XML Schema Part 1: Structures

W3C Working Draft 6-May-1999

This version

http://www.w3.org/1999/05/06-xmlschema-1/
(in XML and HTML, with accompanying schema and DTD)

Latest version:

http://www.w3.org/TR/xmlschema-1/

Editors:

David Beech (Oracle) <dbeech@us.oracle.com>
Scott Lawrence (Agranat Systems) <lawrence@agranat.com>
Murray Maloney (Commerce One) <murray@muzmo.com>
Noah Mendelsohn (Lotus) <noah_mendelsohn@lotus.com>
Henry S. Thompson (University of Edinburgh) <ht@cogsci.ed.ac.uk>

Status of this Document

This is a W3C Working Draft for review by members of the W3C and other interested parties in the general public.

It has been reviewed by the XML Schema Working Group and the Working Group has agreed to its publication. Note that not that all sections of the draft represent the current consensus of the WG. Different sections of the specification may well command different levels of consensus in the WG. Public comments on this draft will be instrumental in the WG's deliberations.

Please review and send comments to www-xml-schema-comments@w3.org (archive)

The facilities described herein are in a preliminary state of design. The Working Group anticipates substantial changes, both in the mechanisms described herein, and in additional functions yet to be described. The present version should not be implemented except as a check on the design and to allow experimentation with alternative designs. *The Schema WG will not allow early implementation to constrain its ability to make changes to this specification prior to final release.*

A list of current W3C working drafts can be found at http://www.w3.org/TR. They may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use W3C Working Drafts as reference material or to cite them as other than "work in progress".

Abstract

XML Schema: Structures is part one of a two part draft of the specification for the XML Schema definition language. This document proposes facilities for describing the structure and constraining the contents of XML 1.0 documents. The schema language, which is itself represented in XML 1.0, provides a superset of the capabilities found in XML 1.0 document type definitions (DTDs.)

Table of Contents

Appendices

1. Introduction

This structural part (*XML Schema: Structures*) of the XML Schema definition language is a distillation of eight months of work by the W3C XML Schema Working Group, including four weeks of intensive work by a group of five editors. This Working Draft draws heavily on the work of [DCD], [DDML], [SOX], [XDR] and [XML-Data]. Requirements for *XML Schema: Structures* can be found in [XML Schema Requirements].

Chapter 2 presents a Conceptual Framework for *XML Schema: Structures*, including an introduction to schema constraints, types, schema composition, and symbol spaces. The abstract and concrete syntax of *XML Schema: Structures* are introduced, along with other terminology used throughout the specification.

Chapter 3 Schema Definitions and Declarations reconstructs the core functionality of XML 1.0, plus a number of extensions, in line with our stated requirements [XML Schema Requirements]. This chapter discusses the declaration and use of datatypes, archetypes, element types, content models, attributes, attribute groups, model groups, refinement, entities and notations.

Chapter 4 presents Schema Composition and Namespaces, including the validation of namespace qualified instance documents, import, inclusion and export of declarations and definitions, schema paths, access to schemas, and related rules for schema-based validity.

Chapter 5 is a placeholder for Documenting schemas, which will eventually provide a standardized means for including documentation in the definition of a schema.

Chapter 6 discusses Conformance, including the rules by which instance documents are validated, and responsibilities of schema-aware processors.

The normative addenda include a (normative) DTD for Schemas and a (normative) Schema for Schemas, which is an XML Schema schema for *XML Schema: Structures*, a Glossary (normative) and References (normative). Non-normative appendixes include a Sample Schema (non-normative) and acknowledgments [Grateful Acknowledgments (non-normative)].

1.1 Documentation Conventions

This Working Draft document was produced using an [XML] DTD and an [XSLT] stylesheet.

The following highlighting is used to present technical material in this document:

[Definition:] A **term** is something we use a lot.

Sample Abstract Syntax Production
`left ::= right1 right2`

Example
A non-normative example illustrating use of the schema language, or a related instance.
`<schema name='http://www.muzmo.com/XMLSchema/1.0/mySchema.xsd' >`
And an explanation of the example.

The following highlighting is used for non-normative commentary in this document:

Issue (dummy): A recorded issue.

Ed. Note: Notes shared among the editorial team.

NOTE: General comments directed to all readers.

1.2 Purpose

The purpose of *XML Schema: Structures* is to provide an inventory of XML markup constructs with which to write schemas.

The purpose of an *XML Schema: Structures* schema is to define and describe a class of XML documents by using these constructs to constrain and document the meaning, usage and relationships of their constituent parts: datatypes, elements and their content, attributes and their values, entities and their contents and notations. Schema constructs may also provide for the specification of implicit information such as default values. Schemas are intended to document their own meaning, usage, and function through a common documentation vocabulary.

Thus, *XML Schema: Structures* can be used to define, describe and catalogue XML vocabularies for classes of XML documents.

Any application that consumes well-formed XML can use the *XML Schema: Structures* formalism to express syntactic, structural and value constraints applicable to its document instances. The *XML Schema: Structures* formalism will allow a useful level of constraint checking to be described and validated for a wide spectrum of XML applications.

The language defined by this specification does not attempt to provide *all* the facilities that might be needed by *any* application. Some applications may require constraint capabilities not expressible in this language, and so may need to perform their own additional validations.

1.3 Relationship To Other Work

The definition of *XML Schema: Structures* is a part of the W3C XML Activity. It is in various ways related to other ongoing parts of that Activity and other W3C WGs

XML Datatype Language

XML Schema: Structures has a dependency on the data typing mechanisms defined in its companion [XML Schemas: Datatypes], published simultaneously with this recommendation.

Document Object Model

XML Schema: Structures has not yet identified requirements or dependencies.

HTML

XML Schema: Structures has a requirement to support modularization of HTML.

Internationalization Working Group

See http://www.w3.org/XML/Group/1999/03/xml-schema-i18n-notes

RDF Schema

XML Schema: Structures has not yet documented requirements or dependencies.

WAI

XML Schema: Structures has a requirement to support accessibility.

XML Information Set

XML Schema: Structures has significant dependencies on [XML-Infoset].

XML Schema: Structures defines its own Information Set Contributions.

XML Schema: Structures will have requirements for subsequent Information Set Working Drafts .

XML Linking WG

XML Schema: Structures has not yet identified requirements or dependencies.

XML Syntax

XML Schema: Structures must interoperate with XML 1.0 and subsequent revisions.

XSL WG

XML Schema: Structures has a requirement to support dimensions and aggregate datatypes.

1.4 Terminology

The terminology used to describe *XML Schema: Structures* is defined in the body of this specification. The terms defined in the following list are used in building those definitions and in describing the actions of *XML Schema: Structures* processors:

[Definition:] **may**

Conforming documents and processors are permitted to but need not behave as described.

[Definition:] **must**

Conforming documents and processors are required to behave as described; otherwise they are in error.

[Definition:] **error**

A violation of the rules of this specification; results are undefined. Conforming software may detect and report an error and may recover from it.

[Definition:] **fatal error**

An error which a conforming processor must detect and report to the application.

[Definition:] **match**

(Of strings or names:) Two strings or names being compared must be character for character the same.

[Definition:] identical

(Of *URI*s or schemaNames:) *identical*, according to the rules for identity in [XML-Namespaces].

2. Conceptual Framework

This specification uses a number of terms that are common to many of the fields of endeavor that have influenced the development of XML Schema. Unfortunately, it is often the case that these terms do not have the same definitions in all of those fields. This section attempts to provide definitions of terms as they are used to describe the conceptual framework, and the remainder of the specification.

2.1 Kinds of XML Documents

Since XML schemas are themselves specified as XML documents, it is useful to clarify the relationships between certain kinds of XML documents:

[Definition:] **Instance**

An XML document whose structure conforms to some schema. See Associating Instance Document Constructs with Corresponding Schemata for the means by which an instance identifies the schema(s) to which it conforms.

[Definition:] **Schema**

A set of rules for constraining the structure and articulating the information set of XML documents.

[Definition:] **Schema Definition**

An XML document that defines a schema. See Access to Schemata for the means by which schema definition documents are accessed during processing. The term "Schema Definition" may also be abbreviated to "schema" where no confusion is likely.

Note that it is possible to specify a schema definition to which schema definitions themselves must conform, and this is given in (normative) Schema for Schemas. An XML 1.0 DTD to which schema definitions must conform is also provided in (normative) DTD for Schemas.

Any schema definition is therefore also an instance. The editors will make an attempt to always use the most precise of these terms, but the reader should note that rules specified herein for instances do apply to all of the following kinds of XML document:

- Documents that are not schema definitions;
- Schema definitions applicable to documents that are not schema definitions;
- The schema definition applicable to schema definitions.

Likewise, rules for schemas do apply to the schema definition for schema definitions, which is an instance of itself.

2.2 On schemas, constraints and contributions

The [XML] specification describes two kinds of constraints on XML documents: *well-formedness* and *validity* constraints. Informally, the well-formedness constraints are those imposed by the definition of XML itself (such as the rules for the use of the < and > characters and the rules for proper nesting of elements), while validity constraints are the further constraints on document structure provided by a particular DTD.

Three kinds of normative statements about the impact of *XML Schema: Structures* components on instances are distinguished in this specification:

[Definition:] **Constraint on Schemas**

Constraints on the form and content of schemas themselves, above and beyond those expressed in (normative) Schema for Schemas;

[Definition:] **Schema-Validity Constraint**

Constraints on the form of instances which they must satisfy to be schema-valid;

[Definition:] **Schema Information Set Contribution**

Augmentations to the information sets of instances which follow as a consequence of schema-validation.

NOTE: Schema Information Set Contributions are not as new as might at first appear: XML 1.0 validation augments the XML 1.0 information set in similar ways, e.g. by providing values for attributes not present in instances, and by implicitly exploiting type information for normalization or access, e.g. consider the effect of `NMTOKENS` on attribute whitespace, and the semantics of `ID` and `IDREF`. By including Schema Information Set Contributions, we are trying to make explicit something XML 1.0 left implicit.

XML Schema: Structures not only reconstructs the DTD constraints of XML 1.0 using XML instance syntax, it also adds the ability to define new kinds of constraints. For example, although the author of an XML 1.0 DTD may declare an element type as containing character data, elements, or mixed content, there is no mechanism with which to constrain the contents of elements to only character data of a particular form, such as only integers in a specified range.

This specification supports the expression of just such constraints by including in the mechanism for the declaration of element types the option of specifying that its contents must consist of a valid string expression of a particular datatype. A number of other mechanisms are added which improve the expressive power, usability and maintainability of schemas as a means to defining the structure of XML documents.

2.3 On 'types'

The use of the word 'type' has caused confusion in earlier discussions of schema technologies. Here, the word is used in several different contexts for a constraint on the form of (parts of) an element or attribute in an instance document. So, there are types that are suitable for constraining individual attributes, others that apply to entire elements (of which attributes may be a part), and so on. We use the words 'define' and 'definition' when speaking of types, and the words 'declare' and 'declaration' when specifying for an attribute or element type the type which constrains its form in instance documents.

2.4 Schemas and their component parts

The next chapter Schema Definitions and Declarations sets out the *XML Schema: Structures* approach to schemas and formal definitions of their component parts. Here we informally summarize the key constructs used in defining schemas. An asterisk (*) in the 'Named?' column indicates that the name will appear in instances -- other names are for schema use only.

XML Schema: Structures **Feature**	**Purpose**	**Named?**
The Schema	A wrapper containing all the definitions and declarations comprising a schema document.	Yes
Datatype Definition	A type (content constraint), such as 'integer', that applies to character data in an instance document, whether it appears as an attribute value or the contents of an element. The mechanisms for defining datatypes are set out elsewhere, in *XML Schemas: Datatypes*.	Yes

Archetype Definition	A complete set of constraints for elements in instance documents, applying to both contents and attributes.	Yes
Element Type Declaration	An association between a name for an element and an archetype. An element type declaration for 'A' is comparable to a DTD declaration `<!ELEMENT A .....>`.	Yes* (local or global)
Attribute Declaration	An association between a a name for an attribute and a datatype, together with occurrence constraints such as 'required' or 'default'. The association is local to its surrounding archetype.	Yes* (local)
Content type	Either a datatype or a content model. A content type applies to the contents of elements in an instance document (but not their attribute values). It provides a unifying abstraction for the constraints which apply to the contents of elements, but introduces no additional features.	No
Element Content Model	A type (content constraint) that applies to the contents of elements in an instance document. Content models do not include attribute declarations.	No
Element-Only Content	Components for constructing content models which allow only element content. Includes facilities for grouping, sequencing, as well as for declaration of and reference to element types.	No (but see below)
Attribute Group Definition	An association between a name and a reusable collection of attribute declarations.	Yes
Named Model Group	Model groups are part of the content model building block abstraction, but are unnamed and cannot be referenced for reuse. A named model group is an association between a name and a model group, allowing for reuse.	Yes
Archetype Refinement	One archetype may be defined as refining one or more other archetypes, acquiring content type and/or attributes therefrom.	Yes
Schema Import	Extends the current schema with definitions and/or declarations from an external schema, retaining the association with the original schema.	No
Schema Inclusion	Integrates definitions and/or declarations from an external schema into the schema being defined, as if they had been defined locally.	No

2.5 Names and Symbol Spaces

As indicated in the third column of the tables above, most of the components listed are given names, which provide for references within the schema, and sometimes from one schema to another. For example, an attribute definition can refer to a named datatype, such as 'integer'. A content model can refer to an element type, and so on.

If all such names were assigned from the same 'pool', then it would be impossible to have e.g. a datatype named 'integer' and an element type with the name 'integer' in the same schema. [Definition:] Accordingly we introduce the idea of a **symbol space** (avoiding 'name space' to avoid confusion with 'Namespaces in XML' [XML-Namespaces]).

There is a single distinct symbol space within a given schema for each of the abstractions named above other than 'Attribute' and 'Element type': within a given symbol space, names are unique, but the same name may appear in more than one symbol space without conflict. In particular note that the same name can refer to both an archetype and an element type, without conflict or necessary relation between the two.

Attributes and local element type declarations are special, in that every archetype defines its own attribute and local element type symbol spaces.

2.6 Abstract and Concrete Syntax

XML Schema: Structures is presented here primarily in the form of an [Definition:] **abstract syntax**, which provides a formal specification of the information provided for each declaration and definition in the schema language. The abstract syntax is presented using a simplified BNF. Defined terms are to the left. Their components are to the right, with a small amount of meta-syntax: ()s for grouping, | to separate alternatives, ? for optionality, * and + for iteration. Terms in italics are primitives, not expanded here, either because they are defined elsewhere (e.g. *URI*, defined by [URI]) or because they can only be grounded once a concrete syntax is decided on (e.g. *choice*).

An abstract syntax production prefixed with a number in brackets (e.g. [3]) is normative; other abstract syntax is either for purposes of explanation, or is a duplicate (for convenience) of a normative definition to be found elsewhere.

The abstract syntax illustrates the expressive power of the language, and the relationships among its component parts. The abstract syntax can be used to evaluate the expressive power of *XML Schema: Structures*, but not its look and feel. In particular, please note that neither ordering within or between productions or choice of names is significant, and that any particular concrete syntax is not constrained by these.

The [Definition:] **concrete syntax** of *XML Schema: Structures*, the exact element and attribute names used in a schema, are a key feature of its proposed design. The concrete syntax is the form in which the schema language is used by schema authors. Though its elements and attributes are often different from the terms of the abstract syntax bnf, the features and expressive power of the two are congruent. The concrete syntax profoundly affects the convenience and usability of the schema language.

We include a preliminary concrete syntax in this draft, via examples and in (normative) Schema for Schemas and (normative) DTD for Schemas. The emphasis in this version has been to stay quite close to the abstract syntax.

3. Schema Definitions and Declarations

The principal purpose of *XML Schema: Structures* is to provide a means for defining schemas that constrain the contents of instance documents and augment the information sets thereof.

3.1 The Schema

A schema contains some preamble information and a set of definitions and declarations.

Schema top level

```
[1]         schema ::= preamble dds*
[2]            dds ::= datatypeDefn | archetypeDefn |
                       elementTypeDecl | attrGroupDefn |
                       modelGroupDefn | notationDecl |
                       entityDecl
[3]       preamble ::= xmlSchemaRef schemaIdentity
                       schemaVersion model export? import?
                       include?
[4]   xmlSchemaRef ::= URI
[5] schemaIdentity ::= URI
[6]  schemaVersion ::= string-value
[7]          model ::= open | refinable | closed
```

preamble consists of an xmlSchemaRef specifying the URI for *XML Schema: Structures*; the schemaIdentity specifying the URI by which *this* schema is to be identified; and a schemaVersion specification for private version documentation purposes and version management.

Example

```
<!DOCTYPE schema
         PUBLIC '-//W3C//DTD XML Schema Version 1.0//EN'
         SYSTEM 'http://www.w3.org/1999/05/06-xmlschema-1/structures.dtd' >

<schema name='file:/usr/schemas/xmlschema/mySchema.xsd'
        version='M.n'
        xmlns='http://www.w3.org/1999/05/06-xmlschema-1/structures.xsd'>

  ...

</schema>
```

Note that the abstract syntax xmlSchemaRef is realised via a default namespace declaration in the concrete syntax.

The schema's model property is discussed in Archetype Refinement. The schema's export, import and include properties are discussed in Schema Composition and Namespaces.

The schema's declarations and definitions, discussed in detail in Schema Definitions and Declarations, provide for the creation of new schema components:

Summary of Definitions and Declarations

datatypeDefn	::=	*NCName*	datatypeSpec
archetypeDefn	::=	*NCName*	archetypeSpec
elementTypeDecl	::=	*NCName*	elementTypeSpec
modelGroupDefn	::=	*NCName*	modelGroupSpec
attrGroupDefn	::=	*NCName*	attrGroupSpec
entityDecl	::=	*NCName*	entitySpec
notationDecl	::=	*NCName*	notationSpec

Example

The following illustrates the basic model for declaring all *XML Schema: Structures* components:

```
<datatype name='myDatatype'>
 ...
</datatype>

<archetype name='myArchetype'>
 ...
</archetype>

<elementType name='myElementType'>
 ...
</elementType>

<attrGroup name='myAttrGroup'>
 ...
</attrGroup>

<modelGroup name='myModelGroup'>
 ...
</modelGroup>

<notation name='myNotation' ... />
```

```
 <textEntity name='myTextEntity'>
  ...
 </textEntity>

 <externalEntity name='myExternalEntity' ... />

 <unparsedEntity name='myUnparsedEntity' ... />

</schema>
```

When creating a new component, we declare that its name is associated with the specification for that component. Each new component definition creates a new entry in the symbol space for that kind of component.

Constraint on Schemas: Unique Definition
The same NCName must not appear in two definitions or declarations of the same type.

> **Issue (no-evolution):** This draft does not deal with the requirement "for addressing the evolution of schemata" (see [XML Schema Requirements]).

3.2 The Document and its Root

> **NOTE:** We have not so far seen any need to reconstruct the XML 1.0 notion of *root*. For the connection from document instances to schemas, see Schema Validity.

3.3 References to Schema Constructs

Uniform means are provided for reference to a broad variety of schema constructs, both within a single schema and to features imported (Schema Import) from external schemas. The name used to reference any component of *XML Schema: Structures* from within a schema consists of an NCName and an optional schemaRef, a reference to an external schema definition. In a few cases, some qualification may be added to a reference: this is made clear as the individual reference forms are introduced below.

Example: Component Names and References

```
 [8]       schemaRef ::= (schemaAbbrev | schemaName)
 [9]    schemaAbbrev ::= NCName
[10]      schemaName ::= URI
         datatypeRef ::= datatypeName datatypeQual
        datatypeName ::= NCName schemaRef?
        archetypeRef ::= archetypeName
       archetypeName ::= NCName schemaRef?
      elementTypeRef ::= elementTypeName
     elementTypeName ::= NCName schemaRef?
        attrGroupRef ::= attrGroupName attrGroupQual
       attrGroupName ::= NCName schemaRef?
       modelGroupRef ::= modelGroupName
      modelGroupName ::= NCName schemaRef?
           entityRef ::= entityName
          entityName ::= NCName schemaRef?
         notationRef ::= notationName
        notationName ::= NCName schemaRef?
```

The BNF above illustrates the reference mechanisms used in this specification.

Example

```
<archetypeRef name='Address'/>

<elementTypeRef name='BLOCKQUOTE' schemaAbbrev='HTML'/>

<datatypeRef name='quantityi' schemaName='http://www.w3.org/xsl.xsd'/>
```

The first of these is a local reference, the other two refer to schemas elsewhere. The elementTypeRef example assumes the schemaAbbrev `HTML` has been declared for import; the datatypeRef example similarly assumes that the given (imaginary as of this writing) URL has been declared for import. See Schema Importfor a discussion of importing.

Constraint on Schemas: Consistent Import
A schemaAbbrev or schemaName in a schemaRef must be declared in an Schema Import of the current schema, and the NCName qualified by that schemaRef must be an import (Import Restrictions) of the appropriate type per that declaration.

Constraint on Schemas: One Reference Only
The concrete syntax uses `schemaAbbrev` and `schemaName` attributes to realise schemaName. It is an error for both these attributes to appear on the same element in a schema.

[Definition:] A **...Ref identifies** a **...Spec** provided there is a definition or declaration of that **...Spec** in the appropriate schema whose NCName matches the NCName of the **...Ref**'s **...Name**. If there is no schemaRef in the **...Name**, the appropriate schema is the current schema or a schema it eventually includes; if there is a schemaRef, the URI contained in or abbreviated by it must resolve successfully to a schema, which is then the appropriate schema. The Preorder Priority for Included Definitions Constraint on Schemas may also obtain.

3.4 Types, Elements and Attributes

Like XML 1.0 DTDs, *XML Schema: Structures* provides facilities for constraining the contents of elements and the values of attributes, and for augmenting the information set of instances, e.g. with defaulted values and type information. [Definition:] We refer hereafter to the combination of schema constraints and information set contributions with the abbreviation **SC**. Compared to DTDs, *XML Schema: Structures* provides for a richer set of SCs, and improved capabilities for sharing SCs across sets of elements and attributes.

3.4.1 Datatype Definition

We start with [Definition:] the simple datatypes whose expression in XML documents consists entirely of character data. As in the current draft of *XML Schemas: Datatypes*, wherever we speak of **datatype**s in this draft, we shall mean these simple datatypes. The treatment of aggregate datatypes (collections and structures) has not yet been resolved.

Datatypes

```
[11]    datatypeDefn ::= NCName datatypeSpec
[12]    datatypeSpec ::= [defined by XML Schemas: Datatypes]
                         exportControl?
[13]    datatypeQual ::= specialize? valueConstraint?
[14]      specialize ::= facet+
[15]           facet ::= is defined by XML Schemas: Datatypes.
                         It might be a range restriction,
                         min/max constraint, etc.
[16] valueConstraint ::= default | fixed
[17]     datatypeRef ::= datatypeName datatypeQual
[18]    datatypeName ::= NCName schemaRef?
           schemaRef ::= (schemaAbbrev | schemaName)
```

```
       schemaAbbrev ::= NCName
         schemaName ::= URI
```

XML Schema: Structures incorporates the datatype definition mechanisms specified by [XML Schemas: Datatypes] in order to express SCs on attribute values and the character data contents of elements.

The first production above is for defining datatypes; datatypeSpec serves to indicate where this chapter connects with *XML Schemas: Datatypes*. exportControl is defined in Exporting Schema Constructs.

The other productions provide for using datatypes once they have been defined, see below under contentType and attrDecl.

We assume that it is appropriate to allow for some local specialization of datatypes at the point of use, and provide for that here (specialize).

As explained in References to Schema Constructs, a schemaRef, if included allows for the referenced definition to be located in some other schema.

Example

```
<datatype name='myDatatype'>
 <basetype name="..."/>
  [ ... TBD]
</datatype>

<datatypeRef name='myDatatype'/>

<datatypeRef name='integer'/>

<datatypeRef name='quantity' schemaName='http://www.w3.org/xsl.xsd'>
 <fixed>12pt</fixed>
</datatypeRef>
```

The first example awaits the *XML Schemas: Datatypes* concrete syntax to be filled in. The first datatypeRef example references the definition above it. The second references a datatype pre-defined by *XML Schemas: Datatypes*. The third references a datatype in an (imaginary) XSL schema and fixes its value.

Constraint on Schemas: Avoid Built-ins
The NCName must not be the same as the name of any of the built-in datatypes (see [XML Schemas: Datatypes]).

[Definition:] A string (possibly empty) **dt-satisfies** a datatypeSpec and an optional datatypeQual if

- The string is a valid instance of the datatype defined by that datatypeSpec, as specialized by the datatypeQual's specialize (if present) (see [XML Schemas: Datatypes] for a definition of when a string is a valid instance of a (possibly specialized) datatypeSpec;

and

- If there is a datatypeQual and it includes a *fixed*, the string matches that fixed value.

Schema Information Set Contribution: Datatype Info
When a string dt-satisfies a datatypeSpec and an optional datatypeQual, the containing attribute or element information item will be augmented to indicate the datatypeSpec and the specialize (if any) which it satisfied.

NOTE: Timing constraints were such that this text may not align completely with *XML Schemas: Datatypes*

3.4.2 Archetype Definition

[Definition:] **Archetype** definitions gather together all SCs pertinent to elements in instance documents, their attributes and their contents. They are called archetypes because there may be more than one element type which shares the same SCs (see Element Type Declaration), and which therefore can be constrained by a common archetype.

Archetypes

```
[19] archetypeDefn ::= NCName archetypeSpec
[20] archetypeSpec ::= refinement* contentType ( attrDecl |
                       attrGroupRef )* model exportControl
[21]   contentType ::= datatypeRef | contentModel |
                       modelGroupRef
             model ::= open | refinable | closed
[22]  archetypeRef ::= archetypeName
[23] archetypeName ::= NCName schemaRef?
```

The first three productions above provide the basic structure of the definition, the last two provide for reference to the things defined. But note that the name of an archetype is not *ipso facto* the name of elements whose appearance in instances will be associated with the SCs of that type. The connection between an element type name and an archetype is made by an elementTypeDecl, see below.

Alongside Attribute Declaration for permitted attributes, SCs for contents are defined in an archetype (contentType). For elements which may contain only character data, content type SCs are specified by reference to a Datatype Definition. Note that doing this by way of datatypeRef means that the character data SCs may provide for specialization and even defaulting in a manner similar to attribute values. For other kinds of elements, a Element Content Model is required.

> **Issue (elt-default):** The extension of defaulting to element content is tentative.

Example

```
<archetype name='myArchetype'>
    <datatypeRef name='myDatatype'/>
    <attrDecl ...>. . .</attrDecl>
</archetype>
```

A simple archetype with character data content constrained by reference to a datatype defined in the same schema and one attribute.

Constraint on Schemas: AttrGroup Unique
The same attrGroupDefn must not be referenced by two or more attrGroupRefs in the same archetypeSpec.

Constraint on Schemas: AttrGroup Identified
Every attrGroupRef in a archetypeSpec must identify an attrGroupSpec.

[Definition:] The **attribute declaration set** of an archetypeSpec consists of all its effective attrDecls together with all the attrDecls contained in the attrGroupSpecs identified by any attrGroupRefs it contains.

[Definition:] The **full name** of an attrDecl in an attribute declaration set is its NCName plus its schemaName, i.e. if it appeared directly in the archetypeSpec, the empty string, if it was acquired by refinement or if it came from an attrGroupSpec, then the schemaName from the schemaRef which identified the relevant archetypeSpec or attrGroupSpec respectively, if any, otherwise the empty string.

Constraint on Schemas: Attribute Locally Unique
The same full name must not appear more than once in any archetypeSpec's attribute declaration set.

[Definition:] An element item **a-satisfies** an archetypeSpec if the element item's attribute items taken together as a set attrs-satisfy the archetypeSpec's attribute declaration set, and either

- The archetypeSpec's contentType is a datatypeRef, the datatypeRef identifies a datatypeSpec, the element item contains only comment, processing instruction and character information items and the string formed by concatenating the characters of each of the character information item children, if any, or else the empty string, dt-satisfies that datatypeSpec as qualified by the datatypeRef's datatypeQual if any;

or

- the contentType is a contentModel, and the sequence of character and element items contained by the element item model-satisfies its effective contentModel.

  Issue (sic-elt-default): The above definitions do not provide for handling a default on an archetype's datatypeRef. Preferred solution: empty element items *ipso facto* satisfy datatypeRefs with defaults and are augmented with the default value. This would have the consequence that you cannot provide the empty string as the explicit value of an element item if it's governed by a datatypeRef with a default.

Schema Information Set Contribution: Archetype Info
When an element item a-satisfies a archetypeSpec, that element information item will be augmented to indicate the archetypeSpec which it satisfied.

Example

Why is the separation of element type and archetype provided for? In certain XML instance documents, the same attribute and content SCs are appropriate for more than one named element. Consider the following instance fragment:

```
<ShippingAddress id='a32' saleable='yes'>
  <Street>1 Main Street</Street>
  <City>New York</City>
  <Zip NineDigit='FALSE'>12345</Zip>
</ShippingAddress>

<BillingAddress id='a29' saleable='no'>
  <Street>2 Rose Lane</Street>
  <City>Anytown</City>
  <Zip NineDigit='TRUE'>12345-6789</Zip>
</BillingAddress>
```

This fragment contains two elements, `ShippingAddress` and `BillingAddress`, that have similar attributes and content. By defining a single appropriate archetype and declaring two element types which reference it, that commonality can be precisely expressed.

NOTE: This draft does not provide any mechanism for applying any SCs to element items whose namespace does not nominate a schema. This may be addressed in a later draft: in the meantime a workaround is possible as follows: Suppose we wish to use some Dublin Core terms in a schema, but all we know is the URI for the Dublin Core document. Perhaps we want to schema-validate

```
<mybook><dc:creator xmlns:dc='...'>Rafael
 Sabattini</dc:creator></mybook>
```

where `mybook` is already known to be covered by my schema. The workaround is to replace the real Dublin Core URI with a local URL for a tiny schema which simply defines `creator`, and references the real URI for documentation.

3.4.3 Attribute Declaration

Attribute declarations associate a name (which will appear as an attribute in start tags in instances) with SCs for the presence and value thereof.

Attributes

```
[24]       attrDecl ::= NCName datatypeRef? required
                        exportControl
        datatypeRef ::= datatypeName datatypeQual
       datatypeQual ::= specialize? valueConstraint?
    valueConstraint ::= default | fixed
       datatypeName ::= NCName schemaRef?
          schemaRef ::= (schemaAbbrev | schemaName)
```

NOTE: Note that the datatypeRef productions are repeated here for easy reference.

Attribute declarations provide for:

- Requiring instances to have attributes;
- Constraining attribute values to express a datatype;

Example

```
<attrDecl name='myAttribute'/>

<attrDecl name='anotherAttribute'>
 <datatypeRef name='integer'>
  <default>42</default>
 </datatypeRef>
</attrDecl>

<attrDecl name='yetAnotherAttribute' required='true'>
 <datatypeRef name='integer'/>
</attrDecl>

<attrDecl name='stillAnotherAttribute'>
 <datatypeRef name='string'>
  <fixed>Hello world!</fixed>
 </datatypeRef>
</attrDecl>
```

Four attributes are declared: one with no explicit SCs at all; two defined by reference to a built-in type, one with a default and one required to be present in instances; and one with a fixed value.

When attribute declarations are used in an archetype specification, each archetype provides its own symbol space for attribute names. E.g. an attribute named `title` within one archetype need not have the same datatypeRef as one declared within another archetype.

[Definition:] An attribute item **attr-satisfies** an attrDecl if

- The attrDecl contains no datatypeRef and the attribute item's value string dt-satisfies the default datatypeSpec for attributes

or

- the attrDecl's datatypeRef identifies a datatypeSpec and the attribute item's value string dt-satisfies that datatypeSpec as qualified by the datatypeRef's datatypeQual if any

where the attribute item's value consists of only character information items and by its "value string" is meant the string formed by concatenating the characters of each of those character information item children, if any, or else the empty string.

Issue (default-attr-datatype): What *is* the default attribute datatypeSpec?

[Definition:] The attribute items of an element item **attrs-satisfy** an attribute declaration set if

- **1)** for each attribute item either

- ❍ **1a)** there is an attrDecl in the attribute declaration set whose full name consists of an NCName which matches the attribute item's name and either
  - ■ **1a1)** the attribute item has no namespace and the attrDecl's full name has no schemaName

  or
  - ■ **1a2)** the attribute item's namespace and the attrDecl's full name's schemaName are identical

  and the attribute item attr-satisfies the declaration

or
- ❍ **1b)** the archetypeSpec being a-satisfied by the attribute item's parent element item is *open*

and

- **2)** every attrDecl in the attribute declaration set which is *required* is used to attr-satisfy an attribute item in the context of (1a) above

Schema Information Set Contribution: Attribute Value Default
For every attrDecl in the attribute declaration set *not* used to attr-satisfy an attribute item in the context of (1a) above which has a datatypeRef which has a *default*, an attribute item with the default value is added to the parent element item.

Issue (namespace-declare): We've got a problem with namespace declarations: they're not attributes at the infoset level, so they can appear without compromising validity, EXCEPT if there is a fixed or required declaration, and defaults should have the apparently desired effect.

3.4.4 Attribute Group Definition

XML Schema: Structures can name a group of attributes so that they may be incorporated as a whole into archetype definitions:

Attribute groups

[25]	attrGroupDefn	::=	*NCName* attrGroupSpec
[26]	attrGroupSpec	::=	attrDecl* exportControl
[27]	attrGroupRef	::=	attrGroupName attrGroupQual
[28]	attrGroupName	::=	*NCName* schemaRef?
[29]	attrGroupQual	::=	attrDecl

Attribute group definitions:

- provide a construct to replace some uses of parameter entities.
- allow for the definition of SCs that relate values of one attribute to others within the group.

Example

```
<attrGroup name='myAttrGroup'>
    <attrDecl .../>
    ...
</attrGroup>

<archetype name='myElementType'>
    <empty/>
    <attrGroupRef name='myAttrGroup'/>
</archetype>
```

Define and refer to an attribute group. The effect is as if the attribute declarations in the group were present in the archetype definition.

NOTE: There needs to be a Constraint on Schema which constrains the attrDecls which appear with an attrGroupRef: the name is the same as one of the attrDecls in the group, datatype and defaulting preserves substitutability, etc.

3.4.5 Element Content Model

When content of elements is not constrained by reference to a datatype (Datatype Definition), it can have any, empty, element-only or mixed content. In the latter cases, the form of the content is specified in more detail.

Content model
[30] contentModel ::= *any* | *empty* | mixed | eltOnly

A content model constrains the content of an archetype or an element type: it says nothing about attributes.

Example

```
<any/>

<empty/>

<mixed>...</mixed>

[Element only content -- see element-only below]
```

Content models do not have names, but appear as a part of the definition of an archetype, which does have a name. Model *groups* can be named and used by name, see below.

[Definition:] A sequence of character and element items (call this *CESeq*) **model-satisfies** an effective contentModel if

- the sequence is empty and the effective contentModel is *empty*;
- the effective contentModel is *any* and every element item in *CESeq* is independently valid, and for every element item in *CESeq* such that the schema which its namespace item resolves to is not the current schema there is an import whose schemaName is identical to that element item's namespace item's *URI* and that import must either be *allElementTypes* or contain an elementTypeRef whose *NCName* matches that element item's name;
- the effective contentModel is mixed and every element item in *CESeq* mixed-satisfies the effective mixed

or

- the effective contentModel is eltOnly and the only character items in *CESeq* are whitespace character items and the sub-sequence of *CESeq* consisting of all the element items therein eo-satisfies the effective eltOnly.

3.4.6 Mixed Content

A content model for mixed content provides for mixing elements with character data in document instances. The allowed element types are named, but neither their order or their number of occurrences are constrainted.

Mixed content
[31] mixed ::= (elementTypeRef | elementTypeDecl)*

The elementTypeRefs and elementTypeDecls determine the element types which may appear as children along with character data.

Example

```
<mixed>
 <elementTypeRef name='name1'/>
 <elementTypeRef name='name2'/>
 <elementTypeRef name='name3'/>
</mixed>
```

Allows character data mixed with any number of `name1`, `name2` and `name3` elements.

NOTE: The fact that mixed allows for there to be *no* elementTypeRefs or elementTypeDecls makes it similar to XML 1.0's Mixed production. Indeed an empty mixed is the only way a schema can allow character data content with no datatype constraint at all.

Constraint on Schemas: Element Type Unique in Mixed
A given NCName must not appear two or more times among the elementTypeDecls and elementTypeRefs with no schemaRefs; a given elementTypeName must not appear two or more times among the elementTypeRefs.

See Element Type Declaration for discussion and examples of the appearance of elementTypeDecl above.

[Definition:] An element item **mixed-satisfies** a mixed if

- the mixed contains an elementTypeRef which the element item ref-satisfies

or

- the mixed contains an elementTypeDecl whose NCName matches the element item's name, in which case the element item must e-satisfy that elementTypeDecl

or

- the archetypeSpec which effectively contains the mixed is *open* and the element item is independently valid.

  Issue (mixed-change-current-schema): There's an implicit change in current schema in the definition of satisfy-mixed above which should be made explicit.

3.4.7 Element-Only Content

A content model for element-only content specifies only child elements (no immediate character data content other than white space is allowed). The content model consists of a simple grammar governing the allowed types of child elements and the order in which they must appear.

Element-only content

```
[32]    eltOnly ::= modelElt
[33]   modelElt ::= occur ( modelGroup | modelGroupRef |
                    elementTypeRef | elementTypeDecl )
[34]      occur ::= min max?
[35] modelGroup ::= compositor modelElt modelEltSeq
[36] compositor ::= sequence | choice | all
[37]modelEltSeq ::= modelElt modelEltSeq?
```

The grammar for element-only content is built on model elements and model groups (modelElt and modelGroup above). A model element provides for some number of occurrences in an instance of either a single element (via elementTypeRef or elementTypeDecl) or a group of elements (via modelGroup or modelGroupRef). A model group is two or more model elements plus a compositor.

A compositor for a model group specifies for a given group whether it is a sequence of its model elements, a choice between its model elements or a set of its model elements which must appear in instances. These options reconstruct the XML 1.0 `,` connector, the XML 1.0 `|` connector and the SGML & connector respectively. In the first case (sequence) all the model elements must appear in the order given in the group; in the second case (choice), exactly one of the model elements must appear in the element content; and in the third case (all), all the model elements must appear in the element content, but may appear in any order.

The occur specification governs how many times the instance material allowed by a modelElt may occur at that point in the grammar. The absence of a *max* specification means that no upper bound is placed on the number of occurrences.

See Element Type Declaration for further discussion and examples of the appearance of elementTypeDecl within modelElt above.

Example

```
<elementTypeRef name='paragraph'/>

<sequence>
 <elementTypeRef name='name1'/>
 <elementTypeRef name='name2' prefix='HTML'/>
</sequence>

<choice minOccur='3' maxOccur='9'>
 <elementTypeRef name='name1'/>
 <elementTypeRef name='name2'/>
</choice>

<all minOccur='3'>
 <elementTypeRef name='name1'/>
 <elementTypeRef name='name2'/>
</all>

<choice>
 <all>
  <elementTypeRef name='name1'/>
  <elementTypeRef name='name2'/>
 </all>
 <all>
  <elementTypeRef name='name3'/>
  <elementTypeRef name='name4'/>
 </all>
</choice>
```

A minimal model which simply requires a `paragraph` (the default is `minOccur=maxOccur=1`; a sequence of two elements (one from another schema); between 3 and 9 elements, each a choice between two; at least three pairs, in any order; one of two pairs, the elements *in* the chosen pair in any order.

[Definition:] A sequence of element items **eltOnly-satisfies** an effective eltOnly if

- it is possible to trace out a path through the content model, obeying the compositor and occur specifications and accepting each element item in the sequence with an elementTypeRef or elementTypeDecl in the content model. Accepting an element item from the sequence with an elementTypeRef requires that it ref-satisfy that elementTypeRef; accepting an element item with an elementTypeDecl requires that the elementTypeDecl's NCName matches the element item's name, in which case the element item must e-satisfy that elementTypeDecl.

  NOTE: The above definition of eltOnly-satisfy does not explicitly incorporate the modifications required when the containing archetype is *open*, as set out at the end of Archetype Refinement, but it should be understood as doing so.

Constraint on Schemas: Element Consistency
A given NCName must not appear both among the elementTypeDecls and among the elementTypeRefs with no schemaRefs, or more than once among the elementTypeDecls.

> **NOTE:** Note that the above permits repeated use of the same elementTypeRef, analogous to DTD usage.

Constraint on Schemas: Unambiguous Content Model
For compatibility, it is an error if a content model is such that there exist element item sequences within which some item can match more than one occurrence of an elementTypeRef or elementTypeDecl in the content model.

> **Issue (still-unambig):** Should this compatibility constraint be preserved?

3.4.8 Named Model Group

This reconstructs another common use of parameter entities.

Named model groups

```
[38] modelGroupDefn ::= NCName modelGroupSpec
[39] modelGroupSpec ::= ( modelGroup | modelGroupRef )
                        exportControl
[40]  modelGroupRef ::= modelGroupName
[41] modelGroupName ::= NCName schemaRef?
```

Example

```
<modelGroup name='myModelGroup'>
 <elementTypeRef name='myElementType'/>
</modelGroup>

<elementType name='myElementType'>
 <modelGroupRef name='myModelGroup'/>
 <attrDecl ...>. . .</attrDecl>
</elementType>

<elementType name='anotherElementType'>
 <choice>
  <elementTypeRef name='yetAnotherElementType'/>
  <modelGroupRef name='myModelGroup'/>
 </choice>
 <attrDecl ...>. . .</attrDecl>
</elementType>
```

A minimal model group is defined and used by reference, first as the whole content model, then as one alternative in a choice.

3.4.9 Element Type Declaration

An [Definition:] **element type** declares the association of an element type name with an archetype, either by reference or by incorporation.

Element type declaration

```
[42] elementTypeDecl ::= NCName elementTypeSpec
[43] elementTypeSpec ::= ( archetypeRef | archetypeSpec )
                         exportControl global?
[44]  elementTypeRef ::= elementTypeName
[45] elementTypeName ::= NCName schemaRef?
```

An element type declaration associates a name with an Archetype Definition. This name will appear in tags in instance documents; the archetype definition provides SCs on the form of elements tagged with the specified name. An element type declaration is comparable to an `<!ELEMENT ...>` declaration in an XML 1.0 DTD.

The last two productions above provide for element types to be referenced by name from content models.

As noted above element type names are in a separate symbol space from the symbol space for the names of archetypes, so there can (but need not be) an archetype with the same name as a top-level element type.

[Definition:] The **full name** of a top-level elementTypeDecl is its NCName plus its schemaName, i.e. if it appeared directly in the current schema or an include, the empty string, if it was imported, then the schemaName of that import, which must successfully resolved to its containing schema.

An elementTypeDecl may also appear within a modelElt. See above (Element-Only Content and Mixed Content) for where this is allowed. This declares a locally-scoped association between an element type name and an archetype. As with attribute names, locally-scoped element type names reside in symbol spaces local to the archetype that defines them. Note however that archetype names are always top-level names within a schema, even when associated with locally-scoped element type names.

NOTE: It is not yet clear whether an archetype defined implicitly by the appearance of an archetypeSpec directly within an elementTypeSpec will have an implicit name, or if so what that name would be, or if not how, if at all, it might be referred to.

Example

```
<elementType name='myElementType'>
 <datatypeRef name='myDatatype'/>
</elementType>

<elementType name='et0'>
 <archetypeRef name='myArchetype'/>
</elementType>

<elementType name='et1'>
 <all>. . .</all>
 <attrDecl ...>. . .</attrDecl>
</elementType>

<elementType name='et2'>
 <any/>
</elementType>

<elementType name='et3'>
 <empty/>
 <attrDecl ...>. . .</attrDecl>
</elementType>

<elementType name='et4'>
 <choice>. . .</choice>
 <attrDecl ...>. . .</attrDecl>
</elementType>

<elementType name='et5'>
 <sequence>. . .</sequence>
 <attrDecl ...>. . .</attrDecl>
</elementType>

<elementType name='et6' model='open'>
 <mixed/>
</elementType>
```

A pretty complete set of alternatives. Note the last one is intended to be equivalent to the idea sometimes called WFXML, for Well-Formed XML: it allows *any* content at all, whether defined in the current schema or not, and *any* attributes.

```
<elementType name='contextOne'>
 <sequence>
  <elementType name='myLocalElementType'
   <archetypeRef name='myFirstArchetype'/>
  </elementType>
  <elementTypeRef name='globalElementType'/>
 </sequence>
</elementType>

<elementType name='contextTwo'
 <sequence>
  <elementType name='myLocalElementType'
   <archetypeRef name='mySecondArchetype'/>
  </elementType>
  <elementTypeRef name='globalElementType'/>
 </sequence>
</elementType>
```

Instances of `myLocalElementType` within `contextOne` will be constrained by `myFirstArchetype`, while those within `contextTwo` will be constrained by `mySecondArchetype`.

NOTE: The possibility that differing attribute definitions and/or content models would apply to elements with the same name in different contexts is an extension beyond the expressive power of a DTD in XML 1.0.

Constraint on Schemas: Nested May Not Be Global
An elementTypeSpec in a nested elementTypeDecl must not be *global*.

Constraint on Schemas: Cannot Shadow Global
If a top-level elementTypeSpec is *global*, then the NCName of its elementTypeDecl must not be redeclared by any nested elementTypeDecl in the same schema or any schema it eventually includes.

[Definition:] An element item **e-satisfies** an elementTypeDecl if the elementTypeDecl:

- contains an archetypeSpec directly, and the element item arch-satisfies that archetypeSpec;

or

- the archetypeRef alternative is used in that elementTypeDecl and it identifies an archetypeSpec and the element item a-satisfies that archetypeSpec.

[Definition:] An element item is **independently valid** if there is a top-level elementTypeDecl whose NCName matches its name in the schema its namespace item resolves to (or a schema that schema includes, in which case see the definition of identify for details on which declaration is used if there is more than one), and the element item must e-satisfy that elementTypeDecl.

[Definition:] An element item **ref-satisfies** an elementTypeRef if

- the elementTypeRef identifies an elementTypeDecl whose NCName matches the element item's name and either
  - the elementTypeDecl's full name has no schemaName, in which case the element item's namespace must be the same as its parent's namespace

  or

  - the element item's namespace and the elementTypeDecl's full name's schemaName are identical

  in which case the element item must also e-satisfy the elementTypeDecl.

or

- the elementTypeRef identifies an elementTypeDecl whose NCName matches the element item's name (call this *ETD1* and there is a top-level elementTypeDecl (call this *ETD2*) which contains or identifies an archetypeSpec which has an ancestor which is itself identified by *ETD1*, in which case *EDT1* must be e-satisfied by the element item.

  NOTE: The last clause above is *much* too complex, it needs to be split apart and built up in stages. It is this which allows elements based on refining archetypes to appear in place of those based on their ancestors.

3.5 Archetype Refinement

> **NOTE:** This section articulates what has only been hinted at above, namely a considerable increase in the power and expressiveness of schema declarations, by explaining what was provided for in the abstract syntax in the previous section, but not explained much if at all at that point.

We provide for the refinement of archetypes declared in a schema. An archetype definition may identify one or more other archetypes from which it specifies the creation of a (joint) refinement.

We provide for interpreting archetypes as imposing constraints on instance elements in either a closed, refinable or open fashion.

- The closed interpretation is as per XML 1.0, where content type and attribute declarations specify all *and only* what must be present in instances.
- The open interpretation removes the 'and only', i.e. it interprets the content type and attribute declarations as requiring (non-optional) items to be present, but *not* as excluding others.
- The refinable interpretation requires the declared (non-optional) items to be present, and also admits such others as have been explicitly declared in refinements.

The relevant abstract syntax productions are as follows:

Refinement

```
              preamble ::= xmlSchemaRef schemaName schemaVersion
                           model export? import? include?
         archetypeSpec ::= refinement* contentType ( attrDecl |
                           attrGroupRef )* model exportControl
                 model ::= open | refinable | closed
[46]        refinement ::= archetypeRef
```

Example

```
<archetype name='chair' model='refinable'>
 <refines>
  <archetypeRef name='furniture'/>
 </refines>
 <attrDecl name='noOfSeats'>. . .</attrDecl>
</archetype>
```

An archetype which not only refines another by adding an attribute, but also is itself (further) refinable. The `furniture` archetype must have been declared *refinable*(or *open*).

We distinguish between explicit, acquired and effective validation constraints for any archetype. [Definition:] **Explicit** constraints are those explicitly present, via attrDecls or contentType, in the definition of the archetype. [Definition:] **Acquired** constraints come from the ancestors of an archetype, if it has any refinements. [Definition:] The **effective** constraints are the union of the explicit and the acquired. The effective constraints are what actually constrain any element type declared with reference to a refining archetype.

Constraint on Schemas: Allowed Refinements
An archetype must not refine one or more other archetypes unless all of the latter have been declared with either *open* or *refinable* (explicitly or by default: the default for model on any archetype which does not explicitly specify one is provided by the model of the schema itself, which in turn defaults to *closed* for compatibility). The same archetype must not be referenced more than once in the refinements list.

> **Issue (default-model):** Should the default model be *open*, *refinable* or *closed*?

For the present, the permitted refinements and the resulting effective constraints are as follows:

- For attribute declarations, any attribute name defined in the explicit constraints must not also be defined in any of the acquired constraints, and any attribute name defined in a constraint acquired from one ancestor must not also be defined in a constraint acquired from another, unless they *both* acquired it from a common ancestor. The effective attribute declarations are defined by the simple set union of the explicit and the acquired attribute definitions.

- For content models, either
    - all the explicit and acquired content models are vacuous in which case the effective model is vacuous;

  or

    - all but one of the explicit and the acquired content models are vacuous, in which case that one becomes the effective model;

  or

    - No element type is referenced by more than one of the explicit and acquired content models (unless two or more acquired models share modelElts acquired from a common ancestor, in which case such modelElts shall be ignored in all but the first for the purpose of constructing the effective model), in which case if the non-vacuous explicit and acquired models are all eltOnly the effective model is a *sequence* of all the non-vacuous acquired models, in the order in which they are specified in the refinements list, followed by the explicit model (if it is non-vacuous), or else if the non-vacuous explicit and acquired models are all mixed, the effective model is a mixed whose elementTypeRefs and elementTypeDecls are the union of the elementTypeRefs and elementTypeDecls of all the non-vacuous explicit and acquired models.

[Definition:] A *refinable* or *open* mixed content model is called **vacuous** if it has no declared elementTypeRefs or elementTypeDecls within it. This is because *any* archetype is substitutable for an archetype with a vacuous content model and no declared attributes.

[Definition:] An archetype AT1 is said to **refine** an archetype AT2 if and only if AT1 is declared to refine either AT2 or (recursively) some archetype that refines AT2. [Definition:] AT2 is then said to be an **ancestor** of AT1.

NOTE: Refinements are by definition substitutable for any of their ancestors.

[Definition:] We define a **substitutability** relation between two archetypes as follows:

One archetype is substitutable for another if any schema-valid instance of the former is necessarily a schema-valid instance of the latter.

Some simple cases:

- Any archetype with no attributes is substitutable for an archetype with no attributes and a contentModel of *any*;
- Any archetype which differs from another only by the specialization of one or more of its attributes (and/or of its datatype if its content is character data) is substitutable for that archetype;
- Any archetype which differs from another only by allowing only character data content (i.e. whose contentType is datatypeRef) where the other has mixed content is substitutable for that archetype.

Example

Refinement by definition creates substitutable archetypes:

```
<archetype name='Address' model='refinable'>
 <sequence>
  <elementTypeRef name='street'/>
  <elementTypeRef name='city'/>
 </sequence>
</archetype>

<archetype name='USAddress'>
 <refines>
  <archetypeRef name='Address'/>
 </refines>
 <sequence>
  <elementTypeRef name='state'/>
  <elementTypeRef name='zip'/>
 </sequence>
</archetype>
```

`USAddress` is substitutable for `Address`: the effective content model (simplifying in the obviously legitmate way) is

```

<sequence>
  <elementTypeRef name='street'/>
  <elementTypeRef name='city'/>
  <elementTypeRef name='state'/>
  <elementTypeRef name='zip'/>
 </sequence>
```

Here is a richer example:

```
<archetype name='transclude' model='refinable'>
 <mixed/>
 <attrDecl name='xml:link'>
  <fixed>simple</fixed>
 </attrDecl>
 <attrDecl name='actuate'>
  <fixed>auto</fixed>
 </attrDecl>
 <attrDecl name='href' required='true'>
  <datatypeRef name='uri'/>
 </attrDecl>
</archetype>

<archetype name='anchor' interp='open'>
 <mixed/>
 <attrDecl name='id'>
  <datatypeRef name='ID'/>
 </attrDecl>
</archetype>

<archetype name='nestHere'>
 <refines>
  <archetypeRef name='transclude'/>
  <archetypeRef name='anchor'/>
 </refines>
 <empty/>
 <attrDecl name='show'>
  <fixed>embed</fixed>
 </attrDecl>
</archetype>

<elementType name='nestHere' archetype='nestHere'/>

<nestHere href='http://www.ltg.ed.ac.uk/~ht/password.xml' id='mpw'/>
```

The final instance element satisfies all the effective constraints associated with the `nestHere` archetype: its `xml:link`, `actuate` and `href` attributes are constrained by constraints acquired from `transclude`, its `id` attribute is constrained by a constraint acquired from `anchor` and its `show` attribute is constrained by its own explicit constraint. The effective content model for `nestHere` is *empty*, under the second clause of the discussion above: only the explicit content model is there at all, so it gets to be the effective one. Note that because they are *refinable* and *open* respectively, both `transclude` and `anchor` are satisfied by the `nestHere` instance, i.e. the substitutability requirement on refining archetypes is satisfied.

Example

Here's a case which merges content models and specializes an attribute definition:

```
<archetype name='polygon' model='refinable'>
 <elementTypeRef name='bbox'/>
 <attrDecl name='n' required='yes'/>
 <attrDecl name='regular'>
  <datatypeRef name='enumeratedSymbol'>
    <enumeration>
     <literal>yes</literal>
     <literal>no</literal>
    </enumeration>
  </datatypeRef>
 </attrDecl>
</archetype>

<elementType name='regularPolygon'>
 <refines>
  <archetypeRef name='polygon'/>
 </refines>
 <elementTypeRef name='side'/>
 <attrDecl name='regular'>
  <fixed>yes</fixed>
 </attrDecl>
</elementType>
```

Here we see not only the merging of two content models, with the two elementTypeRefs being concatenated to produce the effective sequence model for the refined archetype, but also the provision in an explicit constraint of a fixed value for an attribute consistent with its acquired constraint. (Strictly speaking, this goes beyond the constraints on refinement set out above, but it obviously preserves substitutability, and will be allowed as soon as more detailed constraints are drafted.)

```

<regularPolygon n='3'>
   <bbox>...</bbox>
   <side length='3cm'/>
</regularPolygon>
```

NOTE: The intention of the formal validity definition ref-satisfies is that element item should be valid with respect to an archetypeSpec that is *refinable* if it is valid with respect to that archetypeSpec itself or any archetypeSpec that refines it. It might them be possible for validating parser to begin by checking daughters with respect to the declared archetypeSpec if it is element only, since any refinement can only add to the end of a sequence of daughters.

There follows a semi-formal definition of validity with respect to an *open* content model.

To parse against an open content model:

informal statement

ignore all daughters which are NOT mentioned explicitly anywhere in the content model: what's left must match the model in the old, deterministic, XML 1.0 way.

formal statement

Consider the edge-labelled FSM for recognizing the declared content model under a 'closed' (i.e. deterministic, XML 1.0) interpretation.

Call the set of all transition labels anywhere in the FSM (i.e. the tags mentioned anywhere anywhere in the model) *T*.

For each state *s* in the FSM, call the set of tags which label transitions from that state *T[s]*.

Then to turn the FSM into one which interprets the content model as open, for each state s

1. add a transition to failure for every label in *T-T[s]*;
2. add a transition to *s* for the complement of *T*, i.e. a default loopback.

Issue (formal-refinement-FSM): Does refinement need this, or is it covered already?

3.6 Entities and Notations

Entities and notations

```
[47]        entityDecl ::= NCName entitySpec
[48]        entitySpec ::= ( textEntitySpec |
                           externalEntitySpec |
                           unparsedEntitySpec ) exportControl?
[49]    textEntitySpec ::= string-value
[50] externalEntitySpec ::= systemID publicID? exportControl?
[51] unparsedEntitySpec ::= systemID notationRef publicID?
                           exportControl?
[52]         entityRef ::= entityName
[53]        entityName ::= NCName schemaRef?
[54]      notationDecl ::= NCName notationSpec
[55]      notationSpec ::= systemID notationRef publicID?
                           exportControl?
[56]          systemID ::= URI
[57]          publicID ::= see [XML]
[58]       notationRef ::= notationName
[59]      notationName ::= NCName schemaRef?
```

3.6.1 Internal Parsed Entity Declaration

Internal parsed entities are a feature of XML that enables reuse of text fragments by direct reference in an instance document.

In *XML Schema: Structures* documents, internal parsed entities are declared by using the textEntitySpec production.

Example

```
<textEntity name='flavor'>Fresh mint</textEntity'>
```

`flavor` can now be used in an entity reference in instances of the containing schema.

See Schema Validity for SCs covering entities and entity references.

3.6.2 External Parsed Entity Declaration

External parsed entities are a feature of XML that offers a method for including well-formed XML document fragments, including text and markup, by direct reference to the storage object of the parsed entity.

In schemas, external parsed entities are declared by using the externalEntitySpec production.

Example

```
<externalEntity name='FrontMatter'
                system='FrontMatter.xml' />
<externalEntity name='Chapter1'
                system='chapter1.xml' />
<externalEntity name='Chapter2'
                system='Chapter2.xml' />
<externalEntity name='BackMatter'
                system='BackMatter.xml' />
```

These four external entities represent the supposed contents of a book:

```
<book>
  &FrontMatter;
  &Chapter1;
  &Chapter2;
  &BackMatter;
</book>
```

In an instance, the external entities take their familiar XML form. The processor expands the entities for their content.

Again, See Schema Validity for SCs covering entities and entity references.

3.6.3 Unparsed Entity Declaration

External unparsed entities are a feature of XML that offers a baroque method for including binary data by indirect reference to both the storage object and the the notation type of the unparsed entity. In schemas, external parsed entities may be declared by using the unparsedEntitySpec production.

Example

```
<unparsedEntity name='SKU-5782-pic'
                system='http://www.vendor.com/SKU-5782.jpg'
                notation='JPEG' />
```

```
<picture location='SKU-5782-pic'/>
```

The picture element carries an attribute which is (presumably) governed by the unparsed entity declaration.

Schema-validity Constraint: Attribute is Entity
When an attribute value is interpreted as a reference to an unparsed entity [How?!], the attribute value must identifies an unparsedEntitySpec (note that no schemaRef can be specified in this case); the NCName of the notationRef of that unparsedEntitySpec must identify a notationSpec; the resource specified by the systemID and publicID attribute must be available.

Issue (unparsed-entity-gaps): There are lots of gaps and little problems in this design for unparsed entities.

3.6.4 Notation Declaration

A notation may be declared by specifying a name and an identifier for the notation. A notation may be referenced by name in a schema as part of an external entity declaration.

Example

```
<notation name='jpeg'
          public='image/jpeg' system='viewer.exe' />

<elementType name='picture>
 <attrDecl name='entity'>
   <datatypeRef name='NOTATION'/>
 </attrDecl>
</elementtype>
```

```
<picture entity='SKU-5782-pic'/>
```

The notation need not ever be mentioned in the instance document.

Issue (unparsed-entity-attributes): We need to synchronise with XML Schemas: Datatypes regarding how we declare attributes as unparsed entities!

4. Schema Composition and Namespaces

This chapter describes facilities to provide for validation of namespace-qualified instance document elements and attributes, and potentially (subject to enhancements to the Namespaces recommendation), entities and notations.

NOTE: 'Namespaces in XML' [XML-Namespaces] provides an enabling framework for modular composition of schemas. From that document:

We envision applications of Extensible Markup Language (XML) where a single XML document may contain elements and attributes (here referred to as a 'markup vocabulary') that are defined for and used by multiple software modules. One motivation for this is modularity; if such a markup vocabulary exists which is well-understood and for which there is useful software available, it is better to re-use this markup rather than re-invent it.

Such documents, containing multiple markup vocabularies, pose problems of recognition and collision. Software modules need to be able to recognize the tags and attributes which they are designed to process, even in the face of 'collisions' occurring when markup intended for some other software package uses the same element type or attribute name.

These considerations require that document constructs should have universal names, whose scope extends beyond their containing document. This specification describes a mechanism, XML Schema namespaces, which accomplishes this.

XML Schema: Structures provides facilities to enable declaration and modular composition of schemas.

- Each schema is identified by a schema name, which is a namespace-compatible [URI].
- A schema can identify archetypes, datatypes, element types, attribute definitions, attribute groups, named model groups, entities, and notations to be *exported* for use in other schemas.
- A schema can *import* declarations and definitions exported from other schemas. Means are provided for using the imported features in the importing schema.
- Namespace-compatible validation rules are defined for instance documents using imported elements and attributes.
- Conventions are proposed for use of namespace-qualified entities and notations in instance documents. Standardized implementation of these features will depend on corresponding enhancements to the [XML-Namespaces] recommendation. (To be supplied in a future draft of this specification.)
- As specified herein, namespace-qualified attribute definitions are validated only in the context of a corresponding validated elementType. A non-normative appendix proposes changes to the [XML-Namespaces] recommendation that would support more generalized export of attributes. (To be supplied in a future draft of this specification.)

As described in Conformance, full validation is supported in the case where the transitive closure of all schemas referenced by an instance document is available to the validating processor. Furthermore, the schema language is compatible with partial validation, in which validation is done only with respect to a subset of the schemas applicable to a document, or in which a well formed document (I.e. one for which no overall schema is provided) makes selective use of features imported from schemas which are available.

4.1 Associating Instance Document Constructs with Corresponding Schemata

During validation, the standard mechanisms of [XML-Namespaces] are interpreted to create associations between instance document prefixes and schema definitions. Thus, there is in each valid instance document a namespace associated with each schema to which the instance conforms. Each namespace qualified element (or eventually entity or notation), its attributes and its content, is validated against its declaration in the corresponding URI-named schema. In that sense, each schema defines and is coextensive with a namespace. The means by which schemas are located during processing is discussed below in Access to Schemata. Comprehensive rules for validation are discussed in Conformance.

Example

For example (refer to schema definitions in the previous example):

```
<SomeDocument xmlns='http://coolbits.com/someschema.xsd'
              xmlns:o='http://xyzcorp.com/otherschema.xsd'
              xmlns:p='http://xyzcorp.com/yetanotherschema.xsd'>

    <someelement>
        <name1/>
        <o:name2  p:someattr='123'/>
    </someelement>

</SomeDocument>
```

'somelement' and 'name1' (which happens to be empty) are validated against their definitions in 'http://coolbits.com/someschema.xsd'. Similarly, 'name2' is validated against its definition in 'http://xyzcorp.com/otherschema.xsd'. The existence of 'name1' and 'name2' as content of 'somelement', and specifically their URI qualification, is validated against the corresponding content model declaration for 'someelement' in 'http://coolbits.com/someschema.xsd'. Validation definition attrs-satisfy determines whether 'p:someattr' is correctly namespace qualified according to the content model definition for 'name2'.

4.2 Exporting Schema Constructs

> **NOTE:** Experience with programming languages and similar schema definition systems suggests that there is value in distinguishing the internal implementation of a module from the features or interfaces that it provides for reuse by others. For example, a schema defined to describe an automobile might intend that its definitions for 'automobile' and 'engine' be building blocks for use in other schemas, but that other constructs such as 'screw' or 'bolt' be reserved for internal use.

XML Schema: Structures constructs must be exported for external use. By default, all datatypes, archetypes, elementTypes, attribute definitions, model groups, attribute groups, entities and notations are exported. The export declaration can be used within a schema declaration to override the default behavior for each such class of declaration or definition:

Export

```
[60]             export ::= allDatatypes? allArchetypes?
                            allElementTypes? allModelGroups?
                            allAttributeGroups? allEntities?
                            allNotations?
[61]       allDatatypes ::= boolean
[62]      allArchetypes ::= boolean
[63]    allElementTypes ::= boolean
[64]     allModelGroups ::= boolean
[65] allAttributeGroups ::= boolean
[66]        allEntities ::= boolean
[67]       allNotations ::= boolean
```

Example

```
<schema name='http://coolbits.com/someschema.xsd'>
  <export datatypes='false'
          archetypes='false'
          elementTypes='true'  <!-- redundant: default is 'true'>
          ... />
[...schema body here...]
</schema>
```

This declaration restricts export of datatypes and archetypes and explicitly, although redundantly, enables export of element types.

The default for all attributes of the export declaration is 'true'. Therefore, all constructions are exported by default if 'export' is not specified, and only the constructions that are not specifically disabled are exported if 'export' is specified. If a schema A imports some other schema B, then by default, all definitions imported from B are exported from A. See Schema Import below.

Export can also be controlled individually for each particular elementType, archetype, model group, attribute group, datatype, entity or notation. Attribute definitions are implicitly exported (or not) along with the elementTypes in which they appear. An export declaration appearing on an individual elementType, archetype, or other declaration or definition overrides any default export behavior established at the schema level.

The following abstract syntax is repeated from elsewhere in this document:

Export control

```
   datatypeDefn ::=  ... exportControl?
  archetypeSpec ::=  ... exportControl?
elementTypeSpec ::=  ... exportControl?
  modelGroupDefn ::=  ... exportControl?
   attrGroupDefn ::=  ... exportControl?
      entityDecl ::=  ... exportControl?
    notationDecl ::=  ... exportControl?
   exportControl ::= boolean
```

Example

```
<elementType name='myElementType' export='true'>
        <sequence>
            <elementTypeRef name='e1'/>
            <elementTypeRef name='e2'/>
        </sequence>
        <attrGroupRef name='attrgrpname'/>
</elementType>
```

This element type declares that export is enabled.

As with any other construct, an elementType or attribute which is not exported cannot be imported for use in the declaration of a content model in an importing schema. However, an instance document providing a namespace qualified element is required to conform with the full archetype of that element, including attributes and elements which might not have been explicitly exported or imported; validating processors are required to check for conformance with the full archetype as declared in the exporting schema.

Similarly, an exported archetype applied to an elementType definition in an importing schema confers its full content type and attribute definitions, including references to elements, content models and attributes which may not have been explicitly exported.

> **NOTE:** For example, if schema B exports 'myElement', which relies on unexported constructs from schema B, instances of myElement must conform to its declared content model, including to elements and attributes which might themselves not be overtly exported.

4.3 Facilities for Schema Composition

Section Exporting Schema Constructs describes the means by which one schema can offer constructs for use in other schemas. Here we contrast two facilities which are provided for combining two or more schemas into a single schema definition. These facilities are referred to as Schema Import and Schema Inclusion respectively.

Schema import provides for schema validation of content models for instance documents that refer to two or more namespaces. Schema inclusion supports modular development and composition of a schema that is ultimately indistinguishable from an equivalent schema developed as a monolithic unit. Stated another way, imported schema components retain their original URI schema name association; included components are effectively re-declared in the including schema, and are thus indistinguishable from similar constructions declared locally.

> **NOTE:** As described below, import and include operations are effected by declarations at the head of the schema. Therefore, it is possible to discover all of a schema's external dependencies by inspection of its import and include declarations.

4.4 Schema Import

An external schema must be explicitly imported for use via a declaration at the head of the importing schema document.

Import
[68] import ::= schemaAbbrev schemaName importRestrictions?

As explained in References to Schema Constructs, imported features can be accessed through an explict schemaName URI or through a shorthand abbreviation (schemaAbbrev).

Example

```
<schema name='http://coolbits.com/someschema.xsd'>

  <!-- Establish 'other' abbrev for use in schema -->

<import schemaAbbrev='other' schemaName='http://coolbits.com/otherschema.xsd'/>

  <!-- can use 'other' anywhere in schema body -->

<elementType name='myNewElement'>
  <choice>
      <elementTypeRef schemaAbbrev='other' name='myElement'/>
      <elementTypeRef schemaAbbrev='other' name='otherElement'/>
  </choice>
</elementType>

</schema>
```

A valid instance would look like the following, delta any namespace prefix changes:

```
<myNewElement xmlns='http://coolbits.com/someschema.xsd'
              xmlns:pref1='http://coolbits.com/otherschema.xsd'>
    <pref1:myElement/>
    <pref1:otherElement/>
</myNewElement>
```

The default namespace establishes that `myNewElement` is an element name in the default namespace, and the two subordinate elements are from the namespace associated with the `pref1` prefix.

Use of schema abbreviations is explained in detail below.

4.5 Using Abbreviations to Reference Imported Schemas

As described in The Schema, the schema's schemaName property supplies a [URI] to name the schema being declared. When multiple schemas are *composed*, a schemaRef (which is either the full schemaName URI or the shorthand schemaAbbrev) is used to reference definitions and declarations from imported schemas. schemaAbbrevs are introduced via the Schema Import mechanism.

Example

The following example illustrates the general role of abbreviations in qualifying references to composed schemas:

```
<schema name='http://coolbits.com/someschema.xsd'>

    <import schemaAbbrev='schema2'
        schemaName='http://coolbits.com/otherschema.xsd'/>

    <elementType name='someelement'>
        <sequence>
            <elementTypeRef name='name1'/>
            <elementTypeRef name='name2' schemaAbbrev='schema2' />
        </sequence>
    </elementType>

</schema>
```

The example illustrates the use of references, in a single content model, to an unqualified and a abbreviation-qualified element type. A schema named with the URI 'http://coolbits.com/someschema.xsd' is shown importing features from another schema named 'http://coolbits.com/otherschema.xsd'. No abbreviation is used in the content model for element 'someelement', to indicate that 'name1' is a reference to an element defined locally. In contrast, 'name2' is qualified by the 'schema2' abbreviation and is therefore defined in 'otherschema.xsd'.

Further details of schema composition are spelled out formally in the sections below.

4.6 Import Restrictions

Import can be restricted to categories such as datatypes, archetypes, element types, and so on:

Import restrictions

```
[69] importRestrictions ::= allDatatypes? allArchetypes?
                            allElementTypes? allModelGroups?
                            allAttributeGroups? allEntities?
                            allNotations? (datatypeRef |
                            elementTypeRef | archetypeRef |
                            modelGroupRef | attrGroupRef |
                            entityRef | notationRef )*
           allDatatypes ::= boolean
          allArchetypes ::= boolean
        allElementTypes ::= boolean
         allModelGroups ::= boolean
     allAttributeGroups ::= boolean
            allEntities ::= boolean
           allNotations ::= boolean
```

Example

The following example illustrates control of schema import by category:

```
<schema name='http://coolbits.com/someschema.xsd'>

  <!-- Establish abbrev1 abbrev for use in schema -->

  <import schemaAbbrev='abbrev1'
          schemaName='http://coolbits.com/otherschema.xsd'
          archetypes='true'
          datatypes='true' >

        <elementTypeRef name='myElement'/>
        <elementTypeRef name='otherElement'/>

   <import>

   <choice>
        <elementRef schemaAbbrev='abbrev1' name='myElement'/>
        <elementRef schemaAbbrev='abbrev1' name='otherElement'/>
   </choice>

</schema>
```

In the example, all datatypes and achetypes that are exported from 'otherschema', as well as the elements named 'myElement' and 'otherElement' are imported.

Constraint on Schemas: Refer to Schema
The URI associated with a schemaRef in any of the productions above must successfully resolve to a schema.

Constraint on Schemas: Name Consistently Defined
The NCName in each of the above productions must identify a declaration or definition of the corresponding class (elementType, archetype, etc.)

Schema-validity Constraint: Use Only Exported Defns
It is not an error for a schema to explicitly import a construct which has not been exported. However, it is an error for a schema to attempt to use such construct.

> **NOTE:** A question arises as to whether imported definitions are re-exported for use by yet a third schema. Because of the difficulties in managing abbrev associations, the answer is 'no', at least in this draft of the specification.

Imported constructs are not implicitly and cannot explicitly be re-exported. However, any element or archetype is imported with its full content model, even when that content model depends on recursively imported elements or attributes. Similarly, an attribute definition can carry a datatype from an external schema, and so on. In other words, myElement in the example above might have a content model that draws on many schemas. Element1 itself is usable as shown above, but its constituent components are only available for use in new content models if explicitly imported.

Furthermore, if some schema B includes features of schema A, and if schema C includes features of B, then C can also explicitly include the same features of schema A directly.

4.7 Schema Inclusion

Although the declarations for schema inclusion resemble those for importation, the purpose is quite different. Element types, attributes, model groups, etc. are effectively redeclared when included in a schema. Specifically, the resulting construct is named in a symbol space of the including schema. As seen by authors of instance documents and other schemas, the result is indistinguishable from a definition or declaration made in the including schema.

Include

```
[70] include ::= schemaRef allDatatypes? allArchetypes?
                 allElementTypes? allModelGroups?
                 allAttributeGroups? allEntities? allNotations?
                 (datatypeRef | elementTypeRef | archetypeRef |
                 modelGroupRef | attrGroupRef | entityRef |
                 notationRef )*
```

An include specification is equivalent to a local definition of the corresponding elementTypes, archetypes, datatypes, attributes, model groups, entities, notations, and attribute groups.

Example

The following example illustrates inclusion:

```
<schema name='http://coolbits.com/someschema.xsd'>

  <include schemaName='http://coolbits.com/otherschema.xsd'>
        <elementRef name='myElement'/>
        <elementRef name='otherElement'/>
  <include>

  <!-- Note unqualified references below -->
  <choice>
       <elementTypeRef name='myElement'/>
       <elementTypeRef name='otherElement'/>
  </choice>

</schema>
```

The two elements, myElement and otherElement, are deemed to part of the overall schema's namespace. Barring accidental name collision, these two foreign element types can be treated just as though they were defined in the current schema. The definitions of their datatypes, achetypes, elements and attributes, are also implicitly included.

Because the intention is to hide the inclusion hierarchy from users, any declarations or definitions on which the inclusion depends, and which are themselves declared locally in or included by the included schema, are similarly redeclared in the including schema. Conversely, definitions or declarations imported by an included schema are effectively imported when used in an including schema.

> **NOTE:** There is some question as to whether all such features should be implicitly imported, or whether features should be imported only when needed to complete the specification of some explicitly included feature (e.g. an imported attribute needed on an included element type.)

An inclusion is treated as a local definition, so naming collisions are possible with other locally defined or included constructs. [Definition:] All include declarations encountered in a schema form a **schema path** in the order encountered. When two or more includes would cause the definition or declaration of identically named elements (I.e. the same NCName in the same symbol space), the first one encountered in the schema path (I.e. the first included) is used and any others are ignored. A local definition or declaration in an including schema supersedes any definition or declaration from any included schema.

[Definition:] A schema **directly includes** another schema if the first schema has an include and the URI contained in or abbreviated by the schemaRef of that include resolves successfully to the second schema.

[Definition:] A schema **eventually includes** another schema if the first schema directly includes the second, or if the first schema directly includes some other schema which itself eventually includes the second.

Constraint on Schemas: Preorder Priority for Included Definitions
When using a **...Ref** to identify a **...Spec**, if there is no appropriate matching declaration or definition in the current schema, but there is more than one eventually included schema which contains an appropriate matching declaration or definition, the **...Spec** whose declaration or definition occurs first in a preorder traversal of the eventually included schemas is the one identified.

4.8 Access to Schemata

Validation requires that one or more schemas be available to the validating processor. As specified above, schemas are named by URI's, which can but need not resemble in form a [URL].

The means used by some particular processor to access a schema, based on its URI, are beyond the scope of this specification, and are expected to vary from processor to processor. Whether a URI that resembles a URL is indeed usable as a URL during validation is application- and processor-dependent. This specification requires that the transitive closure of all schemas required for validation must be accessible to the processor, through whatever means, based on the URI name of each schema.

> **NOTE:** Why this choice? Several reasons, including:

- URL's are not appropriate to all the storage environments in which schema access is required.
- We cannot assume that a schema which was once accessible via URL will remain so for all time. Companies fail, organizations reorganize, but the schemas they have written and named should remain usable without modification. We cannot require a fixed website in perpetuity to host each schema.
- For widely used schemas (e.g. [HTML-4], or the schema for XML schemas), access to a single copy or even to a centrally managed replica can result in scaleability and availability problems.
- URN's are intended as at least a partial solution, but they are not yet widely deployed, and their effectiveness has yet to be proven on a large scale. Should they prove successful, this design is fully compatible with and will require no modification for URN-based naming of schemas.

5. Documenting schemas

> **NOTE:** Documentation facilities have purposely been left out of this draft of the XML Schema Definition Language specification. The editors chose to concentrate on other topics. It is anticipated that explanation elements will be provided for within any of the Schema elements. Their purpose is to encapsulate documentation for the elements within which they are contained. Elements for narrative exposition at the top level of a schema have also been proposed.

Proposals for XML Schema documentation include defining a custom set of elements, allowing any content at all, allowing all or part of [HTML-4], DocBook or TEI. There are good arguments for each of these proposals.

The Working Group must identify its requirements and constraints.

6. Conformance

> **Issue (error-behavior):** This draft includes extensive discussion of conformance and validity checking, but rules for dealing with errors are missing. In future, we must distinguish errors from fatal errors, and clarify rules for dealing with both.

6.1 Schema Validity

We approach the definition of schema validity one step at a time. In the definitions below we deal primarily in terms of information sets, rather than the documents which give rise to them: see [XML-Infoset] for definitions of *item*, *RUE* and *information set*.) Please note that the formal definitions below are explicitly *not* couched in processing terms: they describe properties of an information set, but do *not* tell you how to check an information set to see if it has those properties.

First we have to get to the schema(s) involved. This is slightly tricky, as not all namespace declarations will resolve to schemas, and not everything that purports to be a schema will be one.

[Definition:] A URI is said to **nominate** a schema if it resolves to an element item in the information set of a well-formed XML 1.0 document whose local name is `schema` and whose namespace item's URI identifies either

- this specification or one of its successors,
- the XML Schema 1.0 DTD (or the equivalent DTD from a successor to this specification)

or

- The XML Schema 1.0 schema (or the equivalent schema from a successor to this specification).

[Definition:] A URI is said to **resolve successfully** to a schema if it nominates a schema, and the element item it resolves to represents an XML schema, that is:

- If it were the document element item of an information set corresponding to an XML document whose DOCTYPE identified the XML Schema 1.0 DTD (or the equivalent DTD from a successor to this specification) as its DTD, that document would be valid per XML 1.0;
- it and its descendants satisfy all Constraints on Schemas stated in this specification.

[Definition:] An element item is **schema-ready** if the URI of any of its namespace declaration items which nominates a schema resolves successfully to a schema.

> **Issue (namespace-declaration-items):** Namespace items associated with namespace declarations have disappeared from the most recent version [XML-Infoset]. Several WGs need them, we expect they'll be back, otherwise we can reconstruct what we need from element and attribute namespace items alone with some effort.

[Definition:] A document is **schema-ready** if every element item anywhere in its information set is schema-ready.

Note that this means that documents with *no* namespace declarations, or only namespace declarations which do not nominate schemas are none-the-less schema-ready.

[Definition:] We say an element item is **schema-governed** if its name is in a namespace, and the URI of the information item for that namespace resolves successfully to a schema.

[Definition:] We use the name **schema root** for any element item which is schema-governed and which is either

- the document element

or

- the daughter of an element which is *not* schema-governed.

The provision within *XML Schema: Structures* of a mechanism for defining parsed entities presents problems for the relationship between schema-validity and XML 1.0 well-formedness, since references to entities defined only in a schema are undefined from the XML 1.0 perspective. Strictly speaking, a well-formed XML document may contain references to undefined entities only if it is declared as `standalone='no'` and contains either an external subset or one or more references to external parameter entities in their internal subset. We get around this by [Definition:] defining a **nearly well-formed** XML document to be one which either is well-formed per XML 1.0, or which fails to be well-formed only because of undefined general entity references, but which would be well-formed if it were `standalone='no'` and identified an external subset. We consider this justified on the grounds that the use of a namespace declaration which refers to a schema functions rather as an external subset, and from the XML 1.0 perspective such a reference almost of necessity renders the document non-standalone when schema-validation is applied.

[Definition:] We use the name **string-infoset-in-context** for the XML 1.0 information set items arising from the interpretation of a string in the context of a particular point in an XML 1.0 information set.

[Definition:] The **effective element item** of an element item (call this *OEI*) is an element item whose

- name and namespace items (if any) are the same as those of *OEI*;
- attribute item names and namespace items (if any) are the same as those of *OEI*;
- attribute value items are the respective value items of *OEI*'s attribute items with the string-infoset-in-context of the declared expansion of every RUE in those values replacing those RUEs;
- children based on the children of *OEI* as follows:
  - Character, PI and comment child items carried over unchanged;
  - Element child items replaced with their respective effective element items;
  - RUE child items replaced by the string-infoset-in-context of their declared expansions, with any RUE or element item in that string-infoset-in-context itself being recursively replaced per this or the above definition respectively.

Schema-validity Constraint: Expansions Schema-Ready
Any element item anywhere within the string-infoset-in-context replacing an RUE child per the above must be schema-ready.

Schema-validity Constraint: Ungoverned RUE
RUEs must not appear in element items which are not schema-governed, that is in the values of attributes of or as children of such elements.

Schema-validity Constraint: RUE Entity Declared
For every RUE appearing in a schema-governed element, there must be a parsed entity declaration in the referenced schema whose name matches the name of the RUE.

Note that the above constraints and definition mean that in error-free documents, *all* element items, even ones which are not schema-governed, have well-defined effective element items.

[Definition:] A document is **schema-valid** if and only if:

1. It is nearly-well-formed;
2. It is schema-ready;
3. Every schema root element item in the set of element items consisting of the effective element item of the document element item in the document's information set and all the element items anywhere inside that effective element item, is independently valid.

   NOTE: The validity of all other schema-governed element items follows from (3) above by the recursive nature of the Schema-validity Constraint referenced there.

   NOTE: It is intentional that the above definition labels as schema-valid a document with *no* namespace declarations or with only namespace declarations which do *not* nominate schemas.

Note that there is no requirement that the schema root mentioned above be the root of its document, or that schemas be the roots of *their* documents, or that schema and schema root be in *different* documents. Accordingly, it is possible for a single schema-valid document to contain both a schema and the material which it validates.

The interaction between XML 1.0 DTDs and XML Schemas is complex but clear:

- A document may be well-formed, (DTD) invalid and schema-valid all at the same time;
- If document has a DTD which includes parsed entity declarations, the schema-validation process may well apply to

material whose original home was in those entities (internal or external);

- Where or not a document has a DTD, the (XML 1.0) infoset we're schema-validating may well include 'RUE items', that is, references to unknown entities, unknown, that is, as far as the definitions of XML 1.0 are concerned. As long as the document's schema provides schema definitions for these 'undefined' entities, it still may be schema-valid.

  NOTE: The above is silent on whether schema-valid documents must be Namespace-conforming.

[Definition:] The **augmented information set** of a schema-valid document is the information set rooted in the effective element item of its document element, augmented by all the information items described in any Schema Information Set Contributions which apply to any information items anywhere within it.

6.2 Responsibilities of Schema-aware processors

NOTE: This section has fallen out of alignment with the rest of the specification, but is included none-the-less to give a feeling for how this section will eventually look: the details should *not* be taken too seriously.

Each step in the following presupposes the successful outcome of the previous step.

A conforming XML Schema processor must:

1. Test for XML 1.0 well-formedness;
2. Construct the XML 1.0 information set. This will include identifying and distinguishing all namespace declarations and uses, and expanding any entity references whose XML 1.0 declarations are accessible;
3. Starting from the root, traverse the information set in pre-order until an element information item with a namespace declaration which refers to an accessible schema is found;
4. Schema-validate the information set subtree rooted at that element information item using that schema, i.e.
   - Expand the information set by replacing the remaining entity references as described in Schema Validity;
   - Schema-validate the resulting information set, as described in Schema Validity;
   - Expand the information set per all Schema Information Set Contributions encountered
     - for each namespace: its prefixes and URI, and access to a schema corresponding to that URI.
     - for each element: its name (URI+GI), its content, its datatype or archetype, its attributes
     - for each attribute: its name, its value(s), its datatype, and whether its presence is required on an element.
     - for each datatype: its names, its heritage (or type lattice), its value space, its refinable facets.
     - for data: its type (and type lattice), whether it's presence is required, and its lexical constraints.
     - for each archetype: its name, its content model or datatype, its heritage (or type lattice), its attributes, and whether it is open/closed or can be refined.
     - for each element type: its name, its datatype or archetype or content model, its attributes, and whether it is open/closed or can be refined.
     - for each model: whether it is any, empty, mixed, or a group of one or more element types -- in which case, the grammar of the group, and whether it is open/closed or can be refined.
5. Go back to (3) above and continue traversing, starting with the successor in document order to the item just schema-validated, unless there is no successor;
6. Provide for an external processing system to have access to the combined information set: document instance plus schema information.

   NOTE: Note that the schema contribution to the information set above is meant to be suggestive only at this point, until we've articulated all the Schema Information Set Contributions in the preceding sections.

6.3 Lexical representation

NOTE: The editors did not get to this.

6.4 Information set

NOTE: The editors did not get to this.

Appendices

A. (normative) Schema for Schemas

The XML Schema definition for *XML Schema: Structures* itself is presented here as normative part of the specification, and as an illustrative example of the XML Schema in defining itself with the very constructs that it defines. The names of XML Schema language archetypes, element types, attributes and groups defined here are evocative of their purpose, but are occasionally verbose.

There is some annotation in comments, but a fuller annotation will require the use of embedded documentation facilities or a hyperlinked external annotation for which tools are not yet readily available.

```
<?xml version='1.0'?>
 <!DOCTYPE schema PUBLIC '-//W3C//DTD XMLSCHEMA 19990506//EN'
'http://www.w3.org/1999/05/06-xmlschema-1/structures.dtd'>

<schema xmlns='http://www.w3.org/1999/05/06-xmlschema-1/structures.xsd'
        name='http://www.w3.org/1999/05/06-xmlschema-1/structures.xsd'
        version='0.4'  >
```

Since an *XML Schema: Structures* is an XML document, it has optional XML and doctype declarations that are provided here for completeness. The root `schema` element defines a new schema. Since this is a schema for *XML Schema: Structures*, the xmlSchemaRef references the schema name itself. and specifies that this is version "0.4".

In the following definition of the `schema` element type, the preamble is reproduced with attributes corresponding to schemaName, schemaVersion and model, and a sequence of nested elements for import, export and include. The xmlns attribute corresponds to xmlSchemaRef. The `schema`'s definitions and declarations are represented by `datatype`, `archetype`, `elementType`, `attrDecl`, `attrGroup`, `modelGroup`, `textEntity`, `externalEntity`, `unParsedEntity` and `notation`.

```
    <!-- ############################################### -->
    <!-- ############################################### -->
<!-- The datatype element type, and all of its members are defined
        in XML Schema: Part 2: Datatypes -->
  <include schemaName="http://www.w3.org/1999/05/06-xmlschema-2/datatypes.xsd">

<!-- The NCName datatype is widely used for names of components -->
  <datatype name="NCName"><basetype name="NMTOKEN"/></datatype>
<!-- The public datatype is used for entities and notations -->
  <datatype name="public"><basetype name="string"/></datatype>



    <!-- ############################################### -->
    <!-- ############################################### -->
<!-- schema element type -->

  <elementType name="schema">
    <sequence>
      <elementTypeRef name="import" minOccur="0"/>
      <elementTypeRef name="export" minOccur="0"/>
      <elementTypeRef name="include" minOccur="0"/>
      <choice minOccur="0">
        <elementTypeRef name="datatype"/>
        <elementTypeRef name="archetype"/>
        <elementTypeRef name="elementType"/>
        <elementTypeRef name="attrDecl"/>
        <elementTypeRef name="attrGroup"/>
        <elementTypeRef name="modelGroup"/>
        <elementTypeRef name="textEntity"/>
        <elementTypeRef name="externalEntity"/>
        <elementTypeRef name="unparsedEntity"/>
        <elementTypeRef name="notation"/>
      </choice>
    </sequence>
    <attrDecl name="name">
      <datatypeRef name="uri">
      </datatypeRef>
    </attrDecl>
    <attrDecl name="version">
      <datatypeRef name="string">
      </datatypeRef>
    </attrDecl>
    <attrDecl name="xmlns">
      <datatypeRef name="uri">
        <default>http://www.w3.org/1999/05/06-xmlschema-1/structures.xsd
        </default>
      </datatypeRef>
    </attrDecl>
    <attrGroupRef name="model"/>
  </elementType>

  <attrGroup name="model">
    <attrDecl name="model">
      <datatypeRef name="NCName">
        <default>closed</default>
      </datatypeRef>
    </attrDecl>
  </attrGroup>

  <!-- ################################################ -->
    <!-- import, export and include -->
<!-- The import, export and include elements all include the
     restrictions attribute group, whose attributes can be used
     to enable or disable import and export restrictions.
     Within import and include elements, references to the
     components of foreign schemas control their importation
     or inclusion, respectively.  -->
    <!-- ################################################ -->
<!-- import element type  -->

  <elementType name="import">
    <choice>
      <elementTypeRef name="datatypeRef"/>
      <elementTypeRef name="archetypeRef"/>
      <elementTypeRef name="elementTypeRef"/>
      <elementTypeRef name="attrGroupRef"/>
      <elementTypeRef name="modelGroupRef"/>
      <elementTypeRef name="entityRef"/>
      <elementTypeRef name="notationRef"/>
    </choice>
    <attrDecl name="schemaAbbrev" required="true">
      <datatypeRef name="NCName">
      </datatypeRef>
    </attrDecl>
    <attrDecl name="schemaName" required="true">
      <datatypeRef name="uri">
      </datatypeRef>
    </attrDecl>
    <attrGroupRef name="restrictions"/>
  </elementType>

<!-- export element type  -->
  <elementType name="export">
    <empty/>
    <attrGroupRef name="restrictions"/>
  </elementType>

<!-- include element type  -->
  <elementType name="include">
    <choice>
      <elementTypeRef name="datatypeRef"/>
      <elementTypeRef name="archetypeRef"/>
      <elementTypeRef name="elementTypeRef"/>
      <elementTypeRef name="attrGroupRef"/>
      <elementTypeRef name="modelGroupRef"/>
      <elementTypeRef name="entityRef"/>
      <elementTypeRef name="notationRef"/>
    </choice>
    <attrGroupRef name="restrictions"/>
  </elementType>

  <attrGroup name="restrictions">
    <attrDecl name="datatypes">
      <datatypeRef name="boolean">
        <default>true</default>
      </datatypeRef>
    </attrDecl>
    <attrDecl name="archetypes">
      <datatypeRef name="boolean">
        <default>true</default>
      </datatypeRef>
    </attrDecl>
    <attrDecl name="elementTypes">
      <datatypeRef name="boolean">
        <default>true</default>
      </datatypeRef>
    </attrDecl>
    <attrDecl name="attrGroups">
      <datatypeRef name="boolean">
        <default>true</default>
      </datatypeRef>
    </attrDecl>
    <attrDecl name="modelGroups">
      <datatypeRef name="boolean">
        <default>true</default>
      </datatypeRef>
    </attrDecl>
    <attrDecl name="entities">
      <datatypeRef name="boolean">
        <default>true</default>
      </datatypeRef>
    </attrDecl>
    <attrDecl name="notations">
      <datatypeRef name="boolean">
        <default>true</default>
      </datatypeRef>
    </attrDecl>
  </attrGroup>

  <!-- ############################################### -->
    <!-- Fundamental archetypes needed to build the schema -->
<!-- Two fundamental archetypes (components and references)
     are used hereafter to build the schema. -->
    <!-- A component specifies its own name and its export control. -->
  <!-- ############################################### -->
  <!-- component archetype  -->

  <archetype name="component" model="refinable">
    <empty/>
    <attrDecl name="name">
      <datatypeRef name="NCName">
      </datatypeRef>
    </attrDecl>
    <attrDecl name="export">
      <datatypeRef name="boolean">
      </datatypeRef>
    </attrDecl>
  </archetype>


<!-- A reference specifies the name of a component, and
     optionally a schemaAbbrev or schemaName attribute.  -->
    <!-- reference archetype  -->
  <archetype name="reference" model="refinable">
    <empty/>
    <attrDecl name="name" required="true">
      <datatypeRef name="NCName">
      </datatypeRef>
    </attrDecl>
    <attrDecl name="schemaAbbrev">
      <datatypeRef name="NCName">
      </datatypeRef>
    </attrDecl>
    <attrDecl name="schemaName">
      <datatypeRef name="uri">
      </datatypeRef>
    </attrDecl>
  </archetype>


  <!-- ############################################### -->
    <!-- ######### COMPONENTS ########################### -->
  <!-- ############################################### -->

<!-- A datatypeRef element type is based on the reference archetype.
     It provides for specification of datatype qualification
     through the ordered and unordered facets and through the
     optional default and fixed elements. The content model
     is tentative, pending future coordination work between the
     is tentative, pending further coordination between the
     schema definition and datatype definition specifications. -->
    <!-- datatypeRef element type  -->
  <archetype name="datatypeRef">
    <refines> <archetypeRef name="reference"/> </refines>
    <all>
      <all>
        <modelGroupRef name="ordered"/>
        <modelGroupRef name="unordered"/>
      </all>
      <choice minOccur="0" maxOccur="1">
        <elementTypeRef name="default"/>
        <elementTypeRef name="fixed"/>
      </choice>
    </all>
  </archetype>

<!-- datatypeRef element type  -->
  <elementType name="datatypeRef">
    <archetypeRef name="datatypeRef"/>
  </elementType>

<!-- default element type  -->
  <elementType name="default">
    <datatypeRef name="string">
    </datatypeRef>
  </elementType>

<!-- fixed element type  -->
  <elementType name="fixed">
    <datatypeRef name="string">
    </datatypeRef>
  </elementType>


<!-- The archetype element type is based on the component archetype.
     It may include a refines element that specifies the archetype
     that is is based on, and either a datatypeRef or a model,
     followed by any number of attrDecl and attrGroupRef elements.  -->
    <!-- ################################################ -->
  <!-- ################################################ -->
    <!-- archetype element type -->
  <elementType name="archetype">
    <refines> <archetypeRef name="component"/> </refines>
    <sequence>
      <elementTypeRef name="refines"/>
      <choice>
        <elementTypeRef name="datatypeRef"/>
        <modelGroupRef name="model"/>
      </choice>
      <elementTypeRef name="attrDecl"/>
      <elementTypeRef name="attrGroupRef"/>
    </sequence>
    <attrGroupRef name="model"/>
  </elementType>

<!-- The model model group, and the modelElt model group that
```

```
     it references, are reusable content model fragments.
     The model model group is used in the definition of the
     archetype, elementType and modelGroup element types.
     Its members are defined later. -->

  <modelGroup name="model">
    <choice>
      <elementTypeRef name="any"/>
      <elementTypeRef name="empty"/>
      <elementTypeRef name="mixed"/>
      <modelGroupRef name="modelElt"/>
    </choice>
  </modelGroup>

  <modelGroup name="modelElt">
    <choice>
      <elementTypeRef name="all"/>
      <elementTypeRef name="choice"/>
      <elementTypeRef name="elementType"/>
      <elementTypeRef name="elementTypeRef"/>
      <elementTypeRef name="sequence"/>
      <elementTypeRef name="modelGroupRef"/>
    </choice>
  </modelGroup>


<!-- The archetypeRef element type is based on the reference archetype.  -->
    <!-- archetypeRef element type -->
  <elementType name="archetypeRef">
    <archetypeRef name="reference"/>
  </elementType>


<!-- The elementType element type is based in the component archetype.
     It is almost identical to the archetype element type except
     that it additionally provides for immediate specification of
     an archetypeRef upon which its definition is based.  -->

<!-- The elementTypeRef is based upon the reference archetype.
     It additionally provides for specification of minOccurs
     and maxOccurs attributes used when defining a content model.  -->
    <!-- ############################################### -->
  <!-- ############################################### -->
    <!-- elementType element type -->
  <elementType name="elementType">
    <refines> <archetypeRef name="component"/> </refines>
    <sequence>
      <elementTypeRef name="refines"/>
      <choice>
        <elementTypeRef name="datatypeRef"/>
        <elementTypeRef name="archetypeRef"/>
        <modelGroupRef name="model"/>
      </choice>
      <elementTypeRef name="attrDecl"/>
      <elementTypeRef name="attrGroupRef"/>
    </sequence>
    <attrGroupRef name="model"/>
  </elementType>
```

```
<!-- elementTypeRef element type -->
  <elementType name="elementTypeRef">
    <archetypeRef name="reference"/>
    <attrGroupRef name="occurrence"/>
  </elementType>

  <attrGroup name="occurrence">
    <attrDecl name="minOccur">
      <datatypeRef name="integer">
        <default>1</default>
      </datatypeRef>
    </attrDecl>
    <attrDecl name="maxOccur">
      <datatypeRef name="integer">
        <default>1</default>
      </datatypeRef>
    </attrDecl>
  </attrGroup>

<!-- The modelGroup element type is based on the component archetype.
     It must contain elements of the model model group, and it
     may specify its model and occurrence attributes. -->
    <!-- The modelGroupRef element type is based on the reference archetype. -->
    <!-- ############################################### -->
  <!-- ############################################### -->
    <!-- modelGroup element type -->
  <elementType name="modelGroup">
    <refines> <archetypeRef name="component"/> </refines>
    <modelGroupRef name="model"/>
    <attrGroupRef name="model"/>
    <attrGroupRef name="occurrence"/>
  </elementType>

<!-- modelGroupRef element type -->
  <elementType name="modelGroupRef">
    <archetypeRef name="reference"/>
    <attrGroupRef name="occurrence"/>
  </elementType>


<!-- The modelGroup element type is based in the component archetype.
     It provides for one or more attrDecl and attrGroupRef elements. -->
  <!-- The attrGroupRef element type is based on the reference archetype.  -->
    <!-- ############################################### -->
  <!-- ############################################### -->
    <!-- attrGroup element type -->
  <elementType name="attrGroup">
    <refines> <archetypeRef name="component"/> </refines>
    <choice minOccur="1" maxOccur="*">
      <elementTypeRef name="attrDecl"/>
      <elementTypeRef name="attrGroupRef"/>
    </choice>
    <attrGroupRef name="model"/>
  </elementType>

<!-- attrGroupRef element type -->
```

```
  <elementType name="attrGroupRef">
    <archetypeRef name="reference"/>
  </elementType>


<!-- The textEntity element type is based on the component archetype.
     It provides for string content to specify the entity value.  -->
    <!-- ############################################### -->
  <!-- ############################################### -->
    <!-- entities -->
  <!-- textEntity element type -->
  <elementType name="textEntity">
    <datatypeRef name="string">
    </datatypeRef>
  </elementType>

<!-- The externalEntity and unparsedEntity element types are
     based on the component archetype. They provide for specification
     of a URI, an optional public identifier, and a notation attribute.  -->
    <!-- externalEntity element type -->
  <elementType name="externalEntity">
    <refines> <archetypeRef name="component"/> </refines>
    <empty/>
    <attrGroupRef name="external"/>
  </elementType>

<!-- unparsedEntity element type -->
  <elementType name="unparsedEntity">
    <refines> <archetypeRef name="component"/> </refines>
    <empty/>
    <attrGroupRef name="external"/>
  </elementType>

  <attrGroup name="external">
    <attrDecl name="system" required="true">
      <datatypeRef name="uri">
      </datatypeRef>
    </attrDecl>
    <attrDecl name="public">
      <datatypeRef name="public">
      </datatypeRef>
    </attrDecl>
    <attrDecl name="notation" required="true">
      <datatypeRef name="notation">
        <default>XML</default>
      </datatypeRef>
    </attrDecl>
  </attrGroup>

<!-- The entityRef elementType is based on the reference archetype.
 -->
    <!-- entityRef element type -->
  <elementType name="entityRef">
    <archetypeRef name="reference"/>
  </elementType>


<!-- The notation element type is based on the component archetype.
```

```
     The notationRef element type is based in the reference archetype. -->
    <!-- notation element type -->
  <elementType name="notation">
    <refines> <archetypeRef name="component"/> </refines>
    <empty/>
    <attrGroupRef name="external"/>
  </elementType>

<!-- notationRef element type -->
  <elementType name="notationRef">
    <archetypeRef name="reference"/>
  </elementType>



  <!-- ############################################### -->
    <!-- ############################################### -->
<!-- Content model atoms -->

  <elementType name="any">
    <empty/>
  </elementType>

  <elementType name="empty">
    <empty/>
  </elementType>

  <elementType name="mixed">
    <choice minOccur="0" maxOccur="*">
      <elementTypeRef name="elementTypeRef"/>
      <elementTypeRef name="elementType"/>
    </choice>
  </elementType>



  <!-- ############################################### -->
    <!-- ############################################### -->
<!-- Element groups -->

  <archetype name="compositor">
    <choice minOccur="2" maxOccur="*">
      <modelGroupRef name="modelElt"/>
      <elementTypeRef name="elementType"/>
    </choice>
    <attrGroupRef name="occurrence"/>
  </archetype>
  <elementType name="all">
    <archetypeRef name="compositor"/>
  </elementType>

  <elementType name="choice">
    <archetypeRef name="compositor"/>
  </elementType>

  <elementType name="sequence">
    <archetypeRef name="compositor"/>
  </elementType>
  <!-- ############################################### -->
    <!-- ############################################### -->
<!-- notations for use within XML Schema: Structures  -->
  <notation name="XMLSchemaStructures" public="structures"
   system="http://www.w3.org/1999/05/06-xmlschema-1/structures.xsd"/>
  <notation name="XML" public="REC-xml-19980210"
   system="http://www.w3.org/TR/1998/REC-xml-19980210"/>
</schema>
```

NOTE: And that is the end of the schema for *XML Schema: Structures*.

B. (normative) DTD for Schemas

The following is the DTD for *XML Schema: Structures*:

```
<!ELEMENT schema ((import*, include*, export?,
                  (comment | datatype | archetype | elementType
                  | attrGroup | modelGroup | notation
                  | textEntity | externalEntity | unparsedEntity)* ))>
<!ATTLIST schema
                 name         CDATA       #IMPLIED
                 version      CDATA       #IMPLIED
                 xmlns        CDATA
"http://www.w3.org/1999/05/06-xmlschema-1/structures.xsd"
                 model      (open|refinable|closed) "closed" >

<!ELEMENT import (( elementTypeRef | archetypeRef | datatypeRef
                  | modelGroupRef | attrGroupRef | entityRef
                  | notationRef )* ) >

<!ATTLIST import
                 schemaAbbrev     NMTOKEN      #REQUIRED
                 schemaName       CDATA        #REQUIRED
                 datatypes        (true|false) "true"
                 archetypes       (true|false) "true"
                 elementTypes     (true|false) "true"
                 attrGroups       (true|false) "true"
                 modelGroups      (true|false) "true"
                 entities         (true|false) "true"
                 notations        (true|false) "true" >

<!ELEMENT export EMPTY >
<!ATTLIST export
                 datatypes        (true|false) "true"
                 archetypes       (true|false) "true"
                 elementTypes     (true|false) "true"
                 attrGroups       (true|false) "true"
                 modelGroups      (true|false) "true"
                 entities         (true|false) "true"
                 notations        (true|false) "true" >

<!ELEMENT include (( elementTypeRef | archetypeRef | datatypeRef
                  | modelGroupRef | attrGroupRef | entityRef
                  | notationRef )* ) >
```

```
<!ATTLIST include
                 schemaName       CDATA        #REQUIRED
                 datatypes        (true|false) "true"
                 archetypes       (true|false) "true"
                 elementTypes     (true|false) "true"
                 attrGroups       (true|false) "true"
                 modelGroups      (true|false) "true"
                 entities         (true|false) "true"
                 notations        (true|false) "true" >


<!-- -->
<!-- comments contain text -->
<!-- -->
<!ELEMENT comment (#PCDATA) >

<!-- -->
<!-- The datatype element is defined in XML Schema: Part 2: Datatypes -->
<!-- -->

<!ENTITY % xs-datatypes PUBLIC "datatypes"
                   "http://www.w3.org/1999/05/06-xmlschema-2/datatypes.dtd" >
%xs-datatypes;

<!ELEMENT datatypeRef ((%ordered; | %unordered)*, (default|fixed)*) >
<!ATTLIST datatypeRef
                 name            NMTOKEN    #REQUIRED
                 schemaAbbrev    NMTOKEN    #IMPLIED
                 schemaName      CDATA      #IMPLIED >

<!ELEMENT default (#PCDATA) >
<!ELEMENT fixed (#PCDATA) >

<!ENTITY % modelElt "all|choice|elementTypeRef|elementType|sequence" >
<!ENTITY % model    "any|empty|%modelElt;|mixed" >
<!-- -->
<!-- an archetype is a named content type specification -->
<!-- -->
<!ELEMENT archetype ( refines?,
                     ( datatypeRef |
                      (%model; | modelGroupRef) )?,
                     (attrDecl | attrGroupRef)* ) >
<!ATTLIST archetype
                 name        NMTOKEN     #REQUIRED
                 export     (true|false) "true"
                 model      (open|refinable|closed) #IMPLIED >

<!ELEMENT refines (archetypeRef)* >

<!ELEMENT archetypeRef EMPTY >
<!ATTLIST archetypeRef
                 name      NMTOKEN    #REQUIRED
                 schemaAbbrev    NMTOKEN    #IMPLIED
                 schemaName CDATA      #IMPLIED >

<!-- -->
<!-- an elementType is a named element and its content type -->
```

```
<!-- -->
<!ELEMENT elementType ( refines?,
                       ( datatypeRef | archetypeRef |
                        (%model; | modelGroupRef) )?,
                       (attrDecl | attrGroupRef)* ) >
<!ATTLIST elementType
                 name        NMTOKEN       #REQUIRED
                 export     (true|false)   "true"
                 model      (open|refinable|closed) #IMPLIED >

<!ELEMENT elementTypeRef EMPTY >
<!ATTLIST elementTypeRef
                 name      NMTOKEN    #REQUIRED
                 schemaAbbrev    NMTOKEN    #IMPLIED
                 schemaName CDATA      #IMPLIED
                 minOccur    CDATA      "1"
                 maxOccur    CDATA      #IMPLIED >

<!-- -->
<!-- a model is a named content model (without attributes) -->
<!-- -->
<!ELEMENT modelGroup  (%modelElt;) >
<!ATTLIST modelGroup
                 name        NMTOKEN     #REQUIRED
                 export     (true|false) "true" >

<!ELEMENT modelGroupRef EMPTY >
<!ATTLIST modelGroupRef
                 name        NMTOKEN    #REQUIRED
                 schemaAbbrev      NMTOKEN    #IMPLIED
                 schemaName   CDATA      #IMPLIED
                 minOccur    CDATA      "1"
                 maxOccur    CDATA      #IMPLIED >

<!ELEMENT any EMPTY >
<!ELEMENT empty EMPTY >
<!ELEMENT mixed (elementTypeRef|elementType)* >

<!ELEMENT sequence (modelGroupRef | sequence | choice | all
                    | elementType | elementTypeRef)+ >
<!ATTLIST sequence
                 minOccur    CDATA      "1"
                 maxOccur    CDATA      #IMPLIED >

<!ELEMENT choice (modelGroupRef | sequence | choice | all
                  | elementType | elementTypeRef)+ >
<!ATTLIST choice
                 minOccur    CDATA      "1"
                 maxOccur    CDATA      #IMPLIED >

<!ELEMENT all (modelGroupRef | sequence | choice | all
               | elementType | elementTypeRef)+ >
<!ATTLIST all
                 minOccur    CDATA      "1"
                 maxOccur    CDATA      #IMPLIED >

<!-- -->
<!-- an attrDecl names an attribute specification -->
```

```
<!-- the datatypeRef default is "string" -->
<!-- the attrValue allows * for NMTOKEN lists -->
<!-- -->
<!ELEMENT attrDecl (datatypeRef?) >
<!ATTLIST attrDecl
                 name        NMTOKEN     #REQUIRED
                 required   (true|false) "false"  >

<!-- an attrGroup is a named collection of attrDecls -->
<!ELEMENT attrGroup (attrDecl | attrGroupRef)+ >
<!ATTLIST attrGroup
                 name        NMTOKEN     #REQUIRED
                 export     (true|false) "true" >

<!ELEMENT attrGroupRef EMPTY >
<!ATTLIST attrGroupRef
                 name      NMTOKEN    #REQUIRED
                 schemaAbbrev    NMTOKEN    #IMPLIED
                 schemaName CDATA      #IMPLIED >

<!--  -->
<!-- Entities and notations in XML Schema -->
<!--  -->

<!-- the three kinds of entities share a symbol space  -->
<!ELEMENT entityRef EMPTY >
<!ATTLIST entityRef
                 name      NMTOKEN    #REQUIRED
                 schemaAbbrev    NMTOKEN    #IMPLIED
                 schemaName CDATA      #IMPLIED >

<!-- a textEntity can be referenced in documents of this type -->
<!ELEMENT textEntity (#PCDATA) >
<!ATTLIST textEntity
                 name        NMTOKEN     #REQUIRED
                 export      (true|false) #FIXED "true" >

<!-- an externalEntity can be referenced in documents of this type -->
<!ELEMENT externalEntity EMPTY >
<!ATTLIST externalEntity
                 name        NMTOKEN     #REQUIRED
                 export     (true|false) #FIXED "true"
                 public      CDATA       #IMPLIED
                 system      CDATA       #REQUIRED
                 notation    NMTOKEN     #FIXED "XML">

<!-- declares notation to be a 1st class element or entity content types -->
<!ELEMENT notation EMPTY >
<!ATTLIST notation
                 name        NMTOKEN     #REQUIRED
                 export     (true|false) #FIXED "true"
                 public      CDATA       #REQUIRED
                 system      CDATA       #IMPLIED>

<!ELEMENT notationRef EMPTY >
<!ATTLIST notationRef
                 name      NMTOKEN    #REQUIRED
                 schemaAbbrev    NMTOKEN    #IMPLIED
                 schemaName CDATA      #IMPLIED >

<!-- an unparsedEntity can be referenced in documents of this type  -->
<!ELEMENT unparsedEntity EMPTY >
<!ATTLIST unparsedEntity
                 name        NMTOKEN     #REQUIRED
                 export     (true|false) #FIXED "true"
                 public      CDATA       #IMPLIED
                 system      CDATA       #REQUIRED
                 notation    NMTOKEN     #REQUIRED >

<!NOTATION XMLSchemaStructures PUBLIC "structures"
"http://www.w3.org/1999/05/06-xmlschema-1/structures.xsd" >
<!NOTATION XML PUBLIC "REC-xml" "http://www.w3.org/TR/1998/REC-xml-19980210" >
```

C. Glossary (normative)

Ed. Note: The Glossary has barely been started. An XSL macro will be used to collect definitions from throughout the spec and gather them here for easy reference.

abstract syntax

[Definition:] the**abstract syntax** of the XML Schema Definition Language is ...

aggregate datatype

[Definition:] an **aggregate datatype** is

archetype

[Definition:] an **archetype** is

archetype reference

[Definition:] an **archetype reference** is

'all' content model group

[Definition:] the **'all' content model group** is

'any' content

[Definition:] the **'any' content** specification ...

atomic datatype

[Definition:] an **atomic datatype** is

attribute

[Definition:] an **attribute** is

attribute group

[Definition:] an **attribute group** is

'choice' content model group

[Definition:] the **'choice' content model group** is

composition

[Definition:] **composition** is

concrete syntax

[Definition:] the **concrete syntax** is

constraint

[Definition:] a **constraint** is

content

[Definition:] **content** is

context

[Definition:] a **context** is

datatype
    [Definition:] an **datatype** is
datatype reference
    [Definition:] an **datatype reference** is
default value
    [Definition:] a **default value** is
document
    [Definition:] a **document** is
element
    [Definition:] an **element** is
element content
    [Definition:] **element content** is
element type
    [Definition:] an **element type** is
element type reference
    [Definition:] an **element type reference** is
'empty' content
    [Definition:] the **'empty' content** specification ...
export
    [Definition:] to **export** is
export control
    [Definition:] an **export control**
external entity
    [Definition:] an **external entity** is
facet
    [Definition:] a **facet** is
fixed value
    [Definition:] a **fixed value**
fragment
    [Definition:] a **fragment** is
import
    [Definition:] to **import** is
include
    [Definition:] to **include** is
information set
    [Definition:] an **information set** is
instance
    [Definition:] an **instance** is
markup
    [Definition:] **markup** is
'mixed' content
    [Definition:] the **'mixed' content** specification ...
model
    [Definition:] a **model** is
model group
    [Definition:] a **model group** is
model group reference

[Definition:] a **model group reference** is

RUE
: [Definition:] **RUE** is short for *reference to undefined entity information item* as defined in [XML-Infoset]

NCName
: [Definition:] an **NCName** is a name with no namespace qualification, as defined in [XML-Namespaces]. Appears in all the definition and declaration productions of this specification.

namespace
: [Definition:] a **namespace** is

notation
: [Definition:] a **notation** is

object model
: [Definition:] an **object model** is

occurrence
: [Definition:] **occurrence** is

parameter entity
: [Definition:] a **parameter entity** is

preamble
: [Definition:] a **preamble** is

presence
: [Definition:] **presence** is

refinement
: [Definition:] **refinement** is

document root
: [Definition:] the **document root** is ...

scope
: [Definition:] **scope** is

'sequence' content model group
: [Definition:] the **'sequence' content model group** is

structure
: [Definition:] **structure** is

symbol space
: [Definition:] a **symbol space** is

text entity
: [Definition:] a **parsed entity** is

unparsed entity
: [Definition:] an **unparsed entity** is

validation
: [Definition:] **validation** is

vocabulary
: [Definition:] a **vocabulary** is

well-formedness
: [Definition:] **well-formedness** is

D. References (normative)

Austin-FTF

Fourth Meeting of the W3C XML Schema Working Group. See
http://www.w3.org/XML/Group/1999/04/xml-schema-ftf.html

DCD

Document Content Description for XML (DCD), Tim Bray et al. W3C, 10 August 1998. See
http://www.w3.org/TR/NOTE-dcd

DDML

Document Definition Markup Language. See http://www.w3.org/TR/NOTE-ddml

HTML-4

HTML 4.0 Specification, Dave Raggett et al. W3C, 1998. See http://www.w3.org/TR/REC-html40

ISO-11404

ISO 11404 -- Information Technology -- Programming Languages, their environments and system software interfaces -- Language-independent datatypes, ISO/IEC 11404:1996(E).

RFC-1808

RFC 1808,*Relative Uniform Resource Locators*. Internet Engineering Task Force. See
http://ds.internic.net/rfc/rfc1808.txt

SOX

Schema for Object-oriented XML, Matt Fuchs, et al. W3C, 1998. See http://www.w3.org/Submission/1998/15/

SOX-1.1

Schema for Object-oriented XML, Version 1.1, Matt Fuchs, et al. W3C, 1999. See ???

URI

Uniform Resource Identifiers (URI): Generic Syntax and Semantics. See
http://www.ics.uci.edu/pub/ietf/uri/draft-fielding-uri-syntax-01.txt

URL

RFC 1738,*Uniform Resource Locators (URL)*. Internet Engineering Task Force. See http://ds.internic.net/rfc/rfc1738.txt

URN

RFC 2141,*URN Syntax*. Internet Engineering Task Force. See http://ds.internic.net/rfc/rfc2141.txt

WAI-PAGEAUTH

WAI Accessibility Guidelines: Page Authoring, Gregg Vanderheiden et al. W3C, 14-Apr-1998. See
http://www.w3.org/TR/WD-WAI-PAGEAUTH

WEBARCH-EXTLANG

Web Architecture: Extensible Languages, Tim Berners-Lee and Dan Connolly. W3C, 10 Feb 1998. See
http://www.w3.org/TR/NOTE-webarch-extlang

WEBSGML

Proposed TC for WebSGML Adaptations for SGML, C. F. Goldfarb, ed., 14 June 1997. See
http://www.sgmlsource.com/8879rev/n1929.htm

XDR

XML-Data Reduced, Frankston, Charles, and Henry S. Thompson, ed. See
http://www.ltg.ed.ac.uk/~ht/XMLData-Reduced.htm

XLink

XML Linking Language (XLink), Eve Maler and Steve DeRose, W3C, 3 March 1998. See
http://www.w3.org/TR/WD-xlink

XML Schema Requirements

XML Schema Requirements , Ashok Malhotra and Murray Maloney, ed., W3C, 15 February 1999. See http://www.w3.org/TR/NOTE-xml-schema-req

XML Schemas: Datatypes

XML Schema Part 2: Datatypes, Paul V. Biron and Ashok Malhotra, eds. See http://www.w3.org/1999/05/06-xmlschema-2/datatypes

XML

Extensible Markup Language (XML) 1.0, Tim Bray, et al. W3C, 10 February 1998. See http://www.w3.org/TR/REC-xml

XML-Data

XML-Data, Andrew Layman, et al. W3C, 05 January 1998. See http://www.w3.org/TR/1998/NOTE-XML-data-0105

XML-Infoset

XML Information Set (first public WD), David Megginson et al., W3C, 1999. See http://www.w3.org/XML/Group/1999/04/WD-xml-infoset-19990428.html

XML-Namespaces

Namespaces in XML, Tim Bray et al., W3C, 1998. See http://www.w3.org/TR/WD-xml-names

XPointer

XML Pointer Language (XPointer), Eve Maler and Steve DeRose, W3C, 3 March 1998. See http://www.w3.org/TR/WD-xptr

XSchema

XSchema Specification, Simon St. Laurent, Ronald Bourret, John Cowan, et al., Version 1.0, Draft, 18 October 1998. See http://www.simonstl.com/xschema/spec/xscspecv4.htm
For more general information, consult http://purl.oclc.org/NET/xschema

XSLT

Extensible Stylesheet Language Transformations, James Clark, W3C, 21 April 1999. See http://www.w3.org/TR/1999/WD-xslt-19990421

E. Grateful Acknowledgments (non-normative)

The editors the acknowledge the members of the W3C XML Schema Working Group, the members of other W3C Working Groups, and industry experts in other forums who have contributed directly or indirectly to the process or content of creating this document. The editors are particularly grateful to Lotus Development Corp. for providing teleconferencing facilities.

F. Sample Schema (non-normative)

NOTE: The editors did not get to this.

Example

An example of a full blown schema:

```
<schema name='http://www.myOrg.com/bob/schema1.xsd' >


[...]
```

Explanation...

```
[...]

</schema>
```

Explanation...

G. Open Issues

A tabulation of open issues flagged above follows:

dummy
no-evolution
elt-default
sic-elt-default
default-attr-datatype
namespace-declare
mixed-change-current-schema
still-unambig
default-model
formal-refinement-FSM
unparsed-entity-gaps
unparsed-entity-attributes
error-behavior
namespace-declaration-items

XML Schema Part 2: Datatypes

World Wide Web Consortium Working Draft 06-May-1999

This version

http://www.w3.org/1999/05/06-xmlschema-2/
(with accompanying schema and DTD)

Latest version:

http://www.w3.org/TR/xmlschema-2/

Editors:

Paul V. Biron (Kaiser Permanente, for Health Level Seven) <Paul.V.Biron@kp.org>
Ashok Malhotra (IBM) <petsa@us.ibm.com>

Status of this Document

This is a W3C Working Draft for review by members of the W3C and other interested parties in the general public.

It has been reviewed by the XML Schema Working Group and the Working Group has agreed to its publication. Note that not that all sections of the draft represent the current consensus of the WG. Different sections of the specification may well command different levels of consensus in the WG. Public comments on this draft will be instrumental in the WG's deliberations.

Please review and send comments to www-xml-schema-comments@w3.org (archive).

The facilities described herein are in a preliminary state of design. The Working Group anticipates substantial changes, both in the mechanisms described herein, and in additional functions yet to be described. The present version should not be implemented except as a check on the design and to allow experimentation with alternative designs. *The Schema WG will not allow early implementation to constrain its ability to make changes to this specification prior to final release.*

A list of current W3C working drafts can be found at http://www.w3.org/TR. They may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use W3C Working Drafts as reference material or to cite them as other than "work in progress".

> **Ed. Note:** Several "note types" are used throughout this draft:
>
> **issue [Issue (issue-name):]**
> : something on which the editors are seeking comment
>
> **editorial note [Ed. Note:]**
> : something the editors wish to call to the attention of the reader. To be removed prior to the recommendation becoming final.
>
> **note [Note:]**
> : something the editors wish to call to the attention of the reader. To remain in the final recommendation.

Abstract

This document specifies a language for defining datatypes to be used in XML Schemas and, possibly, elsewhere.

Table of Contents

Appendices

1. Introduction

1.1 Purpose

The [XML] specification defines a limited array of facilities for applying datatypes to document content in that documents may contain or refer to DTDs that assign types to elements and attributes. However, document authors, including *authors* of traditional *documents* and those transporting *data* in XML, often require a high degree of type checking to ensure robustness in document understanding and data interchange.

The table below offers two typical examples of XML instances in which datatypes are implicit: the instance on the left represents a billing invoice, the instance on the right a memo or perhaps an email message in XML.

Data oriented	Document oriented
`<invoice>` `<orderDate>Jan 21, 1999</orderDate>` `<shipDate>Jan 25, 1999</shipDate>` `<billingAddress>` `<name>Ashok Malhotra</name>` `<street>123 IBM Ave.</street>` `<city>Hawthorne</city>` `<state>NY</state>` `<zip>10532-0000</zip>` `</billingAddress>` `<voice>555-1234</voice>` `<fax>555-4321</fax>` `</invoice>`	`<memo importance="high"` `date="03/23/1999">` `<from>Paul V. Biron</from>` `<to>Ashok Malhotra</to>` `<subject>Latest draft</subject>` `<body>` `We need to discuss the latest` `draft <emph>immediately</emph>.` `Either email me at <email>` `mailto:paul.v.biron@kp.org</email>` `or call <phone>555-9876</phone>` `</body>` `</memo>`

```
<invoice>
   <orderDate>Jan 21, 1999</orderDate>
   <shipDate>Jan 25, 1999</shipDate>
   <billingAddress>
      <name>Ashok Malhotra</name>
      <street>123 IBM Ave.</street>
      <city>Hawthorne</city>
      <state>NY</state>
      <zip>10532-0000</zip>
   </billingAddress>
   <voice>555-1234</voice>
   <fax>555-4321</fax>
</invoice>
```

```
<memo importance="high"
      date="03/23/1999">
   <from>Paul V. Biron</from>
   <to>Ashok Malhotra</to>
   <subject>Latest draft</subject>
   <body>
      We need to discuss the latest
      draft <emph>immediately</emph>.
      Either email me at <email>
      mailto:paul.v.biron@kp.org</email>
      or call <phone>555-9876</phone>
   </body>
</memo>
```

The invoice contains several dates and telephone numbers, a state (which comes from an enumerated list of values), and a zip code (which has a definable regularity of expression). The memo contains many of the same types of information: a date, a telephone number, an email address and an "importance" value (which undoubtedly comes from an enumerated list, such as "low", "medium" or "high"). Applications which process invoices and memos need to raise exceptions if something that was supposed to be a date or a telephone number does not conform to the rules for valid dates or telephone numbers.

In both cases, validity constraints exist on the content of the instances that are not expressible in XML DTDs. The limited datatyping facilities in XML have prevented validating XML processors from supplying the rigorous type checking required in these situations. The result has been that application writers have had to implement type checking in an ad hoc manner. This specification addresses the needs of both document authors and applications writers for a robust, extensible datatype system for XML which could be incorporated into XML processors. As discussed below, these datatypes can be used in other XML-related standards as well.

1.2 Requirements

The [XML Schema Requirements] document spells out concrete requirements to be fulfilled by this specification. This states that the XML Schema Language must:

1. provide for primitive data typing, including byte, date, integer, sequence, SQL & Java primitive data types, etc.;

2. define a type system that is adequate for import/export from database systems (e.g., relational, object, OLAP);
3. distinguish requirements relating to lexical data representation vs. those governing an underlying information set;
4. allow creation of user-defined datatypes, such as datatypes that are derived from existing datatypes and which may constrain certain of its properties (e.g., range, precision, length, mask).

1.3 Scope

This portion of the XML Schema Language discusses datatypes that can be used in a XML Schema. These datatypes can be specified for element content that would be specified as #PCDATA and attribute values of various types in a DTD. It is the intension of this specification that it be usable outside of the context of XML Schemas for a wide range of other XML-related activities such as [XSL] and [RDF Schema].

For the most part, this specification discusses what are sometimes referred to as *scalar datatypes* in that they constrain the lexical representation of a single literal. In some cases, as for example in **[IDREFS]**, **[ENTITIES]** and **[NMTOKENS]**, the value may consist of a list or set of literals separated by spaces. This is an example of what is called an *aggregate datatype*. Future versions of this specification will contain a more general mechanism for defining and using aggregate (collection) datatypes such as sets, bags and records.

1.4 Terminology

Ed. Note: if necessary, insert a terminology list (e.g., may, must, datatype valid, etc.)

1.5 Organization

Following this introduction, the remainder of this specification is organized into three main sections: **[Type System]** describes the conceptual framework upon which the datatype system is constructed; **[Built-in datatypes]** details the complete list of datatypes which all conforming processors must implement; **[User-generated datatypes]** discusses how to define specialized types from the built-in types and **[Conformance]** specifies the general rules concerning conforming processors.

2. Type System

This section describes the conceptual framework behind the type system defined in this specification. The framework has been influenced by the [ISO 11404] standard on language-independent datatypes as well as the datatypes for [SQL] and for programming languages such as Java.

The datatypes discussed in this specification are computer representations of well known abstract concepts such as *integer* and *date*. It is not the place of this specification to define these concepts. Many other publications provide excellent definitions.

Two concepts are essential for an understanding of datatypes as they are discussed here: a *value space* is an abstract collection of permitted values for the datatype. Each datatype also has a space of valid lexical representations or literals. A single value in the value space may map to several valid literals.

2.1 Datatype

[Definition:] In this specification, a **datatype** has a set of distinct values, called its **value space**, and is characterized by facets and/or properties of those values and by operations on or resulting in those values. Further, each datatype is characterized by a space consisting of valid lexical representations for each value in the value space.

2.2 Value space

A *value space* is a abstract collection of permitted values for the datatype. Value spaces have certain properties. For example, they always have the concept of *cardinality* and *equality* and may have the concept of *order* by which individual values within the value space can be compared to one another. Value spaces may also support operations on values such as *addition*.

[Definition:] A **value space** is the collection of permitted values for a given datatype. The value space of a given datatype can be defined in one of the following ways:

- enumerated outright, sometimes referred to as an extensional definition

- defined axiomatically from fundamental notions, sometimes referred to as an intensional definition
- defined as the subset of values from an already defined value space with a given set of properties
- defined as a combination of values from some already defined value space by a specific construction procedure

2.3 Lexical Space

In addition to its *value space*, each datatype has a lexical representation space. [Definition:] The **lexical space** for a datatype consists of a set of valid literals. Each value in the datatype's value space maps to one or more valid literals in its lexical space. For example, "100.0" and "1.0E2" are two different representations for the same value. Depending on the situation, either or both of these representations might be acceptable. The type system defined in this specification provides a mechanism for schema designers to control the value set as well as the acceptable lexical representations of the values in the value space of a datatype. Each **[Primitive datatypes]** definition includes a detailed description of the default lexical space.

2.4 Characterizing operations

Many different datatypes may share the same value space. As a result, a datatype is only partially defined by its value space. [Definition:] The **characterizing operations** for a datatype are those operations (such as "add" or "append") on or resulting in values of the datatype which distinguish this datatype from other datatypes having value spaces which are identical except possibly for substitution of literals.

Characterizing operations can be useful in choosing the appropriate datatype for particular purposes, such as mapping to or from common programming languages or database environments.

> **Ed. Note:** Currently, no characterizing operations are defined on the built-in datatypes provided by this specification; additionally, there is no means to specify characterizing operations on user-generated datatypes. This will be addressed in a future draft.

This discussion of characterizing operations in the definition of datatype is for pedagogical purposes only and does *not* imply that conforming processors must implement those operations, nor does it imply that *expressions* (containing operators) which evaluate to a given datatype will be accepted by conforming XML processors.

2.4.1 Equal

Every value space supports the notion of equality, with the following rules:

- for any two instances of values from the value space (a, b), either a is equal to b, denoted a = b, or a is not equal to b, denoted a &ne; b;
- there is no pair of instances (a, b) of values from the value space such that both a = b and a &ne; b;
- for every value a from the value space, a = a;
- for any two instances (a, b) of values from the value space, a = b if and only if b = a;
- for any three instances (a, b, c) of values from the value space, if a = b and b = c, then a = c.

On every datatype, the operation Equal is defined in terms of the equality property of the value space: for any values a, b drawn from the value space, Equal(a,b) is true if a = b, and false otherwise.

2.5 Facets

[Definition:] A **facet** is a single defining aspect of a concept or an object. Generally speaking, each facet of an item characterizes that item along independent aspects or dimensions.

The facets of a datatype serve to distinguish those aspects of one datatype which *differ* from other datatypes. Rather than being defined solely in terms of a prose description the datatypes in this specification are defined in terms of the *synthesis* of facet values which together determine the value space and properties of the datatype.

Facets are of two types: *fundamental* facets that define the datatype and *non-fundamental* or *constraining* facets that constrain the permitted values of a datatype.

2.5.1 Fundamental facets

Datatypes are characterized by properties of their value spaces. These optional properties are discussed in this section. Each of these properties give rise to a facet that serves to characterize the datatype.

2.5.1.1 Order

[Definition:] A value space, and hence a datatype, is said to be **ordered** if there exists an **order relation** defined for that value space. Order relations have the following rules:

- for every pair (a, b) from the value space, either a &le; b or b &le; a, or a = b;
- for every triple (a, b, c) from the value space, if a &le; b and b &le; c, then a &le; c.

If a value space is ordered, then the datatype will have a corresponding **[Characterizing operations]**, called InOrder(a, b), defined by:

- for every (a, b) from the value space, InOrder(a, b) is true if a &le; b, and false otherwise.

There may exist several possible order relations for a given value space. Additionally, there may exist multiple datatypes with the same value space. In such cases, each datatype will define a different order relation on the value space.

> **Ed. Note:** Currently, no order relations are defined on the built-in datatypes provided by this specification; additionally, there is no means to specify an order relation on user-generated datatypes. This will be addressed in a future draft.

2.5.1.2 Bounds

Some ordered value spaces, and hence some datatypes, are said to be bounded. [Definition:] A value space is **bounded above** if there exists a unique value *U* in the value space such that, for all values *v* in the value space, *v* &le; *U*. The value *U* is said to be an **upper bound** of the value space. [Definition:] A value space is **bounded below** if there exists a unique value *L* in the space such that, for all values *v* in the value space, *L* &le; *v*. The value *L* is then said to be a **lower bound** of the value space.

[Definition:] A datatype is bounded if its value space has both an upper and a lower bound.

2.5.1.3 Cardinality

[Definition:] Every value space has associated with it the concept of **cardinality**. Some value spaces are finite, some are countably infinite while still others are uncountably infinite. A datatype is said to have the cardinality of its value space. It is sometimes useful to categorize value spaces (and hence, datatypes) as to their cardinality. There are three significant cases:

- value spaces that are finite
- value spaces that are countably infinite and exact (see **[Exact and Approximate]**)
- value spaces that are countably infinite and approximate (see **[Exact and Approximate]**)

Every conceptually finite value space is necessarily exact. No computational datatype is uncountably infinite.

> **Ed. Note:** Currently, cardinality is not specified for the built-in datatypes provided by this specification; additionally, there is no means to specify a cardinality on user-generated datatypes. This will be addressed in a future draft.

2.5.1.4 Exact and Approximate

The computational model of a datatype may limit the degree to which values of the datatype can be distinguished. If every value in the value space of the conceptual datatype is distinguishable in the computational model from every other value in the value space, then the datatype is said to be exact.

Certain mathematical datatypes having values which do not have finite representations are said to be approximate, in the following sense:

> Let M be the mathematical datatype and C be the corresponding computational datatype, and let P be the mapping from the value space of M to the value space of C. Then for every value v' in C, there is a corresponding value v in M and a real value h such that P(x) = v' for all X in M such that |v - x| < h. That is, v' is the approximation in C to all values in M which are "within distance h of value v". Furthermore, for at least one value v' in C, there is more than one value y in M such that P(y) = v' And thus C is not an exact model of M.

In this specification, all approximate datatypes have computational models which specify, via parametric values, a degree of approximation, that is, they require a certain minimum set of values of the mathematical datatype to be distinguishable in the computational datatype.

Ed. Note: Currently, exactness is not specified for the built-in datatypes provided by this specification; additionally, there is no means to specify a exactness for user-generated datatypes. This will be addressed in a future draft.

2.5.1.5 Numeric

A datatype is said to be numeric if its values are conceptually quantities (in some mathematical number system). A datatype whose values do not have this property is said to be non-numeric.

2.5.2 Constraining or Non-fundamental facets

Constraining facets are optional properties that can be applied to a datatype to (further) constrain its value space. Constraining the value space consequently constrains the allowed lexical representations. Adding constraining facets to a **[Base type]** is used to define **[User-generated datatypes]**.

Ed. Note: should we consider units/dimensionality now? or wait for a further draft? Note that **[timePeriod]** implicitly has units.

2.5.2.1 Length

[Definition:] For the **[string]** datatype, **length** specifies the maximum number of allowable characters in the string. For the **[binary]** datatype it specifies the maximum length in bytes.

Ed. Note: We need to ultimately reconcile the notion of string length with the resolution of the i18n issues around character, indexing, etc.

2.5.2.2 Maximum Length

[Definition:] The **maxlength** facet indicates the maximum length, in bytes, of a **[string]** datatype for which the length facet is not specified.

2.5.2.3 Lexical representation

The datatypes defined in this specification are defined in terms of abstract value spaces and their properties as opposed to how values are lexically represented in XML instances. However, the lexical representation of values is of prime importance in many applications. Because of this importance, each **[Primitive datatypes]** definition includes a detailed description of its default **[Lexical Space]**. [Definition:] The **lexical representation** facet can be used to constrain the allowable representations, or literals, for values of a datatype. The meaning of the lexical representation facet depends on the datatype to which it is applied.

For example, for **[string]**, values for the lexical representation facet are either **[Pictures]** or **[Regular Expressions]**, while for **[dateTime]**, values are derived from [ISO 8601] and [SQL].

2.5.2.4 Enumeration

[Definition:] Presence of an **enumeration** facet constrains the value space of the datatype to one of the specified list. The enumeration facet can be applied to any datatype. No order or any other relationship is implied between the elements of the enumeration list.

2.5.2.5 maxInclusive

[Definition:] The **maxInclusive** facet determines the upper bound of the value space for a datatype with the **[Order]** property. The maximum value specified with this facet is *inclusive* in the sense that the value specified for the facet is itself included in the value space for the datatype.

2.5.2.6 maxExclusive

[Definition:] The **maxExclusive** facet determines the upper bound of the value space for a datatype with the **[Order]** property. The maximum value specified with this facet is *exclusive* in the sense that the value specified for the facet is itself excluded from the value space for the datatype.

2.5.2.7 minInclusive

[Definition:] The **minInclusive** facet determines the lower bound of the value space for a datatype with the [Order] property. The minimum value specified with this facet is *inclusive* in the sense that the value specified for the facet is itself included in the value space for the datatype.

2.5.2.8 minExclusive

[Definition:] The **minExclusive** facet determines the lower bound of the value space for a datatype with the [Order] property. The minimum value specified with this facet is *exclusive* in the sense that the value specified for the facet is itself excluded from the value space for the datatype.

2.6 Datatype dichotomies

It is useful to categorize the datatypes defined in this specification along various dimensions, forming a set of characterization dichotomies.

2.6.1 Atomic vs. aggregate datatypes

The first distinction to be made is that between **atomic** and **aggregate** datatypes.

- [Definition:] **Atomic** datatypes are those having values which are intrinsically indivisible.
- [Definition:] **Aggregate** datatypes are those having values which can be decomposed into two or more component values.

For example, a date that is represented as a single character string could be the value of an atomic *date* datatype; while another date represented as separate "month", "day" and "year" elements would be the value of an aggregate *date* datatype. Not surprisingly, the distinction is analogous to that between an XML element whose content model is #PCDATA and one with element content.

As discussed above, this specification focuses mainly on atomic datatypes. Later versions will address aggregate datatypes in more detail. Note that the XML attribute types [IDREFS], [ENTITIES] and [NMTOKENS] can be thought of as aggregate (list) types.

A datatype which is atomic in this specification need not be an "atomic" datatype in any programming language used to implement this specification.

2.6.2 Primitive vs. generated datatypes

- [Definition:] **Primitive** datatypes are those that are not defined in terms of other datatypes; they exist *ab initio*.
- [Definition:] **Generated** datatypes are those that are defined in terms of other datatypes.

For example, a [number] is a well defined mathematical concept that cannot be defined in terms of other datatypes while a [date] is a special case of the more general datatype [dateTime].

The datatypes defined by this specification fall into both the primitive and the generated categories. It is felt that a judiciously chosen set of primitive datatypes will serve the widest possible audience by providing a set of convenient datatypes that can be used as is, as well as providing a rich enough base from which the variety of datatypes needed by schema designers can be generated.

A datatype which is primitive in this specification need not be a "primitive" datatype in any programming language used to implement this specification.

2.6.2.1 Base type

[Definition:] Every generated datatype is defined in terms of an existing datatype, referred to as the **base type**. Base types may be either primitive or generated.

In the example above, [date] is referred to as a **subtype** of the base type [dateTime]. The value space of a subtype is a subset of the value space of the base type.

2.6.3 Built-in vs. user-generated datatypes

- [Definition:] **Built-in** datatypes are those which are entirely defined in this specification, and may be either primitive or generated;
- [Definition:] **User-generated** datatypes are those generated datatypes whose base types are built-in datatypes or user-generated datatypes and are defined by individual schema designers by giving values to constraining facets.

Conceptually there is no difference between the built-in generated datatypes included in this specification and the user-generated datatypes which will be created by individual schema designers. The built-in generated datatypes are those which are believed to be so common that if they were not defined in this specification many schema designers would end up reinventing them. Furthermore, including these generated datatypes in this specification serves to demonstrate the mechanics and utility of the datatype generation facilities of this specification.

A datatype which is built-in in this specification need not be a "built-in" datatype in any programming language used to implement this specification.

3. Built-in datatypes

Issue (nulls): A future revision of this specification will provide a general mechanism for specifying the difference between "null" and "not present" for all datatypes. Exactly what that mechanism will be is an open issue at this point.

3.1 Namespace considerations

The built-in datatypes defined by this specification are designed so that systems other than XML Schema may be able to use them. To facilitate such usage, the built-in datatypes in this specification come from the XML Datatype namespace, the namespace defined by this specification. This applies to both built-in primitive and built-in generated datatypes.

NOTE: The exact URLs for the namespace(s) defined by this W3C specification is still an open issue. This issue has been raised with the XML Coordination Group (issue 1999-0201-07 Standardizing W3C namespace URIs) for general coordination and resolution.

Each user-generated datatype is also associated with a unique namespace. However, user-generated datatypes do not come from the XML Datatype namespace; rather, they come from the namespace of the schema in which they are defined. Note that associating a namespace with a user-generated datatype is not a general purpose extensibility mechanism and does not apply to primitive datatypes. Suppose a schema author wanted to introduce a new set of primitive datatypes, say a core set of mathematical datatypes not based on the Number datatype defined as a built-in primitive by this specification. Such a schema author might try to define those datatypes, associate a unique namespace with them and expect schema processors to understand them. Unfortunately, such a scenario would not work. Each such datatype would need specialized validation code and there are still many unresolved issues regarding standard mechanisms for sharing such code.

As described in more detail in **[User-generated datatypes]**, each user-generated datatype must be defined in terms of a base type included in this specification or a user-generated datatype by assigning facets which serve to constrain the value set of the user-generated datatype to a subset of the base type. Such a mechanism works because all schema processors are required to be able to validate datatypes defined by subsetting the value space of a datatype included in this specification.

3.2 Primitive datatypes

The primitive datatypes are described below. For each primitive datatype we discuss the fundamental facets, if any, and the constraining facets, if any.

3.2.1 ID

This is the ID datatype from [XML]. It applies only to attribute values. ID has no fundamental or constraining facets.

Validity Constraint: ID

Values of type **[ID]** must match the Name production. A name must not appear more than once in an XML document as a value of this type; i.e., ID values must uniquely identify the elements which bear them.

Ed. Note: There are several situations in which we need better reference mechanisms than those provided by ID and IDREF/IDREFS. For example, it would be desirable to extend IDs and IDREFs to be typed and scoped to better represent primary key/foreign key relationships in a database. XSL has recently introduced the concept of xsl:key and xsl:keyref whereby a single property of an element can be used as a key. We need such a mechanism for XML as a whole and it would be nice if this were extended to support multi-part keys.

3.2.2 IDREF

This is the IDREF datatype from [XML]. It applies only to attribute values. IDREF has no fundamental or constraining facets.

Validity Constraint: IDREF

Values of type IDREF must match the Name production; each Name must match the value of an ID attribute on some element in the XML document; i.e. IDREF values must match the value of some ID attribute.

3.2.3 IDREFS

This is the IDREFS datatype from [XML]. It applies only to attribute values. IDREFS has no fundamental or constraining facets.

Validity Constraint: IDREFS

Values of type IDREFS must match Names; each Name must match the value of an ID attribute on some element in the XML document; i.e. IDREF values must match the value of some ID attribute.

3.2.4 ENTITY

This is the ENTITY datatype from [XML]. It applies only to attribute values. ENTITY has no fundamental or constraining facets.

Validity Constraint: Entity Name

Values of type ENTITY must match the Name production; each Name must match the name of an unparsed entity declared in the schema.

3.2.5 ENTITIES

This is the ENTITIES datatype from [XML]. It applies only to attribute values. ENTITIES has no fundamental or constraining facets.

Validity Constraint: Entity Names

Values of type ENTITIES must match the Names production; each Name must match the name of an unparsed entity declared in the schema.

3.2.6 NMTOKEN

This is the NMTOKEN datatype from [XML]. It restricts the contents to a valid XML name. NMTOKEN applies only to attribute values and has no fundamental or constraining facets.

Validity Constraint: Name Token

Values of type NMTOKEN must match the Nmtoken production; values of type NMTOKENS must match Nmtokens.

Issue (nmtoken-primitive-or-generated): should NMTOKEN be defined as a primitive (as above) or as a subtype of **[string]** with a regular expression facet such as "[a-zA-Z0-9_-]+" (or whatever the regular expression actually should be to match the Nmtoken production)? A similar issue also applies to all of the XML attribute types, **[ID]**, **[IDREF]**, **[IDREFS]**, **[ENTITY]**, **[ENTITIES]** and **[NOTATION]**.

3.2.7 NMTOKENS

This is the NMTOKENS datatype from [XML]. It restricts the contents to a list of valid XML names. NMTOKENS applies only to attribute values and has no fundamental or constraining facets.

Validity Constraint: Name Tokens

Values of type NMTOKENS must match the Nmtokens production.

3.2.8 NOTATION

This is the NOTATION datatype from [XML]. NOTATION has no fundamental or constraining facets.

Validity Constraint: Notation Attributes

Values of this type must match one of the notation names included in the declaration; all notation names in the declaration must be declared.

3.2.9 string

[Definition:] The **string** datatype represents character strings in XML. The value space of the string datatype is the set of finite sequences of UCS characters ([ISO 10646] and [Unicode]). A UCS character (or just character, for short) is an atomic unit of communication; it is not further specified except to note that every UCS character has a corresponding UCS code point, which is an integer.

3.2.9.1 Lexical Representation

The **string** datatype has an optional constraining facet called **[Lexical representation]** The value of this facet is either a **picture** or a **regular expression**. Picture types are discussed in Appendix **[Pictures]** and regular expression constraints are discussed in Appendix **[Regular Expressions]**. If this facet is not present, there is no restriction on the lexical representation.

> **Issue (picture-or-regex):** Should the values of the **[Lexical representation]** facet be **pictures**, **regexs**, both or some other mechanism?

3.2.9.2 Length

The **string** datatype has an optional constraining facet called length. If length is specified we have a fixed length character string. If length is not specified we have a variable length character string.

3.2.9.3 Maximum Length

The **string** datatype has an optional constraining facet called maxlength. If maxlength is specified for a variable length string it represents an upper bound of the length of the string. Both length and maxlength cannot be defined for the same datatype. The absolute maximum length of variable length character string depends on the XML parser implementation.

3.2.9.4 Maximum and Minimum Values

The **string** datatype also has the following constraining facets:

- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

Clearly, the effect of these constraining facets depends on the collating sequence used to define the order property for strings.

> **Ed. Note:** The issue of collating sequences for strings is complex. It will be discussed in detail in a subsequent version of this specification.

3.2.10 boolean

[Definition:] The **boolean** datatype has the value space required to support the mathematical concept of binary-valued logic: {true, false}.

> **Issue (three-valued-logic):** Do we need to add a third value "unknown" to the value space to support three-valued logic? SQL supports this. Will the general mechanism for "nulls" to be defined in a future revision handle this case?

3.2.10.1 Lexical Representation

An instance of a datatype that is defined as *boolean* can have the following legal lexical values {0, 1, true, false, yes, no}. If a lexical representation facet is not present in the datatype definition then all these lexical values are allowed. A lexical representation facet can be added to the datatype definition to restrict the lexical values to a subset of the above.

3.2.11 number

[Definition:] The **number** datatype is the standard mathematical concept of number, including the integers, reals, rationals, etc.

Number has the following constraining facets:

- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

Ed. Note: The motivation for the *number* datatype was to allow the user to specify a value that could take any legal lexical representation for a numeric quantity. This turns out to be problematic, however, due to the large number of extant representations some of which cannot be distinguished without extra information. Having to define a lexical representation facet for number seems to defeat its purpose and, thus, the number datatype may be removed from this specification.

3.2.12 dateTime

[Definition:] The **dateTime** datatype represents a combination of date and time values as defined [SQL] and in [ISO 8601] encoded as a single string. An optional facet specifies the lexical representation. If this is not specified, all lexical representation formats conforming to [ISO 8601] Section 5 and [SQL] are acceptable.

Issue (non-gregorian-dates): Both standards are limited to Gregorian dates. As an internalization issue, do we want support for non-gregorian dates? This issue also applies to **[date]**, **[time]** and **[timePeriod]**.

3.2.12.1 Lexical Representation

If this facet is specified its value must correspond to a legal representation for combinations of dates and times as defined in [SQL] and sections 5.1 and 5.4 of [ISO 8601].

Issue (dateTime-lexical-representation): We need to spell out the various SQL and ISO 8601 representations (e.g., CCYYMMDD and CCYY-MM-DD, etc.) in detail here, or in a (non-normative) appendix. We may also want to support additional formats e.g. neither SQL or ISO 8601 seems to support the 12/25/1999 format for date. A lexical representation for dateTime as a collection of elements may also be desirable. This issue also applies to **[date]**, **[time]** and **[timePeriod]**.

3.2.13 binary

[Definition:] The **binary** datatype represents strings (blobs) of binary data. It has two fundamental facets. The optional length facet specifies the length of the data. If the length is not specified the default is unlimited length. The optional "encoding" facet specifies the encoding which may be "hex" for hexadecimal digits or "base64" for MIME style Base64 data.

3.2.14 uri

[Definition:] The **uri** datatype represents a Universal Resource Identifier (URI) Reference as defined in [RFC 2396]. It has no fundamental or constraining facets.

Issue (uri-scheme-facet): should we have a facet to allow a limitation to a specific scheme? It might be useful to able to say that something was not only a URI, but that it was a "mailto" and not a "http://...".

3.3 Generated datatypes

This section gives conceptual definitions for all built-in generated datatypes defined by this specification, including a description of the facets which apply to each datatype. The concrete syntax used to define generated datatypes (whether built-in or user-generated) is given in section [User-generated datatypes] and the complete definitions of the built-in generated datatypes (written in that concrete syntax) are provided in Appendix [Built-in Generated Datatype Definitions (normative)].

3.3.1 integer

[Definition:] The **integer** datatype corresponds to the standard mathematical concept of integer numbers. The value space of the integer datatype is the infinite set {-&infin;,...,-2,-1,0,1,2,...,&infin;} although computer implementations restrict this to a finite set. The basetype of integer is [number].

Integer has the following constraining facets:

- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.3.1.1 Lexical representation

If this optional required facet is not specified, standard integer representations are acceptable. These consist of a string of digits with an optional sign or optional parentheses to indicate a negative number. Optional commas may appear after every three digits. For example: -1, 12678967543233, (100,000).

This facet must be specified if other lexical representations are desired such as the European format that allows periods after every three digits.

3.3.2 decimal

[Definition:] The **decimal** datatype restricts allowable values to numbers with an exact fractional part. The basetype of decimal is [number].

Decimal has the following required fundamental facets:

- precision: the total number of digits in the number.
- scale: the number of digits after the decimal point. Must be less than or equal to precision.

Decimal has the following constraining facets:

- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.3.2.1 Lexical representation

If this optional required facet is not specified, standard decimal representations are acceptable. These consist of a string of digits separated by a period as a decimal indicator, in accordance with the scale and precision facets, with an optional sign or optional parentheses to indicate a negative number. Optional commas may appear after every three digits. For example: -1.23, 12678967.543233, (100,000.00).

This facet must be specified if other lexical representations are desired such as the European format that uses a comma instead of a period as the decimal indicator.

3.3.3 real

[Definition:] The **real** datatype is a computational approximation to the standard mathematical concept of real numbers. These are often called floating point numbers. The basetype of real is [number].

Real has the following constraining facets:

- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.3.3.1 Lexical representation

Real values have a single standard lexical representation consisting of a mantissa followed by the character "E" followed by an exponent. The exponent must be an integer with optional sign without parentheses or commas. The mantissa must be a decimal number with optional sign without parentheses or commas. For example: -1E4, 1267.43233E12, 12.78E-2.

3.3.4 date

[Definition:] The **date** datatype represents a date value as defined in [ISO 8601] encoded as a single string. The basetype of date is **[dateTime]**. An optional fundamental facet specifies the lexical format. If this is not specified, all lexical representation formats conforming to [ISO 8601] Section 5.2 and [SQL] are acceptable. For example, 1985-04-12, 19850412.

> **Issue (other-date-representations):** Both ISO and SQL allow only the minus sign, "-", as separator? Do we want to allow other separators such as the solidus, "/" or colon, ":" ? We also need to discuss the aggregate representation for dates.

3.3.4.1 Lexical Representation

This optional facet can be used to restrict the allowed lexical representations. Its value must correspond to a legal representation for dates as defined Section 5.2 of [ISO 8601] or [SQL].

3.3.5 time

[Definition:] The **time** datatype represents a time value as defined in [ISO 8601] encoded as a single string. The basetype of date is **[dateTime]**. An optional fundamental facet specifies the lexical format. If this is not specified, all lexical representation formats conforming to [ISO 8601] Section 5.3 and [SQL] are acceptable. For example, 23:20:50, 232050.

3.3.5.1 Lexical Representation

This optional facet can be used to restrict the allowed lexical representations. Its value must correspond to a legal representation for time as defined Section 5.3 of [ISO 8601] or [SQL].

3.3.6 timePeriod

[Definition:] The **timePeriod** datatype represents a period of time as defined in [ISO 8601] encoded as a single string. The basetype of date is **[dateTime]**. A timePeriod is one of:

- a duration of time with a specific start and end. For example, 19990412T232050/19990415T021000, in [ISO 8601] syntax, where the start and end are separated by the solidus, "/", and the date and time by the letter "T".
- a duration of time without a specified start and end (e.g., 1 second, 3 months) This corresponds to the [SQL] datatype "interval". For example, 2001-01-05-12.00.00.0000 in SQL syntax.
- a duration of time with a specific start but not a specific end. For example,19990412T232050/P1Y3M15D12H30M in [ISO 8601] syntax.
- a duration of time with a specific end but not a specific start, For example, P1Y3M15D12H30M/19990415T021000 in [ISO 8601] syntax.

The [ISO 8601] formats need to be extended to support:

- A minus sign immediately following the "P" to indicate negative periods.

An optional fundamental facet specifies the lexical format. If this is not specified, all lexical representation formats conforming to [ISO 8601] Section 5.5 and [SQL] are acceptable.

3.3.6.1 Lexical Representation

If this facet is specified its value must correspond to a legal representation for time periods as defined in section 5.5 of [ISO 8601] or [SQL].

4. User-generated datatypes

A user-generated datatype can be defined from a built-in datatype by adding optional constraining facets. For example, someone may want to define a datatype called *heightInInches* from the built-in datatype *integer* by supplying *maxInclusive* and *minInclusive* facets. In this case, *heightInInches* is the name of the new user-generated datatype, *integer* is its base type and *maxInclusive* and *minInclusive* are the constraining facets.

```
<datatype name="heightInInches">
   <basetype name="real" URI="http://www.w3.org/xmlschemas/datatypes" />
   <minInclusive>
      0.0
   </minInclusive>
   <maxInclusive>
      120.0
   </maxInclusive>
</datatype>
```

Ed. Note: The abstract syntax proposed here (and the productions) are preliminary as they allow datatype definitions which are logically inconsistent (e.g., they allow numeric facets on non-numeric datatypes). This will be corrected in future drafts, as the XML Schema language comes to allow the specification of tighter constraints.

Ed. Note: This section needs more explanatory text describing the productions and their relationship to the conceptual framework described in sections **[Type System]** and **[Built-in datatypes]**.

Datatype definitions

[1]	datatypeDefn	::=	NCName basetype facets*	[Constraint: Unique datatype definitions]
[2]	basetype	::=	datatypename	
[3]	facets	::=	ordered | unordered	[Constraint: Appropriate facets]

The following is the definition for a possible built-in generated datatype "currency". Such a datatype definition could appear in the schema which defines datatypes for XML Schemas and shows that a generated datatype can have the same value space as its basetype, which might mean that it is just an alias or renaming of the basetype. In this case, the specification would probably also define some "semantics" for currency which went beyond those of decimal.

```
<datatype name="currency">
   <basetype name="dt:decimal"/>
</datatype>
```

Constraint: Unique datatype definitions

The name of the datatype being defined must be unique among the datatypes defined in the containing schema.

Constraint: Appropriate facets

If the value space of the basetype is ordered, then only ordered facets may appear in a datatype definition.

Datatype names

[4]	datatypename	::=	builtinname | usergenname	
[5]	builtinname	::=	ID | IDREF | IDREFS | NMTOKEN | NMTOKENS | ENTITY | ENTITIES | string | uri | binary | number | integer | real | decimal | dateTime | date | time | timePeriod	
[6]	usergenname	::=	NCName schemaRef	[Constraint: Datatype name]

NOTE: The production labeled datatypename above is not to be confused with that labeled datatypeName in Section 3.3.1 of [Structural Schemas].

Constraint: Datatype name

The name specified must be the name of a datatype defined in the schema in which the user-generated datatype is defined.

Facets			
[7]	ordered	::=	bounds | numeric
[8]	unordered	::=	lexicalRepresentation | enumeration | length | maxLength

Ordered facets				
[9]	bounds	::=	(minInclusive | maxInclusive)? (minExclusive | maxExclusive)?	
[10]	maxInclusive	::=	literalValue	[Constraint: Literal type]
[11]	minInclusive	::=	literalValue	[Constraint: Literal type]
[12]	minExclusive	::=	literalValue	[Constraint: Literal type]
[13]	maxExclusive	::=	literalValue	[Constraint: Literal type]

Constraint: Literal type

The literal value specified must be of the same type as the basetype in the datatype definition in which this facet appears.

Numeric facets

```
[14]   numeric ::= precision | scale
[15] precision ::= integerLiteral
[16]     scale ::= integerLiteral
```

The following is the definition for a user-generated datatype which could be used to represent monetary amounts for use in a financial management application which does not allow figures above $1M and allows only whole cents. This definition would appear in a schema authored by an end-user (i.e., not in the schema for schemas) and shows how to define a datatype by specifying facet values which constrain the range of the basetype in a manner specific to the basetype. This is different than specifying max/min values as discussed before.

This type could just as well have been defined with the potential built-in generated type "currency", defined above, as its basetype,

```
<datatype name="amount">
   <basetype name="decimal" URI="http://www.w3.org/xmlschemas/datatypes" />
   <precision>
      8
   </precision>
   <scale>
      2
   </scale>
</datatype>
```

Unordered facets

```
[17]                 length ::= integerLiteral
[18]              maxLength ::= integerLiteral
[19]            enumeration ::= literal+
[20] lexicalRepresentation ::= lexical+
[21]                lexical ::= lexicalSpec    [ Constraint: Lexical
                                                 specification ]
```

Constraint: Lexical specification

The lexical specification must be of the "correct" kind, i.e., a dateTime lexical representation for datatypes generated from **[dateTime]** etc.

The following example is a definition for a user-generated datatype which limits the possible lexical representations of dates to the two main forms found in [ISO 8601] section 5.2.1.1. This datatype definition would appear in a schema authored by an end-user and shows how to define a datatype by restricting the lexical form of its literals. The example also shows how this datatype would be used in an element definition.

```
<datatype name="myDate">
   <basetype name="date"  URI="http://www.w3.org/xmlschemas/datatypes" />
   <lexicalRepresenation>
      <lexical>
         CCYYMMDD
      </lexical>
      <lexical>
         CCYY-MM-DD
      </lexical>
   </lexicalRepresenation>
</datatype>

<elementType name="shippingDate">
   <datatypeRef name="myDate">
```

```
</elementType>
```

Given the definitions above, the following might occur in an instance document.

```
...
<shippingDate>19990510</shippingDate>
...
<shippingDate>1999-05-10</shippingDate>
```

Both of the above shipping dates refer to "abstract" date of *May 10, 1999*

The following example is a datatype definition for a user-generated datatype which limits the possible literal values of dates to the four US holidays enumerated. This datatype definition would appear in a schema authored by an end-user and shows how to define a datatype by enumerating the values in its value space. The enumerated values must be type-valid literals for the basetype.

```
<datatype name="holidays">
   <basetype name="date" URI="http://www.w3.org/xmlschemas/datatypes" />
   <enumeration>
      <literal>
         -0101    <!-- New Year's day -->
      </literal>
      <literal>
         -0704    <!-- 4th of july -->
      </literal>
      <literal>
         -1125    <!-- Thanksgiving -->
      </literal>
      <literal>
         -1225    <!-- Christmas -->
      </literal>
   </enumeration>
</datatype>
```

Literals

```
[22]         literal ::= literalValue
[23]    literalValue ::= stringLiteral | numericLiteral |
                         dateTimeLiteral | uriLiteral
[24]   stringLiteral ::= (see [string])
[25]  numericLiteral ::= integerLiteral | realLiteral |
                         decimalLiteral
[26]  integerLiteral ::= (see [integer])
[27]     realLiteral ::= see [real]
[28]  decimalLiteral ::= see [decimal]
[29] dateTimeLiteral ::= (see [dateTime])
[30]      uriLiteral ::= (see [uri])
```

Issue (definition-overriding): should it be possible to specify a value for a non-fundamental facet on an element or attribute of a given datatype in an instance document or in an element or attribute definition in a schema (see Section 3.4.4 of [Structural Schemas])? If so, what syntax should be used? This needs to be coordinated with the structural schema editorial team.

5. Conformance

The XML specification [XML] defines two levels of conformance. Well-formed documents conform to valid XML syntax but may or may not obey the constraints defined by a DTD. Valid XML documents conform to the structure laid down in a DTD. Thus, if a DTD defines an attribute as an ID, instances of XML documents conforming to the DTD can only be valid if the values of such attributes are valid XML names and are unique in the document. By introducing additional datatypes to XML, this specification extends the notion of validity in the sense that values defined to have a certain datatype in the schema must conform to the lexical representations allowed for that datatype. It also needs to be said that that is all that is expected of an XML processor. There are no expressions on datatypes. Neither are there operations on datatypes.

In some cases, datatypes will not be specified in the schema but will be specified in XML documents. In other cases, datatypes in the documents will be specialized versions of datatypes specified for the same component in the schema. Validating XML processors should be able to validate the format of values in XML documents in these cases as well.

Appendices

A. Schema for Datatype Definitions (normative)

```
<?xml version='1.0'?>
<!DOCTYPE schema PUBLIC '-//W3C//DTD XSDL 19990506//EN'
                        'http://www.w3.org/1999/05/06-xsdl/WD-xsdl.dtd'>

<schema xmlns='http://www.w3.org/TR/1999/WD-xdtl-19990506.xsd'
        name='http://www.w3.org/TR/1999/WD-xdtl-19990506.xsd'
        version='0.1'>
<modelGroup name="ordered">
   <choice>
      <modelGroupRef name="bounds"/>
      <modelGroupRef name="numeric"/>
   </choice>
</modelGroup>
<modelGroup name="bounds">
   <choice>
      <sequence>
         <elementTypeRef name="minInclusive" minOccur="0" maxOccur="1"/>
         <elementTypeRef name="maxInclusive" minOccur="0" maxOccur="1"/>
      </sequence>
      <sequence>
         <elementTypeRef name="minExclusive" minOccur="0" maxOccur="1"/>
         <elementTypeRef name="maxExclusive" minOccur="0" maxOccur="1"/>
      </sequence>
   </choice>
</modelGroup>
<modelGroup name="numeric">
   <choice>
      <elementTypeRef name="precision"/>
      <elementTypeRef name="scale"/>
   </choice>
</modelGroup>
<modelGroup name="unordered">
   <choice>
      <modelGroupRef name="lexicalRepresentation"/>
      <modelGroupRef name="enumeration"/>
      <modelGroupRef name="length"/>
      <modelGroupRef name="maxLength"/>
   </choice>
</modelGroup>

<elementType name="datatype">
   <sequence>
      <elementTypeRef name="basetype"/>
      <choice minOccur="0" maxOccur="*">
         <modelGroupRef name="ordered"/>
         <modelGroupRef name="unordered"/>
      </choice>
   </sequence>
   <attrDecl name="name" required="true">
      <datatypeRef name="NMTOKEN"/>
   </attrDecl>
   <attrDecl name="export">
      <datatypeRef name="boolean">
         <default>true</default>
      </datatype>
   </attrDecl>
</elementType>
<elementType name="basetype">
   <empty/>
   <attrDecl name="name" required="true">
      <datatypeRef name="NMTOKEN"/>
   </attrDecl>
   <attrDecl name="schemaAbbrev">
      <datatypeRef name="NMTOKEN"/>
   </attrDecl>
   <attrDecl name="schemaName">
      <datatypeRef name="uri"/>
   </attrDecl>
</elementType>
<elementType name="maxExclusive">
   <datatypeRef name="string"/> <!-- the datatype depends on the basetype -->
</elementType>
<elementType name="minExclusive">
   <datatypeRef name="string"/> <!-- the datatype depends on the basetype -->
</elementType>
<elementType name="maxInclusive">
   <datatypeRef name="string"/> <!-- the datatype depends on the basetype -->
</elementType>
<elementType name="minInclusive">
   <datatypeRef name="string"/> <!-- the datatype depends on the basetype -->
</elementType>
<elementType name="precision">
   <datatypeRef name="integer"/>
</elementType>
<elementType name="scale">
   <datatypeRef name="integer"/>
</elementType>
<elementType name="length">
   <datatypeRef name="integer"/>
</elementType>
<elementType name="maxLength">
   <datatypeRef name="integer"/>
</elementType>
<elementType name="enumeration">
```

```
   <sequence minOccur="1" maxOccur="*">
      <elementTypeRef name="literal"/>
   </sequence>
</elementType>
<elementType name="literal">
   <datatypeRef name="string"/>  <!-- the datatype depends on the basetype -->
</elementType>
<elementType name="lexicalRepresentation">
   <sequence minOccur="1" maxOccur="*">
      <elementTypeRef name="lexical"/>
   </sequence>
</elementType>
<elementType name="lexical">
   <datatypeRef name="string"/>  <!-- the datatype depends on the basetype -->
</elementType>
</schema>
```

B. DTD for Datatype Definitions (normative)

```

<!ENTITY % numeric "precision | scale">
<!ENTITY % bounds "((minInclusive | minExclusive)?,
  (maxInclusive | maxExclusive)?)">
<!ENTITY % ordered "%bounds; | %numeric;">
<!ENTITY % unordered "lexicalRepresentation | enumeration
  | length | maxLength">

<!ELEMENT datatype (basetype, (%ordered; | %unordered;)*)>
<!ATTLIST datatype
        name NMTOKEN #REQUIRED
        export (true|false) "true">

<!ELEMENT basetype EMPTY>
<!ATTLIST basetype
   name NMTOKEN #REQUIRED
   schemaAbbrev NMTOKEN #IMPLIED
   schemaName CDATA #IMPLIED>

<!ELEMENT maxExclusive (#PCDATA)>
<!ELEMENT minExclusive (#PCDATA)>
<!ELEMENT maxInclusive (#PCDATA)>
<!ELEMENT minInclusive (#PCDATA)>

<!ELEMENT precision (#PCDATA)>
<!ELEMENT scale (#PCDATA)>

<!ELEMENT length (#PCDATA)>
<!ELEMENT maxLength (#PCDATA)>
<!ELEMENT enumeration (literal)+>
<!ELEMENT literal (#PCDATA)>
<!ELEMENT lexicalRepresentation (lexical)+>
<!ELEMENT lexical (#PCDATA)>
```

C. Built-in Generated Datatype Definitions (normative)

This section gives the datatype definitions for all built-in generated datatypes. These definitions are to appear in the "schema for schemas" and *not* in schema instances written by end-users.

> **Ed. Note:** this section needs to be expanded to include all built-in generated datatypes defined in [**Generated datatypes**]

```
<datatype name="date">
   <basetype name="dateTime" URI="http://www.w3.org/xmlschemas/datatypes" />
   <lexicalRepresentation>
      <!--ISO 8601 section 5.2.1.1 and SQL-->
      <lexical>
         CCYYMMDD  <!-- 19850412 ==> April 12, 1985-->
      </lexical>
      <lexical>
         CCYY-MM-DD<!-- 1985-04-12 ==> April 12, 1985-->
      </lexical>
      <!--ISO 8601 section 5.2.1.2-->
      <lexical>
         CCYY-MM   <!-- 1985-04 ==> April, 1985-->
      </lexical>
      <lexical>
         CCYY      <!-- 1985 ==> 1985-->
      </lexical>
      <lexical>
         CC        <!-- 19 ==> the 1900's-->
      </lexical>
      <!--ISO 8601 section 5.2.1.3-->
      <lexical>
         YYMMDD    <!-- 850412 ==> April 12, '85 (in the current century)-->
      </lexical>
      <lexical>
         YY-MM-DD  <!-- 85-04-12 ==> April 12, '85 (in the current century)-->
      </lexical>
      <lexical>
         -YYMM     <!-- -8504 ==> April, '85 (in the current century)-->
      </lexical>
      <lexical>
         -YY-MM    <!-- -85-04 ==> April, '85 (in the current century)-->
      </lexical>
      <lexical>
         -YY       <!-- -85 ==> '85 (in the current century)-->
      </lexical>
      <lexical>
         --MMDD    <!-- --0412 ==> April 12-->
      </lexical>
      <lexical>
         --MM-DD   <!-- --04-12 ==> April 12-->
      </lexical>
      <lexical>
         --MM      <!-- --04 ==> April-->
      </lexical>
      <lexical>
         ---DD     <!-- ---12 ==> the 12th-->
      </lexical>
      <!--ISO 8601 section 5.2.2.1-->
      <lexical>
```

```
         CCYYDDD   <!-- 1985102 ==> April 12, 1985 (i.e., the 102nd day of 1985)-->
      </lexical>
      <lexical>
         CCYY-DDD  <!-- 1985-102 ==> April 12, 1985 (i.e., the 102nd day of 1985)-->
      </lexical>
      <!--ISO 8601 section 5.2.2.2-->
      <lexical>
         YYDDD     <!-- 85102 ==> April 12, '85 (in the current century) (i.e., the
102nd day of '85)-->
      </lexical>
      <lexical>
         YY-DDD    <!-- 85-102 ==> April 12, '85 (in the current century) (i.e., the
102nd day of '85)-->
      </lexical>
      <lexical>
         -DDD      <!-- -102 ==> April 12 (i.e., the 102nd day of the year)-->
      </lexical>
      <!--ISO 8601 section 5.2.3.1-->
      <lexical>
         CCYYWwwD  <!-- 1985W155 ==> April 12, 1985 (i.e., the 5th day of the 15th
week of 1985)-->
      </lexical>
      <lexical>
         CCYY-Www-D<!-- 1985-W15-5 ==> April 12, 1985 (i.e., the 5th day of the 15th
week of 1985)-->
      </lexical>
      <!--ISO 8601 section 5.2.3.2-->
      <lexical>
         CCYYWww   <!-- 1985W15 ==> the 15th week of 1985-->
      </lexical>
      <lexical>
         CCYY-Www  <!-- 1985-W15 ==> the 15th week of 1985-->
      </lexical>
      <!--ISO 8601 section 5.2.3.3-->
      <lexical>
         YYWwwD    <!-- 85W155 ==> April 12, '85 (in the current century) (i.e., the
5th day of the 15th week of '85)-->
      </lexical>
      <lexical>
         YY-Www-D  <!-- 85-W15-5 ==> April 12, '85 (in the current century) (i.e.,
the 5th day of the 15th week of '85)-->
      </lexical>
      <lexical>
         YYWww     <!-- 85W15 ==> the 15th week in the current century-->
      </lexical>
      <lexical>
         YY-Www    <!-- 85-W15 ==> the 15th week in the current century-->
      </lexical>
      <lexical>
         -YWwwD    <!-- -5W155 ==> April 12, ''5 (in the current decade)-->
      </lexical>
      <lexical>
         -Y-Www-D  <!-- -5W15-5 ==> April 12, ''5 (in the current decade)-->
      </lexical>
      <lexical>
         -WwwD     <!-- -W155 ==> April 12 (in the current year)-->
      </lexical>
      <lexical>
```

```
         -Www-D    <!-- -W15-5 ==> April 12 (in the current year)-->
      </lexical>
      <lexical>
         -Www      <!-- -W15 ==> 15th week in the current year-->
      </lexical>
      <lexical>
         -W-D      <!-- -W-5 ==> Friday (of the current week) (i.e., the 5th day)-->
      </lexical>
      <lexical>
         ---D      <!-- ---5 ==> Friday (of any week in any year)-->
      </lexical>
   </lexicalRepresentation>
</datatype>
```

D. Pictures

"Pictures", similar to those in [COBOL] picture clauses, can be used to constrain the format of strings and in some cases control their conversion to numbers. A picture is an alphanumeric string consisting of character symbols. Each symbol, which is usually one character but may be two characters, is a placeholder that stands for a set of characters. For example, the picture "A" stands for a single alphabetic character.

The following is a list of picture symbols and their meanings.

A

A single alphabetic character.

B

A single blank character.

E

The character E, used to indicate floating point numbers.

S

The leftmost character of a picture indicating a signed number. The characters "+" or "-" may appear in the S position.

V

An implied decimal sign. The input 1234 validated by a picture 99V99 is converted into 12.34.

X

Any character.

Z

The leftmost leading numeric character that can be replaced by a space character when the content of that content position is a zero.

9

Any numeric character.

1

Any boolean character (0 or 1).

0,/,-,., and ,

represent themselves.

cs

A placeholder for an appropriate currency symbol.

Here are some examples of picture constraints

```
   $123,45.90 satisfies picture $999,99.99
   $123,45.90 satisfies picture XXXX,XX.XX
   123-45-5678 satisfies picture 999-99-9999 (Social Security Number)
   24E80 satisfies picture 99E99 (floating point)
   23.45 satisfies picture 99.99
   2345 satisfies picture 99V99 (translates to 23.45)
```

E. Regular Expressions

Ed. Note: The following description of regular expressions is copied (with slight modification) by permission from the documentation of the [Perl] programming language.

Issue (perl-regex): Should the final recommendation use Perl's regular expression "extensions"?

[Definition:] **Regular expressions**, similar to those in [Perl], can be used to constrain the format of strings. A regular expression is an alphanumeric string consisting of character symbols. Each symbol, which is usually one character but may be two characters, is a placeholder that stands for a set of characters.

Any single character matches itself, unless it is a metacharacter with a special meaning described here or above. You can cause characters that normally function as metacharacters to be interpreted literally by prefixing them with a "\" (e.g., "\." matches a ".", not any character; "\\" matches a "\"). A series of characters matches that series of characters in the target string, so the pattern blurfl would match "blurfl" in the target string.

You can specify a character class, by enclosing a list of characters in [], which will match any one character from the list. If the first character after the "[" is "^", the class matches any character not in the list. Within a list, the "-" character is used to specify a range, so that a-z represents all characters between "a" and "z", inclusive. If you want "-" itself to be a member of a class, put it at the start or end of the list, or escape it with a backslash. (The following all specify the same class of three characters: [-az], [az-], and [a\-z]. All are different from [a-z], which specifies a class containing twenty-six characters.)

Certain characters as used as metacharacters. The following list contains all of the metacharacters and their meanings.

\

Quote the next metacharacter

^

Match the beginning of the line

.

Match any character (except newline)

$

Match the end of the line (or before newline at the end)

|

Alternation

()

Grouping

[]

Character class

Within a regular expression, the following standard quantifiers are recognized:

*****

Match 0 or more times

+

Match 1 or more times

?

Match 1 or 0 times

{n}

Match exactly n times

{n,}

Match at least n times

{n,m}

Match at least n but not more than m times

The following character sequences also have special meaning within a regular expression.

\t

tab

\n

newline

\r

return

\033

octal char 003

\x1B

hex char 1B

\w

Match a "word" character (alphanumeric plus "_")

\W

Match a non-word character

\s

Match a whitespace character

\S

Match a non-whitespace character

\d

Match a digit character

\D

Match a non-digit character

Ed. Note: we should probably define XML-specific character sequences for things like Nmtoken, Name, etc., as well as ones for the character classes listed in XML 1.0 Appendix B. Character Classes

Regular expressions may also contain the following zero-width assertions:

\b

Match a word boundary

\B

Match a non-(word boundary)

A word boundary (\b) is defined as a spot between two characters that has a \w on one side of it and a \W on the other side of it (in either order), counting the imaginary characters off the beginning and end of the string as matching a \W.

```
  555-1212     is matched by \d{3}-\d{4}           (phone number)
  888-555-1212 is matched by (\d{3}-)?\d{3}-\d{4}  (phone number with optional area
code)
  $123,45.90   is matched by \$\d{3},\d{2}\.\d{2}
  123-45-5678  is matched by \d{3}-?\d{2}-?\d{4}   (Social Security Number)
```

F. References

COBOL

COBOL Standard. See http://www.dkuug.dk/jtc1/sc22/wg4/

ISO 10646

ISO (International Organization for Standardization). *ISO/IEC 10646-1993 (E). Information technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Part 1: Architecture and Basic Multilingual Plane.* [Geneva]: International Organization for Standardization, 1993 (plus amendments AM 1 through AM 7).

ISO 11404

Language-independent Datatypes. Available from http://web.ansi.org/public/catalog/cat_top.html

ISO 8601

Representations of dates and times. See http://www.iso.ch/markete/8601.pdf
Available from http://www.w3.org/TR/1998/WD-xsl-19981216

Namespaces in XML

Namespaces in XML, Tim Bray et al. W3C, 1998 Available at: http://www.w3.org/TR/REC-xml-names

Perl

The Perl Programming Language. See http://www.perl.org

RDF Schema

RDF Schema Specification. See http://www.w3.org/TR/PR-rdf-schema

RFC 2396

Tim Berners-Lee, et. al. *RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax.*. 1998 Available at: http://www.ietf.org/rfc/rfc2396.txt

SQL

SQL Standard. See http://www.jcc.com/SQLPages/jccs_sql.htm

Structural Schemas

XML Schema Part 1: Structures. Available at: http://www.w3.org/XML/Group/1999/05/06-xmlschema-1/

Unicode

The Unicode Consortium. *The Unicode Standard, Version 2.0.* Reading, Mass.: Addison-Wesley Developers Press, 1996.

XML

XML Standard. See http://www.w3.org/TR/REC-xml

XML Schema Requirements

XML Schema Requirements. Available at: http://www.w3.org/TR/NOTE-xml-schema-req

XSL

XSL Working Draft. See http://www.w3.org/TR/1998/WD-xsl-19981216

G. Open Issues

nulls
nmtoken-primitive-or-generated
picture-or-regex
three-valued-logic
non-gregorian-dates

dateTime-lexical-representation
uri-scheme-facet
other-date-representations
definition-overriding
perl-regex

```
<?xml version='1.0'?>
<!DOCTYPE schema PUBLIC '-//W3C//DTD XSDL 19990506//EN'
                        'http://www.w3.org/1999/05/06-xsdl/WD-xsdl.dtd'>

<schema xmlns='http://www.w3.org/TR/1999/WD-xdtl-19990506.xsd'
        name='http://www.w3.org/TR/1999/WD-xdtl-19990506.xsd'
        version='0.1'>
<modelGroup name="ordered">
   <choice>
      <modelGroupRef name="bounds"/>
      <modelGroupRef name="numeric"/>
   </choice>
</modelGroup>
<modelGroup name="bounds">
   <choice>
      <sequence>
         <elementTypeRef name="minInclusive" minOccur="0" maxOccur="1"/>
         <elementTypeRef name="maxInclusive" minOccur="0" maxOccur="1"/>
      </sequence>
      <sequence>
         <elementTypeRef name="minExclusive" minOccur="0" maxOccur="1"/>
         <elementTypeRef name="maxExclusive" minOccur="0" maxOccur="1"/>
      </sequence>
   </choice>
</modelGroup>
<modelGroup name="numeric">
   <choice>
      <elementTypeRef name="precision"/>
      <elementTypeRef name="scale"/>
   </choice>
</modelGroup>
<modelGroup name="unordered">
   <choice>
      <modelGroupRef name="lexicalRepresentation"/>
      <modelGroupRef name="enumeration"/>
      <modelGroupRef name="length"/>
      <modelGroupRef name="maxLength"/>
   </choice>
</modelGroup>

<elementType name="datatype">
   <sequence>
      <elementTypeRef name="basetype"/>
      <choice minOccur="0" maxOccur="*">
         <modelGroupRef name="ordered"/>
         <modelGroupRef name="unordered"/>
      </choice>
   </sequence>
   <attrDecl name="name" required="true">
      <datatypeRef name="NMTOKEN"/>
   </attrDecl>
   <attrDecl name="export">
      <datatypeRef name="boolean">
         <default>true</default>
      </datatype>
   </attrDecl>
</elementType>
<elementType name="basetype">
```

```
      <empty/>
      <attrDecl name="name" required="true">
         <datatypeRef name="NMTOKEN"/>
      </attrDecl>
      <attrDecl name="schemaAbbrev">
         <datatypeRef name="NMTOKEN"/>
      </attrDecl>
      <attrDecl name="schemaName">
         <datatypeRef name="uri"/>
      </attrDecl>
   </elementType>
   <elementType name="maxExclusive">
      <datatypeRef name="string"/> <!-- the datatype depends on the basetype -->
   </elementType>
   <elementType name="minExclusive">
      <datatypeRef name="string"/> <!-- the datatype depends on the basetype -->
   </elementType>
   <elementType name="maxInclusive">
      <datatypeRef name="string"/> <!-- the datatype depends on the basetype -->
   </elementType>
   <elementType name="minInclusive">
      <datatypeRef name="string"/> <!-- the datatype depends on the basetype -->
   </elementType>
   <elementType name="precision">
      <datatypeRef name="integer"/>
   </elementType>
   <elementType name="scale">
      <datatypeRef name="integer"/>
   </elementType>
   <elementType name="length">
      <datatypeRef name="integer"/>
   </elementType>
   <elementType name="maxLength">
      <datatypeRef name="integer"/>
   </elementType>
   <elementType name="enumeration">
      <sequence minOccur="1" maxOccur="*">
         <elementTypeRef name="literal"/>
      </sequence>
   </elementType>
   <elementType name="literal">
      <datatypeRef name="string"/>  <!-- the datatype depends on the basetype -->
   </elementType>
   <elementType name="lexicalRepresentation">
      <sequence minOccur="1" maxOccur="*">
         <elementTypeRef name="lexical"/>
      </sequence>
   </elementType>
   <elementType name="lexical">
      <datatypeRef name="string"/>  <!-- the datatype depends on the basetype -->
   </elementType>
   </schema>
```

```
<!ENTITY % numeric "precision | scale">
<!ENTITY % bounds "((minInclusive | minExclusive)?,
  (maxInclusive | maxExclusive)?)">
<!ENTITY % ordered "%bounds; | %numeric;">
<!ENTITY % unordered "lexicalRepresentation | enumeration
  | length | maxLength">

<!ELEMENT datatype (basetype, (%ordered; | %unordered;)*)>
<!ATTLIST datatype
        name NMTOKEN #REQUIRED
        export (true|false) "true">

<!ELEMENT basetype EMPTY>
<!ATTLIST basetype
   name NMTOKEN #REQUIRED
   schemaAbbrev NMTOKEN #IMPLIED
   schemaName CDATA #IMPLIED>

<!ELEMENT maxExclusive (#PCDATA)>
<!ELEMENT minExclusive (#PCDATA)>
<!ELEMENT maxInclusive (#PCDATA)>
<!ELEMENT minInclusive (#PCDATA)>

<!ELEMENT precision (#PCDATA)>
<!ELEMENT scale (#PCDATA)>

<!ELEMENT length (#PCDATA)>
<!ELEMENT maxLength (#PCDATA)>
<!ELEMENT enumeration (literal)+>
<!ELEMENT literal (#PCDATA)>
<!ELEMENT lexicalRepresentation (lexical)+>
<!ELEMENT lexical (#PCDATA)>
```