattenuator.xml:	<scattering_cross_section units="barns"
..	
NXbeam.xml:	<incident_energy units="meV"
..	
NXbeam.xml:	<incident_wavelength units="Angstroms"
..	
NXbeam.xml:	<incident_beam_divergence units="degree"
NXbeam.xml:	<final_wavelength_spread units="Ang-stroms"
NXbeam.xml:	<final_beam_divergence units="degrees"
..	
NXbeam.xml:	<flux units="s-1cm-2"
..	
NXcollimator.xml:	<soller_angle units="minutes"
NXcrystal.xml:	<cut_angle units="degrees"
NXcrystal.xml:	<unit_cell_volume units="Angstroms3" rank="1"
NXcrystal.xml:	<wavelength units="Angstroms"
NXcrystal.xml:	<lattice_parameter units="Angstrom"
NXcrystal.xml:	<scattering_vector units="Angstrom^-1"
..	
NXcrystal.xml:	<mosaic_horizontal units="arc minutes"
..	
NXdetector.xml:	units="10-6 second|10-7 second"
NXdetector.xml:	<raw_time_of_flight units="clock_pulses"
NXdetector.xml:	units="number"
..	
NXdetector.xml:	units="10-3 meter|10-2 meter"
NXdetector.xml:	<solid_angle units="steradians"
..	
NXdetector.xml:	<x_pixel_size units="mili*metre"
..	
NXdetector.xml:	<gas_pressure units="bars"
NXdetector.xml:	<efficiency units=""
..	
NXdisk_chopper.xml:	<rotation_speed units="rpm|hertz"

..	
NXdisk_chopper.xml:	<phase units="degree"
..	
NXdisk_chopper.xml:	<wavelength_range units="nm"
..	
NXevent_data.xml:	<time_of_flight units="10^n second"
..	
NXinsertion_device.xml:	<gap units="milimetre"
..	
NXpositioner.xml:	<target_value units="{}"
NXsample.xml:	<unit_cell_volume units="Angstroms3"
NXsample.xml:	<mass units="g"
NXsample.xml:	<density units="g cm-3"
NXsample.xml:	<concentration units="g.cm-3"
..	
NXshape.xml:	<size units="meter"
NXsource.xml:	<power units="MW"
NXsource.xml:	<current units="microamps"
NXsource.xml:	<voltage units="MeV"
NXsource.xml:	<frequency units="Hz"
NXsource.xml:	<period units="microseconds"
NXsource.xml:	<pulse_width units="micro.second"

There is almost no consistency at all regarding whether unit symbols or words are used, how unit names are spelt, how dimensions and exponents are indicated, or even what language is used, and in many cases units are given as “” or as “{}”, making a robust validation of scientific data against the current NeXus XML templates impossible.

However, to the extent that the ‘units’ attributes are given as strings, they can be extracted and enforced in the automatically generated NeXML Schema attribute declarations per data item, and in the cases where options are indicated, lists of restricting enumeration literals can be used to represent valid 'units' choices.

```
  <xsd:complexType abstract="false" name="NXattenuatorType">
..
    <xsd:complexContent>
      <xsd:extension base="wca:NXgroupType">
        <xsd:sequence>
          <xsd:element maxOccurs="unbounded" name="distance">
            <xsd:complexType>
              <xsd:simpleContent>
                <xsd:extension base="nx:float32">
..
                  <xsd:attribute name="units" use="required">
                    <xsd:simpleType>
                      <xsd:restriction base="xsd:string">
                        <xsd:enumeration value="m"/>
                      </xsd:restriction>
                    </xsd:simpleType>
                  </xsd:attribute>
                </xsd:extension>
              </xsd:simpleContent>
            </xsd:complexType>
          </xsd:element>
..
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
```

Table 20: A simple single 'units' restriction represented thus in the NeXML schema

The use of a single enumeration for a single ‘units’ value has the advantage (over, for example, an XSD pattern) that it is easily enforced in many validating XML instance editors. In the case of a choice of units, multiple enumeration literals are used (rather than a pattern):

```
  <xsd:complexType abstract="false" name="NXdisk_chopperType">
  ..
  <xsd:complexContent>
      <xsd:extension base="wca:NXgeometricalType">
        <xsd:sequence>
..
          <xsd:element maxOccurs="1" minOccurs="0" name="rota-
tion_speed">
            <xsd:complexType>
              <xsd:simpleContent>
                <xsd:extension base="nx:float32">
..
                  <xsd:attribute name="units" use="required">
                    <xsd:simpleType>
                      <xsd:restriction base="xsd:string">
                        <xsd:enumeration value="rpm"/>
                        <xsd:enumeration value="hertz"/>
                      </xsd:restriction>
                    </xsd:simpleType>
                  </xsd:attribute>
                </xsd:extension>
              </xsd:simpleContent>
            </xsd:complexType>
          </xsd:element>
..
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
```

Table 21: A 'units' restriction with enforced options represented in the NeXML schema

In the validating EMF instance editor bound to the NeXML schema the valid ‘units’ options are then offered thus:

2.4.1) Improving robustness of units though binding to an XML Schema

One candidate for an imported units XML schema is the Units Markup Language (UnitsML) [13] draft XML Schema , which Kelly and has reverse engineered it into UML[8].

Another possibility is the automated conversion of the experimental XML database of the UDUNITS2 packages [12] into an XML Schema, using the same Java, EMF, XSD, and XML technologies presented here, which Kelly proposes as a subproject of NeXML and NeXus. Note that the NeXus design already states:

> “Units of data which must conform to the standard defined by the Unidata UDunits utility (in particular, see udunits.dat)”

Conversion of UDUNITS to an XML Schema would be of interest to many science and engineering projects, including the OMG's SysML systems engineering language effort[14].

8 http://school.nomagicasia.com/unitsml (as part of Kelly's work on robust units for the SysML extension of UML)

2.5) Encapsulating enumerated values in DataItems

Enumeration literals are indicated in the NeXus XML templates by a list of quoted and/or unquoted strings separated by '|':

```
<NXfilter name="filter_name">
      <NXgeometry name="">
      {Geometry of the filter}?
      </NXgeometry>
      <description type="NX_CHAR">
      {"Beryllium" | "Pyrolytic Graphite" | "Graphite" | "Sapphire" |
      "Silicon" | "Supermirror"}?
      </description>
      <status type="NX_CHAR">
      {in | out}?
      </status>
      ..
</NXfilter>
```

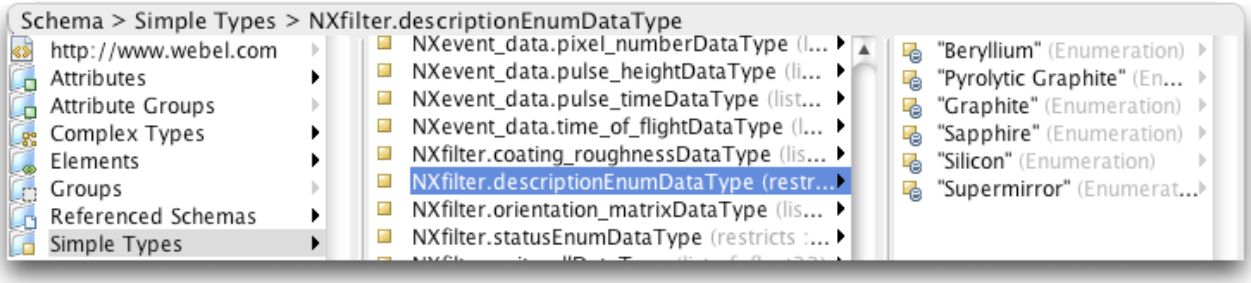
Table 22: Extract from `NXfilter.xml` showing data items with candidate enumeration literals

The automated translation of such candidate enumerations to the NeXML Schema is achieved with intermediate simple types encapsulating the enumeration literals:

```
  <xsd:simpleType name="NXfilter.descriptionEnumDataType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Beryllium"/>
      <xsd:enumeration value="Pyrolytic Graphite"/>
      <xsd:enumeration value="Graphite"/>
      <xsd:enumeration value="Sapphire"/>
      <xsd:enumeration value="Silicon"/>
      <xsd:enumeration value="Supermirror"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="NXfilter.statusEnumDataType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="in"/>
      <xsd:enumeration value="out"/>
    </xsd:restriction>
  </xsd:simpleType>
```

Table 23: Extract from the `NeXML.xsd` schema showing enumeration literals for data item values

The Netbeans IDE XML Schema editor view shows these enumerations thus:



These simple enumeration types for the given DataItems are then referenced as base types in elements of the owning Group (along with extending attribute declarations elided here):

```
<xsd:complexType abstract="false" name="NXfilterType">
..
  <xsd:complexContent>
      <xsd:extension base="wca:NXgeometricalType">
        <xsd:sequence>
          <xsd:element maxOccurs="1" minOccurs="0" name="description">
            <xsd:complexType>
              <xsd:simpleContent>
                <xsd:extension base="wca:NXfilter.descriptionEnumData-
Type">
                  ..
                </xsd:extension>
              </xsd:simpleContent>
            </xsd:complexType>
          </xsd:element>
          <xsd:element maxOccurs="1" minOccurs="0" name="status">
            <xsd:complexType>
              <xsd:simpleContent>
                <xsd:extension base="wca:NXfilter.statusEnumDataType">
                  ..
                </xsd:extension>
              </xsd:simpleContent>
            </xsd:complexType>
          </xsd:element>
..
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
```

Table 24: Extract from the NeXML schema showing data item types referencing value enumerations

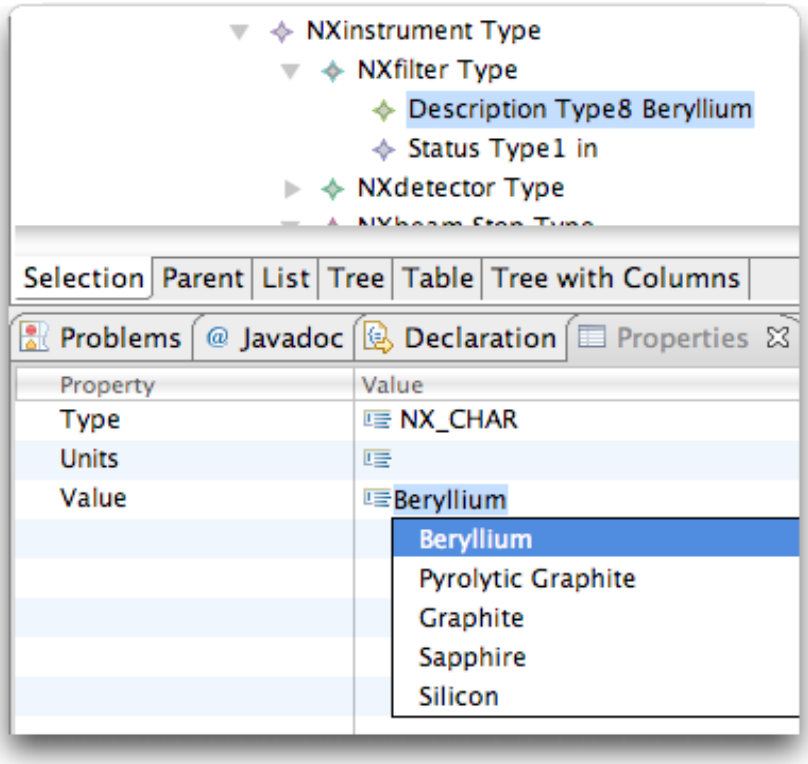


Illustration 10: The validating EMF model editor offers enumeration literal values as options in a property sheet

2.5.1) Issue: inconsistencies in the XML Template representation of enumerations

Unfortunately there are at the time of writing many inconsistencies in the use of enumeration indicators in the NeXus XML templates.

Example: in some XML templates freestyle documentation text is mixed with candidate enumeration literals:

From `NXbeam_stop.xml`:

```
<description type="NX_CHAR">
 {description of beamstop: circular | rectangular}?
</description>
```

From `NXsensor.xml`:

```
<type type="NX_CHAR">
{ The type of hardware we use for the measurement e.g. Temperature:
"J | K | T | E | R | S | Pt100 | Rh/Fe" pH: "Hg/Hg2Cl2 | Ag/AgCl |
ISFET" Ion selective electrode: "specify species; e.g. Ca2+" Magnet-
ic field: "Hall" Surface pressure: "wilhelmy plate" }?
</type>
```

Example: in `NXinsertion_device.xml` the type enumeration is seemingly unbounded, as indicated with '…':

```
<NXinsertion_device name="{Name of insertion device}">
 <type type="NX_CHAR">
      {"undulator"|"wiggler"|...}?
 </type>
```

This case is difficult to capture in the XML Schema language.

Example: in `NXshape.xml` a ',' separator is used instead of a '|'[9]:

```
 {"nxcylinder", "nxbox", "nxsphere", ...}?
```

The generative approach provides systematic diagnostics on such inconsistencies in representations of enumerations in the templates.

2.6) Global and additional data and group attributes

Beyond 'name' for groups and 'units' and 'type' for data items, the NeXus templates employ additional global and data attributes[10], which are not yet handled by the NeXML transformer. The inclusion of most of these attributes (possibly leveraging the `NeXus.xsd` attributes definitions) is straightforward and not addressed yet here.

9 Although NeXus.xsd provides explicit NXshape, NXtranslation, and NXorientation classes, the NeXML generator is able to consistently generate these from correct XML templates anyway.

10 http://www.nexusformat.org/Design#NeXus_Attributes

3) EMF genmodel for the NeXML schema and the validating EMF instance model editor

The EMF provides for generation of a validating EMF instance model editor corresponding to the EMF "genmodel" of a loaded XML Schema. The loaded NeXML and NeXus base schemas (as an EMF genmodel) and an example NeXML instance model are shown in the following screenshots:

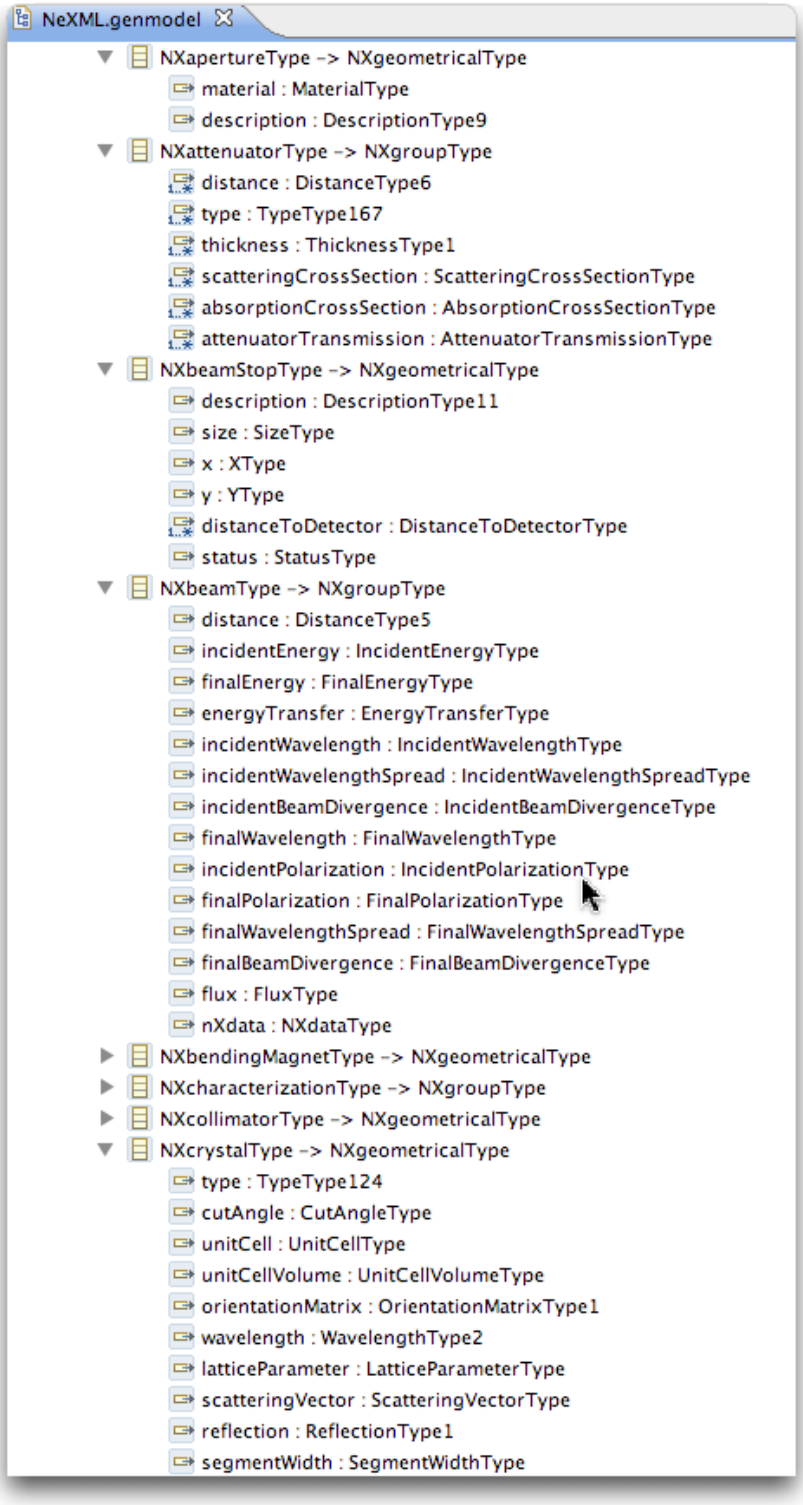


Illustration 11: EMF genmodel of the NeXML schema with NX group classes and data item and group child elements.

In order to capture type, units, and length restrictions specific to data items, the NeXML transformer generates many dedicated anonymous complex types for data items. The EMF genmodel loading process gives these data items unique names independent of the parent group context:

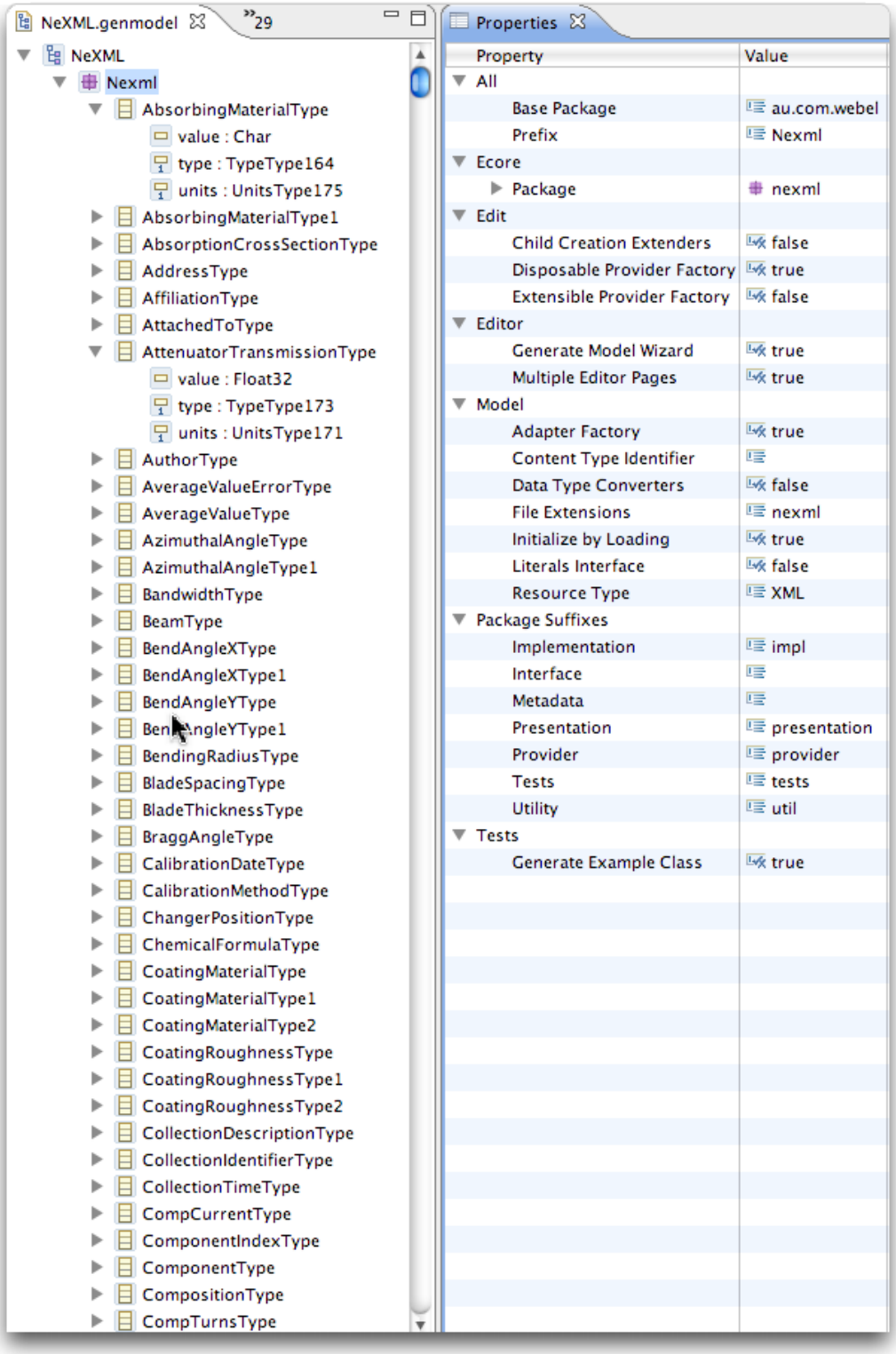


Illustration 12: Selected generated NeXML data items in the EMF schema genmodel. Note the repetition with variation of data item types (used in different parent group contexts, and the reference to data-item specific unit and NAPItype types.

There appears to be significant redundancy in many data item definitions, however closer inspection shows that there are subtle variations between the data item definitions in different parent group context, complicating attempts to reuse global data item definitions such as the `DescriptionType` below:

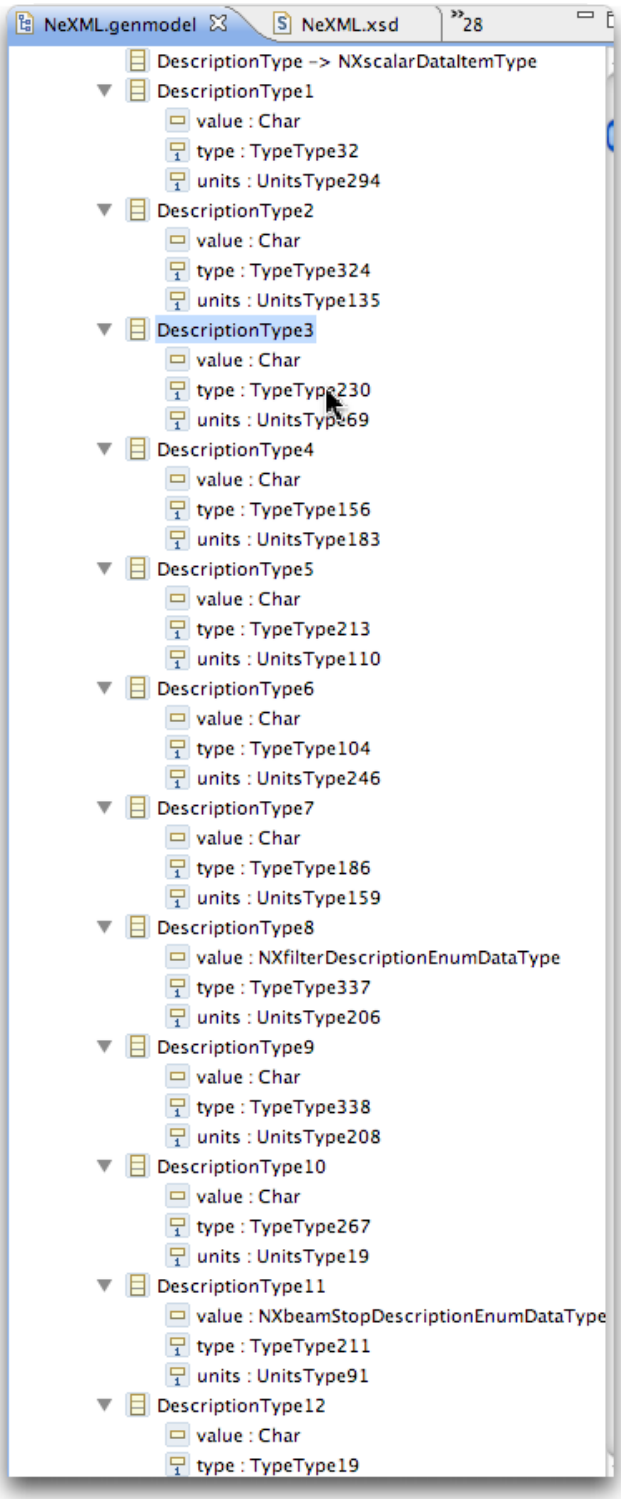


Illustration 13: some (but not all) Description data items are restricted by enumeration

The EMF genmodel correctly identifies enumeration literals in the NeXML schema, however they are compromised by inconsistencies in the NeXus XML templates and the NeXus design:

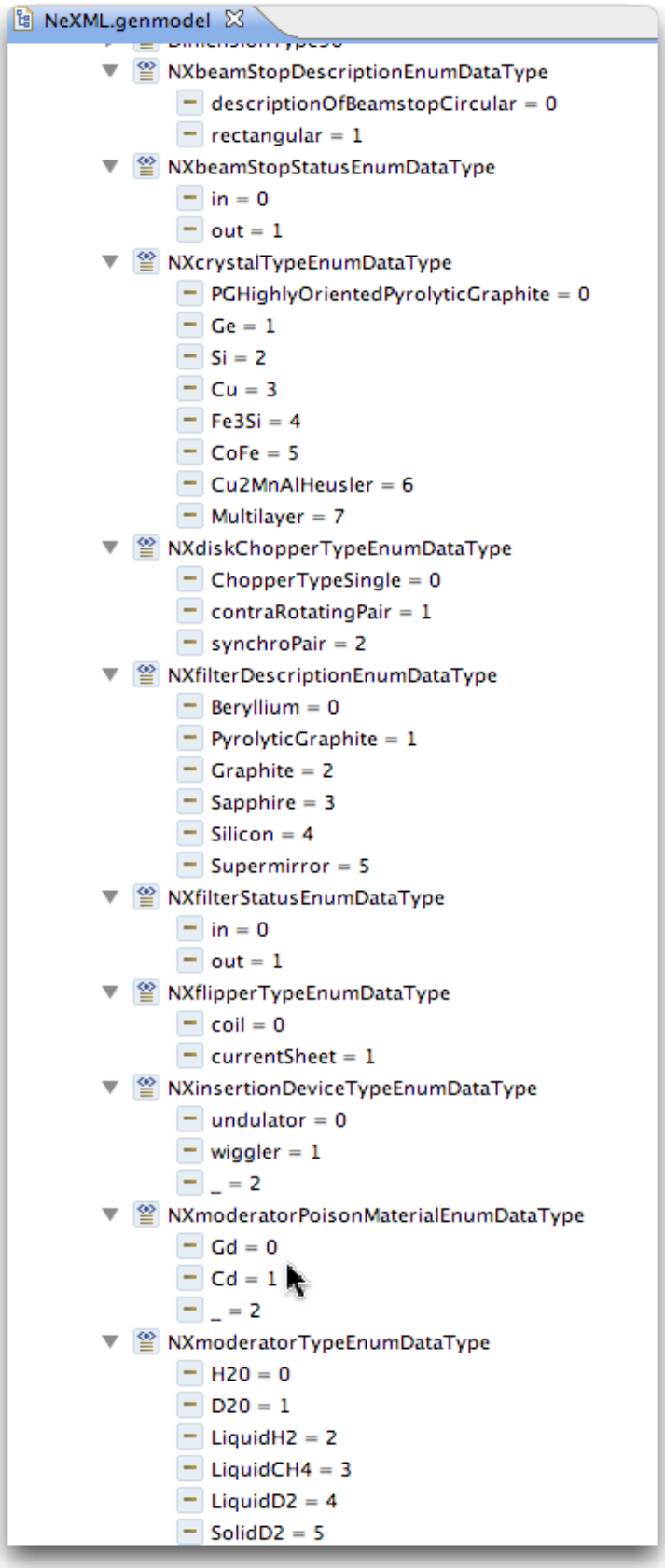


Illustration 14: Enums in the EMF genmodel for NeXML

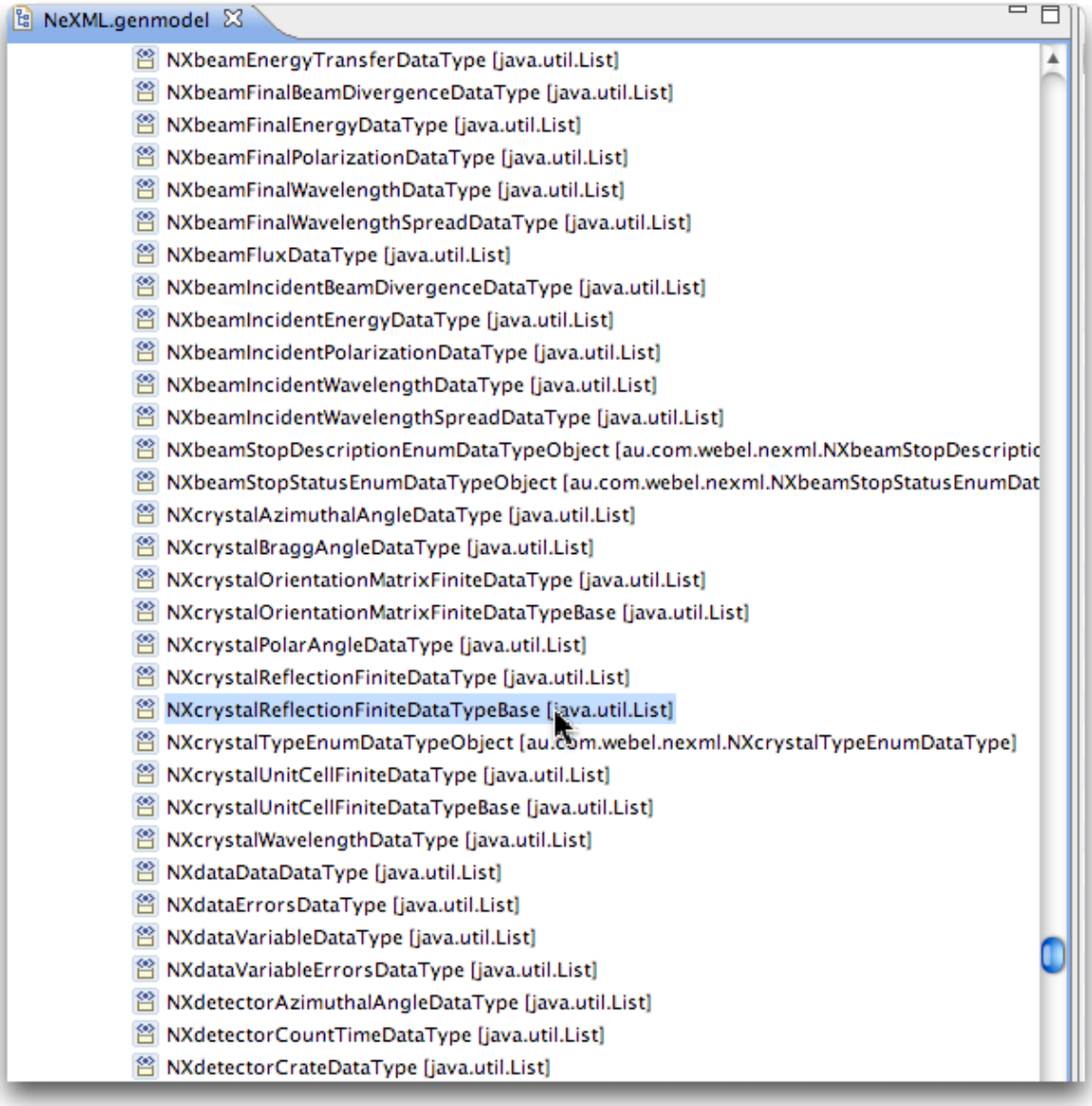


Illustration 15: EMF genmodel encapsulating per-DataItem lists as Java list types

A current inefficiency in the NeXML strategy is the need for hundreds of dataitem-specific simple types to capture restrictions in length, units, and NAPItypes. Future work should identify reusable type definitions while still offering data-item specific schema restrictions:

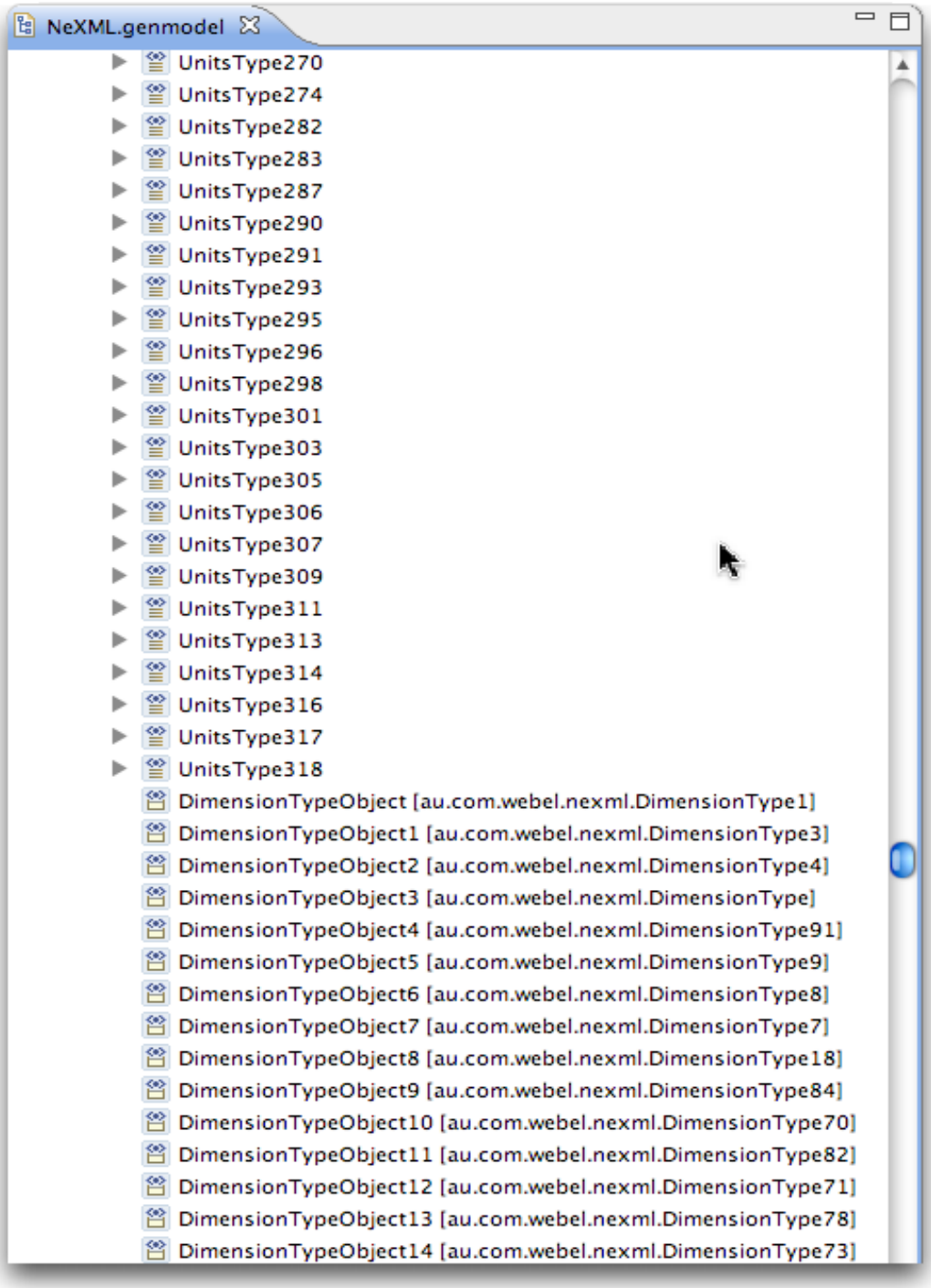


Illustration 16: many per-dataitem simple types capturing units and dimensions.

Since the `NeXML.xsd` imports the `NeXus.xsd` it is also loaded by EMF as a *genmodel*:

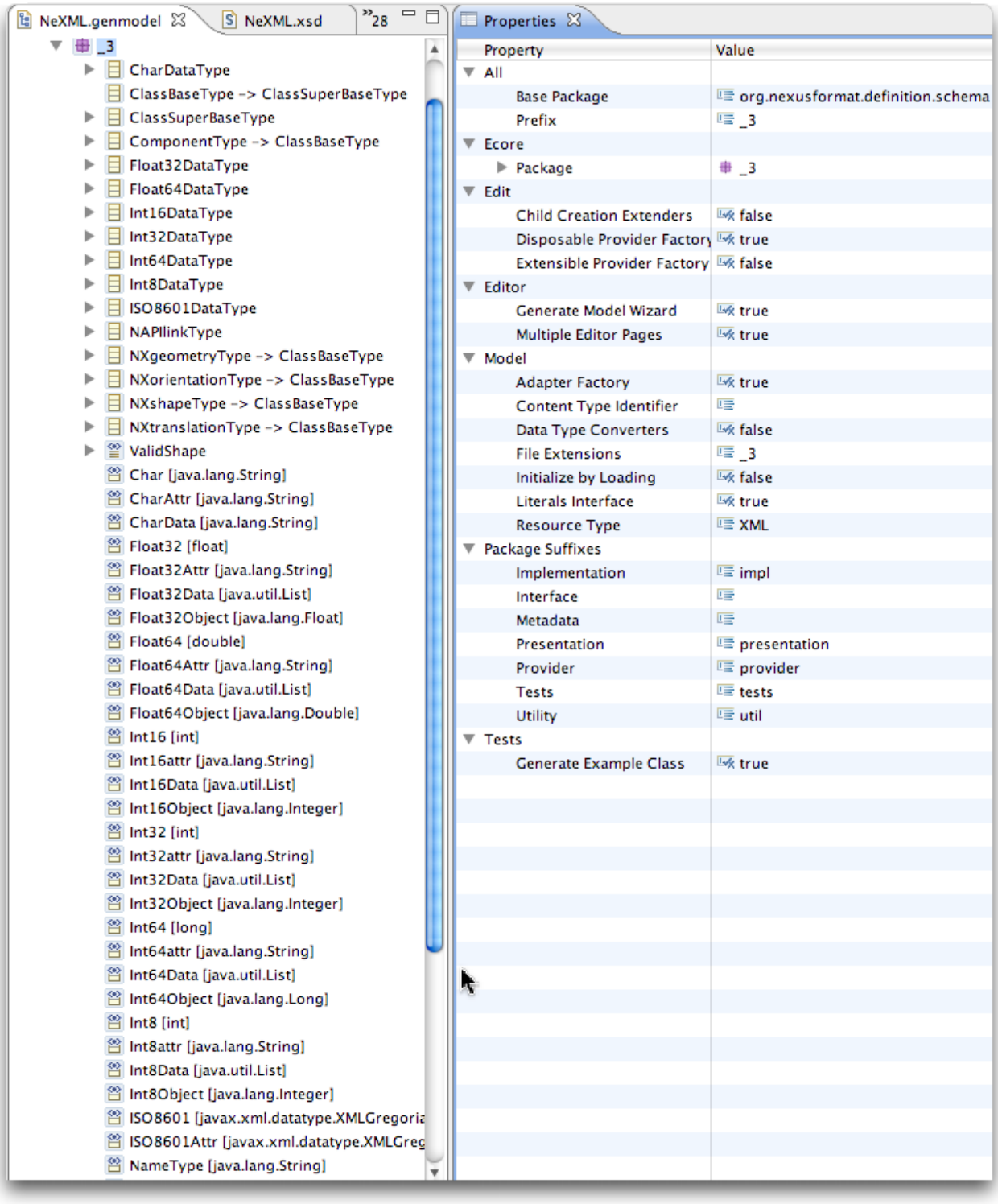


Illustration 17: For completeness the EMF genmodel import of the `NeXus.xsd` schema is shown

3.1) EMF instance model editor validated against the NeXML schema

Once an EMF genmodel has been loaded for the `NeXML.xsd` schema, corresponding `EObject` Java bindings and validating EMF model editor for the schema can be generated, which admits only construction of instance models corresponding to the NeXML schema:

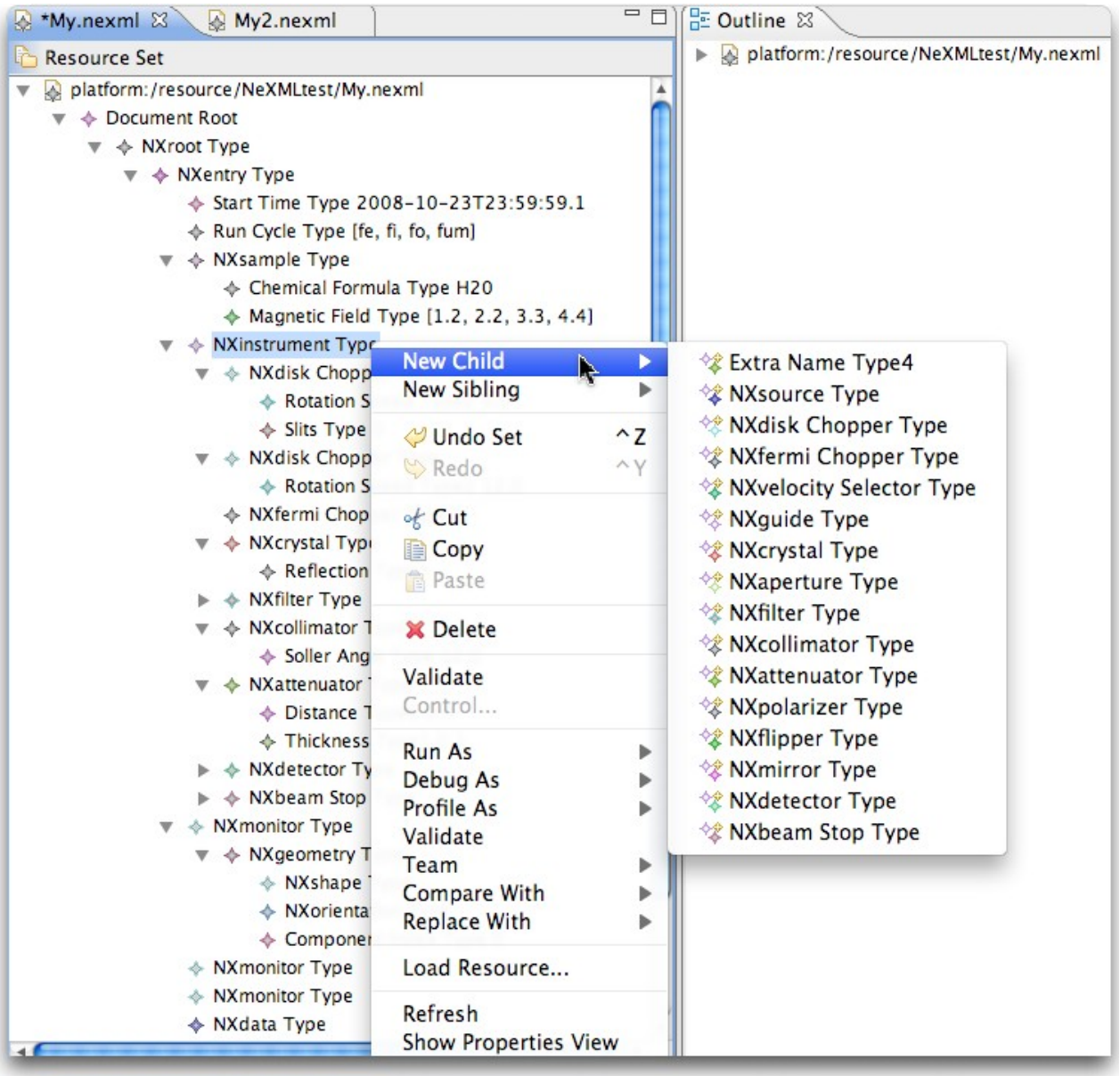


Illustration 18: A valid NeXML model with stub values in the generated EMF instance model

4) UML graphical representations of the reverse-engineered NeXML schema

UML class models of the NeXML schema provide alternative views and insights, and it is possible to also create structurally valid graphical NeXML/NeXus instance models that reflect the topology of real instrument and experiments. The following diagrams were prepared in MagicDraw UML[11], however similar results can be achieved in many UML2-compliant tools.

4.1) The NeXML Schema in UML class diagrams

On reverse to UML, XSD complex and simple types become stereotyped UML Classes, while XSD elements become stereotyped UML Properties of the Classes of those types. In class diagrams Properties can be displayed either as UML attribute Properties in the attributes compartment of a Class symbol, or as the Property end of a UML Association, or as a part Property symbol in the structure compartment of a Class. XSD extension is shown using a stereotyped UML Generalization relationship:

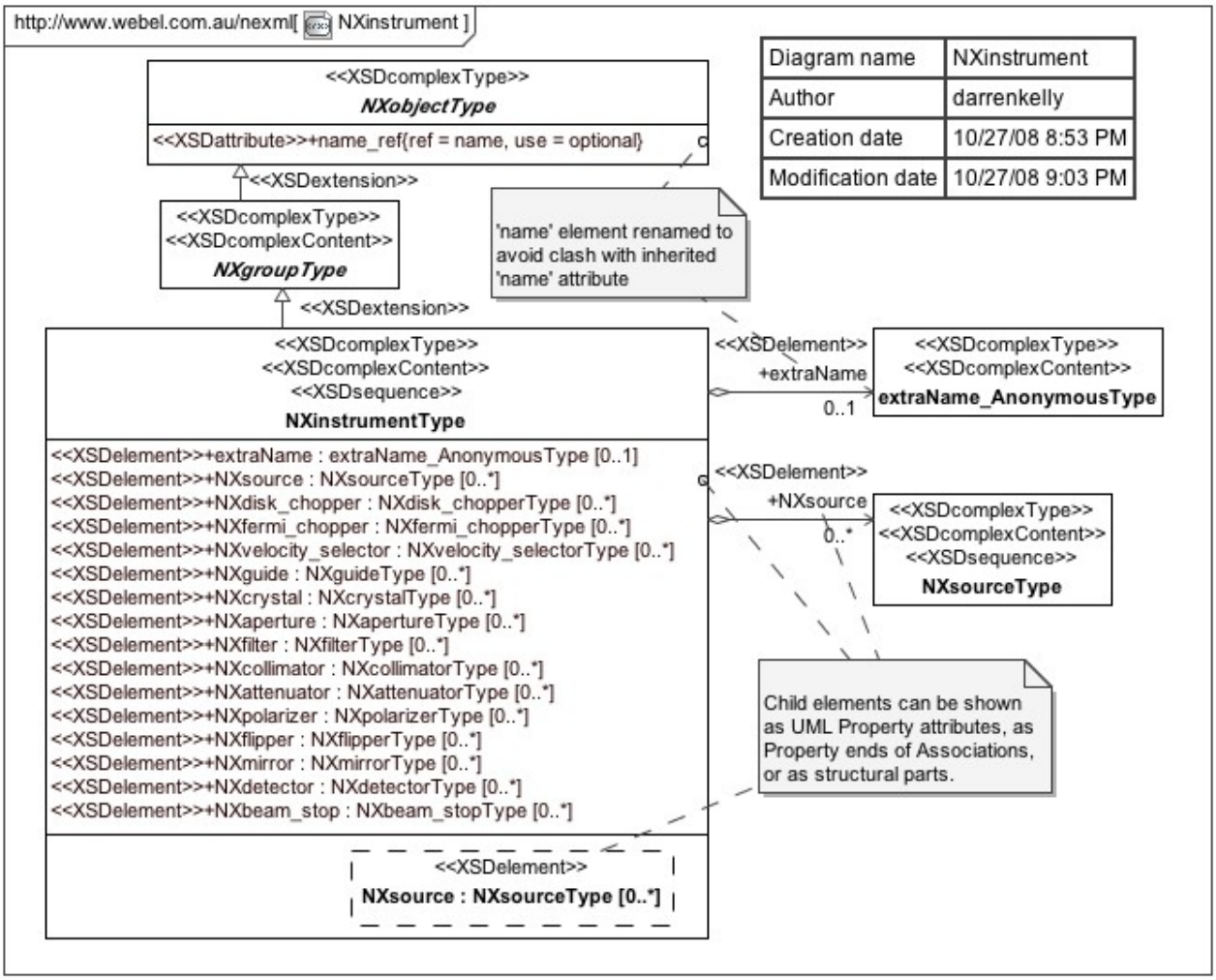


Illustration 19: UML class diagram of aspects of `NXinstrumentType` *from the NeXML schema*

11 http://www.magicdraw.com

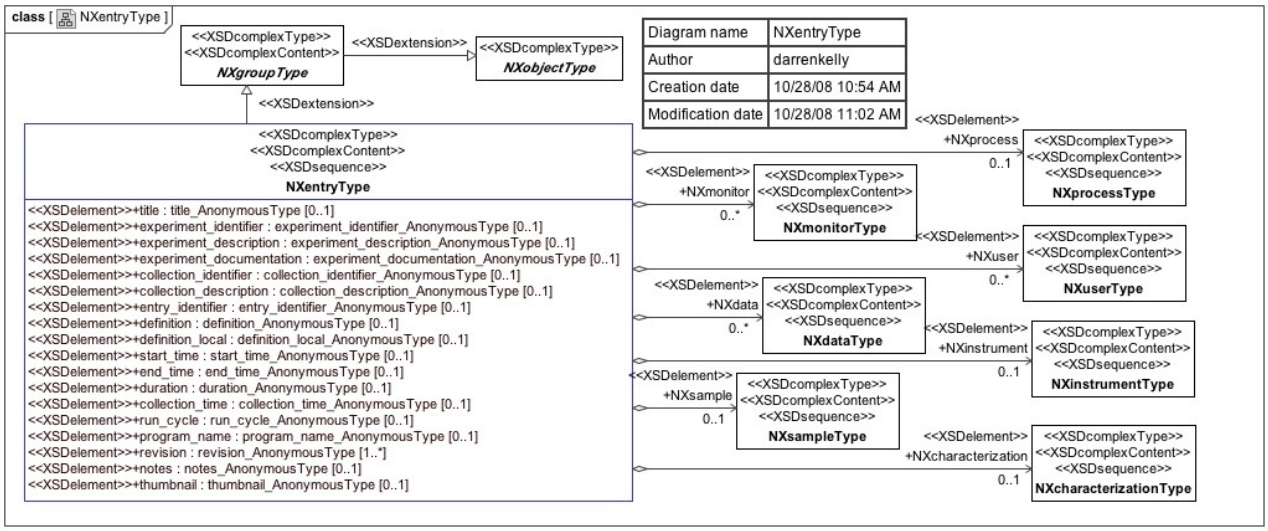


Illustration 20: UML "focus" class diagram of `NXentryType` *from the NeXML schema*

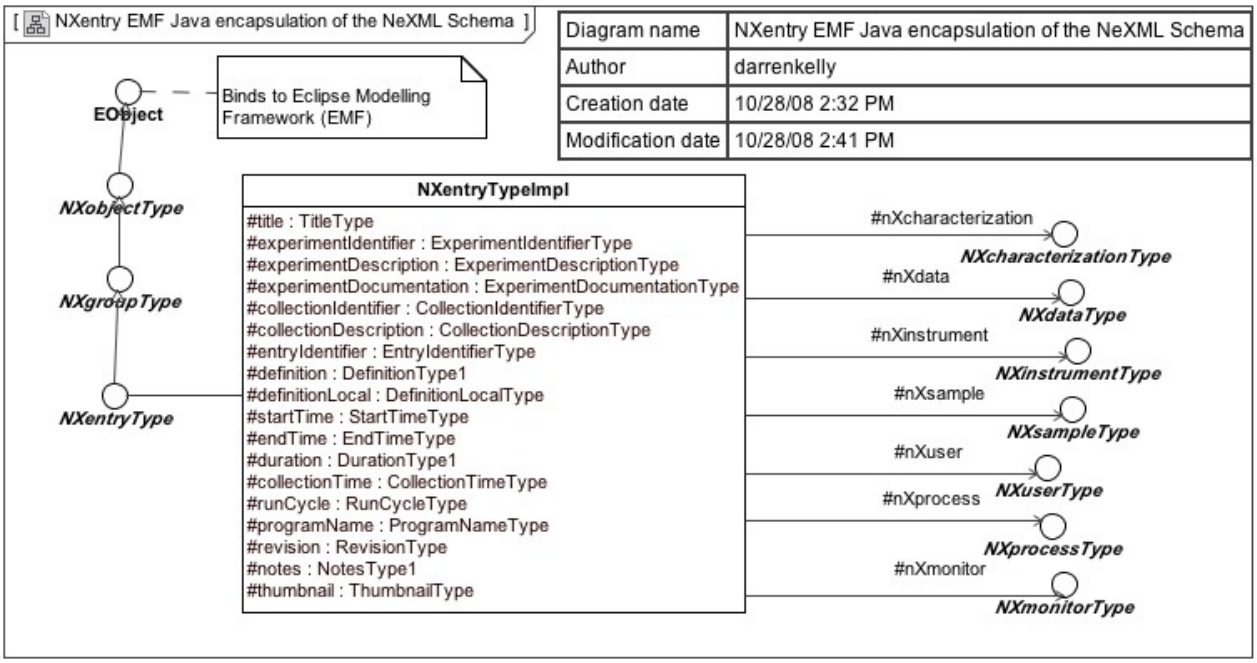


Illustration 21: Java bindings for part of the NeXML Schema in EMF, reverse engineered into UML, with EMF interfaces (that are `EObjects`*) and an automatically generated implementation*

4.2) Per-DataItem units, dimensions, types, and value choices in UML

It has already been demonstrated that NeXML employs a pragmatic recipe for restricting units, possible values, dimensionality of data value lists, and types, by generating specific simple types per data item, which can then be extended as complex types to mix in other attributes. While UML supports redefinition and restriction of properties in inheriting subclasses, this is harder to achieve in XML Schema. The current strategy is illustrated below for selected data item examples:

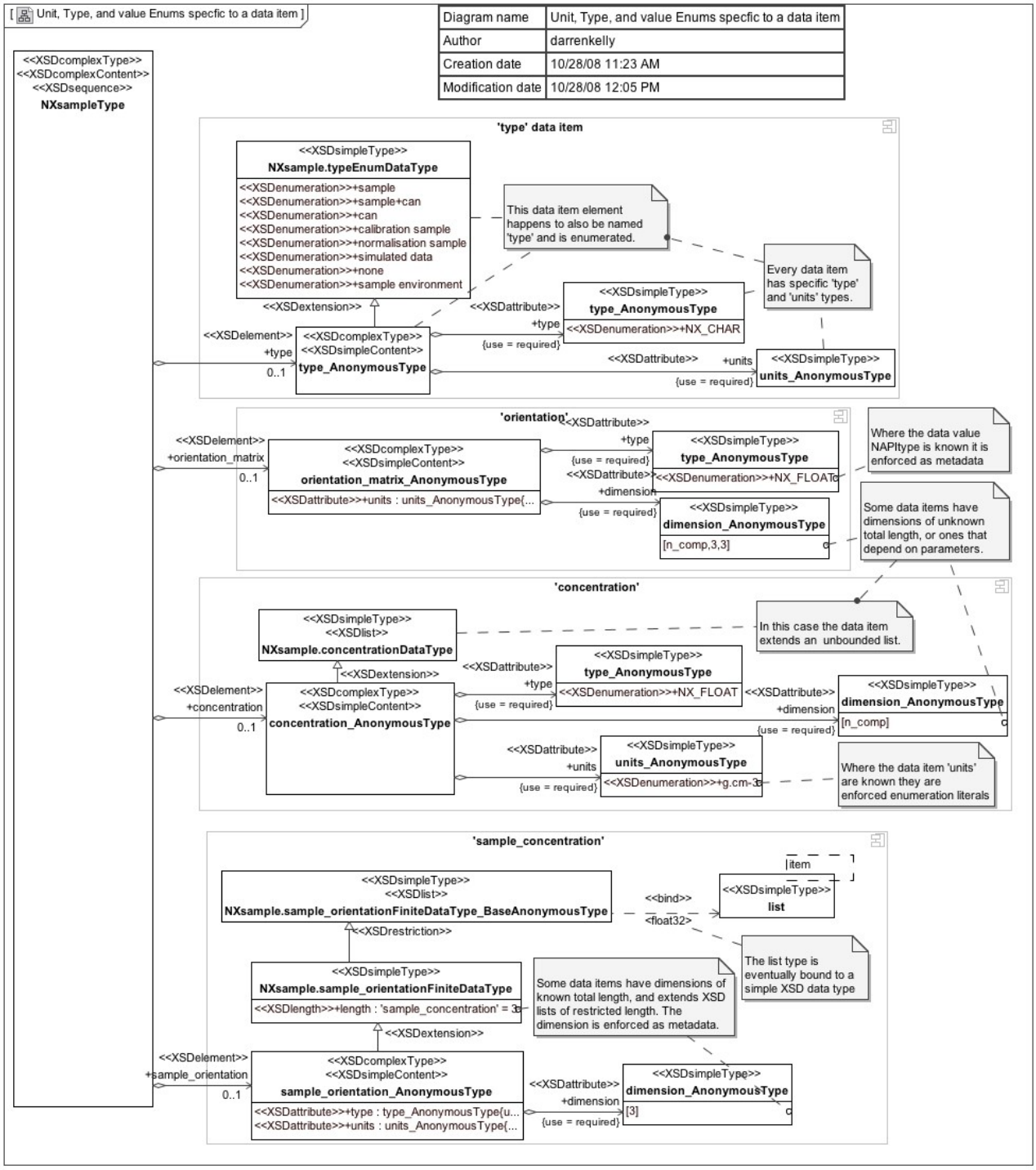


Illustration 22: UML overview of the NeXML recipe for capturing units, types, dimensions, and values specific to each data item, with complex types extending restricted simple types.

4.3) The NeXML schema as UML composite structure

There is an exact correspondence between attribute Properties of a Class (listed in the attributes compartment of a Class symbol) and part Property symbols in the structure compartment of a Class (or alternatively within the frame content in a framed composite structure diagram). It is possible to apply this principle using deep nesting of part Property symbols to create a graphical view of an entire XML schema in UML, or just a subset of the schema, with valid structure ensured by the UML2 compliant tool:

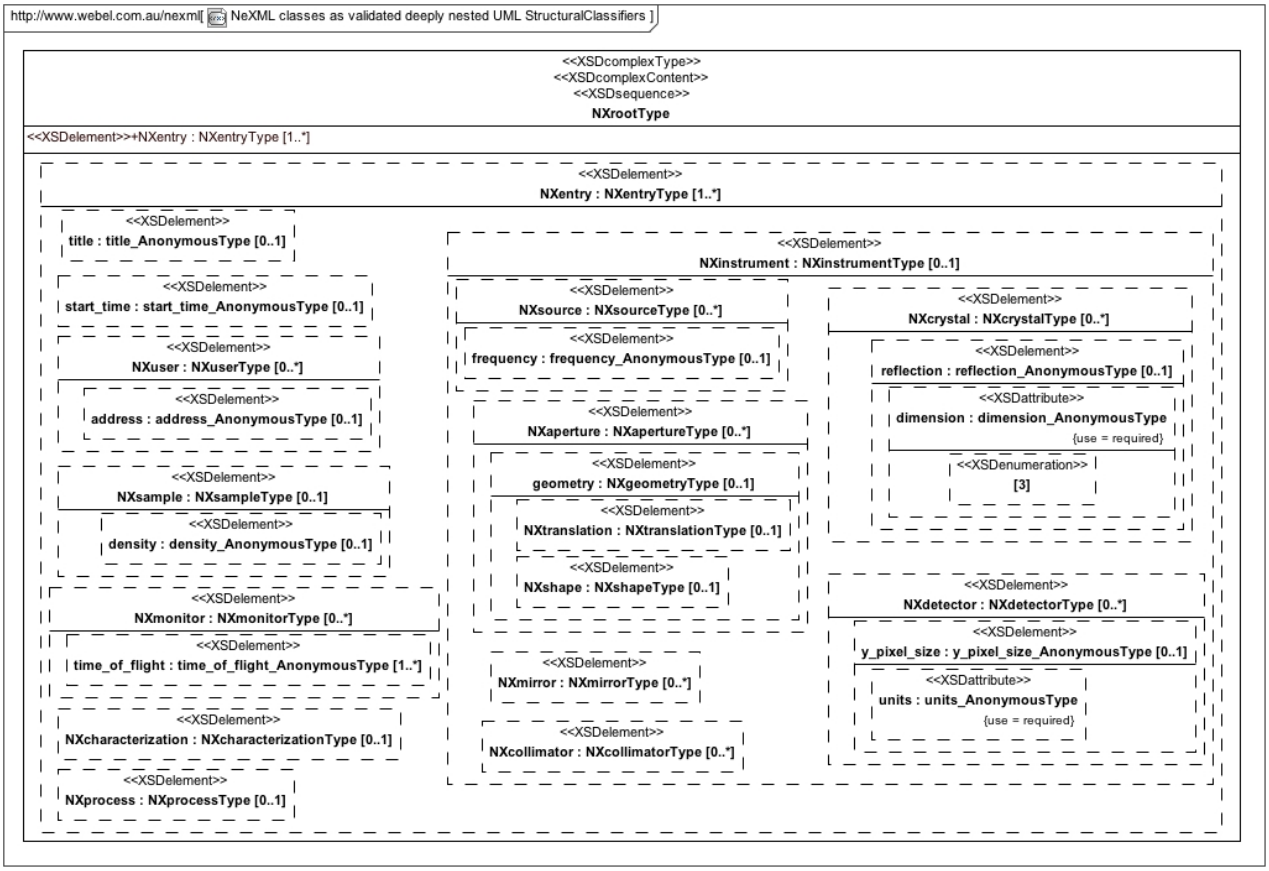


Illustration 23: UML composite structure diagram of reverse engineered NeXML Schema types, showing deep nesting of validated hierarchical schema structure with multiplicities

4.4) UML object diagrams of instruments in NeXML

UML InstanceSpecifications typed by the Classes corresponding to reverse engineered NeXML types may be used as "objects" in class diagrams to construct views of instruments that reflect the beam topology and instrument geometry. The graphical containment of instances within instances is merely cosmetic, so that one is free to group objects logically and/or physically without reference to the NeXML schema structure; the assignment of instances to UML Slots of parent instances is strictly validated against the Property structure of the Classes typing the instances:

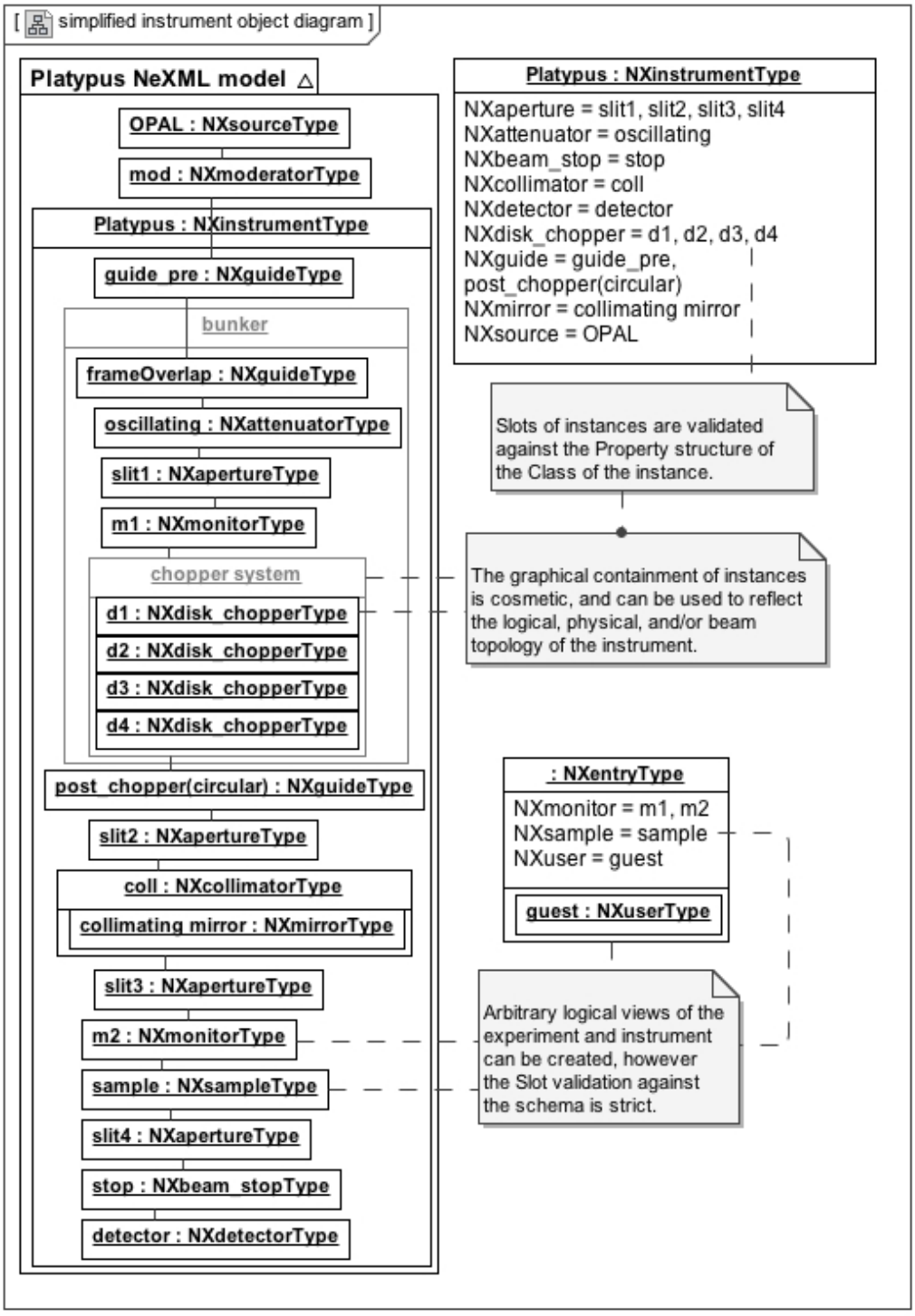


Illustration 24: Simplified UML object model of the Platypus reflectometer at OPAL using InstanceSpecifications typed by Classes of NeXML schema types

5) Conclusion

The advantages of a generative approach to migration and/or transformation of the NeXus base class definitions from “meta-DTD” XML templates to the XML Schema language are compelling. As a general principle, there is nothing that can be done manually that can't be done generatively, while the generative approach brings with it many advantages such as:

1. A very uniform, consistent, and binding-friendly XML schema.
2. The possibility of continuation of NeXus class definitions in improved versions of the NeXus XML templates.
3. Identification of inconsistencies in the existing XML templates.
4. Identification of weaknesses in the existing NeXus design.

The priority should therefore be on establishing the thinnest possible `NeXus.xsd` base schema upon which the generative NeXML approach can build. To this end at least the following improvements in the NeXus XML templates and design are necessary:

1. A clear inheritance notation that can be mapped to XSD extension[12].
2. Separation of NAPItype indications from indications of dimensionality in the 'type'.
3. Formal definition of parameters upon which attributes depend, especially as used in indications of dimensionality, so that XML keyrefs can be used to correlate sizes.
4. Consistent review of documentation and embedded metadata (like multiplicities)[13].
5. Binding of the units to a digitally robust specification.

It has been further suggested that the technologies presented here could equally be employed to transform the UDUNITS XML database to an XML Schema with EMF bindings.

6) Acknowledgements

Thanks to Nick Hauser, NBI Computing and Electronics, Australian Nuclear Science and Technology Organisation (ANSTO), for hosting NIAC2008 in Sydney and for many discussions on the NeXus format and the generative NeXML strategy. Thanks to Peter Petersen, Spallatial Neutron Source (SNS), as chair of NIAC for receiving a presentation on NeXML at NIAC2008.

12 Kelly has promoted this strategy since 2005 and presented numerous UML and Java examples at NIAC2006.

13 Kelly has made the case for a simple multiplicity attribute that can be mapped to UML or XML Schema.

7) References

[1] NeXus project web site
http://www.nexusformat.org

[2] Eclipse Foundation XML Schema Definition (XSD) reference library project
http://www.eclipse.org/modeling/mdt/?project=xsd#xsd

[3] 2007 Kelly, D. R. C., Hauser N.,
"NeXusBeans: object-oriented software components for the NeXus scattering science data format using Java, UML, and XML Schema technologies",
Australian Nuclear Science and Technology Organisation (ANSTO)
NBI Computing and Electronics Group, Technical Report

[4] NeXML: UML-driven Java NeXusBeans and XML schema for the NeXus format
Project web site
http://www.webel.com.au/nexml

[5] Unified Modeling Language (UML)™, Object Management Group (OMG)
http://www.uml.org

[6] Eclipse Modeling Framework Project (EMF)
http://www.eclipse.org/emf

[7] W3C XML Schema Working Group
http://www.w3.org/XML/Schema

[8] XStream Java API for XML serialization
http://xstream.codehaus.org/

[9] http://svn.nexusformat.org/definitions/trunk/schema/

[10] 2004 Bracha G.,,
Generics in the Java Programming Language
http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

[11] XML Schema Part 0: Primer Second Edition
http://www.w3.org/TR/xmlschema-0/#element-all

[12] The Unidata UDUNITS Package (experimental version)
http://www.unidata.ucar.edu/software/udunits
http://www.unidata.ucar.edu/software/udunits/udunits-2/udunits2.html

[13] NIST, Units Markup Language (UnitsML)
http://unitsml.nist.gov/

[14] The OMG Systems Modeling Language (OMG SysML™)
http://www.omgsysml.org/

[15] SUN JavaBeans Technology Specification
http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html