Technical Report

An XML Schema Description of Graph Transformation Systems

by

Dániel Varró,
András Pataricza

December, 2000

Budapest University of Technology and Economics
Department of Measurement and
Information Systems

Abstract

XML Schema is a candidate recommendation of the World Wide Web Consortium introduced to provide a type system that would enhance the construction and validation of XML documents.

During the APPLIGRAPH Subgroup Meeting on Exchange Formats for Graph Transformation, the work package "XML Schema and its Conformance with MOF Metamodels" was constituted in order to assess whether it is possible to make an XML Schema description for graph transformation systems.

The current report summarizes our work on the topic in the following way.

- The main features of XML Schema documents are discussed from a modelling aspect.
- A sample XML Schema encoding of the improving metamodel of graph transformation systems is provided. [1]

Contents

[1] This work was supported by the Hungarian National Scientific Foundation Grant OTKA T030804

1 Introduction

A first step towards a standardized, common model interchange format providing a uniform graph description and rule representation [2] that is capable of handling the most fundamental concepts of graph transformation was carried out in during the APPLIGRAPH Subgroup Meeting on Exchange Formats for Graph Transformation [1]. A potential candidate is the novel standard of the web, the Extensible Markup Language (XML), which allows the interchange of models in a distributed environment (i.e. Internet).

During the meeting a first version of a MOF metamodel for graph transformation systems (GXL-GTXL) was constituted in the form of UML Class Diagrams. Unfortunately, high level UML diagrams do not provide a straightforward encoding to DTDs (describing the structure of XML documents).

- Automatically generated XMI descriptions (which provide the closest correspondence with MOF metamodels with respect to modelling concepts) are verbose in contrast with manual DTD encodings.
- Compact DTDs derived manually from the GXL-GTXL metamodel are inconsistent from certain aspects with the original high abstraction level metamodel, which fact originates in the lack of a fine–grained type and inheritance system in DTDs.

At the Appligraph meeting, a work package was opened to reveal the possibilities and limitations of the novel evolving method for document validation XML Schema. Additionally, a correspondence was to be identified between UML based MOF metamodels and XML Schema constructs.

The aim of the current report is to give an overview on the major concepts of XML Schema, moreover, to construct a partial encoding of the GXL-GTXL metamodel using XML Schema.

The current report is **NOT**

- a precise specification of the XML Schema language;
- a complete description of XML Schema datatypes or structures;
- an exhaustive encoding of the GXL-GTXL metamodel.

It rather provides a general introduction of major concepts from a modelling point of view by several examples taken from different sorts of graphs. In addition, the sample XML Schema description of graph transformation systems is just one possible (and partial) encoding mainly for demonstration purposes. However, we tried to keep the GXL DTD and our XML Schema description as consistent as possible.

The current report is structured as follows. From Section 2 to Section 4, a brief introduction to the major concepts of XML Schema is provided with several examples. Section 5 contains an XML Schema description of graphs conforming to the GXL-GTXL metamodel. Finally, Section 6 concludes our paper.

2 XML Schema: Basic Concepts

XML Schemata are an XML dialect for describing and constraining the content of XML documents. XML Schemata are currently in the Candidate Recommendation phase of the W3C development process which means that the XML Schema Working Group considers it to be stable and encourages to comment on it.

2.1 A Sample XML Document

The concepts of writing an XML Schema are usually introduced through a sample XML document. Let us take a graph model for our purpose.

Example 2.1. *Our sample XML document contains a directed graph (`g_0`) with two nodes (`n_1`, `n_2`) connected by an edge (`e_3`) leading from `n_1` to `n_2`. A unique identifier is attached to each edge as indicated by the boolean attribute of graphs (`edgeids`).*

```
<?xml version="1.0"?>

<!-- Ids are attached to edges in the current graph model -->
<Graph edgeids="true">

  <!-- A node element of type SimpleState -->
  <Node id="n_1">
    <type>
      SimpleState
    </type>
  </Node>

  <!-- A node of type CompositeState -->
  <Node id="n_2">
    <type>
      CompositeState
    </type>
  </Node>

  <!-- An edge between node n_1 and n_2 -->
  <Edge id="e_3" from="n_1" to="n_2" />
</Graph>
```

The example contains a main element `Graph`, and subelements `Node` and `Edge`, which may contain in turn other subelements (e.g. `Type`). Identifiers (`id`) are also attached to these elements.

2.2 XML Schema Basics

An XML Schema itself is a well–formed XML document thus it is composed of a hierarchy of XML elements and XML attributes attached.

An XML Schema is embedded into a top–level `schema` element, which is enclosed between a **start** and an **end tag**.

Example 2.2. *A sample top–level XML Schema element (`xsd:schema`)*

```
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
</xsd:schema>
```

The starting `schema` element has a prefix `xsd:` which is associated with the XML Schema namespace [4] by the attribute declaration

`xmlns:xsd="http://www.w3.org/2000/08/XMLSchema"`, that appears in the schema element. The prefix `xsd:` is used by convention to denote the XML Schema namespace (see Section 3 for namespaces in XML Schema), although any prefix could be used. The same prefix, and hence the same association, will also appear in the names of built-in simple types, like e.g. `xsd:string`.

2.3 Element and Attribute Declarations

The major drawback of DTDs originates in the lack of a fine–grained type system which hinders the use of object–oriented techniques and hierarchical composition for document design. To avoid this drawback, the concepts of simple and compound types were introduced in XML Schemata.

Hence, in XML Schema, there is a major distinction between **definitions** (which create new types), and **declarations** which enable elements and attributes with specific names and types to appear in document instances either by direct composition (i.e. embedded elements) or using references (via e.g. the well–known `IDREFS` mechanism in instance document instances).

- New complex types are defined by using the `complexType` element and such **definitions** typically contain a set of *element declarations*, *element references*, and *attribute declarations.*
- The **declarations** are not themselves types, but rather an *association* between a name and the constraints which govern the appearance of that name in documents.

Elements are declared using the element `element`, while attributes are declared using the `attribute` element similarly to the well–known `!ELEMENT` and `!ATTLIST` tags in traditional DTDs.

Example 2.3. *Sample* ***element declarations*** *(`Node`, `Edge`, and `Graph`) and* ***attribute declarations*** *(`edgeids`) are provided (where `xsd:boolean` is a built–in type, while the definitions of `NodeType`, `EdgeType` and `GraphType` will come later).*

```
<!-- Element declarations: Node, Edge, Graph -->
<xsd:element name="Node" type="NodeType"/>
<xsd:element name="Edge" type="EdgeType"/>
<xsd:element name="Graph" type="GraphType" />

<!-- Attribute declaration edgeids -->
<xsd:attribute name="edgeids" type="xsd:boolean"/>
```

Table 1 gives a guideline for distinguishing between types and elements, declarations and definitions, XML Schemata and XML document instances (which all take the form of special XML elements).

2.4 Complex Type Definitions

As there is a large scale of possible current (and future) applications of XML documents, the content of an XML element may also take its value from common basic types (like integers, strings or e.g booleans), alternatively, they can be constructed from other (usually more basic) elements in a hierarchical way following a strict tree structure.

Thus, in XML Schema, there is a basic difference between **complex types** (which are allowed to contain subelements and carry attributes), and **simple types** (which cannot contain neither subelements nor attributes but numbers, strings, dates, etc.).

XML Schema	XML document (instance)
element declarations (`xsd:element name="Node"`)	elements (`<Node id="n1"> </Node>`)
attribute declarations (`xsd:attribute name="edgeids"`)	attributes (`<Graph edgeids="true">`)
simple type definitions (`xsd:boolean`)	simple data values (`"true"`)
complex type definitions (`xsd:complexType name="GraphType"`)	embedded elements (`<Graph> <Node> </Node> </Graph>`)

Table 1: An overview of elements and types

- Each declared element is assigned to a specific (either simple or complex) type (defining the content of the element in an instance document). In Example 2.3, a complex type called `NodeType` was assigned to the `Node` element.
- Attributes are always associated with a simple type (either built–in or derived). In Example 2.3, the attribute element `edgeids` had a built–in simple type.

New complex types are defined by using the `complexType` element, and such definitions typically contain a set of *element declarations*, *element references*, and *attribute declarations* (see Section 2.3 for these concepts).

Example 2.4. *The following example is a* ***complex type definition*** *for graphs containing two* ***element references*** *(`Node` and `Edge` indicated by the `ref` attribute of their element declaration) and a boolean attribute called `edgeids`.*

Please note that this definition is only valid if schema elements `Node` and `Edge` are declared somewhere in the XML Schema (see Example 2.3 for their declarations).

```
<!-- Complex type definition: GraphType -->
<xsd:complexType name="GraphType">

  <!-- Element contents should appear in the given sequence -->
  <xsd:sequence>

    <!-- A graph may contain an arbitrary number of Node elements -->
    <xsd:element ref="Node" minOccurs="0" maxOccurs="unbounded"/>

    <!-- ...followed by an arbitrary number of Edge elements
    <xsd:element ref="Edge" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>

  <!-- The boolean attribute edgeids has the default value false -->
  <xsd:attribute name="edgeids" type="xsd:boolean"
                 use="default" value="false"/>
</xsd:complexType>
```

The previous type definition also sets up some special constraints on elements appearing in instance documents, which restrictions are discussed later in more details.

- A `Graph` element of type `GraphType` may contain an arbitrary number of `Nodes` followed by an arbitrary number of `Edges`.
- Such a `Graph` element may have a boolean attribute `edgeids` of type `xsd:boolean`. If the attribute is omitted from an element in an instance document its value has to be considered as false (indicated by the attributes `use` and `value` in the definition part prescribing a default value for an attribute).

2.5 Simple Datatypes

In addition to embedded nodes and edges, an attribute called `edgeids` is also attached to a graph controlling the use of identifiers on edges. This attribute has a built–in simple type as indicated by the `xsd:boolean` type, thus in instance documents, `edgeids` may take the value of `true` or `false`.

XML Schema provides a large scale of built–in simple datatypes including for instance:

- string literals (`xsd:string`, etc.);
- number types (`xsd:int, xsd:float, xsd:boolean`, etc.);
- time values (`xsd:time, xsd:date`);
- XML types (`xsd:Name, xsd:ID, xsd:IDREF`);

New simple types can also be derived from existing simple types (built–in's and derived), see Section 4.2 for details.

2.6 Content Models

In traditional DTDs, the construction of a content model (including complex elements) was controlled by the sequence (`,`) and choice operators (`|`). For instance, a DTD prescribing that a `class` element is composed of a `teacher` and list of pupils (either `girls` and `boys`) would take the form:

```
<!ELEMENT class (teacher, (girl | boy)*) >
```

Naturally, the same concepts can be found in XML Schemata as well. XML Schema enables a group of elements to be defined and named, so that the elements can be used to construct complex types (thus mimicking common usage of parameter entities in XML).

- Contents of a **sequence** group are constrained to appear in the same order (sequence) as they are declared.

  In an instance document of the graph schema (Example 2.4), all the `Node` schema elements embedded into a `Graph` element have to precede the first occurrence of an `Edge` element, as prescribed by the built–in `sequence` schema element.

- The **choice** group element allows only one of its children to appear in an instance.

  For instance, we may allow for a `Graph` to contain its `Node` and `Edge` elements in an arbitrary order by the following group declarations:

```
<xsd:sequence>
  <xsd:choice>
    <xsd:element ref="Node" />
    <xsd:element ref="Edge" />
  </xsd:choice>
</xsd:sequence>
```

- The **all** group element (which provides a simplified version of the SGML &-Connector) allows all the elements in the group to appear simultaneously in a document or not at all, and they may appear in any order.

2.7 Element and Attribute Cardinality

Elements As graphs typically contain plenty of node and edge elements, the type definition for `Graph` elements should allow the embedding of nodes and edges in an arbitrary number into them. On the other hand, special graph types can be derived by restricting that a graph should contain at most 5 nodes or must not contain more than 28.

Such restrictions of element cardinality is indicated by two attributes: `minOccurs` and `maxOccurs` prescribing that

- the minimum appearance of an element should be greater than or equal to `minOccurs`;
- the maximum appearance of an element should be less than or equal to `maxOccurs`.

The attributes `minOccurs` and `maxOccurs` extend the concepts of at–most–once (`?`), at–least–once (`+`) and unbound (`*`) semantics widely used in traditional DTDs to restrict the appearance of embedded elements by allowing to prescribe cardinalities of any positive integers.

Example 2.5. *As a result of restrictions in Example 2.4, the appearance of a **`Node`** element in a **`Graph`** element is optional, and its upper limit is not constrained.*

```
<xsd:element ref="Node" minOccurs="0" maxOccurs="unbounded"/>
```

Attributes Attributes may appear once or not at all. Therefore the syntax for specifying occurrences of attributes is different from the syntax for elements. In particular, a `use` attribute is used in an attribute declaration to indicate whether the attribute is `required` or `optional`, and if `optional` whether the attribute's value is `fixed` or whether there is a `default`. A second attribute, `value`, provides any value that is called for according to Table 2.

Attributes (`use`,`value`)	Semantics
(optional, –)	attribute may appear once and it may have any value
(required, –)	attribute must appear once and it may have any value
(required, 19)	attribute must appear once, its value must be 19
(fixed, 19)	attribute may appear once; if it appears, its value must be 19
(default, 19)	attribute may appear once; if it does not appear, its value is 19

Table 2: Restrictions on attributes

Example 2.6. *For instance, the default value of* `edgeids` *in Example 2.4 is* `false` *(however, it was redefined in the sample XML document of Example 2.1).*

```
<xsd:attribute name="edgeids" type="xsd:boolean" use="default" value="false"/>
```

3 XML Schema: Namespaces and Qualification

XML Namespaces used in XML Schemata will surely play an important role in any standardization process. At first, a standardization committee should create a common metamodel of the field (e.g. graph transformation), Afterwards, they build up a standard XML Schema description containing a collection (denoted as **standard vocabulary**) of type definitions and element declarations (whose names belong to a particular namespace) for the major constructs. Finally, the committee should place the standard schema on the Web to be accessed by any application.

However, such a standard can never be complete (i.e. it will not contain those aspects of the field that are completely novel and thus not theoretically stable). Moreover, a large document standard surely decreases legibility and thus makes the construction of tools conforming the standard more difficult.

The concepts of XML Schema combined with XML Namespaces encourage the creation of a standard schema of medium size which can be extended by *local declarations* and type definitions including mainly tool–specific attributes and elements. The use of **target namespace** enables us to *distinguish between definitions and declarations from different vocabularies* (e.g. one target namespace for the standard and and another one for tool–specific local element declarations).

When the conformance of an instance document to one or more schemata is checked (through a process called **schema validation**), we need to identify which element and attribute declarations and type definitions in the schemata should be used to check which elements and attributes in the instance document.

The schema author also has several options that affect how the identities of elements and attributes are represented in instance documents. More specifically, the author can decide whether or not the appearance of locally declared elements and attributes in an instance must be **qualified** by a namespace, using either an *explicit prefix or implicitly by default.*q The schema author's choice regarding qualification of local elements and attributes has a number of implications regarding the structures of schemata and instance documents. A good practice can be to qualify the standard elements by a common prefix while using local elements in an unqualified form.

3.1 Target Namespace and Unqualified Locals

In a new version of our sample graph schema, we explicitly declare a target namespace, and specify that both locally defined elements and locally defined attributes must be unqualified. The target namespace in Example 3.1 is `http://www.example.hu/GXL-GXTL`, as indicated by the value of the `targetNamespace` attribute.

Qualification of local elements and attributes can be globally specified by a pair of attributes, `elementFormDefault` and `attributeFormDefault`, on the schema element, or can be specified separately for each local declaration using the form attribute. All such attribute values may be set to `unqualified` or `qualified` (in other words prefixed), to indicate whether or not locally declared elements and attributes must be qualified.

In Example 3.1, the qualification of elements and attributes are globally specified by setting the values of both `elementFormDefault` and `attributeFormDefault` to `unqualified`. (Strictly speaking, these settings are unnecessary because these values are the defaults for the two attributes.)

The example also demonstrates the concepts of **locally** and **globally declared elements**. The `Node` and `Graph` are global elements as they are declared in the context of the schema as a whole rather than within the context of a particular type. Meanwhile the `Edge` element is declared as local, in the context of the `Graph` element.

Example 3.1. *A graph schema demonstrating the use of target namespaces ("http://www.example.hu/GXL-GXTL"), qualified (e.g. Node) and unqualified names (e.g. Edge).*

```
<!-- Declaring namespaces (target, default) and unqualified element names -->
<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
        xmlns:gxl="http://www.example.hu/GXL-GXTL"
        targetNamespace="http://www.example.hu/GXL-GXTL"
        elementFormDefault="unqualified"
        attributeFormDefault="unqualified">

  <!-- Globally declared elements -->
  <element name="Node" type="gxl:NodeType"/>
  <element name="Graph" type="gxl:GraphType" />
  <complexType name="GraphType">
    <sequence>
      <element ref="Node" minOccurs="0" maxOccurs="unbounded"/>

      <!-- Locally declared element -->
      <element name="Edge" type="gxl:EdgeType"
               minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
  </complexType>
  ...
</schema>
```

- Starting from the end of the schema, a type called `GraphType` is defined first that consists of the globally declared `Node` and locally declared `Edge` elements. One consequence of this type definition is that the `GraphType` type is included in the schema's target namespace.
- Please note that the type references in the element declarations are prefixed, i.e. `gxl:GraphType`, `gxl:NodeType`, `gxl:EdgeType`, and the prefix is associated with the namespace `http://www.example.hu/GXL-GTXL`. This is the same namespace as the schema's target namespace, therefore a processor of this schema will know to search within *this* schema for the definition of the type `GraphType` and the declaration of the element `Node`.
- It is also possible to refer to types in another schema with a different target namespace, hence enabling re-use of definitions and declarations between schemas.
- Please also note, that the XML Schema elements are not prefixed this time, as the default namespace (`xmlns`) of the document is associated with `"http://www.w3.org/2000/10/XMLSchema"`.

When such a schema is populated into a conforming document instance (Example 3.2), the following changes can be observed.

Example 3.2. *A document instance of the XML Schema of Example 3.1 with qualified **`Node`** elements and unqualified local **`Edge`** elements.*

```
<?xml version="1.0"?>
<!-- An explicitly given namespace -->
<Graph xmlns:gxl="www.example.hu/GXL-GTXL" edgeids="true">
  <!-- Global elements are qualified (prefixed) -->
  <gxl:Node id="n_1" />
  <gxl:Node id="n_2" />

  <!-- The local Edge element is not prefixed -->
  <Edge id="e_3" from="n_1" to=""n_2" />
</Graph>
```

The instance document declares one namespace, `http://www.example.hu/GXL-GTXL`, and associates it with the prefix `gxl:`. This prefix applies to the global elements (`Graph` and `Node`). Furthermore, `elementFormDefault` and `attributeFormDefault` require that the prefix is *not* applied to any of the the locally declared elements such as `Edge`, and it is *not* applied to any of the attributes.

When local elements and attributes are not required to be qualified, an instance author may require more or less knowledge about the details of the schema to create schema valid instance documents. More specifically, if the author can be sure that only the root element (such as `Graph`) is global, then it is a simple solution to qualify only the root element.

Alternatively, the author may know that all the elements are declared globally, and so all the elements in the instance document can be prefixed, perhaps taking advantage of a default namespace declaration.

3.2 Qualified Locals

Elements and attributes can independently be required to be qualified, although the qualification of local elements are only described in the paper. To specify that all locally declared elements in a schema must be qualified, the value of elementFormDefault should be set to qualified.

Example 3.3. *Declaring qualified local elements (indicated by the true value attached to `elementFormDefault`)*

```
<!-- Elements have to be qualified in a conforming document -->
<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
        xmlns:gxl="http://www.example.hu/GXL-GXTL"
        targetNamespace="http://www.example.hu/GXL-GXTL"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified">

  <!-- Globally declared elements -->
  <element name="Node" type="gxl:NodeType"/>
  <element name="Graph" type="gxl:GraphType" />
  <complexType name="GraphType">
    <sequence>
      <element ref="Node" minOccurs="0" maxOccurs="unbounded"/>
```

```
      <!-- Locally declared element -->
      <element name="Edge" type="gxl:EdgeType"
               minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
  </complexType>
  ...
</schema>
```

In a conforming documents, all the elements have to be qualified explicitly (Example 3.4), alternatively, the explicit qualification of every element can be replaced with implicit qualification provided by a default namespace (Example 3.5).

Example 3.4. *A document with explicitly qualified elements (`gxl:Graph`, `gxl:Node` and `gxl:Edge`).*

```
<?xml version="1.0"?>
<!-- Elements with explicit qualification -->
<gxl:Graph xmlns:gxl="www.example.hu/GXL-GTXL" edgeids="true">
  <gxl:Node id="n_1" />
  <gxl:Node id="n_2" />
  <gxl:Edge id="e_3" from="n_1" to=""n_2" />
</gxl:Graph>
```

Example 3.5. *A document with implicit qualification provided by default namespace `xmlns="www.example.hu/GXL-GTXL"`*

```
<?xml version="1.0"?>
<!-- Elements with default namespace -->
<Graph xmlns="www.example.hu/GXL-GTXL" edgeids="true">
  <Node id="n_1" />
  <Node id="n_2" />
  <Edge id="e_3" from="n_1" to=""n_2" />
</Graph>
```

In the latter case, all the elements in the instance belong to the same namespace, and the namespace statement declares a default namespace that applies to all the elements in the instance. Hence, it is unnecessary to explicitly prefix any of the elements.

4 XML Schema: Advanced Modelling Concepts

Beside its type constructs, another major advance in XML Schema is its partial support for inheritance mechanisms widely used in object–oriented modelling. XML Schema allows to form more complex types from a base type by means of extension, or more specific types by restriction. In addition to these concepts, elements can be declared abstract thus preventing them to appear in instance documents.

Therefore, when local, tool–specific elements are derived from a global standard vocabulary, the derivation process is characterized by extending or restricting the existing element types to obtain the new local constructs. With this respect, local non–standard elements derived from standard ones can be used in document instances whenever a basic, standard element is expected without conflicts during the validation.

4.1 Extension

When a complex type is derived by **extension**, its effective content model is the content model of the base type plus the content model specified in the type derivation. Furthermore, the two content models are treated as two children of a sequential group.

The concepts of extension in XML Schemata resembles to extensional inheritance in object–oriented modelling where a new class is derived from existing ones by adding new properties and methods.

Extensions are declared by using a `complexContent` and an embedded `extension` element, which has a `base` attribute to indicate the base type from which the extended type is derived.

Continuing our graph example, the notion of graph elements will be introduced with textually typed nodes and edges. As types can be attached to both nodes and edges, a *"common superclass"* `GraphElement` is declared and `Node` and `Edge` elements are derived from them by extension.

Example 4.1. *A graph schema allowing the construction of (textually) typed graph elements (nodes and edges) by introducing the attribute* ***`Type`***.

```
<!-- Graph -->
<xsd:element name="Graph" type="GraphType" />
<xsd:complexType name="GraphType">
  <xsd:sequence>
    <xsd:element ref="GraphElement" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="edgeids" type="xsd:boolean"
                 use="default" value="false"/>
</xsd:complexType>

<!-- Common superclass: GraphElement -->
<xsd:element name="GraphElement" type="GraphElementType"/>
<xsd:complexType name="GraphElementType">
  <xsd:attribute name="Type" type="xsd:string"/>
</xsd:complexType>

<!-- Node element derived by extension-->
  <xsd:element name="Node" type="NodeType" />
  <xsd:complexType name="NodeType">
    <xsd:complexContent>
      <!-- Extension is based on Graph Elements
      <xsd:extension base="GraphElementType" />
    </xsd:complexContent>
  </xsd:complexType>

<!-- Edge element derived by extension -->
  <xsd:element name="Edge" type="EdgeType" />
  <xsd:complexType name="EdgeType">
    <xsd:complexContent>
      <xsd:extension base="GraphElementType">
        <!-- Adding new attributes -->
```

```
      <xsd:attribute name="from" type="xsd:IDREF" />
      <xsd:attrubute name="to"   type="xsd:IDREF" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

The original graph concept (Example 2.4) is altered, graphs contain `GraphElements` instead of `Nodes` and `Edges` distinctly. However, as nodes and edges are an extension of graph elements, they can be used to as a substitution for their "supertyped" objects.

This declaration also implies that nodes and edges can be embedded in a `Graph` element in an arbitrary order as only a sequence of `GraphElements` are prescribed in the `GraphType` definition. As a result, the following document conforms to the schema of Example 4.1.

Example 4.2. *An XML document conforming to the schema of Example 4.1. Please note that this time nodes and edges can be listed in an arbitrary order inside a* ***Graph*** *element.*

```
<?xml version="1.0"?>
<Graph edgeids="true"
       xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
  <!-- Graph element first contains a Node,
       the explicit indication of Node type is required (xsi:type)) -->
  <Node id="n_1" xsi:type="Node">
    <Type>
      SimpleState
    </Type>
  </Node>

  <!-- Secondly, Graph element contains an Edge -->
  <Edge id="e_3" from="n_1" to=""n_2" xsi:type="Edge"/>

  <!-- ... finally, another Node -->
  <Node id="n_2" xsi:type="Node">
    <Type>
      CompositeState
    </Type>
  </Node>
</Graph>
```

XML Schema allows to define the `Node` and `Edge` elements as `GraphElement` types in order to use instances of `Nodes` and `Edges` in place of instances of `GraphElement`. In other words, an instance document whose content conforms to e.g. the `Node` type will be valid if its content appears within the document at a location where a `GraphElement` is expected.

In order to identify exactly which derived type is intended to be used, the derived type must be identified in the instance document. The type is identified using the `xsi:type` attribute which is part of the XML Schema instance namespace. In Example 4.2, the use of `Node` and `Edge` derived types is identified through the values assigned to the `xsi:type` attributes.

4.2 Restriction

In addition to deriving new complex types by extending content models, it is also possible in XML Schema to derive both simple and complex types from existing types by restriction.

Although restrictive inheritance (e.g. a Baby is a Person whose ages is less than 2) in object–oriented modelling is not recommended to be used, it is still a common practice in XML Schemata.

Deriving simple types A common example, let us consider that only a subset of strings can be used as type names in Example 4.1 like `SimpleState`, `CompositeState`, etc. In this case, we may introduce a new simple type containing the enumeration of the strings allowed.

Restrictions are declared via the element `xsd:restriction` where its `base` attribute indicates the base type restricted to gain a new type.

Example 4.3. *Deriving a simple enumeration type (`TypeEnum`) by restriction from `xsd:Strings`.*

```
<!-- Declaration for GraphElement -->
<xsd:element name="GraphElement" type="GraphElementType"/>
<xsd:complexType name="GraphElementType">

  <!-- Attribute Type is of type TypeEnum -->
  <xsd:attribute name="Type" type="TypeEnum"/>
</xsd:complexType>

<!-- Simple type TypeEnum is derived by restriction -->
<xsd:simpleType name="TypeEnum">
  <!-- ... from the built-in string type -->
  <xsd:restriction base="xsd.string">

    <!-- Allowed values for enumaration -->
    <xsd:enumeration value="SimpleState"/>
    <xsd:enumeration value="CompositeState"/>
    <xsd:enumeration value="Transition"/>
  </xsd:restriction>
</xsd:simpleType>
```

An additional use of simple type restrictions is to prescribe a legal range of values for the new type derived from an existing simple type (e.g. Integer).

Supposing that we need to create a new type of integer called `myInteger` whose range of values is between 10000 and 99999. Such an integer type can be defined on the base of the built-in simple type integer, but the range of the integer base type is restricted by employing `minInclusive` and `maxInclusive` attributes:

Example 4.4. *Defining a new simple type `myInteger` for holding values between 10000 and 99999 in instance documents.*

```
<xsd:simpleType name="myInteger">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="10000"/>
    <xsd:maxInclusive value="99999"/>
  </xsd:restriction>
</xsd:simpleType>
```

XML Schema also has the concept of a list type. A list can be created from most of the atomic types, moreover, one can create new list types by a derivation from existing atomic (thus non–complex) types.

Example 4.5. *A list of `myIntegers`*

```
<xsd:simpleType name="listOfMyIntType">
   <xsd:list itemType="myInteger"/>
 </xsd:simpleType>
```

Restricting complex types In addition to deriving new complex types by extending content models, it is also possible to derive new types by restricting the content models of existing types.

A complex type derived by restriction is similar to its base type, except for that its declarations are more limited than the corresponding declarations in the base type. In fact, the values represented by the new type are a subset of the values represented by the base type. In other words, an application prepared for the values of the base type would not be surprised by the values of the restricted type.

When a complex type is derived by restriction, we may *attach types or defaults* to the new type which were undefined in the base type, or *constrain the cardinality* definition of the base type to a more restrictive one (in the new type).

For instance, we may define `UMLGraphElementTypes` as a restriction of `GraphElementTypes` constrained by allowing only UML types attached to nodes and edges and the presence of at least one `Node` or `Edge` element in `NEGraphType` (non–empty graph) elements are also prescribed.

Example 4.6. *Deriving complex types by restriction. A **`NEGraphType`** (non–empty graph) must contain at least one **`GraphElement`**, and **`UMLGraphElementTypes`** are allowed to have node and edge types of **`TypeEnum`**. (The definitions and declarations of Example 4.1 are referred.)*

```
<!-- A new complex type for Non-Empty Graphs (containing at least one node) -->
<xsd:complexType name="NEGraphType">
  <xsd:complexContent>

    <!-- Restriction is based on GraphType
    <xsd:restriction base="GraphType">
      <xsd:sequence>

        <!-- Cardinality is altered -->
        <xsd:element ref="GraphElement" minOccurs="1" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="edgeids" type="xsd:boolean"
                     use="default" value="false"/>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="GraphElementType">
  <!-- Attribute type of Type is currently undefined -->
  <xsd:attribute name="Type" />
</xsd:complexType>
```

```
<xsd:complexType name="UMLGraphElementType">
  <xsd:complexContent>
    <xsd:restriction base="GraphElementType">
      <!-- Attribute type of Type is defined -->
      <xsd:attribute name="Type" type="TypeEnum/>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

Please note that types derived by restriction must repeat all the components of the base type definition.

4.3 Abstract Elements and Types

XML Schema provides a mechanism to force the substitution for a particular element or type for a similar purpose to the abstract class declarations in object–oriented languages.

- When an element is declared to be "abstract", it cannot be used in an instance document.
- When an element's corresponding type definition is declared as abstract, all instances of that element must use `xsi:type` to indicate a derived type that is not abstract (See Section 4.1 for the use of `xsi:type`).

For instance, in our graph example, a `Graph` element should normally contain `Nodes` and `Edges`, while instances of `GraphElements` in graph documents should not be allowed.

Example 4.7. *An element declaration forbidding the appearance of* `GraphElements` *in instance documents. (A substitution of* `GraphElement` *in Example 4.1)*

```
<!-- A GraphElement must not appear in a valid document -->
<xsd:element name="GraphElement" type="GraphElementType"
             abstract="true"/>
```

Please remember that the type of elements in the role of a `GraphElement` (i.e. nodes and edges) has to be indicated in instance documents by the `xsi:type` attribute.

4.4 Summary of XML Schema

This section provided an introductory overview of constructing XML Schemas and documents conforming to a specific schema. Major concepts were discussed on an evolving example describing graph documents advancing step by step to an XML Schema specification of GXL-GTXL models.

However, the current paper is supposed to introduce the XML Schema specification from a modelling aspect, thus several concepts omitted from the paper (as not required to build a first version of GraTra XMLSchema model) are listed below.

- mixed content, empty content, null values
- redefining types and groups,
- substitution groups

- controlling type derivation, importing types
- specifying uniqueness

For these concepts, see [5, 6] for details.

5 An XML Schema Description of Graphs (GXL-GTXL)

In the current section, an XML Schema description of graphs is provided on the basis of the evolving GXL-GTXL metamodel of graph transformation systems. As the GXL-GTXL metamodel is under a rapid development, we have chosen to implement some core constructs, i.e. the description of graphs, which had already gone through a thorough discussion thus considered to be more or less stable for the graph transformation community.

5.1 The GXL Metamodel (Version 0.7.2)

At the APPLIGRAPH Subgroup Meeting on Exchange Formats for Graph Transformation [1] it was started to sketch a conceptual model to describe those aspects of a graphs which have to be exchanged by a common graph interchange format. Figure 1 and 2 show these conceptual models for graphs and the attribute structure (a slightly reduced version of [3]).

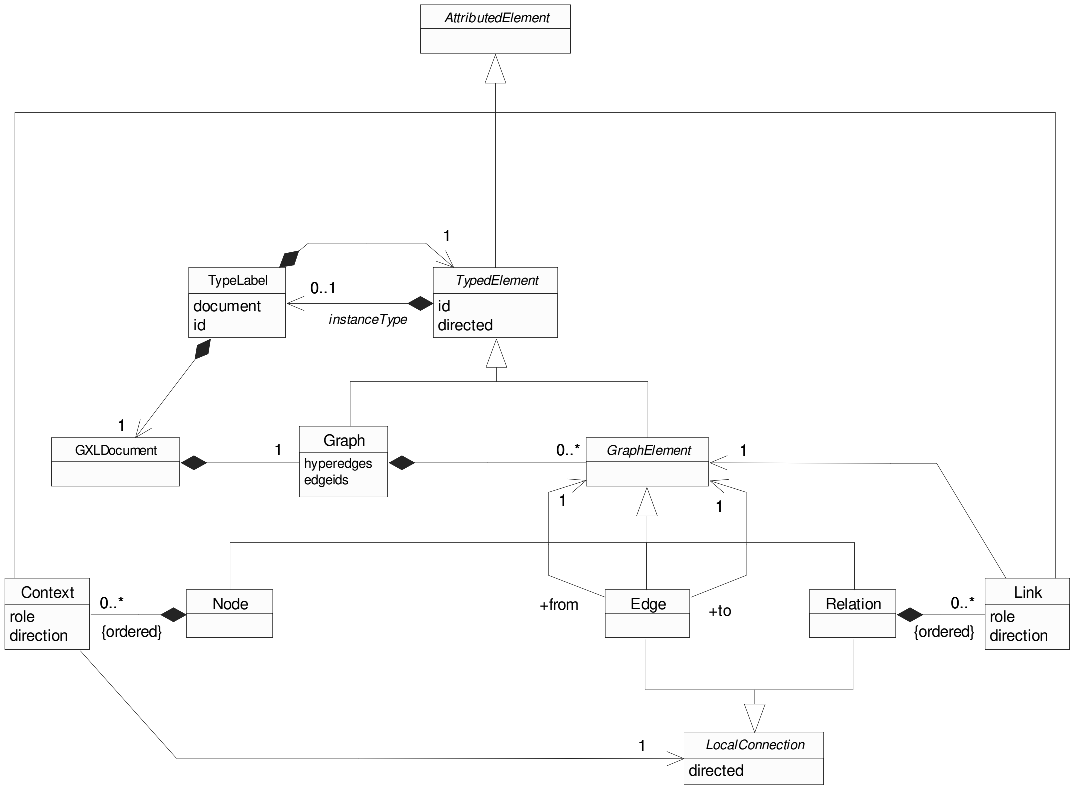

Figure 1: Major graph concepts

In the following sections, a XML Schema implementation of this metamodel is introduced step by step. Please note that several basic data types (included in the original GXL metamodel [3]) are not considered this time.

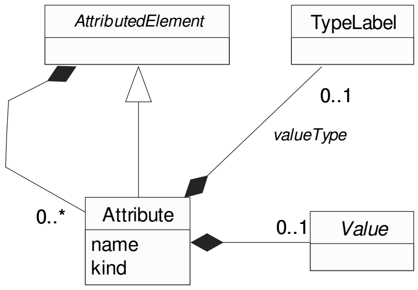

Figure 2: Graph attributes

5.2 Attribute and Type System

The top–most (abstract) class of the metamodel is `AttributedElement`, which is composed of an arbitrary number of `Attribute`s and declared to be abstract.

GXL-GTXL Schema 1. *The abstract class* `AttributedElement`

```
<xsd:element name="AttributedElement" type="AttributedElementType"
             abstract="true"/>
<xsd:complexType name="AttributedElementType">
  <xsd:sequence>
    <xsd:element ref="Attribute" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

The `Attribute` element is constructed from the elements `Type` and `Value` (values may contain any type of elements), thus further embedded attributes (for storing e.g. layout information) is not modelled currently. `Name` and `Type` attributes are both `NMTOKEN`s to conform with the GXL DTD. Element `Value` is not currently elaborated in details.

GXL-GTXL Schema 2. *The class* `Attribute` *and* `Value`

```
<!-- Attribute -->
  <xsd:element name="Attribute" type="AttributeType"/>
  <xsd:complexType name="AttributeType">
    <xsd:sequence>
      <xsd:element name="Type" type="TypeLabelType" />
      <xsd:element ref="Value" />
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:NMTOKEN" use="required"/>
    <xsd:attribute name="kind" type="xsd:NMTOKEN" />
  </xsd:complexType>

<!-- Value -->
  <xsd:element name="Value" type="xsd:anyType" />
```

The `TypedElement` is an abstract subclass of `AttributedElement`. It contains an identifier as an attribute (`id`), and at most one type (`type`) can also be embedded.

The purpose of `directed` attribute will be introduced later.

GXL-GTXL Schema 3. *The abstract class `TypedElement`*

```
<xsd:element name="TypedElement"  type="TypedElementType" abstract="true" />
<xsd:complexType name="TypedElementType">
  <xsd:complexContent>
    <xsd:extension base="AttributedElementType">
      <xsd:sequence>
        <xsd:element name="type" type="TypeLabelType" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID"/>
      <xsd:attribute name="directed"  type="GraphDirectionType"
                     use="default" value="directed"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="GraphDirectionType">
  <xsd:restriction base="xsd.string">
    <xsd:enumeration value="directed"/>
    <xsd:enumeration value="undirected"/>
  </xsd:restriction>
</xsd:simpleType>
```

The `TypeLabel` element (attached to `TypedElements`) is a reference to graphs (represented as web documents), or graph elements (nodes, edges, hyperedges).

GXL-GTXL Schema 4. *The class `TypeLabel`.*

```
<xsd:element name="TypeLabel" type="TypeLabelType"/>
<xsd:complexType name="TypeLabelType">
  <xsd:attribute name="id"       type="xsd:ID" />
  <xsd:attribute name="document" type="xsd:CDATA" />
</xsd:complexType>
```

5.3 Graphs and Graph Elements

A `GXL` document contains *exactly one* graph.

GXL-GTXL Schema 5. *The class `GXL`.*

```
<xsd:element name="GXL" type="GXLType" />
<xsd:complexType name="GXLType">
  <xsd:sequence>
    <xss:element ref="Graph" />
  </xsd:sequence>
</xsd:complexType>
```

A `Graph` element extends the class `TypedElement` and it is composed of `GraphElements` (of an arbitrary number).

- The `edgeids` attribute is true for graphs where `Edges` and `Rels` possess their own identifiers.

- The `hypergraph` parameter allows the existence of hyperedges if its value is "true". Hyperedges are general relationships which connect an arbitrary number of elements instead of two elements (in the case directed or undirected edges).
- The `direction` attribute indicates whether the graph or hypergraph should be interpreted as `directed` (default) or `undirected` graph or hypergraph. This feature holds for all `Nodes`, `Edges` or `Rels` in the graph or hypergraph unless it is overwritten for these graph elements.

GXL-GTXL Schema 6. *The class* ***`Graph`***

```
<xsd:element name="Graph" type="GraphType" />
<xsd:complexType name="GraphType">
  <xsd:complexContent>
    <xsd:extension base="TypedElementType">
      <xsd:sequence>
        <xsd:element ref="GraphElement" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="edgeids"   type="xsd:boolean"
                     use="default" value="false"/>
      <xsd:attribute name="hypergraph" type="xsd:boolean"
                     use="default" value="false"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

`GraphElement` is an abstract class representing nodes, edges and hyperedges (relations).

GXL-GTXL Schema 7. *The abstract class* ***`GraphElement`****.*

```
<xsd:element name="GraphElement" type="GraphElementType" abstract="true"/>
<xsd:complexType name="GraphElementType">
  <xsd:complexContent>
    <xsd:extension base="TypedElementType">
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

5.4 Nodes and Edges

`Node` element represents an ordinary graph node and extends `GraphElement`.

GXL-GTXL Schema 8. *The class* ***`Node`***

```
<xsd:element name="Node" type="NodeType" />
<xsd:complexType name="NodeType">
  <xsd:complexContent>
    <xsd:extension base="GraphElementType">
      <xsd:sequence>
        <xsd:element ref="Context" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

An `Edge` (an extension of `GraphElement`) is the special binary relationship between `Nodes`.

GXL-GTXL Schema 9. *The class* ***`Edge`***

```
<xsd:element name="Edge" type="EdgeType" />
<xsd:complexType name="EdgeType">
  <xsd:complexContent>
    <xsd:extension base="GraphElementType">
      <xsd:attribute name="from" type="xsd:IDREF" use="required"/>
      <xsd:attrubute name="to"   type="xsd:IDREF" use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

The `Rel` element (derived from `GraphElement`) allows the definition of n–ary relationships, which connect n (instead of two) graph elements. In graph theory, these relationships are usually called *hyperedges*.

```
<xsd:element name="Rel" type="RelType" />
<xsd:complexType name="RelType">
  <xsd:complexContent>
    <xsd:extension base="GraphElementType">
      <xsd:sequence>
        <xsd:element name="link" type="LinkType"
                     minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

5.5 Context and Links

`Context` elements describe the incidence to a certain node. Incidences can be `incoming` or `outgoing`. All context elements of a node can be grouped together to indicate the ordering of incidences.

GXL-GTXL Schema 10. *The class* ***`Content`***

```
<xsd:element name="Context"  type="ContextType" />
<xsd:complexType name="ContextType">
  <xsd:complexContent>
    <xsd:extension base="AttributedElementType">
      <xsd:attribute name="ref"       type="xsd:IDREF" />
      <xsd:attribute name="role"      type="xsd:NMTOKEN" />
      <xsd:attribute name="direction" type="DirectionType" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="DirectionType">
```

```
  <xsd:restriction base="xsd.string">
    <xsd:enumeration value="incoming"/>
    <xsd:enumeration value="outgoing"/>
  </xsd:restriction>
</xsd:simpleType>
```

`Link` elements describe the tentacles of `Rel` elements. Both, `Context` and `Link` elements can be associated with role attributes to distinguish different kinds of connections between `Nodes`, `Edges`, and `Rels`.

GXL-GTXL Schema 11. *The class* ***Link***

```
<xsd:element name="Link" type="LinkType" />
<xsd:complexType name="LinkType"
  <xsd:complexContent>
    <xsd:extension base="AttributedElementType">
      <xsd:attribute name="ref"       type="xsd:IDREF" />
      <xsd:attribute name="role"      type="xsd:NMTOKEN" />
      <xsd:attribute name="direction" type="DirectionType" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

6 Conclusion

Advantages of XML Schema In the current paper, the major concepts using XML Schema as a representation and validation method for XML documents were briefly introduced. On that basis, a sample XML Schema encoding of MOF–based GXL-GTXL metamodel (constituted during and after [1]) was given. Though this implementation did not cover all the constructs in the metamodel, they demonstrated several advantages of XML Schema.

- XML Schema provide a fine–grained type concept (the content of each XML element is validated with respect to a simple or complex type) which was a major deficiency in traditional DTDs.
- Moreover, its extension and restriction mechanism makes the encoding of MOF metamodels easier and more consistent.
- Abstract elements can be introduced to support abstract metaclasses in metamodels.
- The type system of XML Schema conforms with XML, thus the same XML document can be checked against both an appropriate DTD and an XML Schema.

Disadvantages of XML Schema However, XML Schema has certain disadvantages originating mainly in its complexity.

- Too complex constructs are defined (datatypes, restrictions, extensions, etc.)
- There are difficulties in implementation (in fact, there is no tool conforming to the entire standard, one of the best tools is XML Spy).

- XML Schema is not a standard yet, just a candidate recommendation, which means that the XML Schema Working Group considers it to be stable and encourages comment on it. However, according to [5] "*Should this the associated specification prove very difficult or impossible to implement, the Working Group will return the specification to Working Draft status and make necessary changes.*"

As a result, we make the following statements to discuss before the subsequent Appligraph Meeting in Bremen:

1. XML Schema clearly provides closer correspondence to a high–level MOF metamodel with respect to modelling concepts than pure DTDs, thus it is a promising candidate for a future representation method.

2. A automatic generation approach (from UML Diagrams to XML Schema) is easier to be built.

3. XML Schema may also serve as an intermediate description in an automatic DTD generation process.

References

[1] *APPLIGRAPH Subgroup Meeting on Exchange Formats for Graph Transformation Systems*, Paderborn, September 2000.

[2] APPLIGRAPH. *XML-based Exchange Formats for Graphs and Graph Transformation System.* http://tfs.cs.tu-berlin.de/.

[3] A. Schürr, S. E. Sim, R. Holt, and A. Winter. The GXL Graph eXchange Language. http://www.gupro.de/GXL/.

[4] World Wide Web Consortium. *XML Namespaces.* http://www.w3.org/TR/REC-xml-names/.

[5] World Wide Web Consortium. *XML Schema Part 0: Primer*, October 2000. http://www.w3.org/TR/2000/CR-xmlschema-0-20001024/.

[6] World Wide Web Consortium. *XML Schema Part 1: Structures*, October 2000. http://www.w3.org/TR/2000/CR-xmlschema-1-20001024/.