XML Schema Part 1: Structures

W3C Working Draft 22 September 2000

This version:
http://www.w3.org/TR/2000/WD-xmlschema-1-20000922/
(in XML (with its own DTD, XSL stylesheet (Nov REC version)) and HTML), with separate provision of the schema and DTD for schemas described herein.
Latest version:
http://www.w3.org/TR/xmlschema-1/
Previous version:
http://www.w3.org/TR/2000/WD-xmlschema-1-20000407/
Editors:
Henry S. Thompson (University of Edinburgh)<ht@cogsci.ed.ac.uk>
David Beech (Oracle Corp.) <dbeech@us.oracle.com>
Murray Maloney (for Commerce One)<murray@muzmo.com>
Noah Mendelsohn (Lotus Development Corporation) <Noah_Mendelsohn@lotus.com>

Abstract

XML Schema: Structures specifies the XML Schema definition language, which offers 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). This specification depends on *XML Schema Part 2: Datatypes*.

Status of this document

This is a public working draft of XML Schema 1.0, issued by the XML Schema Working Group, for review by the public and by members and working groups of the World Wide Web Consortium.

This working draft incorporates most Working Group decisions through 2000-09-19. It has been reviewed by the XML Schema Working Group, and the Working Group has agreed to its publication as a working draft, which includes our proposed resolution of most issues raised during Last Call. The Working Group intends to submit this specification for publication as a Candidate Recommendation very soon, but is issuing this interim public draft as it sets out a number of changes to the XML Representation of XML Schemas, and we wished to make these available as quickly as possible. Readers may find Description of changes (non-normative) (§H) helpful in identifying the major changes since the last Public Working Draft.

Note that this revision incorporates several backwards-incompatible changes to the XML representation of schemas. Accordingly, the XML Schema namespace URI has changed, to `http://www.w3.org/2000/10/XMLSchema`.

Although comments from the public and other W3C working groups are always welcome, and we encourage readers to review the draft and to send comments to www-xml-schema-comments@w3.org comments on changes other than those to the concrete syntax may be premature, as there are still some changes pending to the prose of the specifications. An archive of the comments received is available.

Although the Working Group does not anticipate further changes to the functionality described here, this is still a working draft, subject to change. The present version should be implemented only by those interested in providing a check on its design or by those preparing for an implementation of the Candidate Recommendation. *The Schema WG will not allow early implementation to constrain its ability to make changes to this specification prior to final release.*

During the Candidate Recommendation phase, although feedback based on implementation experience is welcome, there are certain aspects of the design presented herein where the Working Group is particularly interested in feedback. These are designated *priority feedback* aspects of the design, and identified as such in editorial notes throughout this draft.

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".

Table of contents

Appendices

---

1 Introduction

This document sets out the structural part (*XML Schema: Structures*) of the XML Schema definition language.

Chapter 2 presents a Conceptual Framework (§2) for XML Schemas, including an introduction to the nature of XML Schemas and a formal specification of the XML Schema abstract data model, along with other terminology used throughout this document.

Chapter 3, Schema Component Details (§3), specifies the precise semantics of each component of the abstract model.

Chapter 4, XML Representation of Schemas and Schema Components (§4), presents the XML representation that maps to the abstract model, in the form of a DTD and XML Schema for an XML Schema document type, along with rules and conventions for identifying the components needed for any particular validation.

Chapter 5 presents Schema Component Validity Constraints (§5), which provide detailed constraints on the internal structure of each component of the abstract model.

Chapter 6 presents Schema Access and Composition (§6), including the connection between documents and schemas, the import and inclusion of declarations and definitions and the foundations of schema-validity.

Chapter 7 discusses Validation Processing of schemas and documents (§7), including the overall approach to schema-validation of documents, and responsibilities of schema-aware processors.

The normative appendices include a(normative) Schema for Schemas (§A) for the transfer syntax, a Glossary

(normative) * (§B) [not yet written] and References (normative) * (§C).

The non-normative appendices include the(non-normative) DTD for Schemas (§E)

This document is primarily intended as a language definition reference. As such, although it contains a few examples, it is *not* designed primarily to serve as a motivating introduction to the design and its features, but rather as a careful and fully explicit definition of that design, suitable for guiding implementations. For those in search of a step-by-step introduction to the design, the non-normative [XML Schema: Primer] is a much better starting point than this document.

1.1 Purpose

The purpose of *XML Schema: Structures* is to define the nature of XML schemas and their component parts, provide an inventory of XML markup constructs with which to represent schemas, and define the application of schemas to XML documents.

The purpose of an *XML Schema: Structures* schema is to define and describe a class of XML documents by using schema components to constrain and document the meaning, usage and relationships of their constituent parts: datatypes, elements and their content and attributes and their values. Schemas may also provide for the specification of additional document information, such as default values for attributes and elements. Schemas have facilities for self-documentation. 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 allows a useful level of constraint checking to be described and validated for a wide spectrum of XML applications. However, 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.

1.2 Dependencies on Other Specifications

The definition of*XML Schema: Structures* depends on the following specifications:[URI], [XML-Infoset], [XML-Namespaces], [XPath], and [XML Schemas: Datatypes]. If the XML Base proposal is adopted before we go to REC, we will need to account for any changes it makes to the Infoset in the areas of QName interpretation and value space and the interpretation of all aspects of schemas involving values identified as being of type `uriReference`, including in particular`xsi:schemaLocation`, `xsi:noNamespaceSchemaLocation` and `targetNamespace`.

1.3 Documentation Conventions and Terminology

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

Special terms are defined at their point of introduction in the text; hyperlinks connect other uses of the term to the definition. For example, a definition ofterm might read: [Definition:] A **term** is something we use a lot. The definition is labeled as such and the term is highlighted typographically. The end of the definition is not specially marked in the displayed or printed text.

Non-normative examples are set off typographically and accompanied by a brief explanatio

Example

```
<schema
        targetNamespace="http://www.muzmo.com/XMLSchema/1.0/mySchema" >
```

And an explanation of the example.

References to properties of information items as defined in [XML-Infoset] are notated as links to the relevant section thereof, set off with square brackets, for example [children].

The definition of each kind of schema component consists of a list of its properties and their contents, followed by descriptions of the semantics of the properties:

Schema Component: Example

{example property}
Definition of the property.

References to properties of schema components are notated as links to the relevant definition as exemplified above, set off with curly braces, for instance {example property}.

The correspondence between an element information item which is part of the XML representation of a schema and one or more schema components is presented in a tableau which illustrates the element information item(s) involved, followed by a tabulation of the correspondence between properties of the component and properties of the information item. Where context may determine which of several different components may arise, several tabulations, one per component, are given. In the XML representation, bold-face attribute names (e.g. **count** below) indicate a required attribute information item, and the rest are optional. Where an attribute information item has an enumerated type definition, the values are shown separated by vertical bars, as for `size` below; if there is a default value, it is shown following a colon. The allowed content of the information item is shown as a grammar fragment, using the Kleene operators `?`, `*` and `+`. The property correspondences are normative, but the illustration of the XML representation element information items is not.

XML Representation Summary: `example` Element Information Item

Example Schema Component	
Property	**Representation**
{example property}	Description of what the property corresponds to, e.g. the value of the `size` [attribute]

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

Issue (dummy): A recorded issue.

Ed. Note: Notes from the editors to themselves or the Working Group, or identification of priority feedback aspects of this draft.

NOTE: General comments directed to all readers.

2 Conceptual Framework

This chapter gives an overview of *XML Schema: Structures* at the level of its abstract data model. (Schema Component Details (§3) provides details on this model, and subsequent chapters define a normative representation in XML for the components of the model.) Readers interested primarily in learning to write schema documents may wish to first read [XML Schema: Primer] and then consult XML Representation of Schemas and Schema Components (§4), using the sections below as a guide to the underlying formal structure of the schema language.

2.1 Overview of XML Schema

An XML Schema consists of components such as type definitions and element declarations. These can be used to assess the validity of well-formed element information items (as defined in [XML-Infoset]), and furthermore may specify augmentations to those items and their descendants. This augmentation makes explicit information which may have been implicit in the original document, such as default values for attributes and elements and the types of element and attribute information items.

The process of schema validation consists of determining whether an element information item satisfies the constraints embodied in the components of an XML Schema, and if so of adding any appropriate augmentations.

2.2 XML Schema Abstract Data Model

This specification builds on [XML] and [XML-Namespaces]. The concepts and definitions used herein regarding XML are framed at the abstract level of information items as defined in [XML-Infoset]. By definition, this use of the infoset provides *a priori* guarantees of well-formedness (as defined in [XML]) and namespace conformance (as defined in [XML-Namespaces]) for all candidates for schema-validity and for all schema documents.

Just as [XML] and [XML-Namespaces] can be described in terms of information items, XML Schemas can be described in terms of an abstract data model. In defining XML Schemas in terms of an abstract data model, this specification rigorously specifies the information which must be available to a conforming XML Schema processor. The abstract model for schemas is conceptual only, and does not mandate any particular implementation or representation of this information. To facilitate interoperation and sharing of schema information, a normative interchange format for schemas is described in XML Representation of Schemas and Schema Components (§4)

> **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 Layer 3: Web-interoperability (§6.3) and Errors in Schema Construction and Structure (§7.1).

[Definition:] **Schema component** is the generic term for the building blocks that comprise the abstract data model of the schema. [Definition:] An **XML Schema** is a set of schema components. There are 12 kinds of component in all, falling into three groups. The primary components are as follows. They may have names, and (except for some element declarations) may be independently accessed:

- Simple type definitions
- Complex type definitions
- Attribute declarations
- Element declarations

The secondary components are as follows. Like the primary components, they may have names and be independently accessed:

- Attribute group definitions
- Identity-constraint definitions
- Model group definitions
- Notation declarations

Finally, the "helper" components provide small parts of other components; they are not independent of their context and cannot be independently accessed:

- Annotations
- Model groups
- Particles
- Wildcards

During validation, [Definition:] **declaration** components are associated by (qualified) name to information items being validated

On the other hand, [Definition:] **definition** components define internal schema components that can be used in other schema components.

[Definition:] Declarations and definitions may have and be identified by **name**s, which are NCNames as defined by [XML-Namespaces].

[Definition:] Several kinds of component have a **target namespace**, which is either absent or a namespace URI, also as defined by [XML-Namespaces]. The target namespace serves to identify the namespace within which the association between the component and its name exists. In the case of declarations, this in turn determines the namespace URI of, for example, the element information items it may validate.

> **NOTE:** At the abstract level, there is no requirement that the components of a schema share a target namespace. Any schema for use in schema-validation of documents containing names from more than one namespace will of necessity include components with different target namespaces. This contrasts with the situation at the level of the XML Representation of Schemas and Schema Components (§4), in which each schema document contributes definitions and declarations to a single target namespace.

Schema-validity, defined in detail in Validation Processing of schemas and documents (§7), is a relation between information items and schema components. For example, an attribute information item may be schema-valid with respect to an attribute declaration, a list of element information items may be schema-valid with respect to a content model, and so on. The following sections briefly introduce the kinds of components in the schema abstract data model, other major features of the abstract model, and how they contribute to the overall definition of schema-validity.

2.2.1 Type Definition Components

The abstract model provides two kinds of type definition component: simple and complex.

[Definition:] This specification uses the phrase **type definition** in cases where no distinction need be made between simple and complex types.

Type definitions form a hierarchy with a single root. First we describe characteristics of that hierarchy, then provide an introduction to simple and complex type definitions themselves.

2.2.1.1 Type Definition Hierarchy

[Definition:] Except for a distinguished ur-type definition, every type definition is, by construction, either a restriction or an extension of some other type definition. The graph of these relationships forms a tree known as the **Type Definition Hierarchy**.

[Definition:] A type definition whose declarations or facets are in a one-to-one relation with those of another specified type definition, with each in turn restricting the possibilities of the one it corresponds to, is said to be a **restriction**. The specific restrictions might include narrowed ranges or reduced alternatives. Members of a type, A, whose definition is a restriction of the definition of another type, B, are always members of type B as well.

[Definition:] A complex type definition which allows element or attribute content in addition to that allowed by another specified type definition is said to be an **extension**.

[Definition:] A distinguished **ur-type definition** is present in each XML Schema, serving as the root of the type definition hierarchy for that schema. The ur-type definition, whose name is **anyType**, has the unique characteristic that it can function as a complex or a simple type definition, according to context. Specifically, restrictions of the ur-type definition can themselves be either simple or complex type definitions.

[Definition:] A type definition used as the basis for an extension or restriction is known as the **base type definition** of that definition.

2.2.1.2 Simple Type Definition

A simple type definition is a set of constraints on strings and information about the values they encode, applicable to the normalized value of an attribute information item or of an element information item with no element children. Informally, it applies to attribute values and text-only content of elements.

Each simple type definition, whether built-in (that is, defined in [XML Schemas: Datatypes]) or user-defined, is a restriction of some particular simple base type definition. For the built-in primitive types, this is the simple version of the ur-type definition, whose name is **anySimpleType**, which is in turn understood to be a restriction of the ur-type definition. Simple types may also be defined whose members are lists of items themselves constrained by some other simple type definition, or whose membership is the union of the memberships of some other simple type definitions. List and union simple type definitions are also understood as restrictions of the simple ur-type definition.

For details on the composition and schema-validation contributions of simple type definitions, see (non-normative) Simple Type Definition Details (§3.13) and [XML Schemas: Datatypes]. The latter also defines an extensive inventory of pre-defined simple types. See (non-normative) XML Representation of Simple Type Definition Schema Components (§4.3.11) for the XML representation of simple type definitions, and Simple Type Definition Constraints (§5.12) for constraints on simple type definition components as such.

2.2.1.3 Complex Type Definition

A complex type definition is a set of attribute declarations and a content type, applicable to the [attributes] and [children] of an element information item respectively. The content type may require the [children] to contain neither element nor character information items, to be a string which is schema-valid with respect to particular simple type or to contain a sequence of element information items which is schema-valid with respect to a particular model group, with or without character information items as well.

Each complex type definition is either

- A restriction of a complex base type definition

or

- an extension of a simple or complex base type definition

or

- A restriction of the ur-type definition.

A complex type which extends another does so by having additional content model particles at the end of the other definition's content model, or by having additional attribute declarations, or both.

> **NOTE:** This specification allows only appending, and not other kinds of extensions. This decision simplifies application processing required to cast instances from derived to base type. Future versions may allow more kinds of extension, requiring more complex transformations to effect casting.

See Complex Type Definition Details (§3.4) for the composition and schema-validation contributions of complex type definition schema components, XML Representation of Complex Type Definition Schema Components (§4.3.3) for the XML representation of complex type definitions and Complex Type Definition Constraints (§5.11) for constraints on complex type definition components as such.

2.2.2 Declaration Components

There are three kinds of declaration component: element, attribute, and notation. Each described in a section below. Also included is a discussion of element substitution groups, which is a feature provided in conjunction with element declarations.

2.2.2.1 Element Declaration

An element declaration is an association of a name with a type definition, either simple or complex, an (optional) default value and a set of identity-constraint definitions. The association is either global or scoped to a containing complex type definition. A global element declaration with name 'A' is broadly comparable to a pair of DTD declarations as follows, where the associated type definition fills in the ellipsis:

```
<!ELEMENT A . . .>
<!ATTLIST A . . .>
```

Element declarations contribute to schema-validity as part of model group validation, when their defaults and type components are checked against an element information item with a matching name and namespace, and by triggering identity-constraint definition validation.

See Element Declaration Details (§3.3) for the composition and schema-validation contributions of element declaration schema components, XML Representation of Element Declaration Schema Components (§4.3.2) for the XML representation of element declarations and Element Declaration Constraints (§5.2) for constraints on element declaration components as such.

2.2.2.2 Element Substitution Group

In XML 1.0, the name and content of an element must correspond exactly to the element type referenced in the corresponding content model.

[Definition:] Through the new mechanism of **element substitution groups**, XML Schemas provides a more powerful model supporting substitution of one named element for another. Any global element declaration can serve as the defining element, or head, for an element substitution group. Other global element declarations, regardless of target namespace, can be designated as members of the substitution group headed by this element. In a suitably enabled content model, a reference to the head validates not just the head itself, but elements corresponding to any member of the substitution group as well.

All such members must have type definitions which are either the same as the head's type definition or restrictions or extensions of it. Therefore, although the names of elements can vary widely as new namespaces and members of the substitution group are defined, the content of member elements is strictly limited according to the type definition of the substitution group head.

Note that element substitution groups are not represented as separate components. They are specified in the property values for element declarations (see Element Declaration (§2.2.2.1)).

2.2.2.3 Attribute Declaration

An attribute declaration is an association between a name and a simple type definition, together with occurrence information and (optionally) a default value. The association is either global, or local to its containing complex type definition. Attribute declarations contribute to schema-validity as part of complex type definition validation, when their occurrence, defaults and type components are checked against an attribute information item with a matching name and namespace.

See Attribute Declaration Details (§3.2) for the composition and schema validation contributions of attribute declaration schema components, XML Representation of Attribute Declaration Schema Components (§4.3.1) for the XML representation of attribute declarations and Attribute Declaration Constraints (§5.1) for constraints on attribute declaration components as such.

2.2.2.4 Notation Declaration

A notation declaration is an association between a name and an identifier for a notation. For an attribute information item to be schema-valid with respect to a `NOTATION` simple type definition, its value must have been declared with a notation declaration.

See Notation Declaration Details (§3.11) for the composition and schema validation contributions of notation declaration schema components, XML Representation of Notation Declaration Schema Components (§4.3.9) for the XML representation of notation declarations and Notation Declaration Constraints (§5.8) for constraints on notation declaration components as such.

2.2.3 Model Group Components

The model group, particle, and wildcard components contribute to the portion of a complex type definition that controls an element information item's content type.

2.2.3.1 Model Group

A model group is a constraint in the form of a grammar fragment that applies to lists of element information items. It consists of a list of particles, i.e. element declarations, wildcards and model groups. There are three varieties of model group:

- Sequence (the element information items match the particles in sequential order)
- Conjunction (the element information items match the particles, in any order)

- Disjunction (the element information items match one of the particles)

See Model Group Details (§3.7) for the composition and schema-validation contributions of model group schema components, Complex Type Definition Details (§3.4)for the use of model groups as content models, XML Representation of Model Group Schema Components (§4.3.6) for the XML representation of model groups and Model Group Constraints (§5.7) for constraints on model group components as such.

2.2.3.2 Particle

A particle is a term in the grammar for element content, consisting of either an element declaration, a wildcard or a model group, together with occurrence constraints. Particles contribute to schema-validity as part of complex type validation, when they allow anywhere from zero to many element information items or sequences thereof, depending on their contents and occurrence constraints.

[Definition:] A particle can be used in a complex type definition to express a validity constraint on the[children] of an element information item; such a particle is called a**content model**.

> **NOTE:** *XML Schema: Structures* content models are similar to but more expressive than[XML] content models; unlike[XML], *XML Schema: Structures* applies content models to the validation of both mixed and element-only content.

See Particle Details (§3.8) for the composition and schema-validation contributions of particle schema components, XML Representation of Model Group Schema Components (§4.3.6) for the XML representation of particles and Particle Constraints (§5.10) for constraints on particle components as such.

2.2.3.3 Wildcard

A wildcard is a special kind of particle which matches element and attribute information items dependent on their namespace URI, independently of their local names.

See Wildcard Details (§3.9) for the composition and schema-validation contributions of wildcard schema components, XML Representation of Wildcard Schema Components (§4.3.7) for the XML representation of wildcards and Wildcard Constraints (§5.5) for constraints on wildcard components as such.

2.2.4 Identity-constraint Definition Components

A identity-constraint definition is an association between a name and one of several varieties of identity-constraint related to uniqueness and reference. All the varieties use[XPath] expressions to pick out sets of information items relative to particular target element information items which are unique, or a key, or a valid reference, within a specified scope. An element information item is only schema-valid with respect to an element declaration with identity-constraint definitions if those definitions are all satisfied for all the descendants of that element information item which they pick out.

See Identity-constraint Definition Details (§3.10)for the composition and schema-validation contributions of identity-constraint definition schema components,XML Representation of Identity-constraint Definition Schema Components (§4.3.8) for the XML representation of identity-constraint definitions andIdentity-constraint Definition Constraints (§5.3) for constraints on identity-constraint definition components as such.

2.2.5 Group Definition Components

There are two kinds of convenience definitions available for use in reusing pieces of complex type definitions: model group definitions and attribute group definitions.

2.2.5.1 Model Group Definition

A model group definition is an association between a name and a model group, for use in reusing the same model group in several complex type definitions.

See Model Group Definition Details (§3.6) for the composition and schema validation contributions of model group definition schema components, XML Representation of Model Group Definition Schema Components (§4.3.5) for the XML representation of model group definitions and Model Group Definition Constraints (§5.6) for constraints on model group definition components as such.

2.2.5.2 Attribute Group Definition

An attribute group definition is an association between a name and a set of attribute declarations, for use in reusing the same set in several complex type definitions.

See Attribute Group Definition Details (§3.5) for the composition and schema-validation contributions of attribute group definition schema components, XML Representation of Attribute Group Definition Schema Components (§4.3.4) for the XML representation of attribute group definitions and Attribute Group Definition Constraints (§5.4) for constraints on attribute group definition components as such.

2.2.6 Annotation Components

An annotation is information for human and/or mechanical consumers. The interpretation of such information is not defined in this specification.

See Annotation Details (§3.12) for the composition and schema-validation contributions of annotation schema components, XML Representation of Annotation Schema Components (§4.3.10) for the XML representation of annotations and Annotation Constraints (§5.9) for constraints on annotation components as such.

2.3 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.

The preceding section focussed on schema-validity, that is the constraints on information items which schema components supply. In fact however this specification provides four different kinds of normative statements about schema components, their representations in XML and their contribution to the schema-validation of information items:

[Definition:] Constraint on Schemas
Constraints on the schema components themselves, i.e. conditions components must satisfy to be components at all. Largely to be found in Schema Component Validity Constraints (§5).

[Definition:] Schema Representation Constraint
Constraints on the representation of schema components in XML. Some but not all of these are expressed in (non-normative) DTD for Schemas (§E) and (normative) Schema for Schemas (§A). Largely to be found in XML Representation of Schemas and Schema Components (§4).

[Definition:] Validity Contribution
Constraints expressed by schema components which information items must satisfy to be schema-valid. Largely to be found in Schema Component Details (§3).

The definition of the above constraints sometimes involves many clauses, some as alternatives, some as joint requirements. The presentations below number all clauses: clauses at the same level are either clearly identified as alternatives with words such as *either* and *or*, or should be understood as joint.

[Definition:] Schema Information Set Contribution
Augmentations to post-schema-validation information sets expressed by schema components, which follow as a consequence of schema-validation. Largely to be found in Schema Component Details (§3).

Schema information set contributions are not new. XML 1.0 validation augments the XML 1.0 information set in similar ways, for example by providing values for attributes not present in instances, and by implicitly exploiting type information for normalization or access. (As an example of the latter case, consider the effect of `NMTOKENS` on attribute whitespace, and the semantics of `ID` and `IDREF`.) By including schema information set contributions, this specification makes explicit some features that XML 1.0 left implicit.

2.4 Conformance

This specification describes three levels of conformance for schema aware processors. The first is required of all processors. Support for the other two will depend on the application environments for which the processor is intended.

[Definition:] **Minimally conforming** processors must completely and correctly implement the Constraints on Schemas, Validity Contributions, and Schema Information Set Contributions contained in this specification.

[Definition:] Processors which accept schemas in the form of XML documents as described in XML Representation of Schemas and Schema Components (§4) are additionally said to provide **conformance to the XML Representation of Schemas**. Such processors must, when processing schema documents, completely and correctly implement all Schema Representation Constraints in this specification, and must adhere exactly to the specifications in XML Representation of Schemas and Schema Components (§4) for mapping the contents of such documents to schema components for use in validation.

> **NOTE:** By separating the conformance requirements relating to the concrete syntax of XML schema documents, this specification admits processors which validate using schemas stored in optimised binary representations, dynamically created schemas represented as programming language data structures, or implementations in which particular schemas are compiled into executable code such as C or Java. Such processors can be said to be minimally conforming but not necessarily in conformance to the XML Representation of Schemas.

[Definition:] **Fully conforming** processors are network-enabled processors which support both levels of conformance described above, and which must additionally be capable of accessing schema documents from the World Wide Web according to Representation of Schemas on the World Wide Web (§2.7) and How schema definitions are located on the Web (§6.3.2). .

> **NOTE:** Although this specification provides just these three standard levels of conformance, it is anticipated that other conventions can be established in the future. For example, the World Wide Web Consortium is considering conventions for packaging on the Web a variety of resources relating to individual documents and namespaces. Should such developments lead to new conventions for representing schemas, or for accessing them on the Web, new levels of conformance can be established and named at that time. There is no need to modify or republish this recommendation to define such additional levels of conformance.

See Schema Access and Composition (§6) for a more detailed explanation of the mechanisms supporting these

levels of conformance.

2.5 Names and Symbol Spaces

As discussed in XML Schema Abstract Data Model (§2.2), most schema components (may) have names. If all such names were assigned from the same "pool", then it would be impossible to have, for example, a simple type definition and an element declaration both with the name "title" in a given target namespace.

This specification therefore introduces the term [Definition:] **symbol space** to denote a collection of names, each of which is unique with respect to the others. A symbol space is similar to the non-normative concept of namespace partition introduced in [XML-Namespaces]. There is a single distinct symbol space within a given target namespace for each kind of definition and declaration component identified in XML Schema Abstract Data Model (§2.2), except that within a target namespace, simple type definitions and complex type definitions share a symbol space. Within a given symbol space, names are unique, but the same name may appear in more than one symbol space without conflict. For example, the same name can appear in both a type definition and an element declaration, without conflict or necessary relation between the two.

Locally scoped attribute and element declarations are special with regard to symbol spaces. Every complex type definition defines its own local attribute and element declaration symbol spaces, where these symbol spaces are distinct from each other and from any of the other symbol spaces. So, for example, two complex type definitions having the same target namespace can contain a local attribute declaration for the unqualified name "priority", or contain a local element declaration for the name "address", without conflict or necessary relation between the two.

2.6 Schema-Related Markup in Documents Being Schema-Validated

The XML representation of schema components uses a vocabulary identified by the namespace URI `http://www.w3.org/2000/10/XMLSchema`. *XML Schema: Structures* also defines several attributes for direct use in XML documents. These attributes are in a different namespace, which has the namespace URI `http://www.w3.org/2000/10/XMLSchema-instance`. For brevity, the text and examples in this specification use the prefix `xsi:` to stand for this latter namespace; in practice, any prefix can be used.

2.6.1 xsi:type

The Simple Type Definition (§2.2.1.2) or Complex Type Definition (§2.2.1.3) used to validate an element is usually determined by reference to the appropriate schema components. However, when permitted by those components, an element can explicitly assert its type using the attribute `xsi:type`. The value of this attribute is a QName; see QName Interpretation (§4.2) for the means by which the QName is associated with a type definition.

2.6.2 xsi:null

XML Schema: Structures introduces a mechanism for signalling that an element's content is missing, or "null" in the terminology of databases. An element has null content if it has the attribute `xsi:null` with the value `true`. An element so labelled must be empty, but can carry attributes if permitted by the corresponding complex type.

2.6.3 xsi:schemaLocation, xsi:noNamespaceSchemaLocation

The `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` attributes can be used in a document to provide hints as to the physical location of schema documents which may be used for validation. See How schema definitions are located on the Web (§6.3.2) for details on the use of these attributes.

2.7 Representation of Schemas on the World Wide Web

On the World Wide Web, schemas are conventionally represented as documents of MIME type "text/xml", conforming to the specifications in XML Representation of Schemas and Schema Components (§4). For more information on the representation and use of schema documents on the World Wide Web see Standards for representation of schemas and retrieval of schema documents on the Web (§6.3.1) and How schema definitions are located on the Web (§6.3.2).

3 Schema Component Details

The following sections provide full details on the properties and significance of the schema itself and each kind of schema component. For each property, its range, that is the kinds of values it may have, is defined. This can be understood as defining a schema as a labelled directed graph, where the root is a schema, and every other vertex is a schema component or a literal (string, boolean, number) and every labelled edge a property. The graph is *not* acyclic: multiple copies of components with the same name in the same symbol space may not exist, so in some cases re-entrant chains of properties must exist. Equality of components for the purposes of this specification is always addressed at the level of names (including target namespaces) within symbol spaces. Any property not identified as optional is required to be present, optional properties which are absent are taken to have absent as their value. Any property identified as a having a set, subset or list value may have an empty value unless this is explicitly ruled out: this is *not* the same as absent. Any property value identified as superset or subset of some set may be equal to that set, unless a proper superset or subset is explicitly called for. By 'string' in Part 1 of this specification is meant a sequence of ISO 10646 character codes identified as legal XML character codes in [XML].

> **NOTE:** Readers whose primary interest is in the XML representation of schemas may wish to skip this chapter on the first reading, concentrating on XML Representation of Schemas and Schema Components (§4) and [XML Schema: Primer].

Throughout this specification, [Definition:] when we refer to the **initial value** of some attribute information item, we mean by this the value of the normalized value property of that item. Similarly, when we refer to the **initial value** of an element information item, we mean the string composed of, in order, the [character code] of each character information item in the [children] of that element information item

[Definition:] When we refer to the **normalized value** of an element or attribute information item, we mean an initial value whose whitespace, if any, has been normalized according to the value of the whitespace facet of the simple type definition by which its validity is assessed:

preserve
: No normalization is done, the value is the normalized value

replace
: All occurrences of `	` (tab), `
` (linefeed) and `
` (carriage return) are replaced with ` ` (space).

collapse
: Subsequent to the replacements specified above under **replace**, contiguous sequences of ` `s are collapsed to a single ` `, and initial and/or final `&x20;`s are deleted.

These three levels of normalization correspond to the processing mandated in XML 1.0 for element content, CDATA attribute content and tokenized attributed content, respectively. See Attribute Value Normalization in [XML] for the precedent for **replace** and **collapse** for attributes. Extending this processing to element content is necessary to ensure a consistent schema validation semantics for simple types, regardless of whether they are applied to attributes or elements. Performing it twice in the case of attributes whose normalized value has already

been subject to replacement or collapse on the basis of information in a DTD is necessary to ensure consistent treatment of attributes regardless of the extent to which DTD-based information has been made use of during infoset construction.

> **NOTE:** Even when DTD-based information *has* been appealed to, and Attribute Value Normalization has taken place, the above definition of normalized value may mean *further* normalization may take place, as for instance when character entity references in attribute values result in whitespace characters other than spaces in their initial values.

Many properties are identified below as having (sets of) other schema components as values. For the purposes of exposition, the definitions in this section assume that (unless the property is explicitly identified as optional) all such values are in fact present. When schema components are constructed from XML representations involving reference by name to other components, this assumption may be violated if one or more references cannot be resolved. This specification addresses the matter of missing components in a uniform manner, described in Missing Sub-components (§7.3): no mention of handling missing components will be found in the individual component descriptions below.

As the above makes clear, at the level of schema components and schema validation, reference to components by name is normally not involved. In a few cases, however, qualified names appearing in information items being validated must be resolved to schema components by such lookup. The following constraint is appealed to in these cases.

Validation Contribution: QName resolution (Instance)
A pair of a local name and a namespace URI (or absent) resolve to a schema component of a specified kind in the context of schema validation if the component is that member of the value of the appropriate property of the schema being used for the validation, that is:

1.1 the {type definitions} if the kind specified is simple or complex type definition;

1.2 the {attribute declarations} if the kind specified is attribute declaration;

1.3 the {element declarations} if the kind specified is element declaration;

1.4 the {attribute group definitions} if the kind specified is attribute group;

1.5 the {model group definitions} if the kind specified is model group;

1.6 the {notation declarations} if the kind specified is notation declaration;

whose **{local name}** matches the local name and whose **{target namespace}** is identical to the namespace URI of the pair.

> **NOTE:** A schema and its components as defined in this chapter are an idealisation of the information a schema-aware processor requires: implementations are not constrained in how they provide it. In particular, no implications about literal embedding versus indirection follow from the use above of language such as "properties . . . having . . . components as values".

3.1 Schema details

At the abstract level, the schema itself is just a container for its components.

Schema Component: Schema
{type definitions} A set of named simple and complex type definitions {attribute declarations} A set of named global attribute declarations {element declarations} A set of named global element declarations {attribute group definitions} A set of named attribute group definitions {model group definitions} A set of named model group definitions {notation declarations} A set of notation declarations {annotations} A set of annotations

See XML Representations of Schemas (§4.1) for the XML representation of schemas and Schema Constraints (§5.13) for constraints on schemas as such.

Schema Information Set Contribution: Schema Information
In the post-schema validation infoset a **[schema information]** property is added to the element information item at which validity assessment began. Its value is a list of **namespace schema information** information items, one for each namespace URI which appears as the **{target namespace}** of any schema component in the schema used for that assessment. Each **namespace schema information** information item has the following properties and values:

[schema namespace]
a namespace URI
[schema components]
a (possibly empty) list of **schema component** information items, each one isomorphic to a component whose **{target namespace}** is the sibling **[schema namespace]** property above, drawn from the schema used for assessment.
[schema documents]
a (possibly empty) list of **schema document** information items, with properties and values as follows:
[document location]
either a URI reference, if available, otherwise absent
[document]
a **document information item**, if available, otherwise absent
for a schema document which contributed components to the schema, and whose `targetNamespace` matches the sibling **[namespace URI]** property above (or was absent but contributed components to that namespace by being included by a schema document with that `targetNamespace` as per Assembling a schema for a single target namespace from multiple schema definition documents (§6.2.1)).

The **[schema components]** property is provided for processors which wish to provide a single access point to some or all of the components of the schema used during validation. Lightweight processors are free to leave it empty.

3.2 Attribute Declaration Details

Attribute declarations provide for:

- Constraining attribute information item values by a simple type definition;
- Providing default or fixed values for an attribute information item.

The attribute declaration schema component has the following properties:

Schema Component: Attribute Declaration
{name} An NCName as defined by [XML-Namespaces]. {target namespace} Either absent or a namespace URI, as defined in [XML-Namespaces]. {simple type definition} A simple type definition. {scope} Optional. Either *global* or a complex type definition. {value constraint} Optional. A pair consisting of a string and, optionally, one of *default*, *fixed*. {annotation} Optional. An annotation

The {name} property must match the local part of the names of attributes being validated.

A {scope} of *global* identifies attribute declarations available for use in complex type definitions throughout the schema. Locally scoped declarations are available for use only within the complex type definition identified by the {scope} property. This property is also absent in the case of non-*global* declarations within attribute group definitions: their scope will be determined when they are used in the construction of complex type definitions.

A non-absent value of the {target namespace} property provides for validation of namespace-qualified attribute information items (which must be explicitly prefixed in the character-level form of XML documents). absent values of {target namespace} validate unqualified (unprefixed) items.

The value of the attribute must conform to the supplied {simple type definition}.

{value constraint} reproduces the functions of XML 1.0 default and #FIXED attribute values. *fixed* indicates that the attribute value must match the supplied constraint string; *default* specifies that the attribute is to appear unconditionally in the post-schema-validation information set, with the supplied value used whenever the attribute is not actually present.

See Annotation Details (§3.12) for the significance of the {annotation} property.

> **NOTE:** A more complete and formal presentation of the semantics of {name}, {target namespace} and *default* {value constraint} is provided in conjunction with other aspects of complex type validation (see Element Children and Attributes Valid (§3.4))

[XML-Infoset] distinguishes namespace declarations such as `xmlns` or `xmlns:xsl` from attributes. Accordingly, it is unnecessary and in fact not possible for schemas to contain attribute declarations corresponding to such namespace declarations, see xmlns Not Allowed (§5.1). No means is provided in this specification to supply a default value for a namespace declaration.

See XML Representation of Attribute Declaration Schema Components (§4.3.1) for the XML representation of

attribute declarations and Attribute Declaration Constraints (§5.1) for constraints on attribute declaration components as such.

Validation Contribution: Attribute Valid
For an attribute information item to be schema-valid with respect to an attribute declaration, itsnormalized value must

- be schema-valid with respect to the {simple type definition}as per String Valid (§3.13);
- match the string of the {value constraint} if it is present and*fixed*.

Validation Contribution: Attribute Valid (Lax)
An attribute information item is laxly schema-valid if either

1.1.1 The [namespace URI] is not absent and the [local name] and [namespace URI] resolve to an attribute declaration, as defined byQName resolution (Instance) (§3);

1.1.2 The item is schema-valid with respect to that declaration, as defined byAttribute Valid (§3.2)

or

1.2 The [namespace URI] is absent, or the [local name] and [namespace URI] do *not* resolve to an attribute declaration, as defined byQName resolution (Instance) (§3);

Schema Information Set Contribution: Attribute Validated by Type
If an attribute information item is schema-valid with respect to an attribute declaration, in the post-schema validation infoset the attribute information item has

1.1 a **[schema normalized value]** property, whose value is the normalized valueof the item as validated;

and either

1.2.1 a single **[type definition]** property, containing an information item isomorphic to the attribute declaration's {simple type definition}component itself, that is, a **Simple Type Definition** information item with one property per property of the component, with the same name, and value either the same atomic value, or an information item corresponding in the same way to its component value, recursively, as necessary.

1.2.2 if the **[type definition]** has {variety} *union*, then additionally there is a**[member type definition]** property, containing an information item isomorphic to that member of the{member type definitions} which actually validated the attribute item's**[normalized value]**.

or

1.3.1 four properties as described in Element Validated by Type (§3.3), except that the {simple type definition} is used wherever the actual type definitionis called for therein.

1.3.2 if the **[type definition]** has {variety} *union*, then there are three additional properties as described in the parallel case for Element Validated by Type (§3.3), where the actual member type definitionis that member of the {member type definitions}which actually validated the attribute item's**[normalized value]**.

See below under Element Validated by Type (§3.3) for a discussion of the alternatives given above.

Also, if the declaration has a {value constraint}, the item's **[schema default]** is set to the declaration's {value constraint} string.

Finally, if an attribute is laxly but not strictly valid, that isAttribute Valid (§3.2) does not hold but Attribute Valid (Lax) (§3.2) does, the information described above under 1.2.1 or 1.3.1 above is provided with respect to the simple ur-type definition.

Schema Information Set Contribution: Validation Outcome (Attribute)

If an attribute information item's schema-validity as defined by Attribute Valid (§3.2) has been assessed, whether successfully or not, then in the post-schema validation infoset the item has a **[validation attempted]** property with the value *full*.

If an attribute information item's schema-validity as defined by Attribute Valid (§3.2) has not been assessed, but its lax schema-validity as defined by Attribute Valid (Lax) (§3.2) has been assessed, in the post-schema validation infoset the item has a **[validation attempted]** property with the value *partial*.

If an attribute information item's schema-validity, as defined by either Attribute Valid (§3.2) or Attribute Valid (Lax) (§3.2), has been assessed, whether successfully or not, then in the post-schema validation infoset the item has a **[validation context]** property whose value is the lowest containing element information item with a **[schema information]** property.

3.3 Element Declaration Details

Element declarations provide for:

- Establishing the validity of element information items.
- Determining schema information set contributions, such as default values.
- Establishing uniquenesses and reference constraint relationships among the values of related elements and attributes.
- Controlling the substitutability of elements through the mechanism of element substitution groups.

The element declaration schema component has the following properties:

Schema Component: Element Declaration
{name} An NCName as defined by [XML-Namespaces]. {target namespace} Either absent or a namespace URI, as defined in [XML-Namespaces]. {scope} Optional. Either *global* or a complex type definition. {type definition} Either a simple type definition or a complex type definition. {nullable} A boolean {value constraint} Optional. A pair consisting of a string and one of *default*, *fixed*. {identity-constraint definitions} A set of constraint definitions. {substitution group affiliation} Optional. A global element definition. {substitution group exclusions} A subset of {*extension*, *restriction*}. {disallowed substitutions} A subset of {*substitutionGroup*, *extension*, *restriction*}. {abstract} A boolean {annotation} Optional. An annotation

The {name} property must match the local part of the names of element information items being validated.

A {scope} of *global* identifies element declarations available for use in content models throughout the schema. Locally scoped declarations are available for use only within the complex type identified by the {scope} property. This property is absent in the case of non-*global* declarations within named model groups: their scope will be determined when they are used in the construction of complex type definitions.

A non-absent value of the {target namespace} property provides for validation of namespace-qualified element information items. absent values of {target namespace} validate unqualified items.

An element information item is schema-valid if it obeys the schema validity constraints of the {type definition}. For such an item, the schema information set contributions from the {type definition} are applied to the corresponding element information item in the post-schema-validation information set.

If {nullable} is *true*, then an element is also schema-valid if it carries the namespace qualified attribute with [local name] `null` from namespace `http://www.w3.org/2000/10/XMLSchema-instance` and value `true` (see xsi:null (§2.6.2)) even if it has no text or element content despite a {content type} which would otherwise require content. Formal details of element validation are described in Element Valid (Explicit) (§3.3).

{value constraint} establishes a default or fixed value for an element. If *default* is specified, and if the element being validated is empty, then the supplied constraint string becomes the **[schema normalized value]** of the validated element in the post-schema-validation infoset. If *fixed* is specified, then the element's content must either be empty, in which case *fixed* behaves as *default*, or it must match the supplied constraint string.

{identity-constraint definitions} express constraints establishing uniquenesses and reference relationships among the values of related elements and attributes. See Identity-constraint Definition Details (§3.10).

Element declarations are members of the substitution group, if any, identified by {substitution group affiliation}. Membership is transitive but not symmetric; an element declaration is implicitly a member of any group of which its {substitution group affiliation} is a member.

An empty {substitution group exclusions} allows a declaration to be nominated as the {substitution group affiliation} of other element declarations having the same {type definition} or types derived therefrom. The explicit values of {substitution group exclusions} rule out element declarations having types which are *extension*s or *restriction*s respectively of {type definition}. If both values are specified, then the declaration may not be nominated as the {substitution group affiliation} of any other declaration.

The supplied values for {disallowed substitutions} determine whether an element declaration appearing in a content model will be prevented from additionally validating elements (a) with an xsi:type (§2.6.1) that identifies an *extension* or *restriction* of the type of the declared element, and/or (b) from validating elements which are in the same substitution group as the declared element. If {disallowed substitutions} is empty, then all derived types and substitution group members are valid.

Element declarations for which {abstract} is *true* can appear in content models only when substitution is allowed; such declarations may not themselves ever be used to validate element content.

See XML Representation of Element Declaration Schema Components (§4.3.2) for the XML representation of element declarations and Element Declaration Constraints (§5.2) for constraints on element declaration components as such.

Validation Contribution: Element Valid (Explicit)

An element information item is schema-valid with respect to an element declaration if

1.1 If {nullable} is false there is no attribute information item among the element information item's [attributes] whose [namespace URI] is identical to `http://www.w3.org/2000/10/XMLSchema-instance` and whose [local name] is `null`;

1.2 If {nullable} is true and there is such an attribute information item and its normalized value is `true`, then

1.2.1 the element information item must have no character or element information item [children];

1.2.2 there is no *fixed* {value constraint}.

If there is an attribute information item among the element information item's [attributes] whose [namespace URI] is identical to `http://www.w3.org/2000/10/XMLSchema-instance` and whose [local name] is `type`, then

2.1 The normalized value of that attribute information item is schema-valid with respect to the built-in QName simple type, as defined by String Valid (§3.13);

2.2 The local name and namespace URI (as defined in QName Interpretation (§4.2)), of the normalized value of that attribute information item resolve to a type definition, as defined in QName resolution (Instance) (§3) -- [Definition:] call this type definition the **item type definition**;

2.3 The item type definition is validly derived from the {type definition} given the {disallowed substitutions}, as defined in Type Derivation OK (Complex) (§5.11) (if it is a complex type definition), or given *{list}*, as defined in Type Derivation OK (Simple) (§5.12) (if it is a simple type definition).

[Definition:] We refer below to the **actual type definition**. If the above three clauses obtain, this should be understood as referring to the local type definition, otherwise to the {type definition}.

If the declaration has a {value constraint}, then provided clause 1.2 has not obtained

3.1 If the element information item has no character information item [children] and the actual type definition is a local type definition, the {value constraint} string is schema-valid with respect to the actual type definition as defined by String Valid (§3.13) (if the actual type definition is a simple type definition) or else by its {content type} (if that is a simple type definition) or else (the actual type definition is a complex type definition whose {content type} is not a simple type definition) the string must be a valid default for the actual type definition as defined in Element Default Valid (Immediate) (§5.2);

3.2 If the {value constraint} is *fixed*, the element information item must have no element information item [children], and the string composed of the element information item's character information item [children] in order must be either empty or match the string of the {value constraint};

Otherwise (the element information item has character information item [children] or there is no {value constraint}) if the actual type definition is a simple type definition, then

4.1.1 The element information item's [attributes] must be empty, excepting those whose [namespace URI] is identical to `http://www.w3.org/2000/10/XMLSchema-instance` and whose [local name] is one of `type`, `null`, `schemaLocation` or `noNamespaceSchemaLocation`;

4.1.2 The element information item must have no element information item [children];

4.1.3 the string composed of the [character code] of each of the element information item's character information item [children] in order must be schema-valid with respect to the actual type definition as defined by String Valid (§3.13)

otherwise (the actual type definition is a complex type definition)

4.2.1 The element information item must be schema-valid with respect to the actual type definition as per Element Children and Attributes Valid (§3.4);

4.2.2 The element information item must be schema-valid with respect to each of the {identity-constraint definitions} as per Identity-constraint Satisfied (§3.10).

Ed. Note: Priority Feedback Request
The Working Group solicits feedback from implementors and users on the extent to which the xsi:null feature provides useful functionality and satisfactorily addresses requirements in the area of data interchange.

NOTE: The {name} and {target namespace} properties are not mentioned above because they are checked during particle validation, as per Element Sequence Valid (Particle) (§3.8).

Validation Contribution: Element Valid (Strict)
An element information item is strictly schema-valid if

1.1 The [local name] and [namespace URI] resolve to an element declaration, as defined by QName resolution (Instance) (§3);

1.2 The item is schema-valid with respect to that declaration, as defined by Element Valid (Explicit) (§3.3)

Validation Contribution: Element Valid (Lax)
An element information item is laxly schema-valid if either

1.1 The item is strictly schema-valid as defined by Element Valid (Strict) (§3.3)

or

1.2.1 The [local name] and [namespace URI] does *not* resolve to an element declaration, as defined by QName resolution (Instance) (§3);

1.2.2 All the element information item [children] and [attributes] of the item are laxly schema-valid, as defined by this constraint or Attribute Valid (Lax) (§3.2), respectively.

Schema Information Set Contribution: Element Default Value
If an element information item is schema-valid with respect to an element declaration, the {value constraint} is present, clause 1.2 of Element Valid (Explicit) (§3.3) above does not obtain and the element information item has no character or element information item [children], the post-schema validation infoset the {value constraint}'s string as the item's **[schema normalized value]** property and its [specified] is set to *schema*. Otherwise, the item's [specified] is set to *instance*.

Schema Information Set Contribution: Element Validated by Type
If an element information item is schema-valid with respect to a type definition declaration, in the post-schema validation infoset the item has

1.1 a **[schema normalized value]** property, whose value is the normalized value of the item as validated (unless Element Default Value (§3.3) above has obtained);

and either

1.2.1 a single **[type definition]** property, containing an information item isomorphic to the type definition component itself, that is, a **Complex Type Definition** information item with one property per property of the component, with the same name, and value either the same atomic value, or an information item corresponding in the same way to its component value, recursively, as necessary.

1.2.2 if the type definition has a simple type definition {content type}, and that type definition has {variety} *union*, then additionally there is a **[member type definition]** property, containing an information item isomorphic to that member of the {member type definitions} which actually validated the element item's character information item content.

or

1.3.1 four properties as follows:

[type definition type]
simple or *complex*, depending on the actual type definition
[type definition namespace]
the **{target namespace}** of the actual type definition
[type definition anonymous]
true if the **{name}** of the actual type definition is absent, otherwise *false*
[type definition name]
the **{name}** of the actual type definition, if it is not absent. If it is absent, schema processors may, but need not, provide a value unique to the {type definition} of the declaration.

1.3.2 if the type definition has a simple type definition {content type}, and that type definition has {variety} *union*, then calling [Definition:] that member of the {member type definitions} which actually validated the element item's character information item content the **actual member type definition**, there are three additional properties:

[member type definition namespace]
the **{target namespace}** of the actual member type definition
[member type definition anonymous]
true if the **{name}** of the actual member type definition is absent, otherwise *false*
[member type definition name]
the **{name}** of the actual member type definition, if it is not absent. If it is absent, schema processors may, but need not, provide a value unique to the {type definition} of the declaration.

The first alternative above is provided for applications such as query processors which need access to the full range of details about how an item was validated, for example the type hierarchy; the second, for lighter-weight processors for whom representing the significant parts of the type hierarchy as information items might be a significant burden.

Also, if the declaration has a {value constraint}, the item's [schema default] property is set to that {value constraint}'s string.

Finally, if an element is laxly but not strictly valid, that is Element Valid (Explicit) (§3.3) and/or Element Valid (Strict) (§3.3) do not hold but Element Valid (Lax) (§3.3) does, the information described above under 1.2.1 or 1.3.1 above is provided with respect to the ur-type definition.

Schema Information Set Contribution: Element Declaration

If an element information item is schema-valid with respect to an element declaration then in the post-schema validation infoset the element information item has a either

1 a single **[element declaration]** property, containing an information item isomorphic to the declaration component itself, that is, an **Element Declaration** item with one property per property of the component, with the same name, and value either the same atomic value, or an information item corresponding in the same way to its component value, recursively, as necessary.

2 a **[null]** property, with value *true* if clause 1.2 of Element Valid (Explicit) (§3.3) above obtains, otherwise *false*.

Schema Information Set Contribution: Validation Outcome (Element)

If

1.1 an element information item's schema-validity as defined by Element Valid (Explicit) (§3.3) has been assessed, whether successfully or not;

1.2 all its element information item children have the value *full* for their **[validation attempted]** property,

then in the post-schema validation infoset the item has a **[validation attempted]** property with the value *full*.

If an element information item's schema-validity as defined by Element Valid (Explicit) (§3.3) has not been assessed, or has been but the above clause is not satisfied, but its lax schema-validity as defined by Element Valid (Lax) (§3.3) has been assessed, in the post-schema validation infoset the item has a **[validation attempted]** property with the value *partial*.

If an element information item's schema-validity, as defined by either Element Valid (Explicit) (§3.3) or Element Valid (Explicit) (§3.3), has been assessed, whether successfully or not, then in the post-schema validation infoset the item has a **[validation context]** property whose value is the lowest containing element information item with a **[schema information]** property.

3.4 Complex Type Definition Details

Complex Type Definitions provide for:

- Constraining element information items by providing Attribute Declaration (§2.2.2.3)s governing the appearance and content of [attributes]
- Constraining element information item [children] to be empty, or to conform to a specified element-only or mixed content model.
- Using the mechanisms of Type Definition Hierarchy (§2.2.1.1) to derive a complex type from another simple or complex type.
- Specifying contributions to the post-schema-validation information set for elements.
- Limiting the ability to derive additional types from a given complex type.
- Controlling the permission to substitute, in an instance, elements of a derived type for elements declared in a content model to be of a given complex type.
- Determining post-schema-validation information set contributions

A complex type definition schema component has the following properties:

Schema Component: Complex Type Definition

{name}
Optional. An NCName as defined by [XML-Namespaces].
{target namespace}
Either absent or a namespace URI, as defined in [XML-Namespaces].
{base type definition}
Either a simple type definition or a complex type definition.
{derivation method}
Either *extension* or *restriction*.
{final}
A subset of {*extension*, *restriction*}.
{abstract}
A boolean
{attribute declarations}
A set of pairs of a boolean and an attribute declaration.
{attribute wildcard}
Optional. A wildcard.
{content type}
One of *empty*, a simple type definition or a pair consisting of a content model (I.e a Particle (§2.2.3.2)) and one of *mixed*, *element-only*.
{prohibited-substitutions}
A subset of {*extension*, *restriction*}.
{annotations}
A set of annotations.

Complex types definitions are identified by their {name} and {target namespace}. Except for anonymous complex type definitions (those with no {name}), since type definitions (i.e. both simple and complex type definitions taken together) must be uniquely identified within an XML Schema, no complex type definition can have the same name as another simple or complex type definition. Complex type {name}s and {target namespace}s are provided for reference from instances (see xsi:type (§2.6.1)), and for use in the XML Representation of Schemas and Schema Components (§4) (specifically in element). See References to schema components across namespaces (§6.2.3) for the use of component identifiers when importing one schema into another.

> **NOTE:** The {name} of a complex type is not *ipso facto* the [(local) name] of the element information items validated by that definition. The connection between a name and a type definition is described in Element Declaration Details (§3.3).

As described in Type Definition Hierarchy (§2.2.1.1), each complex type is derived from a {base type definition} which is itself either a Simple Type Definition (§2.2.1.2) or a Complex Type Definition (§2.2.1.3). {derivation method} specifies the means of derivation as either *extension* or *restriction* (see Type Definition Hierarchy (§2.2.1.1)).

A complex type with an empty specification for {final} can be used as a {base type definition} for other types derived by either of extension or restriction; the explicit values *extension*, and *restriction* prevent further derivations by extension and restriction respectively. If all values are specified, then the complex type is said to be [Definition:] **final**: no further derivations are possible.

A complex type for which {abstract} is *true* must not appear as the {type definition} of an Element Declaration (§2.2.2.1), and must not be referenced from an xsi:type (§2.6.1) attribute in an instance document; such abstract

complex types can be used as {base type definition}s, but they are never used directly to validate element content.

{attribute declarations} are a set of [Definition:] **attribute use pair**s: each is a pair of a boolean and an individual Attribute Declaration (§2.2.2.3) to be used for schema-validating the [attributes] of element information items, where the boolean determines whether the attribute is required or not See Element Children and Attributes Valid (§3.4) and Attribute Valid (§3.2) for details of attribute validation.

{attribute wildcard}s provide a more flexible specification for validation of attributes not explicitly included in {attribute declarations}. Informally, the specific values of {attribute wildcard} are interpreted as follows:

- *any*: [attributes] can include attributes with any qualified or unqualified name.
- a set whose members are either namespace URIs or absent: [attributes] can include any attribute(s) from the specified namespace(s). If absent is included in the set, then any unqualified attributes are (also) allowed.
- *'not'* and a namespace URI: [attributes] cannot include attributes from the specified namespace.
- *'not'* and absent: [attributes] cannot include unqualified attributes.

See Element Children and Attributes Valid (§3.4) and Wildcard allows Namespace URI (§3.9) for formal details of attribute wildcard validation.

{content type} determines the schema-validation of [children] of element information items. Informally:

- A {content type} with the distinguished value *empty* validates elements with no character or element information item [children].
- A {content type} which is a Simple Type Definition (§2.2.1.2) validates elements with character-only [children].
- An *element-only* {content type} validates elements with [children] that conform to the supplied content model.
- A *mixed* {content type} validates elements whose element information- children (I.e. specifically ignoring other [children] such as character information items) conform to the supplied content model.

{prohibited-substitutions} determine whether an element declaration appearing in a content model is prevented from additionally validating element items with an xsi:type (§2.6.1) attribute that identifies an *extension* or *restriction*, or element items in a substitution group whose type definition is similarly derived (if {prohibited-substitutions} contains one of those). If {prohibited-substitutions} is empty, then all such substitutions are valid.

See Element Children and Attributes Valid (§3.4) for a formal specification of element content validation.

See XML Representation of Complex Type Definition Schema Components (§4.3.3) for the XML representation of complex type definitions and Complex Type Definition Constraints (§5.11) for constraints on complex type definition components as such.

Validation Contribution: Element Children and Attributes Valid
An element information item is schema-valid with respect to a complex type definition if:

1.1 {abstract} is false;

1.2
- 1.2.1 If the {content type} is *empty*, the element information item has no character or element information item [children];
- 1.2.2 If the {content type} is a simple type definition, the element information item has no element information item [children], and the normalized value of the element information item is schema-valid with respect to that simple type definition as defined by String Valid (§3.13);
- 1.2.3 If the {content type} is *element-only*, the element information item has no character information item [children] other than those whose [character code] is defined as a white space in [XML];
- 1.2.4 If the {content type} is *element-only* or *mixed*, the sequence of the element information item's element information item [children], if any, taken in order, is schema-valid with respect to the {content type}'s particle, as defined in Element Sequence Valid (Particle) (§3.8)

1.3 For each attribute information item in the element information item's [children] excepting those whose [namespace URI] is identical to `http://www.w3.org/2000/10/XMLSchema-instance` and whose [local name] is one of `type`, `null`, `schemaLocation` or `noNamespaceSchemaLocation`, if there is among the {attribute declarations} an attribute use pair with an attribute declaration whose {name} matches the attribute information item's [local name] and whose {target namespace} is identical to the attribute information item's [namespace URI] (where an absent {target namespace} is taken to be identical to a [namespace URI] with no value) then

- 1.3.1 the attribute information item is schema-valid with respect to that attribute declaration as defined in Attribute Valid (§3.2);

otherwise (there is no pair with a matching attribute declaration)

- 1.3.2 there is an {attribute wildcard} and
  - 1.3.2 the attribute information item is schema-valid with respect to it as defined in Item Valid (Wildcard) (§3.9)

1.4 The attribute declaration of each attribute use pair in the {attribute declarations} whose boolean is *true* matches one of the attribute information items in the element information item's [children] as per clause 3 above.

Schema Information Set Contribution: Attribute Default Value
For each attribute use pair in the {attribute declarations} whose boolean is *false* and whose attribute declaration has a {value constraint} and does not match one of the attribute information items in the element information item's [children] as per clause 1.3 of Element Children and Attributes Valid (§3.4) above, the post-schema validation infoset has an attribute information item whose [local name] is that attribute declaration's {name} whose [namespace URI] is the attribute declaration's {target namespace} and whose **[schema normalized value]** is the declaration's {value constraint} string, added to the [attributes] of the element information item. Furthermore, the item's [specified] is set to *schema*.

Schema Information Set Contribution: Validation Outcome (Complex Type)
If the schema-validity, as defined by Element Children and Attributes Valid (§3.4) above, of an element information item has been assessed, in the post-schema validation infoset the item has a **[validity]** property, whose value is *complete* if the item is schema-valid, *partial* if it is not schema-valid but some or all of its element information item [children] and/or its [attributes] are either schema-valid or laxly schema-valid, otherwise *not*.

Schema Information Set Contribution: Validation Failure (Complex Type)
If the schema-validity, as defined by Element Children and Attributes Valid (§3.4) above, of an element information item has been assessed, in the post-schema validation infoset the item has a list-valued **[error code]** property. If the item is not schema-valid, applications wishing to provide information as to the reason(s) for this are encouraged to record one or more error codes (see Validity Contributions (§D.2) therein.

There is a complex type definition nearly equivalent to the ur-type definition present in every schema by

definition. It has the following properties:

Complex Type Definition of the Ur-Type

Property	Value
{name}	anyType
{target namespace}	http://www.w3.org/2000/10/XMLSchema
{base type definition}	Itself
{derivation method}	*restriction*
{content type}	A pair consisting of *mixed* and a particle with the following properties: **Property** **Value** {min occurs} 1 {max occurs} 1 {term} a model group with the following properties: **Property** **Value** {compositor} *sequence* {particles} a list containing one particle with the following properties: **Property** **Value** {min occurs} 0 {max occurs} unbounded {term} a wildcard with an *any* {namespace constraint}
{attribute declarations}	The empty set
{attribute wildcard}	{namespace constraint} is *any*
{final}	The empty set
{prohibited-substitutions}	The empty set
{abstract}	false

The `mixed` content specification together with the unconstrained wildcard content model and attribute specification produce the defining property for the ur-type definition, namely that *every* complex type definition is (eventually) a restriction of the ur-type definition: its permissions and requirements are the least restrictive possible.

3.5 Attribute Group Definition Details

A schema can name a group of attribute declarations so that they may be incorporated as a group into complex type definitions.

Attribute group definitions do not participate in schema-validation as such, but the {attribute declarations} and {attribute wildcard} of one or more complex type definitions may be constructed in whole or part by reference to an attribute group. Thus, attribute group definitions provide a replacement for some uses of XML's parameter entities. Attribute group definitions are provided primarily for reference from the XML Representation of Schemas and Schema Components (§4) (see complexType and attributeGroup).

The attribute group definition schema component has the following properties:

Schema Component: Attribute Group Definition
{name} An NCName as defined by [XML-Namespaces]. {target namespace} Either absent or a namespace URI, as defined in [XML-Namespaces]. {attribute declarations} A set of pairs of a boolean and an attribute declaration. {attribute wildcard} Optional. A wildcard. {annotation} Optional. An annotation

Attribute groups are identified by their {name} and {target namespace}; attribute group identities must be unique within an XML Schema. See References to schema components across namespaces (§6.2.3) for the use of component identifiers when importing one schema into another.

{attribute declarations} is a set of attribute use pairs, that is, a set of pairs of a boolean and an attribute declaration specifically identified as members of the attribute group, where the boolean determines whether the corresponding attribute is required or not.

{attribute wildcard} provides for an attribute wildcard to be included in an attribute group. See above under Complex Type Definition Details (§3.4) for the interpretation of attribute wildcards during validation.

See Element Children and Attributes Valid (§3.4) and Item Valid (Wildcard) (§3.9) for formal details of attribute wildcard validation. See XML Representation of Attribute Group Definition Schema Components (§4.3.4) for the XML representation of attribute group definitions, and Attribute Group Definition Constraints (§5.4) for constraints on attribute group definition components as such.

3.6 Model Group Definition Details

A model group definition associates a name and optional annotations with a Model Group (§2.2.3.1). By reference to the name, the entire model group can be incorporated by reference into a {term}.

Model group definitions are provided primarily for reference from the XML Representation of Complex Type Definition Schema Components (§4.3.3) (see complexType and group). Thus, model group definitions provide a replacement for some uses of XML's parameter entities.

The model group definition schema component has the following properties:

Schema Component: Model Group Definition
{name} An NCName as defined by [XML-Namespaces]. {target namespace} Either absent or a namespace URI, as defined in [XML-Namespaces]. {model group} A model group. {annotation} Optional. An annotation

Model group definitions are identified by their {name} and {target namespace}; model group identities must be unique within an XML Schema. See References to schema components across namespaces (§6.2.3) for the use of component identifiers when importing one schema into another.

Model group definitions *per se* do not participate in schema-validation, but the {term} of a particle may correspond in whole or in part to a model group from a model group definition.

{model group} is the Model Group (§2.2.3.1) for which the model group definition provides a name.

See XML Representation of Model Group Definition Schema Components (§4.3.5) for the XML representation of model group definitions and Model Group Definition Constraints (§5.6) for constraints on model group definition components as such.

3.7 Model Group Details

When the [children] of element information items are not constrained to be *empty* or by reference to a simple type definition ((non-normative) Simple Type Definition Details (§3.13)), the sequence of element information item [children] content may be specified in more detail with a model group. Because the {term} property of a particle can be a model group, and model groups contain particles, model groups can indirectly contain other model groups; the grammar for content models is therefore recursive.

The model group schema component has the following properties:

Schema Component: Model Group
{compositor} One of *all*, *choice* or *sequence*. {particles} A list of particles {annotation} Optional. An annotation

{compositor} specifies a sequential (*sequence*), disjunctive (*choice*) or conjunctive (*all*) interpretation of the {particles}. This in turn determines whether the element information item [children] validated by the model group must:

- (*sequence*) correspond, in order, to the specified {particles};
- (*choice*) corresponded to exactly one of the specified {particles};
- (*all*, (in which case {particles} is restricted to contain local and global element declarations, with {min occurs}=0 or 1, {max occurs}=1) contain exactly zero or one of each element specified in {particles}. The elements can occur in any order.

When two or more element declarations with the same identity occur at any level within a model group, their type definitions must be the same.

{annotation} Description to be supplied in a future draft.

Validation Contribution: Element Sequence Valid

[Definition:] We define a **partition** of a sequence as a sequence of sub-sequences, some or all of which may be empty, such that concatenating all the sub-sequences yields the original sequence.

A sequence (possibly empty) of element information items is schema-valid with respect to a model group if

1.1 The {compositor} is *sequence* and there is a partition of the sequence into n sub-sequences where n is the length of {particles} such that each of the sub-sequences in order is schema-valid with respect to the corresponding particle in the {particles} as defined in Element Sequence Valid (Particle) (§3.8);

or

1.2 The {compositor} is *choice* and there is a particle among the {particles} such that the sequence is schema-valid with respect to that particle as defined in Element Sequence Valid (Particle) (§3.8);

or

1.3 The {compositor} is *all* and there is a partition of the sequence into n sub-sequences where n is the length of {particles} such that there is a one-to-one mapping between the sub-sequences and the {particles} where each sub-sequence is schema-valid with respect to the corresponding particle as defined in Element Sequence Valid (Particle) (§3.8);

Nothing in the above should be understood as ruling out groups whose {particles} is empty: although no sequence can be schema-valid with respect to such a group whose {compositor} is *choice*, the empty sequence *is* schema-valid with respect to empty groups whose {compositor} is *sequence* or *all*.

> **NOTE:** The above definition is implicitly non-deterministic, and should not be taken as a recipé for implementations. Note in particular that when {compositor} is *all*, particles is restricted to a list of local and global element declarations (see Model Group Constraints (§5.7)). A much simpler implementation is possible than would arise from a literal interpretation of the definition above; informally, the content is valid when each declared element occurs exactly once (or at most once, if {min occurs} is 0), and each is valid with respect to its corresponding declaration. The elements can occur in arbitrary order.

See XML Representation of Model Group Schema Components (§4.3.6) for the XML representation of model groups and Model Group Constraints (§5.7) for constraints on model group components as such.

3.8 Particle Details

As described in Model Group Details (§3.7), particles contribute to the definition of content models. The particle schema component has the following properties:

Schema Component: Particle
{min occurs} A non-negative integer {max occurs} Either a non-negative integer or *unbounded* {term} One of a model group, a wildcard, or an element declaration.

The following is an informal overview of the properties of a particle. Formal interpretation of these properties is found in Element Sequence Valid (Particle) (§3.8).

In general, multiple element information item [children], possibly with intervening character [children] if the content type is *mixed*, can be validated with respect to a single particle. {min occurs} determines the minimum number of such element [children] that can validly occur. The number of such children must be greater than or

equal to {min occurs}. If {min occurs} is *0*, then occurrence of such children is optional.

The number of such element [children] must be less than or equal to any numeric specification of {max occurs}; if {max occurs} is *unbounded*, then there is no upper bound on the number of such children.

Validation Contribution: Element Sequence Valid (Particle)
A sequence (possibly empty) of element information items is schema-valid with respect to a particle if either

1.1.1 The length of the sequence is greater than or equal to the {min occurs};

1.1.2 If {max occurs} is a number, the length of the sequence is less than or equal to the {max occurs};

1.1.3 Either

1.1.3.1 the {term} is a wildcard and each element information item in the sequence is schema-valid with respect to the wildcard as defined by Item Valid (Wildcard) (§3.9)

or

1.1.3.2.1 the {term} is an element declaration;

1.1.3.2.2 for each element information item in the sequence either

1.1.3.2.2.1 the element declaration is local (i.e. its {scope} is not *global*), its {abstract} is false, the element information item's [namespace URI] is identical to the element declaration's {target namespace} (where an absent {target namespace} is taken to be identical to a [namespace URI] with no value), the element information item's [local name] matches the element declaration's {name} and the element information item is schema-valid with respect to the declaration as defined in Element Valid (Explicit) (§3.3);

or

1.1.3.2.2.2 the element declaration is global (i.e. its {scope} is *global*), {abstract} is false, the element information item's [namespace URI] is identical to the element declaration's {target namespace} (where an absent {target namespace} is taken to be identical to a [namespace URI] with no value), the element information item's [local name] matches the element declaration's {name} and the element information item is schema-valid with respect to the element declaration as defined in Element Valid (Explicit) (§3.3);

or

1.1.3.2.2.3 the element declaration is global (i.e. its {scope} is *global*), its {disallowed substitutions} does not contain *substitution*, the [local] and [namespace URI] of the element information item resolve to an element declaration, as defined in QName resolution (Instance) (§3) -- [Definition:] call this declaration the **substituting declaration**, the substituting declaration together with the particle's element declaration's {disallowed substitutions} is validly substitutable for the particle's element declaration as defined in Substitution Group OK (Transitive) (§5.2) and the element information item is schema-valid with respect to the substituting declaration as defined in Element Valid (Explicit) (§3.3);

or

1.2 the {term} is a model group and there is a partition of the sequence into n sub-sequences such that n is greater than or equal to {min occurs} and, if {max occurs} is a number, less than or equal to {max occurs} and each sub-sequence is schema-valid with respect to that model group as defined in Element Sequence Valid (§3.7);

See XML Representation of Model Group Schema Components (§4.3.6) for the XML representation of particles and Particle Constraints (§5.10) for constraints on particle components as such.

3.9 Wildcard Details

In order to exploit the full potential for extensibility offered by XML plus namespaces, more provision is needed than DTDs allow for targeted flexibility in content models and attribute declarations. A wildcard provides for validation of attribute and element information items dependent on their namespace URI, but independently of their local name. The wildcard schema component has the following properties:

Schema Component: Wildcard
{namespace constraint} One of *any*; a pair of *not* and a namespace URI or absent; or a set whose members are either namespace URIs or absent. {process contents} One of *skip*, *lax* or *strict* {annotation} Optional. An annotation

{namespace constraint} provides for validation of elements that:

1. (*any*) have any namespace or are not namespace qualified;
2. (*not* and a namespace URI) have any namespace other than the specified namespace URI, or are not namespace qualified;
3. (*not* and absent) are namespace qualified;
4. (a set whose members are either namespace URIs or absent) have any of the specified namespaces and/or, if absent is included in the set, are unqualified.

{process contents} controls the impact on schema-validity of the information items allowed by wildcards, as follows:

strict
: There must be a global declaration for the item available, and it must be schema-valid with respect to that definition.

skip
: No constraints at all: the item must simply be well-formed XML.

lax
: If the item, or any items among its [children] if it's an element information item, has a uniquely determined declaration available, it must be laxly schema-valid with respect to that definition, that is, schema-validate where you can, don't worry when you can't.

Validation Contribution: Item Valid (Wildcard)
An element or attribute information item is schema-valid with respect to a wildcard constraint if

1.1 its [namespace URI] is schema-valid with respect to the wildcard constraint, as defined in Wildcard allows Namespace URI (§3.9);

1.2 Either

1.2.1 {process contents} is *skip*

or

1.2.2.1 {process contents} is *strict*;

1.2.2.2 The [local name] and [namespace URI] resolve to an element or attribute declaration, as appropriate to the kind of item, as defined by QName resolution (Instance) (§3);

1.2.2.3 The item is schema-valid with respect to that declaration, as defined by Element Valid (Explicit) (§3.3) or Attribute Valid (§3.2).

or

1.2.3.1 {process contents} is *lax*

1.2.3.2 The information item is laxly valid, as defined by Element Valid (Lax) (§3.3) or Attribute Valid (Lax) (§3.2), as appropriate to the kind of item.

Validation Contribution: Wildcard allows Namespace URI
A value which is either a namespace URI or absent is schema-valid with respect to a wildcard constraint (the value of a {namespace constraint}) if

1.1 the constraint is *any*;

or

1.2 the constraint is a pair of *not* and a namespace URI, and the value is not identical to the namespace URI;

or

1.3 the constraint is a set, and the value is identical to one of the members of the set.

Schema Information Set Contribution: Validation Outcome (skipped)
If clause 1.2.1 of Item Valid (Wildcard) (§3.9) above obtains with respect to an information item, in the post-schema validation infoset the item has a **[validation attempted]** property with the value *none*.

See XML Representation of Wildcard Schema Components (§4.3.7) for the XML representation of wildcards and Wildcard Constraints (§5.5) for constraints on wildcard components as such.

3.10 Identity-constraint Definition Details

Identity-constraint definition components provide for uniqueness and reference constraints with respect to the contents of multiple elements and attributes. The identity-constraint definition schema component has the following properties:

Schema Component: Identity-constraint Definition

{name}
An NCName as defined by [XML-Namespaces].
{target namespace}
Either absent or a namespace URI, as defined in [XML-Namespaces].
{identity-constraint category}
One of *key*, *keyref* or *unique*.
{selector}
An XPath expression, as defined in [XPath]
{fields}
An a non-empty list of XPath expressions, as defined in [XPath]
{referenced key}
Required if {identity-constraint category} is *keyref*, forbidden otherwise. A identity-constraint definition with {identity-constraint category} equal to *key* or *unique*.
{annotation}
Optional. An annotation

Identity-constraint definitions are identified by their {name} and {target namespace}; Identity-constraint definition identities must be unique within an XML Schema. See References to schema components across namespaces (§6.2.3) for the use of component identifiers when importing one schema into another.

Informally, {identity-constraint category} identifies the Identity-constraint definition as playing one of three roles:

- (*unique*) the Identity-constraint definition asserts uniqueness, with respect to the content identified by {selector}, of the tuples resulting from evaluation of the {fields} XPath expression(s).
- (*key*) the Identity-constraint definition asserts uniqueness as for *unique*. *key* further asserts that all selected content actually has such tuples.
- (*keyref*) the Identity-constraint definition asserts a correspondence, with respect to the content identified by {selector}, of the tuples resulting from evaluation of the {fields} XPath expression(s), with those of the {referenced key}.

These constraints are specified independently of the types of the attributes and elements involved, i.e. something declared as of type integer may also serve as a key, unlike `ID` and `IDREF`. Each constraint declaration has a name, which exists in a single symbol space for constraints. The equality and inequality conditions appealed to in checking these constraints applies to the *value* of the fields selected, so that for example `3.0` and `3` would be conflicting keys if they were both decimal, but non-conflicting if they were both strings, or one was a string and one a decimal.

Overall the augmentations to XML's `ID/IDREF` mechanism are:

- Not just attribute values, but also element content and combinations of values and content can be declared to be unique;
- Constraints are specified to hold within the scope of particular elements;
- (Combinations of) attribute values and/or element content can be declared to be keys, that is, not only unique, but always present and non-nullable;
- The comparison between *keyref* {fields} and *key* or *unique* {fields} is by value equality, not by string equality.

{selector} specifies an XPath expression [XPath] relative to instances of the element being declared. This must

identify a node set of subelements (i.e. elements contained within the declared element) to which the constraint applies.

{fields} specifies XPath expressions relative to each element selected by a{selector}. This must identify a single node (element or attribute, not necessarily within the selected element) whose content or value, which must be o a simple type, is used in the constraint. It is possible to specify an ordered list of{fields}s, to cater to multi-field keys, keyrefs, and uniqueness constraints.

> **NOTE:** Provision for multi-field keys etc. goes beyond what is supported by`xsl:key`.
>
> **NOTE:** If reference to a key or unique defined in a scoping element which may occur more than once is envisaged (which reference may be from outside any of the scoping elements), then the scoping elements themselves must have keys (typically unique across the entire document), and the scoped keys must include the key of their scoping element among their fields.

A formal description of Identity-constraint definition validation is given below inIdentity-constraint Satisfied (§3.10)

Validation Contribution: Identity-constraint Satisfied
An element information item is schema-valid with respect to a identity-constraint if

1.1 The {selector}, with the element information item as the context node, evaluates to a node-set (as defined in [XPath]). [Definition:] Call this the **target node set**;

1.2 Each node in the target node set is an element node among the descendants of the context node;

1.3 For each node in the target node set all of the {fields}, with that node as the context node, evaluate to either an empty node-set or a node-set with exactly one member. [Definition:] Call the sequence of the values (as defined in[XML Schemas: Datatypes]) of those node-sets in order the **key-sequence** of the node;

[Definition:] Call the subset of the target node set for which all the {fields} evaluate to a node-set with exactly one member which is an element or attribute node the**qualified node set**;

2.1.1 The {identity-constraint category} is *unique*;

2.1.2 No two members of the qualified node set have key-sequences whose members are pairwise equal, as defined in [XML Schemas: Datatypes];

or

2.2.1 The {identity-constraint category} is *key*;

2.2.2 The target node set and the qualified node set are equal, that is, every member of the target node set is also a member of the qualified node set and *vice versa*;

2.2.3 No two members of the qualified node set have key-sequences whose members are pairwise equal;

2.2.4 No element member of the key-sequence of any member of the qualified node set was assessed as schema-valid by reference to an element declaration whose{nullable} is *true*.

or

2.3.1 The {identity-constraint category} is *keyref*;

2.3.2 For each member of the qualified node set (call this the **keyref member**), there must be a member of the node table associated with the {referenced key} in the **[identity-constraint table]** of the element information item (see Identity-constraint Table (§3.10), which must be understood as logically prior to this clause of this constraint, below) whose key-sequence is equal to the **keyref member's** key-sequence member for member.

NOTE: This specification does not define a post-schema validation infoset contribution which would enable schema-aware processors to implement clause 2.2.4 above. This clause can be read as if there were such a contribution, which recorded for example either the element declaration appealed to in Element Valid (Explicit) (§3.3), or the value of its {nullable} property.

Schema Information Set Contribution: Identity-constraint Table

[Definition:] An **eligible identity-constraint** of an element information item is one such that clauses 2.1.[1-2] or 2.2.[1-3] of Identity-constraint Satisfied (§3.10) obtains with respect to that item and that constraint, or such that any of the element information item [children] of that item have a **[identity-constraint table]** with an entry for that constraint.

[Definition:] A **node table** is a set of pairs each consisting of a key-sequence and an element node.

Whenever an element information item has one or more eligible constraints, a new **[identity-constraint table]** is added to the post-schema-validation infoset for that element information item, consisting of pairs of identity-constraints and node tables, one for each of the item's eligible constraints, with the node table in each pair defined as follows: There is a member in the node table associated with an eligible constraint of an element information item consisting of a key-sequence (call it **k**) and a node (call it **n**) if and only if

1.1 1.1.1 There is a member in one of the node tables associated with the eligible constraint in at least one of the **[identity-constraint tables]** of the element information item [children] of the element information item whose key-sequence is **k** and whose node is **n**;

or

1.1.2 **n** is in the qualified node set for the eligible constraint of the element information item with key-sequence **k**.

1.2 There is no member in one of the node tables associated with the eligible constraint in any of the **[identity-constraint tables]** of the element information item [children] of the element information item whose key-sequence is **k** and whose node is a node other than **n**;

1.3 Some node distinct from **n** is in the qualified node set for the eligible constraint of the element information item with key-sequence **k**.

NOTE: This information set contribution, unlike others in this specification, is essentially an internal bookkeeping mechanism. It is introduced to support the definition of Identity-constraint Satisfied (§3.10) above. Accordingly, conformant processors may, but are *not* required, to expose **[identity-constraint table]**s in the post-schema-validation infoset. In other words, the above constraints may be read as saying validation of identity-constraints proceeds *as if* such an infoset property existed.

See XML Representation of Identity-constraint Definition Schema Components (§4.3.8) for the XML representation of identity-constraint definitions and Identity-constraint Definition Constraints (§5.3) for constraints on identity-constraint definition components as such.

3.11 Notation Declaration Details

The notation declaration schema component has the following properties:

Schema Component: Notation Declaration
{name} An NCName as defined by [XML-Namespaces]. {target namespace} Either absent or a namespace URI, as defined in [XML-Namespaces]. {system identifier} Optional if {public identifier} is present. A URI reference. {public identifier} Optional if {system identifier} is present. A public identifier, as defined in [XML]. {annotation} Optional. An annotation

Notation declarations do not participate in schema-validation as such. They are referenced in the course o schema-validating strings as members of the NOTATION simple type.

See XML Representation of Notation Declaration Schema Components (§4.3.9) for the XML representation of notation declarations and Notation Declaration Constraints (§5.8) for constraints on notation declaration components as such.

3.12 Annotation Details

The annotation schema component has the following properties:

Schema Component: Annotation
{application information} A sequence of element information items. {user information} A set of element information items.

{user information} is intended for human consumption, {application information} for automatic processing. In both cases, provision is made for an optional URI reference to supplement the local information. Schema validation does *not* involve dereferencing these URIs, when present. In the case of {user information}, indication may be given as to the identity of the (human) language used in the contents, using the `xml:lang` attribute.

Annotations do not participate in schema-validation as such. Provided an annotation itself satisfies all relevant Constraints of Schemas it *cannot* affect the schema-validity of element information items.

See XML Representation of Annotation Schema Components (§4.3.10) for the XML representation of annotations and Annotation Constraints (§5.9) for constraints on annotation components as such.

3.13 (non-normative) Simple Type Definition Details

> **NOTE:** This section reproduces a version of material from [XML Schemas: Datatypes], for local cross-reference purposes.

Simple type definitions provide for constraining character information item [children] of element and attribute information items. The simple type definition schema component has the following properties:

Schema Component: Simple Type Definition

{name}
Optional. An NCName as defined by [XML-Namespaces].
{target namespace}
Either absent or a namespace URI, as defined in [XML-Namespaces].
{base type definition}
A simple type definition, which may be the simple ur-type definition
{variety}
One of {*atomic*, *list*, *union*}. Depending on the value of {variety}, further properties are defined as follows:
atomic
{primitive type definition}
A built-in primitive simple type definition (or the simple ur-type definition).
{facets}
A set of constraining facets.
list
{item type definition}
A simple type definition.
{facets}
A set of constraining facets.
union
{member type definitions}
A non-empty sequence of simple type definitions.
{facets}
A set of constraining facets.
{annotation}
Optional. An annotation

Simple types are identified by their {name} and {target namespace}. Except for anonymous simple types (those with no {name}), since type definitions (i.e. both simple and complex type definitions taken together) must be uniquely identified within an XML Schema, no simple type definition can have the same name as another simple or complex type definition. Simple type {name}s and {target namespace}s are provided for reference from instances (see xsi:type (§2.6.1)), and for use in the XML Representation of Schemas and Schema Components (§4) (specifically in element and attribute). See References to schema components across namespaces (§6.2.3) for the use of component identifiers when importing one schema into another.

> **NOTE:** The {name} of a simple type is not *ipso facto* the [(local) name] of the element or attribute information items validated by that definition. The connection between a name and a type definition is described in Element Declaration Details (§3.3) and Attribute Declaration Details (§3.2).

{variety} determines whether the simple type corresponds to an *atomic*, *list* or *union* type as defined by [XML Schemas: Datatypes].

As described in Type Definition Hierarchy (§2.2.1.1), every simple type definition is a restriction of some other simple type (the {base type definition}), which is the simple ur-type definition if and only if the type definition in question is one of the built-in primitive datatypes, or a list or union type definition. Each *atomic* type is ultimately

a restriction of exactly one such built-in simple {primitive type definition}.

{facets} for each simple type definition are selected from those defined in [XML Schemas: Datatypes]. For *atomic* definitions, these are restricted to those appropriate for the corresponding {primitive type definition}. Therefore, the value space and lexical space (I.e. the content validated by) any atomic simple type is determined by the pair ({primitive type definition}, {facets}).

As specified in [XML Schemas: Datatypes], *list* simple type definitions validate space separated tokens, each of which conforms to a specified simple type definition, the {item type definition}. The item type specified must not itself be a *list* type, and must be one of the types identified in [XML Schemas: Datatypes] as a suitable base for a list simple type. In this case the {facets} apply to the list itself, and are restricted to those appropriate for lists.

A *union* simple type definition validates strings which satisfy at least one of its {member type definitions}. As in the case of *list*, the {facets} apply to the union itself, and are restricted to those appropriate for unions.

Simple type definitions for all the built-in primitive datatypes, namely *string*, *boolean*, *float*, *double*, *decimal*, *timeInstant*, *timeDuration*, *recurringInstant*, *binary*, *uriReference* (see the Primitive Datatypes section of [XML Schemas: Datatypes]), as well as for the simple and complex ur-type definitions (as previously described), are present by definition in every schema. All are in the XML Schema {target namespace} (namespace URI `http://www.w3.org/2000/10/XMLSchema`), have an *atomic* {variety} with an empty {facets} and the simple ur-type definition as their base type definition and themselves as {primitive type definition}.

Similarly, simple type definitions for all the built-in derived datatypes (see the Derived Datatypes section of [XML Schemas: Datatypes]) are present by definition in every schema, with properties as specified in [XML Schemas: Datatypes] and as represented in XML in (normative) Schema for Schemas (§A) therein.

There is a separate ur-Type for simple types. As discussed in Type Definition Hierarchy (§2.2.1.1), the ur-type definition functions as a simple type when used as the base type definition for the built-in primitive datatypes and for list and union type definitions. It is considered to have an unconstrained lexical space, and a value space consisting of the union of the value spaces of all the built-in primitive datatypes and the set of all lists of all members of the value spaces of all the built-in primitive datatypes.

The simple ur-type definition must *not* be named as the base type definition of any user-defined simple types: as it has no constraining facets, this would be incoherent.

Validation Contribution: String Valid
A string is schema-valid with respect to a simple type definition if it is schema-valid with respect to that definition as defined by Datatype Valid in [XML Schemas: Datatypes].

There is a simple type definition nearly equivalent to the ur-type definition present in every schema by definition. It has the following properties:

Simple Type Definition of the Ur-Type	
Property	**Value**
{name}	anySimpleType
{target namespace}	http://www.w3.org/2000/10/XMLSchema
{base type definition}	the ur-type definition
{variety}	absent

See (non-normative) XML Representation of Simple Type Definition Schema Components (§4.3.11) for the

XML representation of simple type definitions and Simple Type Definition Constraints (§5.12) for constraints on simple type definition components as such.

4 XML Representation of Schemas and Schema Components

The principal purpose of *XML Schema: Structures* is to define a set of schema components that constrain the contents of instances and augment the information sets thereof. Although no external representation of schemas is required for this purpose, such representations will obviously be widely used. To provide for this in an appropriate and interoperable way, we specify a normative XML representation for schemas which makes provision for every kind of schema component. [Definition:] A document in this form (i.e. a schema element information item) is a **schema document**. For the schema document as a whole, and its constituents, the sections below define correspondences between element information items (with declarations in (normative) Schema for Schemas (§A) and (non-normative) DTD for Schemas (§E)) and schema components. All the element information items in the XML representation of a schema are in the XML Schema namespace, that is their [namespace URI] is `http://www.w3.org/2000/10/XMLSchema`. Although a common way of creating schema documents will be using an XML parser, this is not required: any mechanism which constructs conformant infosets as defined in [XML-Infoset] is a possible starting point.

When we say below that a numeric-valued property of a schema component corresponds to the normalized value of some attribute information item, the number in question is understood to be the base 10 interpretation of that normalized value.

Two aspects of the type definitions for the elements presented in the following sections are constant across them all:

1. All of them allow attributes qualified with namespace URIs other than the XML Schema namespace itself: these correspond to nothing in corresponding schema component;
2. All of them allow an annotation as their first child, for human-readable documentation and/or machine-targetted information.

4.1 XML Representations of Schemas

A schema is represented in XML by one or more schema documents. A schema document contains representations for a collection of schema components, e.g. type definitions and element declarations, which have a common **{target namespace}**. A schema document which has one or more import element information items corresponds to a schema with components with more than one **{target namespace}**, see Import Constraints and Semantics (§6.2.3).

XML Representation Summary: `schema` Element Information Item

```
<schema
  attributeFormDefault = qualified | unqualified : unqualified
  blockDefault = #all or (possibly empty) subset of {substitution,
extension, restriction}
  elementFormDefault = qualified | unqualified : unqualified
  finalDefault = #all or (possibly empty) subset of {extension,
restriction}
  id = ID
  targetNamespace = uriReference
  version = string
  {any attributes with non-schema namespace . . .}>
  Content: ((include | import | redefine | annotation)* ,
((attribute | attributeGroup | complexType | element | group |
notation | simpleType) , annotation*)*)
</schema>
```

Schema Schema Component

Property	Representation
{type definitions}	The simple and complex type definitions corresponding to all the simpleType and complexType element information items in the [children], if any, plus any included or imported definitions, see Assembling a schema for a single target namespace from multiple schema definition documents (§6.2.1) and References to schema components across namespaces (§6.2.3).
{attribute declarations}	The (global) attribute declarations corresponding to all the attribute element information items in the [children], if any, plus any included or imported declarations, see Assembling a schema for a single target namespace from multiple schema definition documents (§6.2.1) and References to schema components across namespaces (§6.2.3).
{element declarations}	The (global) element declarations corresponding to all the element element information items in the [children], if any, plus any included or imported declarations, see Assembling a schema for a single target namespace from multiple schema definition documents (§6.2.1) and References to schema components across namespaces (§6.2.3).
{attribute group definitions}	The attribute group definitions corresponding to all the attributeGroup element information items in the [children], if any, plus any included or imported definitions, see Assembling a schema for a single target namespace from multiple schema definition documents (§6.2.1) and References to schema components across namespaces (§6.2.3).
{model group definitions}	The model group definitions corresponding to all the group element information items in the [children], if any, plus any included or imported definitions, see Assembling a schema for a single target namespace from multiple schema definition documents (§6.2.1) and References to schema components across namespaces (§6.2.3).
{notation declarations}	The notation declarations corresponding to all the notation element information items in the [children], if any, plus any included or imported declarations, see Assembling a schema for a single target namespace from multiple schema definition documents (§6.2.1) and References to schema components across namespaces (§6.2.3).
{annotations}	The annotations corresponding to all the annotation element information items in the [children], if any.

Note that none of the attribute information items displayed above correspond directly to properties of schemas The `blockDefault`, `finalDefault`, `attributeFormDefault`, `elementFormDefault`and `targetNamespace` attributes are appealed to in the sub-sections below, as they provide global information applicable to many representation/component correspondences. The other attributes (`id` and `version`) are for user convenience, and this specification defines no semantics for them.

> **Ed. Note:** Priority Feedback Request
> A number of the attributes (listed above) on the <schema> element provide defaults for attributes on subordinate elements. This allows setting values for attributes we judge likely to have the same value across a whole schema document in only one place. It does constitute a kind of minimisation, and does not provide any new semantics. The Working Group solicits feedback both on whether this aspect of the design is a good thing or not, and on the particular values chosen as the defaults for these default-setting attributes themselves.

The definition of the schema abstract data model inXML Schema Abstract Data Model (§2.2) makes clear that most components have a **{target namespace}**. Most components corresponding to representations within a given schema element information item will have a**{target namespace}** which corresponds to the `targetNamespace` attribute.

Since the empty string is a legal (relative) URI reference, supplying an empty string for`targetNamespace` is *not* the same as not specifying it at all. The appropriate form of schema document corresponding to aschema whose components have no {target namespace} is one which has no `targetNamespace` attribute specified at all.

> **NOTE:** The XML namespaces recommendation discusses only instance document syntax for elements and attributes; it therefore provides no direct framework for managing the names of type definitions, attribute group definitions, and so on. Nevertheless, we apply the target namespace facility uniformly to all schema components, i.e. not only declarations but also definitions have a **{target namespace}**.

Example

```
<xs:schema
    xmlns:xs="http://www.w3.org/2000/10/XMLSchema
    targetNamespace="http://purl.org/metadata/dublin_core"
    version="M.n">

  ...

</xs:schema>
```

A modest beginning to a schema.

Although the schema above might be a complete XML document,schema need not be the document element, but can appear within other documents. Indeed there is no requirement that a schema correspond to a (text) document at all: it could correspond to an element information item constructed 'by hand', for instance via a DOM-conformant API.

Aside from include and import, which do not correspond directly to any schema component at all, each of the element information items which may appear in the content ofschema corresponds to a schema component, and all except annotation are named. The sub-sections of XML Representation of Schema Components (§4.3) present each such item in turn, setting out the components to which it may correspond.

4.2 References to Schema Components

Reference to schema components from a schema document is managed in a uniform way, whether the component corresponds to an element information item from the same schema document or is imported (References to schema components across namespaces (§6.2.3)) from an external schema (which may, but need not, correspond to an actual schema document). The form of all such references is a QName. In each of the XML representation expositions in the following sections, an attribute is shown as having type `QName` if and only if it is interpreted as referencing a schema component.

Example

```
<xs:schema xmlns:xs="http://www.w3.org/2000/10/XMLSchema"
           xmlns:xhtml="http://www.w3.org/1999/xhtml"
           xmlns="http://www.foo.com"
           targetNamespace="http://www.foo.com">
  . . .

  <xs:element name="elem1" type="Address"/>

  <xs:element name="elem2" type="xhtml:blockquote"/>

  <xs:attribute name="attr1"
                type="xsl:quantity"/>
  . . .
</xs:schema>
```

The first of these is most probably a local reference, i.e. a reference to a type definition corresponding to a complexType element information item located elsewhere in the schema document, the other two refer to type definitions from schemas for other namespaces and assume that their namespaces have been declared for import. See References to schema components across namespaces (§6.2.3) for a discussion of importing.

Schema Representation Constraint: QName Interpretation

Where the type of an attribute information item in a document involved in schema validation is identified as QName, its normalized value is uniformly interpreted as consisting of a [Definition:] **local name** consisting of the character information items after the colon, if any, otherwise all the character information items, in the normalized value, and of a [Definition:] **namespace URI**, derived from its normalized value and the containing element information item's [in-scope namespaces] as follows:

1.1 If the normalized value contains a colon, then the [namespace URI] of the member of the [in-scope namespaces] whose [prefix] matches the character information items before the colon. If no such member is present, the QName is uninterpretable and the Element Declaration Representation OK (§4.3.2) of element information item containing it is not satisfied;

1.2 otherwise (i.e. the normalized value contains no colon)

1.2.1 if there is a member of the [in-scope namespaces] whose [prefix] is absent, then its [namespace URI];

1.2.2 otherwise absent.

In the absence of the non-core properties [in-scope namespaces] and/or [namespace URI] from the infoset for the schema document in question, processors must reconstruct equivalent information as necessary, using the [declared namespaces] of the containing element information item and its ancestors in the first case and using the

namespace declaration in question's[children] in the second.

Whenever the word "resolve" in any form is used in this chapter in connection with aQName in a schema document, the following definition should be understood as obtaining:

Schema Representation Constraint: QName resolution (Schema Document)
A string known to be a QName resolves to a schema component of a specified kind if:

1.1 that component is a member of the value of the appropriate property of the schema which corresponds to the schema document within which theQName appears, that is

1.1.1 the {type definitions} if the kind specified is simple or complex type definition;

1.1.2 the {attribute declarations} if the kind specified is attribute declaration;

1.1.3 the {element declarations} if the kind specified is element declaration;

1.1.4 the {attribute group definitions} if the kind specified is attribute group;

1.1.5 the {model group definitions} if the kind specified is model group;

1.1.6 the {notation declarations} if the kind specified is notation declaration;

1.2 its **{local name}** matches the local name of the string;

1.3 its **{target namespace}** is identical to the namespace URI of the string;

1.4 its namespace URI is either the target namespace of the schema document containing theQName or that schema document contains an import element information item thenormalized value of whose `namespace` [attribute] is identical to that namespace URI.

4.2.1 References to Schema Components from Elsewhere

The names of schema components such as type definitions and element declarations are not of type `ID`: they are not unique within a schema, just within a symbol space. This means that simple fragment identifiers will not always work to reference schema components from outside the context of schema documents.

There is currently no provision in the definition of the interpretation of fragment identifiers for the `text/xml` MIME type, which is the MIME type for schemas, for referencing schema components as such. However, we observe that [XPointer] provides a mechanism which maps well onto our notion of symbol spaces as it is reflected in the XML representation of schema components. A fragment identifier of the form `#xpointer(xs:schema/xs:element[@name="person"])` will uniquely identify the representation of a global element declaration with name`person`, and similar fragment identifiers can obviously be constructed for the other global symbol spaces.

Short-form fragment identifiers may also be used in some cases, that is when a DTD or XML Schema is available for the schema in question, and the provision of an`id` attribute for the representations of all primary and secondary schema components, which*is* of type `ID`, has been exploited.

It is a matter for applications to specify whether they interpret document-level references of either of the above varieties as being to the relevant element information item (i.e. without special recognition of the relation of schema documents to schema components) or as being to the corresponding schema component.

4.3 XML Representation of Schema Components

For each kind of schema component there is a corresponding normative XML representation. The sections below describe the correspondences between the properties of each kind of schema component on the one hand and the properties of information items in that XML representation on the other, together with constraints on that

representation above and beyond those implicit in the(normative) Schema for Schemas (§A) and (non-normative) DTD for Schemas (§E).

The language used is as if the correspondences were mappings from XML representation to schema component, but the mapping in the other direction, and therefore the correspondence in the abstract, can always be constructed therefrom.

4.3.1 XML Representation of Attribute Declaration Schema Components

The XML representation for an attribute declaration schema component is anattribute element information item. It specifies a simple type definition for an attribute either by reference or explicitly, and may provide default information. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: `attribute` Element Information Item

```
<attribute
  form = qualified | unqualified
  id = ID
  name = NCName
  ref = QName
  type = QName
  use = prohibited | optional | required | default |
fixed : optional
  value = string
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , (simpleType?))
</attribute>
```

If the attribute element information item hasschema as its parent, the corresponding schema component is as follows:

Attribute Declaration Schema Component

Property	Representation
{name}	The normalized value of the `name` [attribute]
{target namespace}	The normalized value of the `targetNamespace` [attribute] of the parent schema element information item, or absent if there is none.
{simple type definition}	The simple type definition corresponding to the simpleType element information item in the [children], if present, otherwise the simple type definition resolved to by the normalized value of the `type` [attribute], if present, otherwise the simple ur-type definition
{scope}	*global*
{value constraint}	If there is a `value` [attribute], then a pair consisting of the normalized value (with respect to the {simple type definition}) of that [attribute] and absent, otherwise absent.
{annotation}	The annotation corresponding to the annotation element information item in the [children], if present, otherwise absent.

otherwise if the attribute element information item hascomplexType or attributeGroup as an ancestor and the `ref` [attribute] is absent, it corresponds to an attribute use pair of a boolean and an attribute declaration (unless `use='prohibited'`, in which case the item corresponds to nothing at all).

The boolean is *true* if the `use` [attribute] is present with normalized value `required`, otherwise *false*.

The attribute declaration is as follows:

Attribute Declaration Schema Component	
Property	**Representation**
{name}	The normalized value of the `name` [attribute]
{target namespace}	If `form` is present and its normalized value is `qualified`, or if `form` is absent and the normalized value of `attributeFormDefault` on the schema ancestor is `qualified`, then the normalized value of the `targetNamespace` [attribute] of the parent schema element information item, or absent if there is none, otherwise absent.
{simple type definition}	The simple type definition corresponding to the simpleType element information item in the [children], if present, otherwise the simple type definition resolved to by the normalized value of the `type` [attribute], if present, otherwise the simple ur-type definition
{scope}	If the attribute element information item has complexType as an ancestor, the complex definition corresponding to that item, otherwise (the attribute element information item is within a top-level attributeGroup definition), absent.
{value constraint}	If there is no `value` [attribute], then absent, otherwise a pair consisting of the normalized value (with respect to the {simple type definition}) of that [attribute] and *default*, if the normalized value of the `use` [attribute] is `default`, otherwise *fixed*.
{annotation}	The annotation corresponding to the annotation element information item in the [children], if present, otherwise absent.

otherwise (the attribute element information item has complexType or attributeGroup as an ancestor and the `ref` [attribute] is present), it corresponds to an attribute use pair of a boolean and an attribute declaration (unless `use='prohibited'`, in which case the item corresponds to nothing at all).

The boolean is *true* if the `use` [attribute] is present with normalized value `required`, otherwise *false*. The attribute declaration is the (global) attribute declaration resolved to by the normalized value of the `ref` [attribute].

Attribute declarations can appear at the top level of a schema document, or within complex types, either as complete (local) declarations, or by reference to top-level declarations. For complete declarations, top-level or local, the `type` attribute is used when the declaration can use a built-in or pre-declared simple type definition. Otherwise an anonymous simpleType is provided inline.

The default when no simple type definition is referenced or provided is the simple ur-type definition, which imposes no constraints at all.

Attribute items validated by a global declaration must be qualified with a namespace URI. Control over whether attribute items validated by a local declaration must be namespace-qualified or not is provided by the `form` [attribute], whose default is provided by the `attributeFormDefault` [attribute] on the enclosing schema, via its determination of {target namespace}.

The names for top-level attribute declarations are in their own symbol space. The names of locally-scoped attribute declarations with no {target namespace} reside in symbol spaces local to the type definition which contains them.

Example

```
<xs:attribute name="myAttribute"/>

<xs:attribute name="yetAnotherAttribute" type="xs:integer" use="required"/>

<xs:attribute name="anotherAttribute" use="default" value="42">
 <xs:simpleType>
  <xs:restriction base="xs:integer">
   <xs:minExclusive value="0"/>
  </xs:restriction>
 </xs:simpleType>
</xs:attribute>

<xs:attribute name="stillAnotherAttribute" type="xs:string" use="fixed" value="Hell
```

Four attributes are declared: one with no explicit constraints at all; two more each declared by reference t simple datatype `integer`, one required to be present in instances and one with a default and a subrange q a fourth with a fixed value.

Schema Representation Constraint: Attribute Declaration Representation OK
In addition to the conditions imposed on attribute element information items by the DTD and schema for schemas, the following must also hold:

1.1 If the item's parent is not schema, then

1.1.1 One of `ref` or `name` must be present, but not both;

1.1.2 If `ref` is present, then all of simpleType, `form`, `type` and `value` must be absent;

1.2 `type` and simpleType must not both be present;

1.3 The corresponding attribute declaration must satisfy the conditions set out in Attribute Declaration Constraints (§5.1).

4.3.2 XML Representation of Element Declaration Schema Components

The XML representation for an element declaration schema component is an element element information item. It specifies a type definition for an element either by reference or explicitly, and may provide occurrence and default information. The correspondences between the properties of the information item and properties of the component(s) it corresponds to are as follows:

XML Representation Summary: `element` Element Information Item

```
<element
  abstract = boolean : false
  block = #all or (possibly empty) subset of {substitution,
extension, restriction}
  default = string
  final = #all or (possibly empty) subset of {extension,
restriction}
  fixed = string
  form = qualified | unqualified
  id = ID
  maxOccurs = for maxOccurs : 1
  minOccurs = nonNegativeInteger : 1
  name = NCName
  nullable = boolean : false
  ref = QName
  substitutionGroup = QName
  type = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , ((simpleType | complexType)? , (key |
keyref | unique)*))
</element>
```

If the element element information item hasschema as its parent, the corresponding schema component is as follows:

Element Declaration Schema Component

Property	Representation
{name}	The normalized value of the `name` [attribute]
{target namespace}	The normalized value of the `targetNamespace` [attribute] of the parent schema element information item, or absent if there is none.
{scope}	*global*
{type definition}	The type definition corresponding to the simpleType or complexType element information item in the[children], if either is present, otherwise the type definition resolved to by the normalized value of the `type` [attribute], otherwise the {type definition} of the element declaration resolved to by the normalized value of the `substitutionGroup` [attribute], if present, otherwise the ur-type definition.
{nullable}	The normalized value of the `nullable` [attribute], if present, otherwise false.
{value constraint}	If there is a `default` or a `fixed` [attribute], then a pair consisting of the normalized value (with respect to the {type definition}, if it is a simple type definition, or the {type definition}'s {content type}, if that is a simple type definition, or else with respect to thesimple ur type) of that [attribute] and either *default* or *fixed*, as appropriate, otherwise absent.
{identity-constraint definitions}	A set consisting of the identity-constraint-definitions corresponding to all the key, unique and keyref element information items in the [children], if any, otherwise the empty set
{substitution group affiliation}	the element declaration resolved to by the normalized value of the `substitutionGroup` [attribute], if present, otherwise absent
{disallowed substitutions}	A set corresponding to the normalized normalized value of the `block`

{[illegible]}	[illegible] corresponding to the normalized [illegible] of the `block` [attribute], if present, otherwise on the normalized value of the `blockDefault` [attribute] of the parent schema element information item, if present, otherwise on the empty string, as follows: **the empty string** the empty set; **`#all`** {*substitution*, *extension*, *restriction*}; ***otherwise*** a set with members drawn from the set above, each being present or absent depending on whether the string contains an equivalently named space-delimited substring.
{substitution group exclusions}	As for {disallowed substitutions} above, but using the `final` [attribute] in place of the `block` [attribute] and with the relevant set being {*extension*, *restriction*}
{abstract}	The normalized value of the `abstract` [attribute], if present, otherwise false
{annotation}	The annotation corresponding to the annotation element information item in the [children], if present, otherwise absent

otherwise if the element element information item has complexType or group as an ancestor and the `ref` [attribute] is absent, the corresponding schema components are as follows (unless `minOccurs=maxOccurs=0`, in which case the item corresponds to no component at all):

Particle Schema Component

Property	Representation
{min occurs}	The numeric normalized value of the `minOccurs` [attribute], if present, otherwise `1`
{max occurs}	*unbounded*, if the `maxOccurs` [attribute] equals *unbounded*, otherwise the numeric normalized value of the `maxOccurs` [attribute], if present, otherwise `1`.
{term}	A (local) element declaration as given below

An element declaration as in the first case above, with the exception of its {target namespace} and {scope} properties, which are as below

Element Declaration Schema Component

Property	Representation
{target namespace}	If `form` is present and its normalized value is `qualified`, or if `form` is absent and the normalized value of `elementFormDefault` on the schema ancestor is `qualified`, then the normalized value of the `targetNamespace` [attribute] of the parent schema element information item, or absent if there is none, otherwise absent.
{scope}	If the element element information item has complexType as an ancestor, the complex definition corresponding to that item, otherwise (the element element information item is within a top-level group definition), absent.

otherwise (the element element information item has complexType or group as an ancestor and the `ref` [attribute] is present), the corresponding schema component is as follows (unless `minOccurs=maxOccurs=0`, in which case the item corresponds to no component at all):

Particle Schema Component	
Property	**Representation**
{min occurs}	The numeric normalized value of the `minOccurs` [attribute], if present, otherwise `1`
{max occurs}	*unbounded*, if the `maxOccurs` [attribute] equals *unbounded*, otherwise the numeric normalized value of the `maxOccurs` [attribute], if present, otherwise `1`.
{term}	The (global) element declaration resolved to by the normalized value of the `ref` [attribute]

element corresponds to an element declaration, and allows the type definition of that declaration to be specified either by reference or by explicit inclusion.

elements within schema produce *global* element declarations; elements within group or complexType produce either particles which contain *global* element declarations (if there's a `ref` attribute) or local declarations (otherwise). For complete declarations, top-level or local, the `type` attribute is used when the declaration can use a built-in or pre-declared type definition. Otherwise an anonymous simpleType or complexType is provided inline.

Element items validated by a global declaration must be qualified with a namespace URI. Control over whether element items validated by a local declaration must be namespace-qualified or not is provided by the `form` [attribute], whose default is provided by the `elementFormDefault` [attribute] on the enclosing schema, via its determination of {target namespace}.

> **Ed. Note:** Priority Feedback Request
> The provision of local element declarations is in part intended to simplify mapping between programming language and database structures where locally scoped name-type bindings are commonplace. It is a departure from XML 1.0 DTDs, in which the name-type binding for elements (but not for attributes) is constant across a document. The Working Group solicits feedback both on whether this aspect of the design is a good thing or not, and in particular on whether it does in fact simplify mappings as intended.

As noted above the names for top-level element declarations are in a separate symbol space from the symbol spaces for the names of type definitions, so there can (but need not be) a simple or complex type definition type with the same name as a top-level element. As with attribute names, the names of locally-scoped element declarations with no {target namespace} reside in symbol spaces local to the type definition which contains them.

Note that the above allows for two levels of defaulting for unspecified type definitions. An element with no referenced or included type definition will correspond to an element declaration which has the same type definition as the head of its substitution group if it identifies one, otherwise the ur-type definition.

See below at XML Representation of Identity-constraint Definition Schema Components (§4.3.8) for key, unique and keyref.

Example

```
<xs:element name="myelement" type="mySimpleType"/>

<xs:element name="et0" type="myComplexType"/>

<xs:element name="et1">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="et0"/>
   . . .
  <xs:sequence>
  <xs:attribute ...>. . .</xs:attribute>
 </xs:complexType>
</xs:element>

<xs:element name="et2">
 <xs:complexType content="empty">
  <xs:attribute ...>. . .</xs:attribute>
 </xs:complexType>
</xs:element>
```

The first two examples above declare elements by reference to a simple and a complex type definition respectively. The third and fourth use embedded anonymous complex type definitions, the first of which in turn refers to one of the top-level element declarations in its content model.

```
<xs:element name="contextOne">
 <xs:complexType>
  <xs:element name="myLocalelement" type="myFirstType"/>
  <xs:element ref="globalelement"/>
 </xs:complexType>
</xs:element>

<xs:element name="contextTwo">
 <xs:complexType>
  <xs:element name="myLocalelement" type="mySecondType"/>
  <xs:element ref="globalelement"/>
 </xs:complexType>
</xs:element>
```

Instances of `myLocalelement` within `contextOne` will be constrained by `myFirstType`, while those within `contextTwo` will be constrained by `mySecondType`.

NOTE: The possibility that differing attribute declarations 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.

Example

```
<xs:complexType name="facet">
 <xs:complexContent
  <xs:extension base="xs:annotated">
   <xs:attribute name="value" minOccurs="1"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:element name="facet" type="xs:facet" abstract="true"/>

<xs:element name="encoding" substitutionGroup="xs:facet">
 <xs:complexType
  <xs:complexContent>
   <xs:restriction base="xs:facet">
    <xs:attribute name="value" type="xs:encodings"/>
   </xs:restriction>
  </xs:complexContent>
 </xs:complexType>
</xs:element>

<xs:element name="period" substitutionGroup="xs:facet">
 <xs:complexType>
  <xs:complexContent>
   <xs:restriction base="xs:facet">
    <xs:attribute name="value" type="xs:timeDuration"/>
   </xs:restriction>
  </xs:complexContent>
 </xs:complexType>
</xs:element>

<xs:complexType name="datatype">
 <xs:sequence>
  <xs:element ref="facet" minOccurs="0" maxOccurs="unbounded"/>
 </xs:sequence>
 <xs:attribute name="name" type="xs:NCName" minOccurs="0"/>
 . . .
</xs:complexType>
```

An example from the schema for datatypes from[XML Schemas: Datatypes]. The `facet` type is defined and the `facet` element is declared to use it. The `facet` element is abstract -- it's *only* defined to stand as the head for a substitution group. Two further elements are declared, each a member of the `facet` substitution group. Finally a type is defined which refers to`facet`, thereby allowing *either* `period` or `encoding` (or any other member of the group).

Schema Representation Constraint: Element Declaration Representation OK

In addition to the conditions imposed on element element information items by the schema for schemas, the following must also hold:

1.1 `default` and `fixed` must not both be present;

1.2 If the item's parent is not schema, then

1.2.1 One of `ref` or `name` must be present, but not both;

1.2.2 If `ref` is present, then all of complexType, simpleType, key, keyref, unique, `nullable`, `default`, `fixed`, `block` and `type` must be absent, i.e. only `minOccurs`, `maxOccurs`, `id` are allowed in addition to `ref`, along with annotation;

1.3 `type` and either simpleType or complexType are mutually exclusive;

1.4 The corresponding particle and/or element declarations must satisfy the conditions set out in Element Declaration Constraints (§5.2) and Particle Constraints (§5.10).

4.3.3 XML Representation of Complex Type Definition Schema Components

The XML representation for a complex type definition schema component is a complexType element information item. It provides validation information for the [attributes] and [children] of an element information item in the form of attribute declarations and a content type.

The XML representation for complex type definitions with a simple type definition {content type} is significantly different from that of those with other {content type}s, and this is reflected in the presentation below, which displays first the elements involved in the first case, then those for the second. The property mapping is shown once for each case.

XML Representation Summary: `complexType` Element Information Item

```
<complexType
  abstract = boolean : false
  block = #all or (possibly empty) subset of {extension,
restriction}
  final = #all or (possibly empty) subset of {extension,
restriction}
  id = ID
  mixed = boolean : false
  name = NCName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , (simpleContent | complexContent | ((group
| all | choice | sequence)? , ((attribute | attributeGroup)* ,
anyAttribute?))))
</complexType>
```

Whichever alternative for the content of complexType is chosen, the following property mappings obtain:

Complex Type Definition Schema Component	
Property	**Representation**
{name}	The normalized value of the `name` [attribute] if present, otherwise absent
{target namespace}	The normalized value of the `targetNamespace` [attribute] of the schema ancestor element information item if present, otherwise absent
{abstract}	The normalized value of the `abstract` [attribute], if present, otherwise false
{prohibited-substitutions}	A set corresponding to the normalized value of the `block` [attribute], if present, otherwise on the normalized value of the `blockDefault` [attribute] of the parent schema element information item, if present, otherwise on the empty string, as follows: **the empty string** the empty set; **`#all`** {*extension*, *restriction*}; ***otherwise*** a set with members drawn from the set above, each being present or absent depending on whether the string contains an equivalently named space-delimited substring. **NOTE:** Although the `blockDefault` [attribute] of schema may include the value *substitution*, this value is ignored in the determination of {prohibited-substitutions} for type definitions (it *is* used in the determination of {disallowed substitutions} for element declarations)
{final}	As for {prohibited-substitutions} above, using the `final` [attribute] in place of the `block` [attribute]
{annotations}	The annotations corresponding to the annotation element information item in the [children], if present, in the simpleContent and complexContent [children], if present, and in their restriction and extension [children], if present, otherwise absent

When the simpleContent alternative is chosen, the following elements are relevant, and the remaining property mappings are as below. Note that either restriction or extension must be chosen as the content of simpleContent.

```
<simpleContent
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , (restriction | extension))
</simpleContent>
```

```
<restriction
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , ((duration | encoding | enumeration |
length | maxExclusive | maxInclusive | maxLength | minExclusive |
minInclusive | minLength | pattern | period | precision | scale)*)?
, ((attribute | attributeGroup)* , anyAttribute?))
</restriction>
```

```
<extension
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , ((attribute | attributeGroup)* ,
anyAttribute?))
</extension>
```

```
<attributeGroup
  id = ID
  ref = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</attributeGroup>
```

```
<anyAttribute
  id = ID
  namespace = ##any | ##other | list of {uri, ##targetNamespace,
##local} : ##any
  processContents = skip | lax | strict : strict
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</anyAttribute>
```

Complex Type Definition with simple content Schema Component	
Property	**Representation**
{base type definition}	The type definition resolved to by the normalized value of the `base` [attribute]
{derivation method}	If the restriction alternative is chosen, then *restriction*, otherwise (the extension alternative is chosen) *extension*
{attribute declarations}	The union of the sets of attribute use pairs as follows 1 the set of pairs corresponding to the attribute [children], if any 2 the {attribute declarations} of the attribute groups resolved to by the normalized values of the `ref` [attribute] of the attributeGroup [children], if any 3 if the type definition resolved to by the normalized value of the `base` [attribute] is a complex type definition, the {attribute declarations} of that type definition, unless the restriction alternative is chosen, in which case those members of the that type definition's {attribute declarations} whose attribute declaration's {name} and {target namespace} are the same as the declaration of an attribute use pair which would be in the set per clause 1 or 2 above are not included.
{attribute wildcard}	1 If there are no attributeGroup [children] corresponding to attribute groups with non-absent {attribute wildcard}s, then 1.1 if there is an anyAttribute present, a wildcard based on the normalized values of the `namespace` and `processContents` [attributes] and the annotation [children], exactly as for the wildcard corresponding to an any element as set out in XML Representation of Wildcard Schema Components (§4.3.7), 1.2 otherwise if the type definition resolved to by the normalized value of the `base` [attribute] is a complex type definition with a {attribute wildcard}, then that {attribute wildcard}, 1.3 otherwise absent. 2 Otherwise a wildcard whose {process contents} and {annotation} are those of a wildcard as defined in 1.1 above, and whose {namespace constraint} is the intensional intersection of the {namespace constraint} of a wildcard as defined in 1.1 above and all the non-absent {attribute wildcard}s of the attribute groups corresponding to the attributeGroup [children], as defined in Attribute Wildcard Intersection (§5.4).
{content type}	1 if the type definition resolved to by the normalized value of the `base` [attribute] is a complex type definition (whose own {content type} must be a simple type definition, see below) and the restriction alternative is chosen, then a simple type definition which restricts that simple type definition with the the set of facet components corresponding to the non-attribute-related [children] (those covered by the alternation after annotation in the content model for restriction above), if any, as defined in Simple Type Restriction (Facets) (§4.3.11); 2 otherwise (the type definition resolved to by the normalized value of the `base` [attribute] is a simple type definition and the the extension alternative is chosen), then that simple type definition;

When the complexContent alternative is chosen, the following elements are relevant (as are the attributeGroup and anyAttribute elements, not repeated here), and the additional property mappings are as below. Note that either restriction or extension must be chosen as the content of complexContent, but their content models are different in this case from the case above when they occur as children of simpleContent.

The property mappings below are *also* used in the case where the third alternative (neither simpleContent nor complexContent) is chosen. This case is understood as shorthand for complex content restricting the ur-type definition, and the details of the mappings should be modified as necessary.

```
<complexContent
  id = ID
  mixed = boolean
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , (restriction | extension))
</complexContent>
```

```
<restriction
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , (group | all | choice | sequence)? ,
((attribute | attributeGroup)* , anyAttribute?))
</restriction>
```

```
<extension
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , ((group | all | choice | sequence)? ,
((attribute | attributeGroup)* , anyAttribute?)))
</extension>
```

Complex Type Definition with complex content Schema Component

Property	Representation
{base type definition}	The type definition resolved to by the normalized value of the `base` [attribute]
{derivation method}	If the restriction alternative is chosen, then *restriction*, otherwise (the extension alternative is chosen) *extension*
{attribute declarations}	The union of the sets of attribute use pairs as follows: 1 the set of pairs corresponding to the attribute [children], if any 2 the {attribute declarations} of the attribute groups resolved to by the normalized values of the `ref` [attribute] of the attributeGroup [children], if any 3 the {attribute declarations} of the type definition resolved to by the normalized value of the `base` [attribute], unless the restriction alternative is chosen, in which case those members of that type definition's {attribute declarations} whose attribute declaration's {name} and {target namespace} are the same as a declaration which would be in a pair in the set per clause 1 or 2 above are not included.
{attribute wildcard}	As above for the simpleContent alternative

{content type}

1 If the restriction alternative is chosen, then

1.1 If there is no group, all, choice or sequence among the [children], then *empty*;

1.2 otherwise a pair consisting of

- if the `mixed` [attribute] is present on complexContent, then *mixed* if its normalized value is `true`, otherwise *elementOnly*;

  otherwise if the `mixed` [attribute] is present on complexType and its normalized value is `true`, then *mixed*;

  otherwise *elementOnly*.
- the particle corresponding to the all, choice, group or sequence among the [children]

2 otherwise (the extension alternative is chosen), [Definition:] let the **explicit content** be the particle corresponding to the all, choice, group or sequence among the [children], if any, otherwise *empty*.

2.1 if the explicit content is *empty*, then the {content type} of the type definition resolved to by the normalized value of the `base` [attribute]

otherwise

2.2 if the type definition resolved to by the normalized value of the `base` [attribute] has a {content type} of *empty*, then a pair of *mixed* or *elementOnly* (determined as above) and the explicit content itself

otherwise

2.3 a pair of *mixed* or *elementOnly* (determined as above) and a particle whose properties are as follows:

{min occurs}
`1`
{max occurs}
`1`
{term}
A model group whose {compositor} is *sequence* and whose {particles} are the particle of the {content type} of the type definition resolved to by the normalized value of the `base` [attribute] followed by the explicit content.

Schema Representation Constraint: Complex Type Definition Representation OK
In addition to the conditions imposed on complexType element information items by the schema for schemas, the following must also hold:

1.1 If the complexContent alternative is chosen, the type definition resolved to by the normalized value of the `base` [attribute] must be a complex type definition;

1.2 If the simpleContent alternative is chosen, the type definition resolved to by the normalized value of the `base` [attribute] must be either a complex type definition whose {base type definition} is a simple type definition or, only if the extension alternative is also chosen, a simple type definition;

1.3 The corresponding complex type definition component must satisfy the conditions set out in Complex Type Definition Constraints (§5.11);

1.4 If clause 2 in the correspondence specification above for {attribute wildcard} obtains, the intensional intersection must be expressible, as defined in Attribute Wildcard Intersection (§5.4).

NOTE: Aside from the simple coherence requirements enforced above, constraining type definitions identified as restrictions to actually *be* restrictions, that is, to schema-validate a subset of the what is schema-validated by their base type definition, is enforced in Complex Type Definition Constraints (§5.11).

Careful consideration of the above concrete syntax reveals that a type definition need consist of no more than a name, i.e. that `<complexType name="anything"/>` is allowed. In this case the type definition constructed is a named copy of the ur-type definition, as it is derived therefrom by (vacuous) restriction.

Example

```
<xs:complexType name="length1">
 <xs:simpleContent>
  <xs:extension base="xs:non-negative-integer">
   <xs:attribute name="unit" type="xs:NMTOKEN"/>
  </xs:extension>
 </xs:simpleContent>
</xs:complexType>

<xs:element name="width" type="length1"/>

  <width unit="cm">2.54</width>

<xs:complexType name="length2">
 <xs:complexContent>
  <xs:restriction base="xs:anyType">
   <xs:sequence>
    <xs:element name="size" type="xs:non-positive-integer"/>
    <xs:element name="unit" type="xs:NMTOKEN"/>
   </xs:sequence>
  </xs:restriction>
 </xs:complexContent>
</xs:complexType>

<xs:element name="depth" type="length2"/>

  <depth>
   <size>2.54</size><unit>cm</unit>
  </depth>

<xs:complexType name="length3">
 <xs:sequence>
  <xs:element name="size" type="xs:non-positive-integer"/>
  <xs:element name="unit" type="xs:NMTOKEN"/>
 </xs:sequence>
</xs:complexType>
```

Three approaches to defining a type for length: one with character data content constrained by referenc

Three approaches to defining a type for length: one with character data content constrained by reference to a built-in datatype, and one attribute, the other two using two elements. `length3` is the abbreviated alternative to `length2`: they correspond to identical type definition components.

Example

```
<xs:complexType name="personName">
 <xs:sequence>
  <xs:element name="title" minOccurs="0"/>
  <xs:element name="forename" minOccurs="0" maxOccurs="unbounded"/>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="extendedName">
 <xs:complexContent>
  <xs:extension base="personName">
   <xs:sequence>
    <xs:element name="generation" minOccurs="0"/>
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:element name="addressee" type="extendedName"/>

  <addressee>
   <forename>Albert</forename>
   <forename>Arnold</forename>
   <surname>Gore</surname>
   <generation>Jr</generation>
  </addressee>
```

A type definition for personal names, and a definition derived by extension which adds a single element; an element declaration referencing the derived definition, and a valid instance thereof.

Example

```
<xs:complexType name="simpleName">
 <xs:complexContent>
  <xs:restriction base="personName">
   <xs:sequence>
    <xs:element name="title" maxOccurs="0"/>
    <xs:element name="forename" minOccurs="1" maxOccurs="1"/>
    <xs:element name="surname"/>
   </xs:sequence>
  </xs:restriction>
 </xs:complexContent>
</xs:complexType>

<xs:element name="who" type="simpleName"/>

   <who>
    <forename>Bill</forename>
    <surname>Clinton</surname>
   </who>
```

A simplified type definition derived from the base type from the previous example by restriction, eliminating one optional daughter and fixing another to occur exactly once; an element declared by reference to it, and a valid instance thereof.

Example

```
<xs:complexType name="paraType" mixed="true">
 <xs:choice minOccurs="0" maxOccurs="unbounded">
  <xs:element ref="emph"/>
  <xs:element ref="strong"/>
 </xs:choice>
 <xs:attribute name="version" type="xs:decimal"/>
</xs:complexType>
```

A further illustration of the abbreviated form, with the `mixed` attribute appearing on `complexType` itself.

4.3.4 XML Representation of Attribute Group Definition Schema Components

The XML representation for an attribute group definition schema component is an attributeGroup element information item. It provides for naming a group of attribute declarations and an attribute wildcard for use by reference in the XML representation of complex type definitions and other attribute group definitions. The correspondences between the properties of the information item and properties of the component it corresponds to are as follows:

XML Representation Summary: `attributeGroup` Element Information Item

```
<attributeGroup
  id = ID
  name = NCName
  ref = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , ((attribute | attributeGroup)* ,
anyAttribute?))
</attributeGroup>
```

When an attributeGroup appears as a daughter of schema, it corresponds to an attribute group definition as below. When it appears as a daughter of complexType or attributeGroup, it does not correspond to any component as such.

Attribute Group Definition Schema Component

Property	Representation
{name}	The normalized value of the `name` [attribute]
{target namespace}	The normalized value of the `targetNamespace` [attribute] of the parent `schema` element information item.
{attribute declarations}	The union of the set of attribute use pairs corresponding to the attribute [children], if any, with the {attribute declarations} of the attribute groups resolved to by the normalized values of the `ref` [attribute] of the attributeGroup [children], if any.
{attribute wildcard}	As for {attribute wildcard} as described in XML Representation of Complex Type Definition Schema Components (§4.3.3), except that clause 1.2 is irrelevant and cannot obtain
{annotation}	The annotation corresponding to the annotation element information item in the [children], if present, otherwise absent

Example

```
<xs:attributeGroup name="myAttrGroup">
    <xs:attribute .../>
    ...
</xs:attributeGroup>

<xs:complexType name="myelement" content="empty">
    <xs:attributeGroup ref="myAttrGroup"/>
</xs:complexType>
```

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

The example above illustrates a pattern which recurs in the XML representation of schemas: The same element, in this case attributeGroup, serves both to define and to incorporate by reference. In the first case the `name` attribute is required, in the second the `ref` attribute is required, and the element must be empty. These two are mutually exclusive, and also conditioned by context: the defining form, with a `name`, must occur at the top level of a schema, whereas the referring form, with a `ref`, must occur within a complex type definition or an attribute group definition.

Schema Representation Constraint: Attribute Group Definition Representation OK
In addition to the conditions imposed on attributeGroup element information items by the schema for schemas, the following must also hold:

1.1 The corresponding attribute group definition, if any, must satisfy the conditions set out in Attribute Group Definition Constraints (§5.4).

1.2 If clause 2 in the correspondence specification in XML Representation of Complex Type Definition Schema Components (§4.3.3) for {attribute wildcard}, as referenced above, obtains, the intensional intersection must be expressible, as defined in Attribute Wildcard Intersection (§5.4).

4.3.5 XML Representation of Model Group Definition Schema Components

The XML representation for a model group definition schema component is a group element information item. It provides for naming a model group for use by reference in the XML representation of complex type definitions and model groups. The correspondences between the properties of the information item and properties of the component it corresponds to are as follows:

XML Representation Summary: `group` Element Information Item

```
<group
  id = ID
  maxOccurs = for maxOccurs : 1
  minOccurs = nonNegativeInteger : 1
  name = NCName
  ref = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , (all | choice | sequence)?)
</group>
```

If there is a `name` [attribute] (in which case the item will have schema as parent), then the item corresponds to a model group definition component with properties as follows:

Model Group Definition Schema Component

Property	Representation
{name}	The normalized value of the `name` [attribute]
{target namespace}	The normalized value of the `targetNamespace` [attribute] of the parent `schema` element information item
{model group}	A model group which is the {term} of a particle corresponding to the all, choice or sequence among the [children] (there must be one)
{annotation}	The annotation corresponding to the annotation element information item in the [children], if present, otherwise absent

Otherwise, the item will have `aref` [attribute], in which case it corresponds to a particle component with properties as follows (unless `minOccurs=maxOccurs=0`, in which case the item corresponds to no component at all):

Particle Schema Component

Property	Representation
{min occurs}	The normalized value of the `minOccurs` [attribute], if present, otherwise `1`
{max occurs}	*unbounded*, if the `maxOccurs` [attribute] equals *unbounded*, otherwise the numeric normalized value of the `maxOccurs` [attribute], if present, otherwise `1`.
{term}	the normalized value of the {model group} of the model group definition resolved to by the normalized value of the `ref` [attribute]

Given the constraints on its appearance in content models, an all should only occur as the only item in the [children]: see Model Group Constraints (§5.7).

Example

```
<xs:group name="myModelGroup">
 <xs:sequence>
  <xs:element ref="someThing"/>
 </xs:sequence>
</xs:group>

<xs:complexType name="trivial">
 <xs:sequence>
  <xs:group ref="myModelGroup"/>
 </xs:sequence>
 <xs:attribute .../>
</xs:complexType>

<xs:complexType name="moreSo">
 <xs:choice>
  <xs:element ref="anotherThing"/>
  <xs:group ref="myModelGroup"/>
 </xs:choice>
 <xs:attribute .../>
</xs:complexType>
```

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

Schema Representation Constraint: Model Group Definition Representation OK
In addition to the conditions imposed on group element information items by the schema for schemas, the following must also hold:

1 The corresponding model group definition, if any, must satisfy the conditions set out in Model Group Constraints (§5.7).

4.3.6 XML Representation of Model Group Schema Components

The XML representation for a model group schema component is either an all, a choice or a sequence element information item. The correspondences between the properties of those information items and properties of the component they correspond to are as follows:

XML Representation Summary: `all` Element Information Item

```
<all
  id = ID
  maxOccurs = for maxOccurs : 1
  minOccurs = nonNegativeInteger : 1
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , element*)
</all>
```

```
<choice
  id = ID
  maxOccurs = for maxOccurs : 1
  minOccurs = nonNegativeInteger : 1
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , (element | group | choice | sequence |
any)*)
</choice>
```

```
<sequence
  id = ID
  maxOccurs = for maxOccurs : 1
  minOccurs = nonNegativeInteger : 1
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , (element | group | choice | sequence |
any)*)
</sequence>
```

Each of the above items corresponds to a particle containing a model group, with properties as follows (unless `minOccurs=maxOccurs=0`, in which case the item corresponds to no component at all):

Particle Schema Component

Property	Representation
{min occurs}	The normalized value of the `minOccurs` [attribute], if present, otherwise `1`
{max occurs}	*unbounded*, if the `maxOccurs` [attribute] equals *unbounded*, otherwise the numeric normalized value of the `maxOccurs` [attribute], if present, otherwise `1`.
{term}	A model group as given below

Model Group Schema Component

Property	Representation
{compositor}	One of *all*, *choice*, *sequence* depending on the element information item
{particles}	a sequence of particles corresponding to all the all, choice, sequence, any, group or element items among the [children], in order
{annotation}	The annotation corresponding to the annotation element information item in the [children], if present, otherwise absent

Schema Representation Constraint: Model Group Representation OK

In addition to the conditions imposed on all, choice and sequence element information items by the schema for schemas, the following must also hold:

1 The corresponding particle and model group must satisfy the conditions set out in Model Group Constraints (§5.7) and Particle Constraints (§5.10).

4.3.7 XML Representation of Wildcard Schema Components

The XML representation for a wildcard schema component is an any or anyAttribute element information item. The correspondences between the properties of an any information item and properties of the components it corresponds to are as follows (see complexType and attributeGroup for the correspondences for anyAttribute):

XML Representation Summary: any Element Information Item

```
<any
  id = ID
  maxOccurs = for maxOccurs : 1
  minOccurs = nonNegativeInteger : 1
  namespace = ##any | ##other | list of {uri, ##targetNamespace,
##local} : ##any
  processContents = skip | lax | strict : strict
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</any>
```

A particle containing a wildcard, with properties as follows (unless `minOccurs=maxOccurs=0`, in which case the item corresponds to no component at all):

Particle Schema Component

Property	Representation
{min occurs}	The normalized value of the `minOccurs` [attribute], if present, otherwise `1`
{max occurs}	*unbounded*, if the `maxOccurs` [attribute] equals *unbounded*, otherwise the numeric normalized value of the `maxOccurs` [attribute], if present, otherwise `1`.
{term}	A wildcard as given below

Wildcard Schema Component

Property	Representation
{namespace constraint}	Dependent on the normalized value of the `namespace` [attribute]: if absent, then *any*, otherwise as follows: **##any** *any* **##other** a pair of *not* and the normalized value of the `targetNamespace` [attribute] of the schema ancestor element information item if present, otherwise absent ***otherwise*** a set whose members are namespace URIs corresponding to the space-delimited substrings of the string, except 1 if one such substring is `##targetNamespace`, the corresponding member is the normalized value of the `targetNamespace` [attribute] of the schema ancestor element information item if present, otherwise absent 2 if one such substring is `##local`, the corresponding member is absent

{process contents}	One of *lax*, *skip*, *strict*, corresponding to the normalized value of the `processContents` [attribute], if present, otherwise *strict*
{annotation}	The annotation corresponding to the annotation element information item in the [children], if present, otherwise absent

Wildcards are subject to the same ambiguity constraints (Unique Particle Attribution (§5.7)) as other content model particles: If an instance element could match either an explicit particle and a wildcard, or one of two wildcards, within the content model of a type, that model is in error.

Example

```
<xs:any processContents="skip"/>

<xs:any namespace="##other" processContents="lax"/>

<xs:any namespace="http://www.w3.org/1999/XSL/Transform"/>

<xs:any namespace="##targetNamespace"/>

<xs:anyAttribute namespace="http://www.w3.org/XML/1998/namespace"/>
```

Concrete examples of the four basic types of wildcard, plus one attribute wildcard.

Schema Representation Constraint: Wildcard Representation OK
In addition to the conditions imposed on any element information items by the schema for schemas, the following must also hold:

1 The corresponding particle and model group must satisfy the conditions set out in Model Group Constraints (§5.7) and Particle Constraints (§5.10).

4.3.8 XML Representation of Identity-constraint Definition Schema Components

The XML representation for an identity-constraint definition schema component is either a key, a keyref or a unique element information item. The correspondences between the properties of those information items and properties of the component they correspond to are as follows:

XML Representation Summary: `unique` Element Information Item

```
<unique
  id = ID
  name = NCName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , (selector , field+))
</unique>
```

```
<key
  id = ID
  name = NCName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , (selector , field+))
</key>
```

```
<keyref
  id = ID
  name = NCName
  refer = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , (selector , field+))
</keyref>
```

```
<selector
  id = ID
  xpath = An XPath expression
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</selector>
```

```
<field
  id = ID
  xpath = An XPath expression
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</field>
```

Identity-constraint Definition Schema Component

Property	Representation
{name}	The normalized value of the `name` [attribute]
{target namespace}	The normalized value of the `targetNamespace` [attribute] of the parent `schema` element information item.
{identity-constraint category}	One of *key*, *keyref* or *unique*, depending on the item
{selector}	An XPath expression corresponding to the normalized value of the selector element information item among the [children]
{fields}	A sequence of XPath expressions, corresponding to the normalized values of the field element information item [children], in order.
{referenced key}	If the item is a keyref, the identity-constraint definition resolved to by the normalized value of the `refer` [attribute], otherwise absent
{annotation}	The annotation corresponding to the annotation element information item in the [children], if present, otherwise absent

The `XPathExprApprox` simple type referenced above is defined in (normative) Schema for Schemas (§A). It is a permissive approximation to the syntax of XPath expressions as defined in [XPath], and is *not* an accurate reconstruction of that syntax. Its use by reference from other schema documents is deprecated: In due course a schema for XPath will be published which includes a simple type definition appropriate for widespread use.

Example

```
<xs:element name="state">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="stateCode" type="twoLetterCode"/>
   <xs:element name="vehicle">
    <xs:complexType>
     . . .
     <xs:attribute name="regNo" type="xs:integer"/>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>

<xs:element name="root"
 . . .
 <xs:key name="regKey">
  <xs:selector>.//vehicle[@regNo]</xs:selector>
  <xs:field>@regNo</xs:field>
  <xs:field>ancestor::state/stateCode</xs:field>
      <!-- scope needs to be involved -->
 </xs:key>
</xs:element>

<xs:element name="person">
 <xs:complexType>
  <xs:sequence>
   . . .
   <xs:element name="car">
    <xs:complexType model="empty">
     . . .
     <xs:attribute name="regRef" type="xs:integer"/>
     <xs:attribute name="regState" type="twoLetterCode"/>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>

 <xs:keyref name="carRef" refer="regKey">
  <xs:selector>.//car[@regRef]</xs:selector>
   <xs:field>@regRef</xs:field>
   <xs:field>@regState</xs:field>
 </xs:keyref>
</xs:element>
```

A `state` element is defined, which*inter alia* contains a `stateCode` descendant and some `vehicle` descendants. A `vehicle` in turn has a `regNo` attribute, which is an integer. The combination of `stateCode` and `regNo` is asserted to be a *key* for `vehicle` within `state`. Furthermore, a `person` element has inter-alia an empty`car` element, with `regRef` and `regState` attributes, which are then asserted together to refer to `vehicles` via the `regKey` constraint.

Schema Representation Constraint: Identity-constraint Definition Representation OK
In addition to the conditions imposed on key, keyref and unique element information items by the schema for

schemas, the following must also hold:

1 The corresponding identity-constraint definition must satisfy the conditions set out in Identity-constraint Definition Constraints (§5.3).

4.3.9 XML Representation of Notation Declaration Schema Components

The XML representation for a notation declaration schema component is a notation element information item. The correspondences between the properties of that information item and properties of the component it corresponds to are as follows:

XML Representation Summary: `notation` Element Information Item

```
<notation
  id = ID
  name = NCName
  public = A public identifier, per ISO 8879
  system = uriReference
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</notation>
```

Notation Declaration Schema Component

Property	Representation
{name}	The normalized value of the `name` [attribute]
{target namespace}	The normalized value of the `targetNamespace` [attribute] of the parent `schema` element information item.
{system identifier}	A URI reference corresponding to the normalized value of the `system` [attribute], if present, otherwise absent
{public identifier}	A URI reference corresponding to the normalized value of the `public` [attribute]
{annotation}	The annotation corresponding to the annotation element information item in the [children], if present, otherwise absent

Example

```
<xs:notation name="jpeg"
             public="image/jpeg" system="viewer.exe" />

<xs:element name="picture">
 <xs:complexType base="xs:binary" derivedBy="extension">
  <xs:attribute name="pictype" type="xs:NOTATION"/>
 </xs:complexType>
</xs:element>

<picture pictype="jpeg">...</picture>
```

Schema Representation Constraint: Notation Definition Representation OK
In addition to the conditions imposed on notation element information items by the schema for schemas, the following must also hold:

1 The corresponding notation definition must satisfy the conditions set out in Notation Declaration Constraints (§5.8).

4.3.10 XML Representation of Annotation Schema Components

Annotation of schemas and schema components, with material for human or computer consumption, is provided for by allowing application information and human information at the beginning of most major schema elements, and anywhere at the top level of schemas. The XML representation for an annotation schema component is an annotation element information item. The correspondences between the properties of that information item and properties of the component it corresponds to are as follows:

XML Representation Summary: `annotation` Element Information Item

```
<annotation>
  Content: (appinfo | documentation)*
</annotation>
```

```
<appinfo
  source = uriReference>
  Content: ({any})*
</appinfo>
```

```
<documentation
  source = uriReference>
  Content: ({any})*
</documentation>
```

Annotation Schema Component

Property	Representation
{application information}	A sequence of the appinfo element information items from among the [children], in order, if any, otherwise the empty sequence.
{user information}	A sequence of the documentation element information items from among the [children], in order, if any, otherwise the empty sequence.

Schema Representation Constraint: Annotation Definition Representation OK
In addition to the conditions imposed on annotation element information items by the schema for schemas, the following must also hold:

1 The corresponding annotation must satisfy the conditions set out in Annotation Constraints (§5.9).

4.3.11 (non-normative) XML Representation of Simple Type Definition Schema Components

NOTE: This section reproduces a version of material from [XML Schemas: Datatypes], for local cross-reference purposes.

XML Representation Summary: `simpleType` Element Information Item

```
<simpleType
  id = ID
  name = NCName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , ((list | restriction | union)))
</simpleType>
```

```
<restriction
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , (simpleType? , ((duration | encoding |
enumeration | length | maxExclusive | maxInclusive | maxLength |
minExclusive | minInclusive | minLength | pattern | period |
precision | scale)*)))
</restriction>
```

```
<list
  id = ID
  itemType = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , (simpleType?))
</list>
```

```
<union
  id = ID
  memberTypes = List of [anon]
  {any attributes with non-schema namespace . . .}>
  Content: (annotation? , (simpleType*))
</union>
```

Simple Type Definition Schema Component

Property	Representation
{name}	The normalized value of the `name` [attribute] if present, otherwise absent
{target namespace}	The normalized value of the `targetNamespace` [attribute] of the schema ancestor element information item if present, otherwise absent
{base type definition}	The type definition resolved to by the normalized value of the `base` [attribute], if present, otherwise the type definition corresponding to the simpleType among the [children] of restriction, if present, otherwise (the restriction alternative is not chosen), the simple ur-type definition
{variety}	If there is a list among the [children], then *list*, otherwise if there is a union among the [children], then *union*, otherwise (there is a restriction among the [children]), then the {variety} of the {base type definition}

If the {variety} is *atomic*, the following additional property mappings also apply:

Atomic Simple Type Definition Schema Component

Property	Representation
{primitive type definition}	The built-in primitive type definition from which the {base type definition} is derived.
{facets}	a set of facet components corresponding to the non-attribute-related [children] (those covered by the alternation after simpleType in the content model for restriction above).

If the {variety} is *list*, the following additional property mappings also apply:

List Simple Type Definition Schema Component	
Property	**Representation**
{item type definition}	Starting from the list among the [children] of simpleType or the list among the [children] of restriction, whichever is present, the type definition resolved to by the normalized value of the `itemType` [attribute], if present, otherwise the type definition corresponding to the simpleType among the [children].
{facets}	If the restriction alternative is chosen, a set of facet components corresponding to the non-attribute-related [children] (those covered by the alternation after simpleType in the content model for restriction above), otherwise the empty set.

If the {variety} is *union*, the following additional property mappings also apply:

Union Simple Type Definition Schema Component	
Property	**Representation**
{member type definitions}	[Definition:] Define the **explicit members** as follows: Starting from the union among the [children] of simpleType or the union among the [children] of restriction, whichever is present, the type definitions resolved to by the space-delimited items in the normalized value of the `memberTypes` [attribute], if any, followed by the type definitions corresponding to the simpleTypes among the [children], if any. The actual value is then formed by replacing any union type definition in the explicit members with its {member type definitions}.
{facets}	If the restriction alternative is chosen, a set of facet components corresponding to the non-attribute-related [children] (those covered by the alternation after simpleType in the content model for restriction above), otherwise the empty set.

When the {variety} is *atomic* or *list*, in the absence of an explicit or inherited `whitespace` facet, one is added with a value based on the {variety} as follows (*union* types have no `whitespace` facet, the `whitespace` facet of its members are what matters):

1. If the {variety} is *list*, then *collapse*
2. otherwise (the {variety} is *atomic*), then depending on the {primitive type definition}, as follows:
   1. if it is the simple ur type, then *preserve*;
   2. if it is the built-in primitive `string` datatype, and there are no explicit or inherited `enumeration` facets, then *replace*;
   3. otherwise, *collapse*.

Schema Representation Constraint: Simple Type Restriction (Facets)
A simple type definition (call it **R**) restricts another simple type definition (call it **B**) with a set of facets (call this **S**) if:

1.1 The {variety} and {primitive type definition} of **R** are the same as those of **B**;

1.2 For each facet in the {facets} of **B**, there is a a facet of the same kind in **R**, which is the facet of the same kind in **S**, if there is one, otherwise a facet of the same kind whose **{value}** is the **{value}** of the facet of the same kind in the {facets} of **B** and whose **{annotation}** is absent.

5 Schema Component Validity Constraints

This chapter presents the constraints each kind of schema component must satisfy to *be* a component.

5.1 Attribute Declaration Constraints

All attribute declarations (see Attribute Declaration Details (§3.2)) must satisfy the following constraints.

Constraint on Schemas: Attribute Declaration Properties Correct

1 The values of the properties of an attribute declaration must be as described in the property tableau in Attribute Declaration Details (§3.2), modulo the impact of Missing Sub-components (§7.3);

2 if there is a {value constraint},

2.1 its string must be schema-valid with respect to the {simple type definition} as defined in String Valid (§3.13);

2.2 its second part (*default* or *fixed*) must be present if and only if its {scope} is not `global`.

Constraint on Schemas: xmlns Not Allowed
The {name} of an attribute declaration must not match `xmlns`.

> **NOTE:** The {name} of an attribute is an NCName, which implicitly prohibits attribute declarations of the form `xmlns:*`

Constraint on Schemas: xsi: Not Allowed
The {target namespace} of an attribute declaration, whether local or global, must not match `http://www.w3.org/2000/10/XMLSchema-instance`.

> **NOTE:** This reinforces the special status of these attributes, so that they not only *need* not be declared to be allowed in instances, but *must* not be declared. It also removes any temptation to experiment with supplying global or fixed values for e.g. `xsi:type` or `xsi:null`, which would be seriously misleadingly, as they would have no effect.

5.2 Element Declaration Constraints

All element declarations (see Element Declaration Details (§3.3)) must satisfy the following constraint.

Constraint on Schemas: Element Declaration Properties Correct

1 The values of the properties of an element declaration must be as described in the property tableau in Element Declaration Details (§3.3), modulo the impact of Missing Sub-components (§7.3);

2 If there is a {value constraint}, its string must be schema-valid with respect to the {type definition} as defined in Element Default Valid (Immediate) (§5.2);

3 If there is an {substitution group affiliation}, the {type definition} of the element declaration must be validly derived from the {type definition} of the {substitution group affiliation}, given the value of the {substitution group exclusions} of the {substitution group affiliation}, as defined in Type Derivation OK (Complex) (§5.11) (if the {type definition} is complex) or given {*list*), as defined in Type Derivation OK (Simple) (§5.12) (if the {type definition} is simple).

The following constraints define relations appealed to elsewhere in this specification.

Constraint on Schemas: Element Default Valid (Immediate)
A string is a valid default with respect to a type definition if

1.1 The type definition is a simple type definition, and the string is schema-valid with respect to that definition as defined by String Valid (§3.13)

or

1.2 The type definition is a complex type definition whose {content type} is a simple type definition, and the string is schema-valid with respect to that simple type definition as defined by String Valid (§3.13)

or

1.3 The type definition is a complex type definition whose {content type} is *mixed*, and the {content type}'s particle is emptiable as defined by Particle Emptiable (§5.10).

Constraint on Schemas: Substitution Group OK (Transitive)
An element declaration (call it **D**) together with a blocking constraint (a subset of {*substitution*, *extension*, *restriction*}, the value of a {disallowed substitutions}) is validly substitutable for another element declaration (call it **C**) if

1.1 the blocking constraint does not contain *substitution*;

1.2 There is a chain of {substitution group affiliation}s from **D** to **C**, that is, either **D**'s {substitution group affiliation} is **C**, or **D**'s {substitution group affiliation}'s {substitution group affiliation} is **C**, or . . .;

1.3 The set of all {derivation method}s involved in the derivation of **D**'s {type definition} from **C**'s {type definition} does not intersect with the union of the blocking constraint, **C**'s {prohibited-substitutions} and the {prohibited-substitutions} of any intermediate {type definition}s in the derivation of **D**'s {type definition} from **C**'s {type definition}.

Constraint on Schemas: Substitution Group
[Definition:] Every element declaration in the {element declarations} of a schema defines a **substitution group**, a subset of those {element declarations}, as follows:

1.1 The element declaration itself is in the group;

1.2 the group is closed with respect to {substitution group affiliation}, that is, if any element declaration in the {element declarations} has a {substitution group affiliation} in the group, then it is also in the group itself.

5.3 Identity-constraint Definition Constraints

All identity-constraint definitions (see Identity-constraint Definition Details (§3.10)) must satisfy the following constraint.

Constraint on Schemas: Identity-constraint Definition Properties Correct

1 The values of the properties of a identity-constraint definition must be as described in the property tableau in Identity-constraint Definition Details (§3.10), modulo the impact of Missing Sub-components (§7.3);

2 If the {identity-constraint category} is *keyref*, the cardinality of the {fields} must equal that of the {fields} of the {referenced key}

5.4 Attribute Group Definition Constraints

All attribute group definitions (see Attribute Group Definition Details (§3.5)) must satisfy the following constraint.

Constraint on Schemas: Attribute Group Definition Properties Correct

1 The values of the properties of an attribute group definition must be as described in the property tableau in Attribute Group Definition Details (§3.5), modulo the impact of Missing Sub-components (§7.3);

2 Two distinct members of the {attribute declarations} may not contain attribute declarations both of whose {name}s match and whose {target namespace}s are identical.

The following constraint defines a relation appealed to elsewhere in this specification.

Constraint on Schemas: Attribute Wildcard Intersection
An wildcard's {namespace constraint} value is the intensional intersection of two other such values (call them **O1** and **O2**) if it is the value determined as follows

1.1 If **O1** and **O2** are the same value, then that value;

1.2 Otherwise if either **O1** or **O2** is *any*, then the other;

1.3 Otherwise if either **O1** or **O2** is a pair of *not* and a namespace URI and the other is a set of (namespace URIs or absent), then that set, minus the negated namespace URI if it was in the set;

1.4 Otherwise if both **O1** and **O2** are sets of (namespace URIs or absent), then the intersection of those sets;

1.5 Otherwise (the two are negations of different namespace URIs) the intersection is not expressible.

In the case where there are more than two values, the intensional intersection is determined by identifying the intensional intersection of two of the values as above, then the intensional intersection of that value with the third (providing the first intersection was expressible), and so on as required.

5.5 Wildcard Constraints

All wildcards (see Wildcard Details (§3.9)) must satisfy the following constraint.

Constraint on Schemas: Wildcard Properties Correct

The values of the properties of a wildcard must be as described in the property tableau in Wildcard Details (§3.9), modulo the impact of Missing Sub-components (§7.3).

The following constraint defines a relation appealed to elsewhere in this specification.

Constraint on Schemas: Wildcard Subset
A namespace constraint (call it **sub**) is an intensional subset of another namespace constraint (call it **super**) if

1.1 **super** is *any*

or

1.2 **sub** is a pair of *not* and a namespace URI or absent and super is a pair of *not* and the same value;

or

1.3 **sub** is a set whose members are either namespace URIs or absent and either

1.3.1 **super** is the same set or a superset thereof

or

1.3.2 **super** is a pair of *not* and a namespace URI or absent and that value is not in **sub**'s set.

5.6 Model Group Definition Constraints

All model group definitions (see Model Group Definition Details (§3.6)) must satisfy the following constraint.

Constraint on Schemas: Model Group Definition Properties Correct

The values of the properties of a model group definition must be as described in the property tableau in Model Group Definition Details (§3.6), modulo the impact of Missing Sub-components (§7.3).

5.7 Model Group Constraints

All model groups (see Model Group Details (§3.7)) must satisfy the following constraints.

Constraint on Schemas: Model Group Correct

The values of the properties of a model group must be as described in the property tableau in Model Group Details (§3.7), modulo the impact of Missing Sub-components (§7.3).

Constraint on Schemas: All Group Limited

A model group with {compositor} *all* must only appear either

1.1 as the model group of a model group definition

or

1.2 in a particle with {min occurs}={max occurs}=1, and that particle must be part of a pair which constitutes the {content type} of a complex type definition.

Furthermore, the {max occurs} of all the particles in the {particles} of an *all* group must be 0 or 1.

Constraint on Schemas: Element Declarations Consistent

If the {particles} contains, either directly, indirectly (that is, within the {particles} of a contained model group, recursively) or implicitly two or more element declaration particles with the same {name} and {target namespace}, all their {type definition}s must be the same.

[Definition:] We say that a list of particles **implicitly contains** an element declaration if a member of the list contains that element declaration in its substitution group.

Constraint on Schemas: Unique Particle Attribution

A content model must be formed such that during schema validation of an element information item sequence, the particle contained directly, indirectly or implicitly therein with which to attempt to schema-validate each item in the sequence in turn can be uniquely determined without examining the content or attributes of that item, and without any information about the items in the remainder of the sequence.

> **NOTE:** This constraint reconstructs for XML Schema the equivalent constraints of [XML] and SGML. Given the presence of element substitution groups and wildcards, the concise expression of this constraint is difficult, see (non-normative) Analysis of the Unique Particle Attribution constraint (§F) for further discussion.

> **NOTE:** Because locally-scoped element declarations may or may not have a {target namespace}, the scope of declarations is *not* relevant to enforcing either of the two preceding constraints.

The following constraints define relations appealed to elsewhere in this specification.

Constraint on Schemas: Effective Total Range (all and sequence)

The effective total range of a particle whose {term} is a group whose {compositor} is *all* or *sequence* is a pair of minimum and maximum, as follows

minimum

The product of the particle's {min occurs} and the sum of the {min occurs} of every wildcard or element declaration particle in the group's {particles} and the minimum part of the effective total range of each of the group particles in the group's {particles} (or 0 if there are no {particles})

maximum

unbounded if the {max occurs} of any wildcard or element declaration particle in the group's {particles} or the maximum part of the effective total range of any of the group particles in the group's {particles} is *unbounded*, or if any of those is non-zero and the {max occurs} of the particle itself is *unbounded*, otherwise the product of the particle's {max occurs} and the sum of the {max occurs} of every wildcard or element declaration particle in the group's {particles} and the maximum part of the effective total range

of each of the group particles in the group's {particles} (or `0` if there are no {particles})

Constraint on Schemas: Effective Total Range (choice)
The effective total range of a particle whose {term} is a group whose {compositor} is *choice* is a pair of minimum and maximum, as follows

minimum
The product of the particle's {min occurs} and the minimum of the {min occurs} of every wildcard or element declaration particle in the group's {particles} and the minimum part of the effective total range of each of the group particles in the group's {particles} (or `0` if there are no {particles})

maximum
unbounded if the {max occurs} of any wildcard or element declaration particle in the group's {particles} or the maximum part of the effective total range of any of the group particles in the group's {particles} is *unbounded*, or if any of those is non-zero and the {max occurs} of the particle itself is *unbounded*, otherwise the product of the particle's {max occurs} and the maximum of the {max occurs} of every wildcard or element declaration particle in the group's {particles} and the maximum part of the effective total range of each of the group particles in the group's {particles} (or `0` if there are no {particles})

5.8 Notation Declaration Constraints

All notation declarations (see Notation Declaration Details (§3.11)) must satisfy the following constraint.

Constraint on Schemas: Notation Declaration Correct
The values of the properties of a notation declaration must be as described in the property tableau in Notation Declaration Details (§3.11), modulo the impact of Missing Sub-components (§7.3).

5.9 Annotation Constraints

All annotations (see Annotation Details (§3.12)) must satisfy the following constraint.

Constraint on Schemas: Annotation Correct
1 The values of the properties of an annotation must be as described in the property tableau in Annotation Details (§3.12), modulo the impact of Missing Sub-components (§7.3);
2 The normalized value of the `xml:lang` [attribute], if present for any documentation element information items in {user information}, must conform to the requirements set out in Language Identification (§ 2.12) in [XML].

5.10 Particle Constraints

All particles (see Particle Details (§3.8)) must satisfy the following constraints.

Constraint on Schemas: Particle Correct
1 The values of the properties of a particle must be as described in the property tableau in Particle Details (§3.8), modulo the impact of Missing Sub-components (§7.3);
2 If {max occurs} is not *unbounded*, that is, it has a numeric value, then
2.1 {min occurs} must not be greater than {max occurs};
2.2 {max occurs} must greater than or equal to 1;

The following constraints define relations appealed to elsewhere in this specification.

Constraint on Schemas: Particle Valid (Extension)
[Definition:] A particle (call it **E**, for extension) is a **valid extension** of another particle (call it **B**, for base) if either

1.1 They are the same particle

or

1.2 **E**'s {min occurs}={max occurs}=1 and its {term} is a *sequence* group whose {particles}' first member is a particle all of whose properties, recursively, are identical to those of **B**, with the exception of **{annotation}** properties.

Constraint on Schemas: Particle Valid (Restriction)
[Definition:] A particle (call it **R**, for restriction) is a **valid restriction** of another particle (call it **B**, for base) if either

1.1 They are the same particle

or

1.2 depending on the kind of particle, per the table below, with the qualification that any global element declara **B**) which is the {substitution group affiliation} of one or more other element declarations is treated as if it w whose {min occurs} and {max occurs} are those of the particle, and whose {particles} consists of one partic occurs} and {max occurs} of 1 for the global element declaration and for each of the declarations in itssubs

		Base Particle			
		elt	**any**	**all**	**choice**
Derived Particle	**elt**	NameAndTypeOK	NSCompat	RecurseAsIfGroup	RecurseAsIfGroup
	any	Forbidden	NSSubset	Forbidden	Forbidden
	all	Forbidden	NSRecurseCheckCardinality	Recurse	Forbidden
	choice	Forbidden	NSRecurseCheckCardinality	Forbidden	RecurseLax
	sequence	Forbidden	NSRecurseCheckCardinality	RecurseUnordered	MapAndSum

Constraint on Schemas: Occurrence Range OK
A particle's occurrence range is a valid restriction of another's occurrence range if

1.1 Its {min occurs} is greater than or equal to the other's {min occurs};

1.2 Either

1.2.1 The other's {max occurs} is *unbounded*

or

1.2.2 both {max occurs} are numbers, and the particle's is less than or equal to the other's.

Constraint on Schemas: Particle Restriction OK (Elt:Elt -- NameAndTypeOK)
An element declaration particle is avalid restriction of another element declaration particle if

1.1 The declarations' {name}s, {target namespace}s and {nullable} are the same;

1.2 **R**'s occurrence range is a valid restriction of **B**'s occurrence range as defined by Occurrence Range OK (§5.10);

1.3 either **B**'s declaration's {value constraint} absent, or is not *fixed*, or **R**'s declaration's {value constraint} is *fixed* with the same string;

1.4 **R**'s declaration's {identity-constraint definitions} is a subset of **B**'s declaration's {identity-constraint definitions}, if any.

1.5 **R**'s declaration's {disallowed substitutions} is a superset of **B**'s declaration's {disallowed substitutions}.

1.6 **R**'s {type definition} is validly derived given {*list*, *extension*} from **B**'s {type definition} as defined by Type Derivation OK (Complex) (§5.11) or Type Derivation OK (Simple) (§5.12), as appropriate.

NOTE: The above constraint on {type definition} means that in deriving a type by refinement, any contained type definitions must themselves be explicitly derived by refinement from the corresponding type definitions in the base definition.

Constraint on Schemas: Particle Derivation OK (Elt:Any -- NSCompat)
An element declaration particle is a valid restriction of a wildcard particle if

1.1 The element declaration's {target namespace} is schema-valid with respect to the wildcard's {namespace constraint} as defined by Wildcard allows Namespace URI (§3.9)

1.2 **R**'s occurrence range is a valid restriction of **B**'s occurrence range as defined by Occurrence Range OK (§5.10);

Constraint on Schemas: Particle Derivation OK (Elt:All/Choice/Sequence -- RecurseAsIfGroup)
An element declaration particle is a valid restriction of a group particle (*all*, *choice* or *sequence*) if a group particle of the variety corresponding to **B**'s, with {min occurs} and {max occurs} of `1` and with {particles} consisting of a single particle the same as the element declaration is a valid restriction of the group as defined by Particle Derivation OK (All:All,Sequence:Sequence -- Recurse) (§5.10), Particle Derivation OK (Choice:Choice -- RecurseLax) (§5.10) or Particle Derivation OK (All:All,Sequence:Sequence -- Recurse) (§5.10), depending on whether the group is *all*, *choice* or *sequence*.

Constraint on Schemas: Particle Derivation OK (Any:Any -- NSSubset)
A wildcard particle is a valid restriction of another wildcard particle if

1.1 **R**'s occurrence range is a valid restriction of **B**'s occurrence range as defined by Occurrence Range OK (§5.10);

1.2 **R**'s {namespace constraint} is an intensional subset of **B**'s {namespace constraint} as defined by Wildcard Subset (§5.5).

Constraint on Schemas: Particle Derivation OK (All/Choice/Sequence:Any -- NSRecurseCheckCardinality)
A group particle is a valid restriction of a wildcard particle if

1.1 Every member of the {particles} of the group is a valid restriction of the wildcard as defined by Particle Valid (Restriction) (§5.10)

1.2 The effective total range of the group, as defined by Effective Total Range (all and sequence) (§5.7) (if the group is *all* or *sequence*) or Effective Total Range (choice) (§5.7) (if it is *choice*) is a valid restriction of **B**'s occurrence range as defined by Occurrence Range OK (§5.10).

Constraint on Schemas: Particle Derivation OK (All:All,Sequence:Sequence -- Recurse)
An *all* or *sequence* group particle is a valid restriction of another group particle with the same {compositor} if

1.1 **R**'s occurrence range is a valid restriction of **B**'s occurrence range as defined by Occurrence Range OK (§5.10);

1.2 there is a complete order-preserving functional mapping from the particles in the {particles} of **R** to the particles in the {particles} of **B** such that

1.2.1 Each particle in the {particles} of **R** is a valid restriction of the particle in the {particles} of **B** it maps to as defined by Particle Valid (Restriction) (§5.10);

1.2.2 All particles in the {particles} of **B** which are not mapped to by any particle in the {particles} of **R** are emptiable as defined by Particle Emptiable (§5.10).

NOTE: Although the validation semantics of an *all* group does not depend on the order of its particles, we require derived *all* groups to match the order of their base to simplify checking that the derivation is OK.

[Definition:] A complete functional mapping is **order-preserving** if each particle **r** in the domain **R** maps to a particle **b** in the range **B** which follows (not necessarily immediately) the particle in the range **B** mapped to by the predecessor of **r**, if any, where "predecessor" and "follows" are defined with respect to the order of the lists which constitute **R** and **B**.

Constraint on Schemas: Particle Derivation OK (Choice:Choice -- RecurseLax)

A *choice* group particle is a valid restriction of another *choice* group particle if

1.1 **R**'s occurrence range is a valid restriction of **B**'s occurrence range as defined by Occurrence Range OK (§5.10);

1.2 There is a complete order-preserving functional mapping from the particles in the {particles} of **R** to the particles in the {particles} of **B** such that each particle in the {particles} of **R** is a valid restriction of the particle in the {particles} of **B** it maps to as defined by Particle Valid (Restriction) (§5.10).

NOTE: Although the validation semantics of a *choice* group does not depend on the order of its particles, we require derived *choice* groups to match the order of their base to simplify checking that the derivation is OK.

Constraint on Schemas: Particle Derivation OK (Sequence:All -- RecurseUnordered)

A *sequence* group particle is a valid restriction of an *all* group particle if

1.1 **R**'s occurrence range is a valid restriction of **B**'s occurrence range as defined by Occurrence Range OK (§5.10);

1.2 there is a complete functional mapping from the particles in the {particles} of **R** to the particles in the {particles} of **B** such that

1.2.1 No particle in the {particles} of **B** is mapped to by more than one of the particles in the {particles} of **R**;

1.2.2 Each particle in the {particles} of **R** is a valid restriction of the particle in the {particles} of **B** it maps to as defined by Particle Valid (Restriction) (§5.10);

1.2.3 All particles in the {particles} of **B** which are not mapped to by any particle in the {particles} of **R** are emptiable as defined by Particle Emptiable (§5.10).

NOTE: Although this clause allows reordering, because of the limits on the contents of *all* groups the checking process can still be deterministic.

Constraint on Schemas: Particle Derivation OK (Sequence:Choice -- MapAndSum)

A *sequence* group particle is a valid restriction of a *choice* group particle if

1.1 there is a complete functional mapping from the particles in the{particles} of **R** to the particles in the {particles} of **B** such that each particle in the {particles} of **R** is a valid restriction of the particle in the {particles} of **B** it maps to as defined by Particle Valid (Restriction) (§5.10);

1.2 The pair consisting of the product of the {min occurs} of **R** and the length of its {particles} and *unbounded* if {max occurs} is *unbounded* otherwise the product of the {max occurs} of **R** and the length of its {particles} is a valid restriction of **B**'s occurrence range as defined by Occurrence Range OK (§5.10).

> **NOTE:** This clause is in principle more restrictive than absolutely necessary, but in practice will cover all the likely cases, and is much easier to specify than the fully general version.

> **NOTE:** This case allows the "unfolding" of iterated disjunctions into sequences. It may be particularly useful when the disjunction is an implicit one arising from the use of substitution groups.

Constraint on Schemas: Particle Emptiable
[Definition:] A particle is **emptiable** if either

1.1 its {min occurs} is `0`

or

1.2 its {term} is a group and the minimum part of the effective total range of that group, as defined by Effective Total Range (all and sequence) (§5.7) (if the group is *all* or *sequence*) or Effective Total Range (choice) (§5.7) (if it is *choice*), is `0`.

5.11 Complex Type Definition Constraints

All complex type definitions (see Complex Type Definition Details (§3.4)) must satisfy the following constraints.

Constraint on Schemas: Complex Type Definition Properties Correct
All complex type definitions (see Complex Type Definition Details (§3.4)) must satisfy the following constraints:

1 The values of the properties of a complex type definition must be as described in the property tableau in Complex Type Definition Details (§3.4), modulo the impact of Missing Sub-components (§7.3);

2 If the {base type definition} is a simple type definition, the {derivation method} must be *extension*;

3 No two attribute declarations in the {attribute declarations} may have identical {name}s and {target namespace}s.

Constraint on Schemas: Derivation Valid (Extension)
If the {derivation method} is *extension*:

1.1 If the {base type definition} is a complex type definition:

1.1.1 The {final} of the {base type definition} must not contain *extension*

1.1.2 Its {attribute declarations} must be a subset of the {attribute declarations} of the complex type definition itself, that is, for everyattribute use pair in the {attribute declarations} of the {base type definition}, there must be a pair in the {attribute declarations} of the complex type definition itself whose attribute declaration has the same {name}, {target namespace} and {simple type definition} as its attribute declaration;

1.1.3 If it has an {attribute wildcard}, the complex type definition must also have one, and the base type definition's {attribute wildcard}'s {namespace constraint} must be a subset of the complex type definition's {attribute wildcard}'s {namespace constraint}, as defined by Wildcard Subset (§5.5);

1.1.4 Either the {content type} of the {base type definition} and the {content type} of the complex type definition itself must be the same simple type definition, or else the{content type} of the complex type definition itself must specify a particle and either the{content type} of the {base type definition} must be *empty* or

1.1.4.1 both {content type}s must be *mixed* or both must be *element-only*;

1.1.4.2 the particle of the complex type definition must be avalid extension of the {base type definition}'s particle, as defined inParticle Valid (Extension) (§5.10).

1.1.5 It must in principle be possible to derive the complex type definition in two steps, the first an extension and the second a restriction (possibly vacuous), from that type definition among its ancestors whose {base type definition} is the ur-type definition.

> **NOTE:** This requirement ensures that nothing removed by a restriction is subsequently added back by an extension. It is trivially to check if the extension in question is the only extension in its derivation, or if there are no restrictions bar the first from the ur-type definition.
>
> Constructing the intermediate type definition to check this constraint is straightforward: simply re-order the derivation to put all the extension steps first, then collapse them into a single extension. If the resulting definition can be the basis for a valid restriction to the desired definition, the constraint is satisfied.

1.2 If the {base type definition} is a simple type definition, the{content type} must be the same simple type definition.

[Definition:] If this constraint holds of a complex type definition, we say it is a**valid extension** of its {base type definition}.

Constraint on Schemas: Derivation Valid (Restriction, Complex)

If the {derivation method} is *restriction*:

1.1 The {base type definition} must be a complex type definition whose {final} does not contain *restriction*;

1.2 For each attribute use pair in the {attribute declarations}:

1.2.1 there must be a pair whose attribute declaration has the same {name} and {target namespace} in the {attribute declarations} of the {base type definition} from whose {simple type definition} the attribute in question's {simple type definition} must be validly derived given {*list*} as defined in Type Derivation OK (Simple) (§5.12);

or

1.2.2 the {base type definition} must have an {attribute wildcard} and the {target namespace} of the attribute in question must be schema-valid with respect to that wildcard, as defined in Wildcard allows Namespace URI (§3.9).

1.3 For each attribute use pair in the {attribute declarations} of the {base type definition} whose boolean part is *true*, there must be a pair with an attribute declaration with the same {name} and {target namespace} as its attribute declaration in the {attribute declarations} of the complex type definition itself;

1.4 If there is an {attribute wildcard}, the {base type definition} must also have one, and the complex type definition's {attribute wildcard}'s {namespace constraint} must be a subset of the {base type definition}'s {attribute wildcard}'s {namespace constraint}, as defined by Wildcard Subset (§5.5);

1.5 Either

1.5 the {content type} of the complex type definition is a simple type definition and either

1.5.1 the {content type} of the {base type definition} is a simple type definition of which the {content type} is a valid restriction as defined in Derivation Valid (Restriction, Simple) (§5.12)

or

1.5.2 The {base type definition} is *mixed* and has a particle which is emptiable as defined in Particle Emptiable (§5.10).

or

- The {content type} of the complex type itself is *empty* and either

1.5.1 the {content type} of the {base type definition} is also *empty*

or

1.5.2 the {content type} of the {base type definition} is *elementOnly* or *mixed* and has a particle which is emptiable as defined in Particle Emptiable (§5.10).

NOTE: To restrict a complex type definition with a simple base type definition to *empty*, use a simple type definition with a *fixed* value of the empty string: this preserves the type information.

or

- The {content type} of the {base type definition} is *mixed* or the {content type} of the complex type definition itself is *element-only*; the particle of the complex type definition itself is a valid restriction of the particle of the {content type} of the {base type definition} as defined in Particle Valid (Restriction) (§5.10).

[Definition:] If this constraint holds of a complex type definition, we say it is a **valid restriction** of its {base type definition}.

The following constraint defines a relation appealed to elsewhere in this specification.

Constraint on Schemas: Type Derivation OK (Complex)

A complex type definition (call it **D**, for derived) is validly derived from a type definition (call this **B**, for base) given a subset of {*extension*, *restriction*} if:

1.1 The {derivation method} of **D** is not in the subset, or in the {final} of its {base type definition};

1.2 Either

1.2.1 They are the same type definition;

or

1.2.2 **B** is the {base type definition}

or

1.2.3 the {base type definition} is not the ur-type definition and is validly derived from **B** given the subset as defined by this constraint (if the {base type definition} is complex) or validly derived from **B** given the subset unioned with {*list*} as defined in Type Derivation OK (Simple) (§5.12) (if the {base type definition} is simple).

NOTE: This constraint is used to check that when someone uses a type in a context where another type was expected (either via `xsi:type` or substitution groups), that the type used is actually derived from the expected type, and that that derivation does not involve a form of derivation which was ruled out by the expected type.

5.12 Simple Type Definition Constraints

All simple type definitions (see (non-normative) Simple Type Definition Details (§3.13)) must satisfy the following constraints.

Constraint on Schemas: Simple Type Definition Properties Correct

The values of the properties of a simple type definition must be as described in the property tableau in (non-normative) Simple Type Definition Details (§3.13), modulo the impact of Missing Sub-components (§7.3);

Constraint on Schemas: Derivation Valid (Restriction, Simple)

1 If the {variety} is *atomic*:
  1.1 The {base type definition} must be an atomic simple type definition or a built-in primitive datatype;
  1.2 For each facet in the {facets}there must be a facet of the same kind in the {facets} of the {base type definition} of whose **{value}** the facet in question's **{value}** must be a valid restriction as defined in [XML Schemas: Datatypes];

2 If the {variety} is *list*:
  2.1 The {item type definition} must have a {variety} of *atomic* or *union* and not have any members in its value space which contain spaces;
  2.2 Only *length*, *minLength*, *maxLength*, *pattern* and *enumeration* facet components are allowed among the {facets}.
  2.3 If the {base type definition} is not the simple ur-type definition, then
    2.3.1 the {base type definition} must have a {variety} of *list*
    2.3.2 for each facet in the {facets} there must be a facet of the same kind in the {facets} of the {base type definition} of whose **{value}** the facet in question's **{value}** must be a valid restriction as defined in [XML Schemas: Datatypes];

3 If the {variety} is *union*:
  3.1 The {item type definition} must have {variety} of *atomic* or *list*;
  3.2 Only *pattern* and *enumeration* facet components are allowed among the {facets}.
  3.3 If the {base type definition} is not the simple ur-type definition, then
    3.3.1 the {base type definition} must have a {variety} of *union*
    3.3.2 for each facet in the {facets} there must be a facet of the same kind in the {facets} of the {base type definition} of whose **{value}** the facet in question's **{value}** must be a valid restriction as defined in [XML Schemas: Datatypes];

[Definition:] If this constraint holds of a simple type definition, we say it is a **valid restriction** of its base type definition.

The following constraints define relations appealed to elsewhere in this specification.

Constraint on Schemas: Type Derivation OK (Simple)
A simple type definition (call it **D**, for derived) is validly derived from a simple type definition (call this **B**, for base) given a subset of {*list*, *extension*, *restriction*} (of which only *list* is actually relevant) if:

1.1 The {variety} of **D** is not in the subset;

1.2
  1.2.1 They are the same type definition

  or

  1.2.2 the {variety} is *atomic* and the base type definition is **B**

  or

  1.2.3 the {variety} is *atomic* and the base type definition is not the simple ur-type definition and is validly derived from **B** given the subset, as defined by this constraint.

5.13 Schema Constraints

All schemas (see Schema details (§3.1)) must satisfy the following constraint.

Constraint on Schemas: Schema Properties Correct

1 The values of the properties of a schema must be as described in the property tableau in Schema details (§3.1), modulo the impact of Missing Sub-components (§7.3);

2 Each of the {type definitions}, {element declarations}, {attribute group definitions}, {model group definitions} and {notation declarations} must not contain two or more schema components with the same **{name}** and **{target namespace}**.

6 Schema Access and Composition

This chapter defines the mechanisms by which we establish the necessary precondition for establishing schema-validity, namely access to one or more schemas. This chapter also describes in detail related mechanisms for using in one schema, definitions and declarations from another, possibly with modifications.

Conformance (§2.4) describes three levels of conformance for schema processors, and Validation Processing of schemas and documents (§7) provides a formal definition of schema-validation. Here we set out in detail the 3-layer architecture implied by the three conformance levels. The layers are:

1. The schema-validation core, relating schema components and instance information items;
2. Schema representation: the connections between XML representations and schema components, including the relationships between namespaces and schema components;
3. XML Schema web-interoperability guidelines: instance->schema and schema->schema connections for the WWW.

Layer 1 specifies the manner in which a schema composed of schema components can be applied to validate an instance element information item. Layer 2, which is primarily defined in XML Representation of Schemas and Schema Components (§4), specifies the use of schema elements in XML documents as the standard XML representation for schema information in a broad range of computer systems and execution environments. To support interoperation over the World Wide Web in particular, layer 3 provides a set of conventions for schema reference on the Web. Additional details on each of the three layers is provided in the sections below.

6.1 Layer 1: Summary of the schema-validation core

The fundamental purpose of the schema-validation core is to define schema-validatity for a single element information item and its descendants with respect to a complex type definition. All processors are required to implement this core predicate in a manner which conforms exactly to this specification.

Schema-validity is defined with reference to an XML Schema (note *not* a schema document) which consists of (at a minimum) the set of schema components (definitions and declarations) required for that validation. This is not a circular definition, but rather a *post facto* observation: no element information item can be fully schema-valid unless all the components required by any aspect of its (potentially recursive) validation are present in the schema.

As specified above, each schema component is associated directly or indirectly with a target namespace, or explicitly with no namespace. In the case of multi-namespace documents, components for more than one target namespace will co-exist in a schema.

Processors have the option to assemble (and perhaps to optimise or pre-compile) the entire schema prior to the start of a validation episode, or to gather the schema lazily as individual components are required. In all cases it is required that:

- The processor succeed in locating the schema components transitively required to complete a validation (note that components derived from schema documents can be integrated with components obtained through other means);
- no definition or declaration changes once it has been established;
- if the processor chooses to acquire declarations and definitions dynamically, that there be no side effects o such dynamic acquisition that would cause the results of validation to differ from that which would have been obtained from the same schema components acquired in bulk.

  NOTE: the validation core is defined in terms of schema components at the abstract level, and no mention is made of the schema definition syntax (i.e schema). Although many processors will acquire schemas in this format, others may operate on compiled representations, on a programmatic representation as exposed in some programming language, etc.

The obligation of a schema-aware processor as far as the schema-validation core is concerned is to implement the definitions of schema-valid given below in Schema Validation of Documents (§7.2). Neither the choice of element information item to be schema-validated, nor which of three means of initiating validation are used, is within the scope of this specification.

Although assessing schema-validity is defined recursively, it is also intended to be implementable in streaming processors. Such processors may choose to incrementally assemble the schema during processing in response, for example, to encountering new namespaces. The implication of the invariants expressed above is that such incremental assembly must result in a validation outcome that is the *same* as would be given if schema-validity was re-assessed with the final, fully assembled schema.

6.2 Layer 2: Schema definitions in XML

XML Representation of Schemas and Schema Components (§4) defines an XML representation for type definitions and element declarations and so on, specifying their target namespace and collecting them into schema documents. The two following sections relate to assembling a complete schema for validation from multiple sources. They should *not* be understood as a form of text substitution, but rather as providing mechanisms for distributed definition of schema components, with appropriate schema-specific semantics.

> **NOTE:** The core validation architecture requires that a complete schema with all the necessary declarations and definitions be available. This may involve resolving both instance->schema and schema->schema references. As observed earlier in Conformance (§2.4), we anticipate that the precise mechanisms for resolving such references will evolve over time. In support of such evolution, we have attempted to observe the design principle that references from one schema document to another schema use mechanisms that directly parallel those used to reference a schema from an instance document.
>
> **NOTE:** In the sections below, "schemaLocation" really belongs at layer 3. For convenience, we document it with the layer 2 mechanisms of import and include, with which it is closely associated.
>
> **Ed. Note:** Priority Feedback Request
> Each of the subsequent sections ends with a note about multiple inclusion/redefinition/importing. The space of possibilities here, particular once nesting is considered, is very large: we solicit feedback on ease of implementation, and any interoperability issues which arise.

6.2.1 Assembling a schema for a single target namespace from multiple schema definition documents

Schema components for a single target namespace can be assembled from several schema documents, that is several schema element information items:

XML Representation Summary: `include` Element Information Item

```
<include
  id = ID
  schemaLocation = uriReference
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</include>
```

A schema information item may contain any number of include elements. Their `schemaLocation` attributes, consisting of a URI reference, identify other schema documents, that is schema information items.

The XML Schema corresponding to schema contains not only the components specified elsewhere in this specification, but also all the components of all the XML Schemas corresponding to any included schema documents. Such included components must either (a) have the same `targetNamespace` as the including schema document, or (b) no `targetNamespace` at all, in which case all the top-level included components are converted to the including schema document's `targetNamespace`.

Schema Representation Constraint: Inclusion Constraints and Semantics
In addition to the conditions imposed on include element information items by the schema for schemas, the following must also hold:

1.1 If the normalized value of the `schemaLocation` [attribute] successfully resolves, it resolves either

1.1.1 to (a fragment of) a resource of type `text/xml`, which in turn corresponds to a schema element information item in a well-formed information set, which in turn corresponds to a valid schema

or

1.1.2 to a schema element information item in a well-formed information set, which in turn corresponds to a valid schema.

In either case call the schema item **SII** and the valid schema **I**.

1.2 The `targetNamespace` of **SII** is either

1.2.1 absent

or

1.2.2 identical to the `targetNamespace` of the include item's parent schema.

It is *not* an error for the normalized value of the `schemaLocation` [attribute] to fail to resolve it all, in which case no corresponding inclusion is performed. It *is* an error for it to resolve but the rest of clause 1.1 above to fail to obtain. Failure to resolve may well cause less than complete schema-validation outcomes, of course.

The schema components (that is {type definitions} {attribute declarations}, {element declarations}, {attribute group definitions}, {model group definitions}, {notation declarations}) of a schema corresponding to a schema element information item with one or more include element information items must include not only definitions or declarations corresponding to the appropriate members of its [children], but also, for each of those include element information items for which clause 1.1 above obtains successfully, a set of components identical to all the schema components of **I** (if clause 1.2.2 above obtains), or identical in all respects except their **{target namespace}**, which is that of the including schema (if clause 1.2.1 above obtains).

> **NOTE:** The above is carefully worded so that multiple including of the same schema document will not constitute a violation of clause 2 of Schema Properties Correct (§5.13), but applications are allowed, indeed encouraged, to avoid including the same schema document more than once to

forestall the necessity of establishing identity component by component.

6.2.2 Including modified component definitions

In an effort to provide some support for evolution and versioning, it is possible to incorporate components corresponding to a schema document *with modifications*. The modifications have a pervasive impact, that is, only the redefined components are used, even when referenced from other incorporated components, whether redefined themselves or not.

> **Ed. Note:** Priority Feedback Request
> This facility is very powerful, perhaps too powerful. Reports of implementation experience, in terms of useability for particular purposes, of the constraints on redefinition imposed below and of implementation difficulty, would be very welcome.

XML Representation Summary: `redefine` Element Information Item

```
<redefine
  schemaLocation = uriReference
  {any attributes with non-schema namespace . . .}>
  Content: (annotation | (attributeGroup | complexType | group |
simpleType))*
</redefine>
```

A schema information item may contain any number of redefine elements. Their `schemaLocation` attributes, consisting of a URI reference, identify other schema documents, that is schema information items.

The XML Schema corresponding to schema not only the components specified elsewhere, but also all the components of all the XML Schemas corresponding to any redefined schema documents. Such components must either (a) have the same `targetNamespace` as the including schema document, or (b) no `targetNamespace` at all, in which case all the top-level components are converted to the redefineing schema document's `targetNamespace`.

The definitions within the redefine element itself are restricted to be redefinitions of components from the redefined schema document, *in terms of themselves*. That is, type definitions must use themselves as their base type definition, and attribute group definitions and model group definitions must include references to themselves. Not all the definitions of the redefined schema document need be redefined.

This mechanism is intended to provide a declarative and modular approach to schema modification, with functionality no different except in scope from what would be achieved by wholesale text copying and redefinition by editing. In particular redefining a type is not guaranteed to be side-effect free: it may in particular have unexpected impacts on other type definitions which are based on the redefined one, even to the extent that some such definitions become ill-formed.

Example

```
v1.xsd:
 <xs:complexType name="personName">
  <xs:sequence>
   <xs:element name="title" minOccurs="0"/>
   <xs:element name="forename" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>

 <xs:element name="addressee" type="personName"/>

v2.xsd:
 <xs:redefine schemaLocation="v1.xsd">
  <xs:complexType name="personName">
   <xs:complexContent>
    <xs:extension base="personName">
     <xs:sequence>
      <xs:element name="generation" minOccurs="0"/>
     </xs:sequence>
    </xs:extension>
   </xs:complexContent>
  </xs:complexType>
 </xs:redefine>

 <xs:element name="author" type="personName"/>
```

The schema corresponding to `v2.xsd` has everything specified by`v1.xsd`, with the `personName` type redefined, as well as everything it specifies itself. According to this schema, elements constrained by the `personName` type may end with a `generation` element. This includes not only the`author` element, but also the `addressee` element.

Schema Representation Constraint: Redefinition Constraints and Semantics

In addition to the conditions imposed on redefine element information items by the schema for schemas, the following must also hold:

1.1 If the normalized value of the `schemaLocation` [attribute] successfully resolves, it resolves either

1.1.1 to (a fragment of) a resource of type `text/xml`, which in turn corresponds to a schema element information item in a well-formed information set, which in turn corresponds to a valid schema

or

1.1.2 to a schema element information item in a well-formed information set, which in turn corresponds to a valid schema.

In either case call the schema item **SII** and the valid schema **I**.

1.2 The `targetNamespace` of **SII** is either

1.2.1 absent

or

1.2.2 identical to the `targetNamespace` of the include item's parent schema.

1.3 Within the [children], each simpleType must have a restriction among its [children] and each complexType must have a restriction or extension among its grand-[children] the normalized value of whose `base` [attribute] must be the same as the normalized value of its own `name` attribute plus target namespace;

1.4 Within the [children], each group must have exactly one group among its contents at some level the normalized value of whose `ref` [attribute] must be the same as the normalized value of its own `name` attribute plus target namespace;

1.5 Within the [children], each attributeGroup must have exactly one attributeGroup among its [children] the normalized value of whose `ref` [attribute] must be the same as the normalized value of its own `name` attribute plus target namespace;

It is *not* an error for the normalized value of the `schemaLocation` [attribute] to fail to resolve it all, in which case no corresponding redefinition is performed. It *is* an error for it to resolve but the rest of clause 1.1 above to fail to obtain. Failure to resolve may well cause less than complete schema-validation outcomes, of course.

The schema components (that is {type definitions} {attribute declarations}, {element declarations}, {attribute group definitions}, {model group definitions}, {notation declarations}) of a schema corresponding to a schema element information item with one or more redefine element information items must include not only definitions or declarations corresponding to the appropriate members of its [children], but also, for each of those redefine element information items for which clause 1.1 above obtains successfully, with the exception of those components explicitly redefined, as described in Individual Component Redefinition (§6.2.2) below, a set of components identical to all the schema components of **I** (if clause 1.2.2 above obtains), or identical in all respects except their **{target namespace}**, which is that of the including schema (if clause 1.2.1 above obtains).

Schema Representation Constraint: Individual Component Redefinition

Corresponding to each non-annotation member of the [children] of a redefine there are one or two schema components in the redefining schema:

1 The simpleType and complexType [children] information items each correspond to two components:

1.1 One component which corresponds to the top-level definition item with the same `name` in the redefined schema document, as defined in XML Representation of Schemas and Schema Components (§4), except that its **{name}** is absent;

1.2 One component which corresponds to the information item itself, as defined in XML Representation of Schemas and Schema Components (§4), except that its **{base type definition}** is the component defined in 1.1 above

This pairing ensures the the coherence constraints on type definitions are respected, while at the same time achieving the desired effect, namely that references to names of redefined components in both the redefineing and redefined schema documents resolve to the redefined component as specified in 1.2 above.

2 The group and attributeGroup [children] each correspond to a single component, as defined in XML Representation of Schemas and Schema Components (§4), except that when the `ref` [attribute] whose normalized value is the same as the item's `name` plus target namespace is resolved, a component which corresponds to the top-level definition item of that name and the appropriate kind in the redefined schema document, as defined in XML Representation of Schemas and Schema Components (§4), is used.

In all cases there must be a top-level definition item of the appropriate name and kind in the redefined schema document.

> **NOTE:** The above is carefully worded so that multiple equivalent redefineing of the same schema document will not constitute a violation of clause 2 of Schema Properties Correct (§5.13), but applications are allowed, indeed encouraged, to avoid redefineing the same schema document in the same way more than once to forestall the necessity of establishing identity component by component (although this will have to be done for the individual redefinitions themselves).

6.2.3 References to schema components across namespaces

As described in XML Schema Abstract Data Model (§2.2), every global schema component is associated with a target namespace (or, explicitly, with none). In this section we set out the exact mechanism and syntax in the XML form of schema definition by which a reference to a foreign component is made, that is, a component with a different target namespace from that of the referring component.

We require not only a means of addressing such foreign components but also a signal to schema-aware processors that a schema document contains such references:

XML Representation Summary: `import` Element Information Item

```
<import
  id = ID
  namespace = uriReference
  schemaLocation = uriReference
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</import>
```

The import element information item identifies namespaces used in external references, i.e. those whose QName identifies them as coming from a different namespace (or none) than the enclosing schema document's `targetNamespace`. The normalized value of its `namespace` [attribute] indicates that the containing schema document may contain qualified references to schema components in that namespace (via one or more prefixes declared with namespace declarations in the normal way). If that attribute is absent, then the import allows unqualified reference to components with no target namespace. Note that components to be imported need not

be in the form of a schema document; the processor is free to access or construct components using means of its own choosing.

The normalized value of the `schemaLocation`, if present, gives a hint as to where a schema document with declarations and definitions for that namespace (or none) may be found. When no `schemaLocation` [attribute] is present, the schema author is leaving the identification of that schema to the instance, application or user, via the mechanisms described below in Layer 3: Web-interoperability (§6.3). When a `schemaLocation` is present, it must contain a single URI reference which the schema author warrants will resolve to a schema document containing the component(s) in the imported namespace referred to elsewhere in the containing schema document.

> **NOTE:** Since both the `namespace` and `schemaLocation` [attribute] are optional, a bare `<import/>` information item is schema-valid. This simply allows unqualified reference to foreign components without giving any hints as to where to find them.

Example

We may use the same namespace both for real work, and in the course of defining schema components in terms of foreign components:

```
<schema xmlns="http://www.w3.org/1999/XMLSchema"
        xmlns:html="http://www.w3.org/1999/xhtml"
        targetNamespace="uri:mywork" xmlns:my="uri:mywork">

 <import namespace="http://www.w3.org/1999/xhtml"/>

 <annotation>
  <documentation
   <html:p>[Some documentation for my schema]</html:p>
  </documentation>
 <annotation>

 . . .

 <complexType name="myType">
  <element ref="html:p" minOccurs="0"/>
  . . .
 </complexType>

 <element name="myElt" type="my:myType"/>
</schema>
```

The treatment of references as QNames implies that since (with the exception of the schema for schemas) the target namespace and the XML Schema namespace differ, without massive redeclaration of the default namespace *either* internal references to the names being defined in a schema document *or* the schema declaration and definition elements themselves must be explicitly qualified. This example takes the first option -- most other examples in this specification have taken the second.

Schema Representation Constraint: Import Constraints and Semantics
In addition to the conditions imposed on import element information items by the schema for schemas, the following must also hold:

1.1 If the application schema reference strategy using the normalized values of the `schemaLocation` and `namespace` [attributes], provides a referent, as defined by Schema Document Location Strategy (§6.3.2), it is either

1.1.1 (a fragment of) a resource of type `text/xml`, which in turn corresponds to a schema element information item in a well-formed information set, which in turn corresponds to a valid schema

or

1.1.2 a schema element information item in a well-formed information set, which in turn corresponds to a valid schema.

In either case call the schema item **SII** and the valid schema **I**.

1.2 The `targetNamespace` of **SII** is identical to the normalized value of the `namespace` [attribute]

It is *not* an error for the application schema reference strategy to fail. It *is* an error for it to resolve but the rest of clause 1.1 above to fail to obtain. Failure to find a referent may well cause less than complete schema-validation outcomes, of course.

The schema components (that is {type definitions} {attribute declarations}, {element declarations}, {attribute group definitions}, {model group definitions}, {notation declarations}) of a schema corresponding to a schema element information item with one or more import element information items must include not only definitions or declarations corresponding to the appropriate members of its [children], but also, for each of those import element information items for which clause 1.1 above obtains successfully, a set of schema components identical to all the schema components of **I**.

> **NOTE:** The above is carefully worded so that multiple importing of the same schema document will not constitute a violation of clause 2 of Schema Properties Correct (§5.13), but applications are allowed, indeed encouraged, to avoid importing the same schema document more than once to forestall the necessity of establishing identity component by component. Given that the `schemaLocation` [attribute] is only a hint, it is open to applications to ignore all but the first import for a given namespace, regardless of the normalized value of `schemaLocation`, but such a strategy risks missing useful information when new `schemaLocations` are offered.

6.3 Layer 3: Web-interoperability

Layers 1 and 2 provide a framework for validation and XML definition of schemas in a broad variety of environments. Over time, we expect that a range of standards and conventions will evolve to support interoperability of XML Schema implementations on the World Wide Web. Layer 3 defines the minimum level function required of all conformant processors operating on the Web: it is intended that, over time, future standards (e.g. XML Packages) for interoperability on the Web and in other environments can be introduced without the need to republish this specification.

6.3.1 Standards for representation of schemas and retrieval of schema documents on the Web

For interoperability, schema documents like all other Web resources may be identified by URI and retrieved using the standard mechanisms of the Web (e.g. http, https, etc.) Schema documents on the Web must be part of documents with the mime type `text/xml`, and are represented in the standard XML schema definition form described by layer 2 (that is as schema element information items).

> **NOTE:** there will often be times when a schema document will be a complete XML 1.0 document whose document element is schema. There will be other occasions in which schema items will be contained in other documents, perhaps referenced using fragment and/or Xpointer notation.

6.3.2 How schema definitions are located on the Web

As described in Layer 1: Summary of the schema-validation core (§6.1), processors are responsible for providing the schema components (definitions and declarations) needed for validation. This section introduces a set of normative conventions to facilitate interoperability for instance and schema documents retrieved and validated from the Web.

> **NOTE:** As discussed above in Layer 2: Schema definitions in XML (§6.2), other non-Web mechanisms for delivering schemas for validation may exist, but are outside the scope of this recommendation.

Processors on the Web are free to assess schema-validity against arbitrary schemas in any of the ways set out in Schema Validation of Documents (§7.2). However, it is useful to have a common convention for determining the schema to use. For this purpose, we require that for general-purpose schema-aware processors (i.e. those not specialised to one or a fixed set of pre-determined schemas) validating a document on the web:

- unless directed otherwise by the user, validation is performed on the document element information item of the specified document;
- unless directed otherwise by a user, the processor is required to construct a schema corresponding to a schema document whose `targetNamespace` is identical to the namespace URI, if any, of the element information item to be validated.

The composition of the complete schema for use in assessing schema-validity is discussed in Layer 2: Schema definitions in XML (§6.2) above. The means used to locate appropriate schema document(s) are processor and application dependent, subject to the following requirements:

1. Schemas are represented on the Web in the form specified above in Standards for representation of schemas and retrieval of schema documents on the Web (§6.3.1);
2. The author of a document uses namespace declarations to indicate the intended interpretation of names appearing therein; there may or may not be a schema retrievable via the namespace URI. Accordingly whether a processor's default behaviour is or is not to attempt such dereferencing, it must always provide for user-directed overriding of that default.

   > **NOTE:** Experience suggests that it is not in all cases safe or desirable from a performance point of view to dereference NS URIs as a matter of course. User community and/or consumer/provider agreements may establish circumstances in which such dereference is a sensible default strategy: this recommendation allows but does not require particular communities to establish and implement such conventions. Users are always free to supply namespace URIs as schema location information when dereferencing *is* desired: see below.

3. On the other hand, in case a document author (human or not) created a document with a particular schema in view, and warrants that some or all of the document is schema-valid per that schema, we provide the `schemaLocation` and `noNamespaceSchemaLocation` [attributes] (in the XML Schema instance namespace, that is, `http://www.w3.org/2000/10/XMLSchema-instance`) (hereafter `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation`). The first records the author's warrant with pairs of URI references (one for the namespace URI, and one for a hint as to the location of a schema document defining names for that namespace URI). The second similarly provides a URI reference as a hint as to the location of a schema document with no `targetNamespace` [attribute].

   Unless directed otherwise, for example by the invoking application or by command line option, processors should attempt to dereference each schema document location URI in the normalized value of such `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` [attributes], see details below.

4. `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` [attributes] can occur on any element. However, it is an error if such an attribute occurs *after* the first appearance of an element or attribute information item within element information item initially assessed for schema-validity whose [namespace URI] it addresses. According to the rules of Layer 1: Summary of the schema-validation core (§6.1), the corresponding schema may be lazily assembled, but is otherwise stable throughout a validation. Although schema location attributes can occur on any element, and can be processed incrementally as discovered, their effect is essentially global to the validation. Definitions and declarations remain in effect beyond the scope of the element on which the binding is declared.

Example

Multiple schema bindings can be declared using a single attribute. For example consider a stylesheet:

```
<stylesheet xmlns="http://www.w3.org/1999/XSL/Transform"
            xmlns:html="http://www.w3.org/1999/xhtml"
            xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
            xsi:schemaLocation="http://www.w3.org/1999/XSL/Transform
                                http://www.w3.org/1999/XSL/Transform.xsd
                                http://www.w3.org/1999/xhtml
                                http://www.w3.org/1999/xhtml.xsd">
```

The namespace URIs used in `schemaLocation` can, but need not be identical to those actually qualifying the element within whose start tag it is found or its other attributes. For example, as above, all schema location information can be declared on the document element of a document, if desired, regardless of where the namespaces are actually used.

Schema Representation Constraint: Schema Document Location Strategy

Given a namespace URI (or none) and (optionally) a URI reference from `xsi:schemaLocation` or `xsi:noNamespaceSchemaLocation`, schema-aware processors may implement any combination of the following strategies, in any order:

1.1 Do nothing, for instance because a schema containing components for the given namespace URI is already known to be available, or because it is known in advance that no efforts to locate schema documents will be successful (for example in embedded systems);

1.2 Based on the location URI, identify an existing schema document, either as a text/xml resource or a schema element information item, in some local schema repository;

1.3 Based on the namespace URI, identify an existing schema document, either as a text/xml resource or a schema element information item, in some local schema repository;

1.4 Attempt to resolve the location URI, to locate a resource on the web which is or contains or references a schema element;

1.5 Attempt to resolve the namespace URI to locate such a resource.

Whenever possible configuration and/or invocation options for selecting and/or ordering the implemented strategies should be provided.

We note that improved or alternative conventions for Web interoperability can be standardised in the future without reopening this recommendation. For example, the W3C is currently considering initiatives to standardise the packaging of resources relating to particular documents and/or namespaces: this would be an addition to the mechanisms described here for layer 3. This architecture also facilitates innovation at layer 2: for example, it would be possible in the future to define an additional standard for the representation of schema components which allowed e.g. type definitions to be specified piece by piece, rather than all at once.

7 Validation Processing of schemas and documents

The architecture of schema validation allows for a rich characterisation of XML documents: schema validity is not a binary predicate.

We distinguish between errors in schema construction and structure, on the one hand, and schema validation outcomes, on the other.

7.1 Errors in Schema Construction and Structure

Before schema validation can be attempted, a schema is required. Special-purpose applications are free to determine a schema for use in validation by whatever means are appropriate, but general purpose processors should implement the strategy set out in Schema Document Location Strategy (§6.3.2), starting with the namespaces declared in the document to be validated, and the normalized values of the `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` [attributes] thereof, if any.

It is an error if a schema and all the components which are the value of any of its properties, recursively, fail to satisfy all the relevant Constraints on Schemas set out in Schema Component Validity Constraints (§5).

If a schema is derived from one or more schema documents (that is, one or more schema element information items) based on the correspondence rules set out in XML Representation of Schemas and Schema Components (§4) and Schema Access and Composition (§6), two additional conditions hold:

- It is an error if any such schema document would not be schema-valid with respect to a schema corresponding to the (normative) Schema for Schemas (§A), that is, following schema-validation with such a schema, the schema element information items would have a **[validation attempted]** property with value *full* or *partial* and a **[validity]** property with value *complete*.
- It is an error if any such schema document is or contains any element information items which violate any of the relevant Schema Representation Constraints set out in XML Representation of Schemas and Schema Components (§4) and Schema Access and Composition (§6).

The three cases described above are the only types of error which this specification defines. With respect to the processes of the checking of schema structure and the construction of schemas corresponding to schema documents, this specification imposes no restrictions on processors after an error is detected. However schema-validity with respect to schema-like entities which do *not* satisfy all the above conditions is incoherent. Accordingly, conformant processors must not attempt to schema-validate using such non-schemas.

7.2 Schema Validation of Documents

With a schema which satisfies the conditions expressed in Errors in Schema Construction and Structure (§7.1) above, the schema-validity of an element information item can be assessed. Three primary approaches to this are possible:

1. The user or application identifies a complex type definition from among the {type definitions} of the schema, and assesses the schema-validity of the item by appealing to Element Children and Attributes Valid (§3.4);
2. The user or application identifies a element declaration from among the {element declarations} of the schema, and assesses the schema-validity of the item by appealing to Element Valid (Explicit) (§3.3);
3. The strict schema-validity of the item is assessed by appealing to Element Valid (Strict) (§3.3)
4. The (possibly lax) schema-validity of the item is assessed by appealing to Element Valid (Lax) (§3.3).

The outcome of this effort, in any case, will be manifest in the **[validation attempted]** and **[validity]** properties on the element information item and its [attributes] and [children], recursively, as defined by Validation Outcome

(skipped) (§3.9), Validation Outcome (Element) (§3.3), Validation Outcome (Attribute) (§3.2) and Validation Outcome (Complex Type) (§3.4). It is up to applications to decide what constitutes a successful outcome.

Note that every element and attribute information item whose validation was assessed will also have **[validation context]** property which refers back to the element information item at which assessment began.

[Definition:] Throughout this document we use the phrase **schema-valid** loosely to refer to a successful outcome to any of the above-listed assessments of an element information item with respect to a schema.

7.3 Missing Sub-components

At the beginning of Schema Component Details (§3), attention is drawn to the fact that most kinds of schema components have properties which are described therein as having other components, or sets of other components, as values, but that when components are constructed on the basis of their correspondence with element information items in schema documents, such properties usually correspond to QNames, and the resolution of such QNames may fail, resulting in one or more values of or containing absent where a component is mandated.

If at any time during schema-validation, schema-validity of an information item is being assessed with respect to a component of any kind any of whose properties has or contains such an absent value, the schema-validation effort is modified, as following:

- In the post-schema validation infoset, the **[validation attempted]** of the item has the value *partial*;
- Schema-validation resumes attempting to satisfy clause 1.2.2 of Element Valid (Lax) (§3.3), if the item is an element, or as if clause 1.2 of Attribute Valid (Lax) (§3.2) obtained, it the item is an attribute.

Because of clause 1.2 of Validation Outcome (Element) (§3.3), if this situation ever arises, the document as a whole will not show a **[validation attempted]** of *full*, unless the problem arises within an area of lax validation (see clause 1.2.3 of Item Valid (Wildcard) (§3.9)).

7.4 Responsibilities of Schema-aware processors

Schema-aware processors are responsible for processing XML documents, schemas and schema documents, as appropriate given the level of conformance (as defined in Conformance (§2.4)) they support, consistently with the conditions set out above.

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 types, elements, 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.

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 `targetNamespace` references the XML Schema namespace itself.

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- XML Schema schema for XML Schemas: Part 1: Structures -->
<!DOCTYPE schema PUBLIC "-//W3C//DTD XMLSCHEMA 200010//EN" "XMLSchema.dtd" >
<schema targetNamespace="http://www.w3.org/2000/10/XMLSchema" blockDefault="#all" elementF

 <annotation>
   <documentation source="http://www.w3.org/TR/2000/WD-xmlschema-1-20000922/structures.htm
   The schema corresponding to this document is normative,
   with respect to the syntactic constraints it expresses in the
   XML Schema language.  The documentation (within &lt;documentation> elements)
   below, is not normative, but rather highlights important aspects of
   the W3C Recommendation of which this is a part</documentation>
 </annotation>

 <annotation>
   <documentation>The simpleType element and all of its members are defined
       in datatypes.xsd</documentation>
 </annotation>

 <include schemaLocation="datatypes.xsd"/>

 <element name="schemaTop" abstract="true" type="annotated">
  <annotation>
   <documentation>This abstract element defines an substitution group over the
elements which occur freely at the top level of schemas.  These are:
simpleType, complexType, element, attribute, attributeGroup, group, notation
All of their types are based on the "annotated" type by extension.</documentation>
  </annotation>
 </element>

 <element name="redefinable" abstract="true" substitutionGroup="schemaTop">
  <annotation>
   <documentation>This abstract element defines a substitution group for the
elements which can self-redefine (see &lt;redefine> below).</documentation>
  </annotation>
 </element>

 <simpleType name="formChoice">
  <annotation>
   <documentation>A utility type, not for public use</documentation>
  </annotation>
  <restriction base="NMTOKEN">
   <enumeration value="qualified"/>
   <enumeration value="unqualified"/>
  </restriction>
 </simpleType>

 <element name="schema" id="schema">
  <annotation>
    <documentation
           source="http://www.w3.org/TR/xmlschema-1/#element-schema"/>
  </annotation>
  <complexType>
   <complexContent>
    <extension base="openAttrs">
     <sequence>
      <choice minOccurs="0" maxOccurs="unbounded">
       <element ref="include"/>
       <element ref="import"/>
       <element ref="redefine"/>
       <element ref="annotation"/>
      </choice>
      <sequence minOccurs="0" maxOccurs="unbounded">
       <element ref="schemaTop"/>
       <element ref="annotation" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
```

```
    </sequence>
    <attribute name="targetNamespace" type="uriReference"/>
    <attribute name="version" type="string"/>
    <attribute name="finalDefault" type="derivationSet" use="default" value=""/>
    <attribute name="blockDefault" type="blockSet" use="default" value=""/>
    <attribute name="attributeFormDefault" type="formChoice" use="default" value="unquali
    <attribute name="elementFormDefault" type="formChoice" use="default" value="unqualifi
    <attribute name="id" type="ID"/>
   </extension>
  </complexContent>
 </complexType>

 <key name="element">
  <selector xpath="element"/>
  <field xpath="@name"/>
 </key>

 <key name="attribute">
  <selector xpath="attribute"/>
  <field xpath="@name"/>
 </key>

 <key name="type">
  <selector xpath="complexType|simpleType"/>
  <field xpath="@name"/>
 </key>

 <key name="group">
  <selector xpath="group"/>
  <field xpath="@name"/>
 </key>

 <key name="attributeGroup">
  <selector xpath="attributeGroup"/>
  <field xpath="@name"/>
 </key>

 <key name="notation">
  <selector xpath="notation"/>
  <field xpath="@name"/>
 </key>

 <key name="identityConstraint">
  <selector xpath=".//key|.//unique|.//keyref"/>
  <field xpath="@name"/>
 </key>

</element>

<simpleType name="allNNI">
 <annotation><documentation>for maxOccurs</documentation></annotation>
 <union memberTypes="nonNegativeInteger">
  <simpleType>
   <restriction base="NMTOKEN">
    <enumeration value="unbounded"/>
   </restriction>
  </simpleType>
 </union>
</simpleType>

<attributeGroup name="occurs">
 <annotation><documentation>for all particles</documentation></annotation>
 <attribute name="minOccurs" type="nonNegativeInteger" use="default" value="1"/>
 <attribute name="maxOccurs" type="allNNI" use="default" value="1"/>
</attributeGroup>

<attributeGroup name="defRef">
 <annotation><documentation>for element, group and attributeGroup,
```

```
              which both define and reference</documentation></annotation>
 <attribute name="name" type="NCName"/>
 <attribute name="ref" type="QName"/>
</attributeGroup>

<group name="typeDefParticle">
 <annotation>
   <documentation>'complexType' uses this</documentation></annotation>
 <choice>
  <element name="group" type="groupRef"/>
  <element ref="all"/>
  <element ref="choice"/>
  <element ref="sequence"/>
 </choice>
</group>

<group name="groupDefParticle">
 <annotation>
   <documentation>'topLevelGroup' uses this</documentation></annotation>
 <choice>
  <element ref="all"/>
  <element ref="choice"/>
  <element ref="sequence"/>
 </choice>
</group>

<group name="nestedParticle">
 <choice>
  <element name="element" type="localElement"/>
  <element name="group" type="groupRef"/>
  <element ref="choice"/>
  <element ref="sequence"/>
  <element ref="any"/>
 </choice>
</group>

<group name="particle">
 <choice>
  <element name="element" type="localElement"/>
  <element name="group" type="groupRef"/>
  <element ref="all"/>
  <element ref="choice"/>
  <element ref="sequence"/>
  <element ref="any"/>
 </choice>
</group>

<complexType name="attribute">
 <complexContent>
  <extension base="annotated">
   <sequence>
    <element name="simpleType" minOccurs="0" type="localSimpleType"/>
   </sequence>
   <attributeGroup ref="defRef"/>
   <attribute name="type" type="QName"/>
   <attribute name="use" use="default" value="optional">
    <simpleType>
     <restriction base="NMTOKEN">
      <enumeration value="prohibited"/>
      <enumeration value="optional"/>
      <enumeration value="required"/>
      <enumeration value="default"/>
      <enumeration value="fixed"/>
     </restriction>
    </simpleType>
   </attribute>
   <attribute name="value" use="optional" type="string"/>
   <attribute name="form" type="formChoice"/>
```

```
    </extension>
   </complexContent>
  </complexType>

  <complexType name="topLevelAttribute">
   <complexContent>
    <restriction base="attribute">
     <sequence>
      <element ref="annotation" minOccurs="0"/>
      <element name="simpleType" minOccurs="0" type="localSimpleType"/>
     </sequence>
     <attribute name="ref" use="prohibited"/>
     <attribute name="form" use="prohibited"/>
     <attribute name="use" use="prohibited"/>
     <attribute name="name" use="required" type="NCName"/>
    </restriction>
   </complexContent>
  </complexType>

  <group name="attrDecls">
   <sequence>
    <choice minOccurs="0" maxOccurs="unbounded">
     <element name="attribute" type="attribute"/>
     <element name="attributeGroup" type="attributeGroupRef"/>
    </choice>
    <element ref="anyAttribute" minOccurs="0"/>
   </sequence>
  </group>

  <element name="anyAttribute" type="wildcard" id="anyAttribute">
   <annotation>
    <documentation
         source="http://www.w3.org/TR/xmlschema-1/#element-anyAttribute"/>
   </annotation>
  </element>

  <group name="complexTypeModel">
   <choice>
       <element ref="simpleContent"/>
       <element ref="complexContent"/>
       <sequence>
        <annotation>
         <documentation>This branch is short for
          &lt;complexContent>
            &lt;restriction base="anyType">
             ...
            &lt;/restriction>
          &lt;/complexContent></documentation>
        </annotation>
        <group ref="typeDefParticle" minOccurs="0"/>
        <group ref="attrDecls"/>
       </sequence>
   </choice>
  </group>

  <complexType name="complexType" abstract="true">
   <complexContent>
    <extension base="annotated">
     <group ref="complexTypeModel"/>
     <attribute name="name" type="NCName">
      <annotation>
       <documentation>Will be restricted to required or
forbidden</documentation>
      </annotation>
     </attribute>
     <attribute name="mixed" type="boolean" use="default" value="false">
      <annotation>
       <documentation>Not allowed if simpleContent child is chosen.
```

```
                    May be overriden by setting on complexContent child.
      </documentation>
     </annotation>
    </attribute>
    <attribute name="abstract" type="boolean" use="default" value="false"/>
    <attribute name="final" type="derivationSet"/>
    <attribute name="block" type="derivationSet" use="default" value=""/>
   </extension>
  </complexContent>
 </complexType>

 <complexType name="topLevelComplexType">
  <complexContent>
   <restriction base="complexType">
    <sequence>
     <element ref="annotation" minOccurs="0"/>
     <group ref="complexTypeModel"/>
    </sequence>
    <attribute name="name" type="NCName" use="required"/>
   </restriction>
  </complexContent>
 </complexType>

 <complexType name="localComplexType">
  <complexContent>
   <restriction base="complexType">
    <sequence>
     <element ref="annotation" minOccurs="0"/>
     <group ref="complexTypeModel"/>
    </sequence>
    <attribute name="name" use="prohibited"/>
   </restriction>
  </complexContent>
 </complexType>

 <complexType name="restrictionType">
  <complexContent>
   <extension base="annotated">
    <sequence>
     <choice>
      <group ref="typeDefParticle" minOccurs="0"/>
      <group ref="simpleRestrictionModel" minOccurs="0"/>
     </choice>
     <group ref="attrDecls"/>
    </sequence>
    <attribute name="base" type="QName" use="required"/>
   </extension>
  </complexContent>
 </complexType>

 <complexType name="complexRestrictionType">
  <complexContent>
   <restriction base="restrictionType">
    <sequence>
     <element ref="annotation" minOccurs="0"/>
     <group ref="typeDefParticle" minOccurs="0"/>
     <group ref="attrDecls"/>
    </sequence>
   </restriction>
  </complexContent>
 </complexType>

 <complexType name="extensionType">
  <complexContent>
   <extension base="annotated">
    <sequence>
     <group ref="typeDefParticle" minOccurs="0"/>
     <group ref="attrDecls"/>
```

```
   </sequence>
   <attribute name="base" type="QName" use="required"/>
  </extension>
 </complexContent>
</complexType>

<element name="complexContent" id="complexContent">
 <annotation>
  <documentation
       source="http://www.w3.org/TR/xmlschema-1/#element-complexContent"/>
 </annotation>
 <complexType>
  <complexContent>
   <extension base="annotated">
    <choice>
     <element name="restriction" type="complexRestrictionType"/>
     <element name="extension" type="extensionType"/>
    </choice>
    <attribute name="mixed" type="boolean">
     <annotation>
      <documentation>Overrides any setting on complexType parent.
      </documentation>
     </annotation>
   </attribute>
   </extension>
  </complexContent>
 </complexType>
</element>

<complexType name="simpleRestrictionType">
 <complexContent>
  <restriction base="restrictionType">
   <sequence>
    <element ref="annotation" minOccurs="0"/>
    <group ref="simpleRestrictionModel" minOccurs="0"/>
    <group ref="attrDecls"/>
   </sequence>
  </restriction>
 </complexContent>
</complexType>

<complexType name="simpleExtensionType">
 <complexContent>
  <restriction base="extensionType">
   <sequence>
    <annotation>
     <documentation>No typeDefParticle group reference</documentation>
    </annotation>
    <element ref="annotation" minOccurs="0"/>
    <group ref="attrDecls"/>
   </sequence>
  </restriction>
 </complexContent>
</complexType>

<element name="simpleContent" id="simpleContent">
 <annotation>
  <documentation
     source="http://www.w3.org/TR/xmlschema-1/#element-simpleContent"/>
 </annotation>
 <complexType>
  <complexContent>
   <extension base="annotated">
    <choice>
     <element name="restriction" type="simpleRestrictionType"/>
     <element name="extension" type="simpleExtensionType"/>
    </choice>
   </extension>
```

```
    </complexContent>
   </complexType>
  </element>

  <element name="complexType" substitutionGroup="redefinable" type="topLevelComplexType" id
   <annotation>
    <documentation
       source="http://www.w3.org/TR/xmlschema-1/#element-complexType"/>
   </annotation>
  </element>

  <simpleType name="derivationControls">
   <restriction base="string">
    <enumeration value="#all"/>
    <enumeration value="substitution"/>
    <enumeration value="extension"/>
    <enumeration value="restriction"/>
   </restriction>
  </simpleType>

  <simpleType name="derivationChoiceOrAll">
   <annotation>
    <documentation>A utility type, not for public use</documentation>
   </annotation>
   <restriction base="derivationControls">
    <enumeration value="#all"/>
    <enumeration value="extension"/>
    <enumeration value="restriction"/>
   </restriction>
  </simpleType>

  <simpleType name="derivationChoice">
   <annotation>
    <documentation>A utility type, not for public use</documentation>
   </annotation>
   <restriction base="derivationChoiceOrAll">
    <enumeration value="extension"/>
    <enumeration value="restriction"/>
   </restriction>
  </simpleType>

  <simpleType name="blockSet">
   <annotation>
    <documentation>#all or (possibly empty) subset of {substitution, extension,
restriction}</documentation>
    <documentation>A utility type, not for public use</documentation>
   </annotation>
   <list itemType="derivationControls"/>
  </simpleType>

  <simpleType name="derivationSet">
   <annotation>
    <documentation>#all or (possibly empty) subset of {extension,
restriction}</documentation>
    <documentation>A utility type, not for public use</documentation>
   </annotation>
   <list itemType="derivationChoiceOrAll"/>
  </simpleType>



  <complexType name="element" abstract="true">
   <annotation>
    <documentation>The element element can be used either
         at the toplevel to define an element-type binding globally,
         or within a content model to either reference a globally-defined
         element or type or declare an element-type binding locally.
        The ref form is not allowed at the top level.</documentation>
```

```
 </annotation>

 <complexContent>
  <extension base="annotated">
   <sequence>
    <choice minOccurs="0">
     <element name="simpleType" type="localSimpleType"/>
     <element name="complexType" type="localComplexType"/>
    </choice>
    <element ref="identityConstraint" minOccurs="0" maxOccurs="unbounded"/>
   </sequence>
   <attributeGroup ref="defRef"/>
   <attribute name="type" type="QName"/>
   <attribute name="substitutionGroup" type="QName"/>
   <attributeGroup ref="occurs"/>
   <attribute name="default" type="string"/>
   <attribute name="fixed" type="string"/>
   <attribute name="nullable" type="boolean" use="default" value="false"/>
   <attribute name="abstract" type="boolean" use="default" value="false"/>
   <attribute name="final" type="derivationSet" use="default" value=""/>
   <attribute name="block" type="blockSet" use="default" value=""/>
   <attribute name="form" type="formChoice"/>
  </extension>
 </complexContent>
</complexType>

<complexType name="topLevelElement">
 <complexContent>
  <restriction base="element">
   <sequence>
    <element ref="annotation" minOccurs="0"/>
    <choice minOccurs="0">
     <element name="simpleType" type="localSimpleType"/>
     <element name="complexType" type="localComplexType"/>
    </choice>
    <element ref="identityConstraint" minOccurs="0" maxOccurs="unbounded"/>
   </sequence>
   <attribute name="ref" use="prohibited"/>
   <attribute name="form" use="prohibited"/>
   <attribute name="minOccurs" use="prohibited"/>
   <attribute name="maxOccurs" use="prohibited"/>
   <attribute name="name" use="required" type="NCName"/>
  </restriction>
 </complexContent>
</complexType>

<complexType name="localElement">
 <complexContent>
  <restriction base="element">
   <sequence>
    <element ref="annotation" minOccurs="0"/>
    <choice minOccurs="0">
     <element name="simpleType" type="localSimpleType"/>
     <element name="complexType" type="localComplexType"/>
    </choice>
    <element ref="identityConstraint" minOccurs="0" maxOccurs="unbounded"/>
   </sequence>
   <attribute name="substitutionGroup" use="prohibited"/>
   <attribute name="final" use="prohibited"/>
  </restriction>
 </complexContent>
</complexType>

<element name="element" type="topLevelElement" substitutionGroup="schemaTop" id="element"
 <annotation>
  <documentation
     source="http://www.w3.org/TR/xmlschema-1/#element-element"/>
 </annotation>
```

```
</element>

<complexType name="group" abstract="true">
 <annotation>
  <documentation>group type for explicit groups, named top-level groups and
         group references</documentation>
 </annotation>
 <complexContent>
  <extension base="annotated">
   <group ref="particle" minOccurs="0" maxOccurs="unbounded"/>
   <attributeGroup ref="defRef"/>
   <attributeGroup ref="occurs"/>
  </extension>
 </complexContent>
</complexType>

<complexType name="realGroup">
 <complexContent>
  <restriction base="group">
   <sequence>
    <element ref="annotation" minOccurs="0"/>
    <group ref="groupDefParticle" minOccurs="0" maxOccurs="1"/>
   </sequence>
  </restriction>
 </complexContent>
</complexType>

<complexType name="namedGroup">
 <complexContent>
  <restriction base="realGroup">
   <sequence>
    <element ref="annotation" minOccurs="0"/>
    <group ref="groupDefParticle" minOccurs="1" maxOccurs="1"/>
   </sequence>
   <attribute name="name" use="required" type="NCName"/>
   <attribute name="ref" use="prohibited"/>
   <attribute name="minOccurs" use="prohibited"/>
   <attribute name="maxOccurs" use="prohibited"/>
  </restriction>
 </complexContent>
</complexType>

<complexType name="groupRef">
 <complexContent>
  <restriction base="realGroup">
   <sequence>
    <element ref="annotation" minOccurs="0"/>
   </sequence>
   <attribute name="ref" use="required" type="QName"/>
   <attribute name="name" use="prohibited"/>
  </restriction>
 </complexContent>
</complexType>

<complexType name="explicitGroup">
 <annotation>
  <documentation>group type for the three kinds of group</documentation>
 </annotation>
 <complexContent>
  <restriction base="group">
   <sequence>
    <element ref="annotation" minOccurs="0"/>
    <group ref="nestedParticle" minOccurs="0" maxOccurs="unbounded"/>
   </sequence>
   <attribute name="name" type="NCName" use="prohibited"/>
   <attribute name="ref" type="QName" use="prohibited"/>
  </restriction>
 </complexContent>
```

```
</complexType>

<element name="all" id="all">
 <annotation>
  <documentation
     source="http://www.w3.org/TR/xmlschema-1/#element-all"/>
 </annotation>
 <complexType>
  <annotation>
   <documentation>Only elements allowed inside</documentation>
  </annotation>
  <complexContent>
   <restriction base="explicitGroup">

    <sequence>
     <element ref="annotation" minOccurs="0"/>
     <element name="element" minOccurs="0" maxOccurs="unbounded">
      <complexType>
       <annotation>
        <documentation>restricted max/min</documentation>
       </annotation>
       <complexContent>
        <restriction base="localElement">

         <sequence>
          <element ref="annotation" minOccurs="0"/>
          <choice minOccurs="0">
           <element name="simpleType" type="localSimpleType"/>
           <element name="complexType" type="localComplexType"/>
          </choice>
          <element ref="identityConstraint" minOccurs="0" maxOccurs="unbounded"/>
         </sequence>
         <attribute name="minOccurs" use="default" value="1">
          <simpleType>
           <restriction base="nonNegativeInteger">
            <enumeration value="0"/>
            <enumeration value="1"/>
           </restriction>
          </simpleType>
         </attribute>
         <attribute name="maxOccurs" use="default" value="1">
          <simpleType>
           <restriction base="allNNI">
            <enumeration value="0"/>
            <enumeration value="1"/>
           </restriction>
          </simpleType>
         </attribute>
        </restriction>
       </complexContent>
      </complexType>
     </element>
    </sequence>
   </restriction>
  </complexContent>
 </complexType>
</element>

<element name="choice" type="explicitGroup" id="choice">
 <annotation>
  <documentation
     source="http://www.w3.org/TR/xmlschema-1/#element-choice"/>
 </annotation>
</element>

<element name="sequence" type="explicitGroup" id="sequence">
 <annotation>
  <documentation
```

```
      source="http://www.w3.org/TR/xmlschema-1/#element-sequence"/>
  </annotation>
 </element>

 <element name="group" substitutionGroup="redefinable" type="namedGroup" id="group">
  <annotation>
   <documentation
      source="http://www.w3.org/TR/xmlschema-1/#element-group"/>
  </annotation>
 </element>

 <complexType name="wildcard">
  <complexContent>
   <extension base="annotated">
    <attribute name="namespace" type="namespaceList" use="default" value="##any"/>
    <attribute name="processContents" use="default" value="strict">
     <simpleType>
      <restriction base="NMTOKEN">
       <enumeration value="skip"/>
       <enumeration value="lax"/>
       <enumeration value="strict"/>
      </restriction>
     </simpleType>
    </attribute>
   </extension>
  </complexContent>
 </complexType>

 <element name="any" id="any">
  <annotation>
   <documentation
      source="http://www.w3.org/TR/xmlschema-1/#element-any"/>
  </annotation>
  <complexType>
   <complexContent>
    <extension base="wildcard">
     <attributeGroup ref="occurs"/>
    </extension>
   </complexContent>
  </complexType>
 </element>

  <annotation>
   <documentation> simple type for the value of the 'namespace' attr of
      'any' and 'anyAttribute'</documentation>
  </annotation>
  <annotation>
   <documentation>Value is
                 ##any      - - any non-conflicting WFXML/attribute at all

                 ##other    - - any non-conflicting WFXML/attribute from
                                  namespace other than targetNS

                 ##local    - - any unqualified non-conflicting WFXML/attribute

                 one or     - - any non-conflicting WFXML/attribute from
                 more URI        the listed namespaces
                 references
                 (space separated)

                ##targetNamespace or ##local may appear in the above list, to
                    refer to the targetNamespace of the enclosing
                    schema or an absent targetNamespace respectively</documentation>
  </annotation>

 <simpleType name="namespaceList">
  <annotation>
   <documentation>##any | ##other | list of {uri, ##targetNamespace,
```

```
##local}</documentation>
   <documentation>A utility type, not for public use</documentation>
  </annotation>
  <restriction base="string"/>
 </simpleType>

 <element name="attribute" substitutionGroup="schemaTop" type="topLevelAttribute" id="attr
  <annotation>
   <documentation
      source="http://www.w3.org/TR/xmlschema-1/#element-attribute"/>
  </annotation>
 </element>

 <complexType name="attributeGroup" abstract="true">
  <complexContent>
   <extension base="annotated">
    <group ref="attrDecls"/>
    <attributeGroup ref="defRef"/>
   </extension>
  </complexContent>
 </complexType>

 <complexType name="namedAttributeGroup">
  <complexContent>
   <restriction base="attributeGroup">
    <sequence>
     <element ref="annotation" minOccurs="0"/>
     <group ref="attrDecls"/>
    </sequence>
    <attribute name="name" use="required" type="NCName"/>
    <attribute name="ref" use="prohibited"/>
   </restriction>
  </complexContent>
 </complexType>

 <complexType name="attributeGroupRef">
  <complexContent>
   <restriction base="attributeGroup">
    <sequence>
     <element ref="annotation" minOccurs="0"/>
    </sequence>
    <attribute name="ref" use="required" type="QName"/>
    <attribute name="name" use="prohibited"/>
   </restriction>
  </complexContent>
 </complexType>

 <element name="attributeGroup" type="namedAttributeGroup" substitutionGroup="redefinable"
  <annotation>
   <documentation
      source="http://www.w3.org/TR/xmlschema-1/#element-attributeGroup"/>
  </annotation>
 </element>

 <element name="include" id="include">
  <annotation>
   <documentation
      source="http://www.w3.org/TR/xmlschema-1/#element-include"/>
  </annotation>
  <complexType>
   <complexContent>
    <extension base="annotated">
     <attribute name="schemaLocation" type="uriReference" use="required"/>
    </extension>
   </complexContent>
  </complexType>
 </element>
```

```
 <element name="redefine" id="redefine">
  <annotation>
   <documentation
      source="http://www.w3.org/TR/xmlschema-1/#element-redefine"/>
  </annotation>
  <complexType>
   <complexContent>
    <extension base="openAttrs">
     <choice minOccurs="0" maxOccurs="unbounded">
      <element ref="annotation"/>
      <element ref="redefinable"/>
     </choice>
     <attribute name="schemaLocation" type="uriReference" use="required"/>
    </extension>
   </complexContent>
  </complexType>
 </element>

 <element name="import" id="import">
  <annotation>
   <documentation
      source="http://www.w3.org/TR/xmlschema-1/#element-import"/>
  </annotation>
  <complexType>
   <complexContent>
    <extension base="annotated">
     <attribute name="namespace" type="uriReference"/>
     <attribute name="schemaLocation" type="uriReference"/>
    </extension>
   </complexContent>
  </complexType>
 </element>

 <simpleType name="XPathExprApprox">
  <annotation>
   <documentation>An XPath expression</documentation>
   <documentation>A utility type, not for public use</documentation>
  </annotation>
  <restriction base="string">
   <pattern value="(/|//|\.|\.\.|:|::|\||(\c-[.:/|])+)+">
    <annotation>
     <documentation>A VERY permissive definition, probably not even
right</documentation>
    </annotation>
   </pattern>
  </restriction>
 </simpleType>

 <complexType name="XPathSpec">
  <complexContent>
   <extension base="annotated">
     <attribute name="xpath" type="XPathExprApprox"/>
   </extension>
  </complexContent>
 </complexType>

 <element name="selector" type="XPathSpec" id="selector">
  <annotation>
   <documentation
      source="http://www.w3.org/TR/xmlschema-1/#element-selector"/>
  </annotation>
 </element>

 <element name="field" id="field" type="XPathSpec">
  <annotation>
   <documentation
      source="http://www.w3.org/TR/xmlschema-1/#element-field"/>
  </annotation>
```

```
</element>

<complexType name="keybase">
 <complexContent>
  <extension base="annotated">
   <sequence>
    <element ref="selector"/>
    <element ref="field" minOccurs="1" maxOccurs="unbounded"/>
   </sequence>
   <attribute name="name" type="NCName" use="required"/>
  </extension>
 </complexContent>
</complexType>

<element name="identityConstraint" type="keybase" abstract="true" id="identityConstraint"
 <annotation>
  <documentation
     source="http://www.w3.org/TR/xmlschema-1/#element-identityConstraint"/>
 </annotation>
</element>

<element name="unique" substitutionGroup="identityConstraint" id="unique">
 <annotation>
  <documentation
     source="http://www.w3.org/TR/xmlschema-1/#element-unique"/>
 </annotation>
</element>
<element name="key" substitutionGroup="identityConstraint" id="key">
 <annotation>
  <documentation
     source="http://www.w3.org/TR/xmlschema-1/#element-key"/>
 </annotation>
</element>
<element name="keyref" substitutionGroup="identityConstraint" id="keyref">
 <annotation>
  <documentation
     source="http://www.w3.org/TR/xmlschema-1/#element-keyref"/>
 </annotation>
 <complexType>
  <complexContent>
   <extension base="keybase">
    <attribute name="refer" type="QName" use="required"/>
   </extension>
  </complexContent>
 </complexType>
</element>

<element name="notation" substitutionGroup="schemaTop" id="notation">
 <annotation>
  <documentation
     source="http://www.w3.org/TR/xmlschema-1/#element-notation"/>
 </annotation>
 <complexType>
  <complexContent>
   <extension base="annotated">
    <attribute name="name" type="NCName" use="required"/>
    <attribute name="public" type="public" use="required"/>
    <attribute name="system" type="uriReference"/>
   </extension>
  </complexContent>
 </complexType>
</element>

<simpleType name="public">
 <annotation>
  <documentation>A public identifier, per ISO 8879</documentation>
  <documentation>A utility type, not for public use</documentation>
 </annotation>
```

```
  <restriction base="string"/>
 </simpleType>

 <annotation>
  <documentation>notations for use within XML Schema schemas</documentation>
 </annotation>

 <notation name="XMLSchemaStructures" public="structures" system="http://www.w3.org/2000/1
 <notation name="XML" public="REC-xml-19980210" system="http://www.w3.org/TR/1998/REC-xml-

 <complexType name="anyType" mixed="true">
  <annotation>
   <documentation>Not the real urType, but as close an approximation as we can
get in the XML representation</documentation>
  </annotation>
  <sequence>
   <any minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
  <anyAttribute/>
 </complexType>
</schema>
```

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

B 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
type
[Definition:] an **type** is
type reference
[Definition:] an **type 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 reference

[Definition:] an **element 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

null

[Definition:] A distinguished value for properties of schema components in the abstract schema data

model whose value is absent.

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 colon, as defined in [XML-Namespaces]. Appears in all the definition and declaration productions of this specification.

QName

[Definition:] a **QName** is a name with an optional namespace qualification, as defined in [XML-Namespaces]. When used in connection with the XML representation of schema components in this specification, this refers to the simple type QName as defined in [XML Schemas: Datatypes].

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

C References (normative) *

Cambridge Communiqué

The Cambridge Communiqué, Ralph Swick and Henry S. Thompson, editors, 7 October 1999. See

http://www.w3.org/TR/schema-arch

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

XML Schema: Primer

XML Schema: Primer, David C. Fallside, ed. See http://www.w3.org/TR/2000/WD-xmlschema-0-20000922/primer.html

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://www.ietf.org/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/~fielding/url/draft-fielding-uri-syntax-01.txt

URL

RFC 1738,*Uniform Resource Locators (URL)*. Internet Engineering Task Force. See http://www.ietf.org/rfc/rfc1738.txt

URN

RFC 2141,*URN Syntax*. Internet Engineering Task Force. See http://www.ietf.org/rfc/rfc2141.txt

WAI-PAGEAUTH

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

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

XML Schemas: Datatypes

XML Schema Part 2: Datatypes, Paul V. Biron and Ashok Malhotra, eds. See http://www.w3.org/TR/2000/WD-xmlschema-2-20000922/datatypes.html

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

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

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

XSLT

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

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 (public WD), David Megginson et al., W3C, 1999. See http://www.w3.org/TR/xml-infoset

XML-Namespaces

Namespaces in XML, Tim Bray et al., W3C, 1998. Seehttp://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/xptr

XPath

XML Path Language, James Clark and Steve DeRose, editors, 16 November 1999. See http://www.w3.org/TR/xpath

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.htmFor more general information, consult http://purl.oclc.org/NET/xschema

D Outcome Tabulations (normative)

To facilitate consistent reporting of schema errors and schema-validation failures, this section tabulates and provides unique names for all the constraints listed in this document. Wherever such constraints have numbered parts, reports should use the name given below plus the part number, separated by a period ('.'). Thus for example `cos-ct-extends.1.2` should be used to report a violation of the second clause of Derivation Valid (Extension) (§5.11).

D.1 Constraints on Schemas and Schema Representation Constraints

ag-props-correct
: Attribute Group Definition Properties Correct

an-props-correct
: Annotation Correct

a-props-correct
: Attribute Declaration Properties Correct

cos-all-limited
: All Group Limited

cos-aw-intersect
: Attribute Wildcard Intersection

cos-choice-range
: Effective Total Range (choice)

cos-ct-derived-ok
: Type Derivation OK (Complex)

cos-ct-extends
: Derivation Valid (Extension)

cos-element-consistent

Element Declarations Consistent
cos-equiv-class
Substitution Group
cos-equiv-derived-ok-rec
Substitution Group OK (Transitive)
cos-group-emptiable
Particle Emptiable
cos-nonambig
Unique Particle Attribution
cos-ns-subset
Wildcard Subset
cos-particle-extend
Particle Valid (Extension)
cos-particle-restrict
Particle Valid (Restriction)
cos-seq-range
Effective Total Range (all and sequence)
cos-st-derived-ok
Type Derivation OK (Simple)
cos-st-restricts
Derivation Valid (Restriction, Simple)
cos-valid-default
Element Default Valid (Immediate)
c-props-correct
Identity-constraint Definition Properties Correct
ct-props-correct
Complex Type Definition Properties Correct
derivation-ok-restriction
Derivation Valid (Restriction, Complex)
e-props-correct
Element Declaration Properties Correct
mgd-props-correct
Model Group Definition Properties Correct
mg-props-correct
Model Group Correct
no-xmlns
xmlns Not Allowed
no-xsi
xsi: Not Allowed
n-props-correct
Notation Declaration Correct
p-props-correct
Particle Correct
range-ok
Occurrence Range OK
rcase-MapAndSum
Particle Derivation OK (Sequence:Choice -- MapAndSum)
rcase-NameAndTypeOK
Particle Restriction OK (Elt:Elt -- NameAndTypeOK)
rcase-NSCompat
Particle Derivation OK (Elt:Any -- NSCompat)
rcase-NSRecurseCheckCardinality

Particle Derivation OK (All/Choice/Sequence:Any -- NSRecurseCheckCardinality)
rcase-NSSubset
Particle Derivation OK (Any:Any -- NSSubset)
rcase-Recurse
Particle Derivation OK (All:All,Sequence:Sequence -- Recurse)
rcase-RecurseAsIfGroup
Particle Derivation OK (Elt:All/Choice/Sequence -- RecurseAsIfGroup)
rcase-RecurseLax
Particle Derivation OK (Choice:Choice -- RecurseLax)
rcase-RecurseUnordered
Particle Derivation OK (Sequence:All -- RecurseUnordered)
schema_reference
Schema Document Location Strategy
sch-props-correct
Schema Properties Correct
src-annotation
Annotation Definition Representation OK
src-attribute
Attribute Declaration Representation OK
src-attribute_group
Attribute Group Definition Representation OK
src-ct
Complex Type Definition Representation OK
src-element
Element Declaration Representation OK
src-expredef
Individual Component Redefinition
src-identity-constraint
Identity-constraint Definition Representation OK
src-import
Import Constraints and Semantics
src-include
Inclusion Constraints and Semantics
src-model_group
Model Group Representation OK
src-model_group_defn
Model Group Definition Representation OK
src-notation
Notation Definition Representation OK
src-qname
QName Interpretation
src-redefine
Redefinition Constraints and Semantics
src-resolve
QName resolution (Schema Document)
src-wildcard
Wildcard Representation OK
st-props-correct
Simple Type Definition Properties Correct
st-restrict-facets
Simple Type Restriction (Facets)
w-props-correct

Wildcard Properties Correct

D.2 Validity Contributions

cvc-attribute
Attribute Valid
cvc-attr-lax
Attribute Valid (Lax)
cvc-complex-type
Element Children and Attributes Valid
cvc-elt
Element Valid (Explicit)
cvc-elt-lax
Element Valid (Lax)
cvc-elt-strict
Element Valid (Strict)
cvc-identity-constraint
Identity-constraint Satisfied
cvc-model-group
Element Sequence Valid
cvc-particle
Element Sequence Valid (Particle)
cvc-resolve-instance
QName resolution (Instance)
cvc-simple-type
String Valid
cvc-wildcard
Item Valid (Wildcard)
cvc-wildcard-namespace
Wildcard allows Namespace URI

D.3 Post-Schema-Validation Infoset Contributions

sic-a-outcome
Validation Outcome (Attribute)
sic-attrDefault
Attribute Default Value
sic-attrType
Attribute Validated by Type
sic-ct-error-code
Validation Failure (Complex Type)
sic-ct-outcome
Validation Outcome (Complex Type)
sic-elt-decl
Element Declaration
sic-eltDefault
Element Default Value
sic-eltType
Element Validated by Type
sic-e-outcome
Validation Outcome (Element)
sic-key

Identity-constraint Table
sic-schema
Schema Information
sic-skipped
Validation Outcome (skipped)

E (non-normative) DTD for Schemas

The DTD for *XML Schema: Structures* is given below. Note there is *no* implication here the `schema` must be the root element of a document.

Although this DTD is non-normative, any XML document which is not valid per this DTD, given redefinitions in its internal subset of the 'p' and 's' parameter entities below appropriate to its namespace declaration of the XML Schema namespace, is almost certainly not a valid schema document, with the exception of documents with multiple namespace prefixes for the XML Schema namespace itself. Accordingly authoring XML Schema documents using this DTD and DTD-based authoring tools, and specifying it as the DOCTYPE of documents intended to be XML Schema documents and validating them with a validating XML parser, are sensible development strategies which users are encouraged to adopt until XML Schema-based authoring tools and validators are more widely available.

```
<!-- DTD for XML Schemas: Part 1: Structures
     Public Identifier: "-//W3C//DTD XMLSCHEMA 200010//EN"
     Official Location: http://www.w3.org/2000/10/XMLSchema.dtd -->
<!-- Id: XMLSchema.dtd,v 1.15 2000/09/22 13:04:59 ht Exp  -->
<!-- With the exception of cases with multiple namespace
     prefixes for the XML Schema namespace, any XML document which is
     not valid per this DTD given redefinitions in its internal subset of the
     'p' and 's' parameter entities below appropriate to its namespace
     declaration of the XML Schema namespace is almost certainly not
     a valid schema. -->

<!-- The the datatype element and its components
     are defined in XML Schema: Part 2: Datatypes -->
<!ENTITY % xs-datatypes PUBLIC 'datatypes' 'datatypes.dtd' >

<!ENTITY % p ''> <!-- can be overriden in the internal subset of a
                      schema document to establish a namespace prefix -->
<!ENTITY % s ''> <!-- if %p is defined (e.g. as foo:) then you must
                      also define %s as the suffix for the appropriate
                      namespace declaration (e.g. :foo) -->
<!ENTITY % nds 'xmlns%s;'>

<!-- Define all the element names, with optional prefix -->
<!ENTITY % schema "%p;schema">
<!ENTITY % complexType "%p;complexType">
<!ENTITY % complexContent "%p;complexContent">
<!ENTITY % simpleContent "%p;simpleContent">
<!ENTITY % extension "%p;extension">
<!ENTITY % element "%p;element">
<!ENTITY % unique "%p;unique">
<!ENTITY % key "%p;key">
<!ENTITY % keyref "%p;keyref">
<!ENTITY % selector "%p;selector">
<!ENTITY % field "%p;field">
<!ENTITY % group "%p;group">
<!ENTITY % all "%p;all">
<!ENTITY % choice "%p;choice">
<!ENTITY % sequence "%p;sequence">
<!ENTITY % any "%p;any">
<!ENTITY % anyAttribute "%p;anyAttribute">
<!ENTITY % attribute "%p;attribute">
```

```
<!ENTITY % attributeGroup "%p;attributeGroup">
<!ENTITY % include "%p;include">
<!ENTITY % import "%p;import">
<!ENTITY % redefine "%p;redefine">
<!ENTITY % notation "%p;notation">

<!-- Customisation entities for the ATTLIST of each element type.
     Define one of these if your schema takes advantage of the
     anyAttribute='##other' in the schema for schemas -->

<!ENTITY % schemaAttrs ''>
<!ENTITY % complexTypeAttrs ''>
<!ENTITY % complexContentAttrs ''>
<!ENTITY % simpleContentAttrs ''>
<!ENTITY % extensionAttrs ''>
<!ENTITY % elementAttrs ''>
<!ENTITY % groupAttrs ''>
<!ENTITY % allAttrs ''>
<!ENTITY % choiceAttrs ''>
<!ENTITY % sequenceAttrs ''>
<!ENTITY % anyAttrs ''>
<!ENTITY % anyAttributeAttrs ''>
<!ENTITY % attributeAttrs ''>
<!ENTITY % attributeGroupAttrs ''>
<!ENTITY % uniqueAttrs ''>
<!ENTITY % keyAttrs ''>
<!ENTITY % keyrefAttrs ''>
<!ENTITY % selectorAttrs ''>
<!ENTITY % fieldAttrs ''>
<!ENTITY % includeAttrs ''>
<!ENTITY % importAttrs ''>
<!ENTITY % redefineAttrs ''>
<!ENTITY % notationAttrs ''>

<!ENTITY % complexDerivationChoice "(extension|restriction)">
<!ENTITY % complexDerivationSet "CDATA">
      <!-- #all or space-separated list drawn from derivationChoice -->
<!ENTITY % blockSet "CDATA">
      <!-- #all or space-separated list drawn from
                       derivationChoice + 'substitution' -->

<!ENTITY % mgs '%all; | %choice; | %sequence;'>
<!ENTITY % cs '%choice; | %sequence;'>
<!ENTITY % formValues '(qualified|unqualified)'>


<!ENTITY % attrDecls    '((%attribute;| %attributeGroup;)*,(%anyAttribute;)?)'>

<!ENTITY % particleAndAttrs '((%mgs; | %group;)?, %attrDecls;)'>

<!-- This is used in part2 -->
<!ENTITY % restriction1 '((%mgs; | %group;)?)'>

%xs-datatypes;

<!-- the duplication below is to produce an unambiguous content model
     which allows annotation everywhere -->
<!ELEMENT %schema; ((%include; | %import; | %redefine; | %annotation;)*,
                    ((%simpleType; | %complexType;
                      | %element; | %attribute;
                      | %attributeGroup; | %group;
                      | %notation; ),
                     (%annotation;)*)* )>
<!ATTLIST %schema;
   targetNamespace      %URIref;              #IMPLIED
   version              CDATA                 #IMPLIED
   %nds;                %URIref;              #FIXED 'http://www.w3.org/2000/10/XMLSchema
   finalDefault         %complexDerivationSet; ''
```

```
   blockDefault         %blockSet;              ''
   id                   ID                      #IMPLIED
   elementFormDefault   %formValues;            'unqualified'
   attributeFormDefault %formValues;            'unqualified'
   %schemaAttrs;>
<!-- Note the xmlns declaration is NOT in the Schema for Schemas,
     because at the Infoset level where schemas operate,
     xmlns(:prefix) is NOT an attribute! -->

<!-- The id attribute here and below is for use in external references
     from non-schemas using simple fragment identifiers.
     It is NOT used for schema-to-schema reference, internal or
     external. -->

<!-- a type is a named content type specification which allows attribute
     declarations-->
<!-- -->

<!ELEMENT %complexType; ((%annotation;)?,
                         (%simpleContent;|%complexContent;|
                          %particleAndAttrs;))>

<!ATTLIST %complexType;
          name      %NCName;                        #IMPLIED
          id        ID                              #IMPLIED
          abstract  %boolean;                       'false'
          final     %complexDerivationSet;          #IMPLIED
          block     %complexDerivationSet;          ''
          mixed (true|false) 'false'
          %complexTypeAttrs;>

<!-- particleAndAttrs is shorthand for a root type -->
<!-- mixed is disallowed if simpleContent, overriden if complexContent
     has one too. -->

<!-- If anyAttribute appears in one or more referenced attributeGroups
     and/or explicitly, the intersection of the permissions is used -->

<!ELEMENT %complexContent; (%restriction;|%extension;)>
<!ATTLIST %complexContent;
          mixed (true|false) #IMPLIED
          %complexContentAttrs;>

<!-- restriction should use the branch defined above, not the simple
     one from part2; extension should use the full model  -->

<!ELEMENT %simpleContent; (%restriction;|%extension;)>
<!ATTLIST %simpleContent; %simpleContentAttrs;>

<!-- restriction should use the simple branch from part2, not the
     one defined above; extension should have no particle  -->

<!ELEMENT %extension; (%particleAndAttrs;)>
<!ATTLIST extension
          base      %QName;                         #REQUIRED
          %extensionAttrs;>

<!-- an element is declared by either:
 a name and a type (either nested or referenced via the type attribute)
 or a ref to an existing element declaration -->

<!ELEMENT %element; ((%annotation;)?, (%complexType;| %simpleType;)?,
                     (%unique; | %key; | %keyref;)*)>
<!-- simpleType or complexType only if no type|ref attribute -->
<!-- ref not allowed at top level -->
<!ATTLIST %element;
            name              %NCName;              #IMPLIED
            id                ID                    #IMPLIED
```

```
            ref                 %QName;                 #IMPLIED
            type                %QName;                 #IMPLIED
            minOccurs           %nonNegativeInteger;    #IMPLIED
            maxOccurs           CDATA                   #IMPLIED
            nullable            %boolean;               'false'
            substitutionGroup   %QName;                 #IMPLIED
            abstract            %boolean;               'false'
            final               %complexDerivationSet; #IMPLIED
            block               %blockSet;              ''
            default             CDATA                   #IMPLIED
            fixed               CDATA                   #IMPLIED
            form                %formValues;            #IMPLIED
            %elementAttrs;>
<!-- type and ref are mutually exclusive.
     name and ref are mutually exculsive, one is required -->
<!-- In the absence of type AND ref, type defaults to type of
     substitutionGroup, if any, else the ur-type, i.e. unconstrained -->
<!-- default and fixed are mutually exclusive -->

<!ELEMENT %group; ((%annotation;)?,(%mgs;)?)>
<!ATTLIST %group;
          name        %NCName;               #IMPLIED
          ref         %QName;                #IMPLIED
          minOccurs   %nonNegativeInteger;   #IMPLIED
          maxOccurs   CDATA                  #IMPLIED
          id          ID                     #IMPLIED
          %groupAttrs;>

<!ELEMENT %all; ((%annotation;)?, (%element;)*)>
<!ATTLIST %all;
          minOccurs   (0|1)                  '1'
          maxOccurs   (0|1)                  '1'
          id          ID                     #IMPLIED
          %allAttrs;>

<!ELEMENT %choice; ((%annotation;)?, (%element;| %group;| %cs; | %any;)*)>
<!ATTLIST %choice;
          minOccurs   %nonNegativeInteger;   '1'
          maxOccurs   CDATA                  '1'
          id          ID                     #IMPLIED
          %choiceAttrs;>

<!ELEMENT %sequence; ((%annotation;)?, (%element;| %group;| %cs; | %any;)*)>
<!ATTLIST %sequence;
          minOccurs   %nonNegativeInteger;   '1'
          maxOccurs   CDATA                  '1'
          id          ID                     #IMPLIED
          %sequenceAttrs;>

<!-- an anonymous grouping in a model, or
     a top-level named group definition, or a reference to same -->

<!-- Note that if order is 'all', group is not allowed inside.
     If order is 'all' THIS group must be alone (or referenced alone) at
     the top level of a content model -->
<!-- If order is 'all', minOccurs==maxOccurs==1 on element/any inside -->
<!-- Should allow minOccurs=0 inside order='all' . . . -->

<!ELEMENT %any; (%annotation;)?>
<!ATTLIST %any;
            namespace       CDATA                   '##any'
            processContents (skip|lax|strict)       'strict'
            minOccurs       %nonNegativeInteger;    '1'
            maxOccurs       CDATA                   '1'
            %anyAttrs;>

<!-- namespace is interpreted as follows:
                 ##any      - - any non-conflicting WFXML at all
```

```
                  ##other    - - any non-conflicting WFXML from namespace other
                                  than targetNamespace

                  ##local    - - any unqualified non-conflicting WFXML/attribute
                  one or     - - any non-conflicting WFXML from
                  more URI        the listed namespaces
                  references

                  ##targetNamespace ##local may appear in the above list,
                    with the obvious meaning -->

<!ELEMENT %anyAttribute; (%annotation;)?>
<!ATTLIST %anyAttribute;
            namespace       CDATA              '##any'
            processContents (skip|lax|strict)  'strict'
            %anyAttributeAttrs;>
<!-- namespace is interpreted as for 'any' above -->

<!-- simpleType only if no type|ref attribute -->
<!-- ref not allowed at top level, name iff at top level -->
<!ELEMENT %attribute; ((%annotation;)?, (%simpleType;)?)>
<!ATTLIST %attribute;
          name      %NCName;      #IMPLIED
          id        ID            #IMPLIED
          ref       %QName;       #IMPLIED
          type      %QName;       #IMPLIED
          use       (prohibited|optional|required|fixed|default) #IMPLIED
          value     CDATA         #IMPLIED
          form      %formValues;  #IMPLIED
          %attributeAttrs;>
<!-- type and ref are mutually exclusive.
     name and ref are mutually exculsive, one is required -->
<!-- value only if use is fixed, required or default, or name -->
<!-- name and use are mutually exclusive -->
<!-- default for use is optional when nested, none otherwise -->
<!-- type attr and simpleType content are mutually exclusive -->

<!-- an attributeGroup is a named collection of attribute decls, or a
     reference thereto -->
<!ELEMENT %attributeGroup; ((%annotation;)?,
                      (%attribute; | %attributeGroup;)*,
                      (%anyAttribute;)?) >
<!ATTLIST %attributeGroup;
                  name       %NCName;       #IMPLIED
                  id         ID             #IMPLIED
                  ref        %QName;        #IMPLIED
                  %attributeGroupAttrs;>

<!-- ref iff no content, no name.  ref iff not top level -->

<!-- better reference mechanisms -->
<!ELEMENT %unique; ((%annotation;)?, %selector;, (%field;)+)>
<!ATTLIST %unique; name     %NCName;       #REQUIRED
                   id       ID             #IMPLIED
                   %uniqueAttrs;>

<!ELEMENT %key;    ((%annotation;)?, %selector;, (%field;)+)>
<!ATTLIST %key;    name     %NCName;       #REQUIRED
                   id       ID             #IMPLIED
                   %keyAttrs;>

<!ELEMENT %keyref; ((%annotation;)?, %selector;, (%field;)+)>
<!ATTLIST %keyref;
                   name     %NCName;       #REQUIRED
                   id       ID             #IMPLIED
                   refer    %QName;        #REQUIRED
                   %keyrefAttrs;>
```

```
<!ELEMENT %selector; ((%annotation;)?)>
<!ATTLIST %selector;
          xpath %XPathExpr; #REQUIRED
          %selectorAttrs;>
<!ELEMENT %field; ((%annotation;)?)>
<!ATTLIST %field;
          xpath %XPathExpr; #REQUIRED
          %fieldAttrs;>

<!-- Schema combination mechanisms -->
<!ELEMENT %include; (%annotation;)?>
<!ATTLIST %include; schemaLocation %URIref; #REQUIRED
                    %includeAttrs;>

<!ELEMENT %import; (%annotation;)?>
<!ATTLIST %import; namespace      %URIref; #IMPLIED
                   schemaLocation %URIref; #IMPLIED
                   %importAttrs;>

<!ELEMENT %redefine; (%annotation; | %simpleType; | %complexType; |
                      %attributeGroup; | %group;)*>
<!ATTLIST %redefine; schemaLocation %URIref; #REQUIRED
                     %redefineAttrs;>

<!ELEMENT %notation; (%annotation;)?>
<!ATTLIST %notation;
                 name        %NCName;    #REQUIRED
                 id          ID          #IMPLIED
                 public      CDATA       #REQUIRED
                 system      %URIref;    #IMPLIED
                 %notationAttrs;>

<!NOTATION XMLSchemaStructures PUBLIC 'structures'
           'http://www.w3.org/2000/10/XMLSchema.xsd' >
<!NOTATION XML PUBLIC 'REC-xml-1998-0210'
               'http://www.w3.org/TR/1998/REC-xml-19980210' >
```

F (non-normative) Analysis of the Unique Particle Attribution constraint

A specification of the import of Unique Particle Attribution (§5.7) which does not appeal to a processing model is difficult. What follows is intended as guidance, without claiming to be complete.

[Definition:] We say that two non-group particles **overlap** if

- They are both element declaration particles whose declarations have the same {name} and {target namespace}

or

- They are both element declaration particles one of which is in the other's substitution group

or

- They are both wildcards, and the intensional intersection of their {namespace constraint}s as defined in Attribute Wildcard Intersection (§5.4) is not the empty set

or

- One is a wildcard and the other an element declaration, and the {target namespace} of the element declaration, or of any member of its substitution group, is schema-valid with respect to the {namespace constraint} of the wildcard.

A content model will violate the unique attribution constraint if it contains two particles which overlap and which either

- are both in the {particles} of a *choice* or *all* group

or

- may schema-validate adjacent information items and the first has {min occurs} less than {max occurs}

Two particles may validate adjacent information items if they are separated by at most epsilon transitions in the most obvious transcription of a content model into a finite-state automaton.

A precise formulation of this constraint can also be offered in terms of operations on finite-state automaton: transcribe the content model into an automaton in the usual way using epsilon transitions for optionality and unbounded maxOccurs, unfolding other numeric occurrence ranges and treating the heads of substitution groups as if they were choices over all elements in the group,*but* using not element QNames as transition labels, but rather pairs of element QNames and positions in the model. Determinise this automaton, treating wildcard transitions as opaque. Now replace all QName+position transition labels with the element QNames alone. If the result has any states with two or more identical-QName-labelled transitions from it, or a QName-labelled transition and a wildcard transition which subsumes it, or two wildcard transitions whose intentional intersection is non-empty, the model does not satisfy the Unique Attribution constraint.

G Acknowledgements (non-normative)

The following have contributed material to this draft:

- David Fallside, IBM
- Scott Lawrence, Agranat Systems
- Andrew Layman, Microsoft
- Eve L. Maler, Sun Microsystems

The editors acknowledge the members of the 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 Working Group is particularly grateful to Lotus Development Corp. and IBM for providing teleconferencing facilities.

The current members of the XML Schema Working Group are:

Jim Barnette, Defense Information Systems Agency (DISA); David Beech, Oracle Corp.; Paul V. Biron, Health Level Seven; Don Box, DevelopMentor; Allen Brown, Microsoft; Lee Buck, Extensibility; Charles E. Campbell, Informix; Peter Chen, Bootstrap Alliance and LSU; David Cleary, Progress Software; Dan Connolly, W3C (staff contact); Roger L. Costello, MITRE; Ugo Corda, Xerox; Andrew Eisenberg, Progress Software; David Ezell, Hewlett Packard Company; David Fallside, IBM; Matthew Fuchs, Commerce One; Andrew Goodchild, Distributed Systems Technology Centre (DSTC Pty Ltd); Paul Grosso, ArborText, Inc; Martin Gudgin, DevelopMentor; Dave Hollander, CommerceNet (co-chair); Mary Holstege, Calico Commerce; Jane Hunter, Distributed Systems Technology Centre (DSTC Pty Ltd); Rick Jelliffe, Academia Sinica; Andrew Layman, Microsoft; Dmitry Lenkov, Hewlett Packard Company; Eve Maler, Sun Microsystems; Ashok Malhotra, IBM; Murray Maloney, Commerce One; John McCarthy, Lawrence Berkeley National Laboratory; Noah Mendelsohn,

Lotus Development Corporation; Don Mullen, Extensibility; Frank Olken, Lawrence Berkeley National Laboratory; Dave Peterson, Graphic Communications Association; Mark Reinhold, Sun Microsystems; Jonathan Robie, Software AG; Lew Shannon, NCR; C. M. Sperberg-McQueen, W3C (co-chair); Bob Streich, Calico Commerce; Henry S. Thompson, University of Edinburgh; Matt Timmermans, Microstar; Jim Trezzo, Oracle Corp.; Steph Tryphonas, Microstar; Mark Tucker, Health Level Seven; Asir S. Vedamuthu, webMethods, Inc; Priscilla Walmsley, XMLSolutions; Norm Walsh, ArborText, Inc; Aki Yoshida, SAP AG

The XML Schema Working Group has benefited in its work from the participation and contributions of a number of people not currently members of the Working Group, including in particular those named below. Affiliations given are those current at the time of their work with the WG.

Paula Angerstein, Vignette Corporation; Gabe Beged-Dov, Rogue Wave Software; Greg Bumgardner, Rogue Wave Software; Dean Burson, Lotus Development Corporation; Rob Ellman, Calico Commerce; George Feinberg, Object Design; Charles Frankston, Microsoft; Ernesto Guerrieri, Inso; Michael Hyman, Microsoft; Renato Iannella, Distributed Systems Technology Centre (DSTC Pty Ltd); Dianne Kennedy, Graphic Communications Association; Janet Koenig, Sun Microsystems; Setrag Khoshafian, Technology Deployment International (TDI); Ara Kullukian, Technology Deployment International (TDI); Murata Makoto, Xerox; Chris Olds, Wall Data; Shriram Revankar, Xerox; John C. Schneider, MITRE; William Shea, Merrill Lynch; Ralph Swick, W3C; Tony Stewart, Rivcom

H Description of changes (non-normative)

This section gives brief summaries of the substantive changes to this specification since the public working draft of 7 April 2000.

H.1 Equivalence classes renamed

Equivalence classes have been renamed substitution groups, to reflect the fact that their semantics is not symmetrical.

H.2 Content model of `complexType` element changed

The content model of the complexType element has been significantly changed, allowing for tighter content models and a better fit between the abstract component and its XML Representation. The side conditions on well-formed representations not captured in the schema for schemas have accordingly been downsized. No changes to the abstract complex type definition component were involved.

H.3 Declaring empty and mixed content models

Part of the change to the complexType element described immediately above involved eliminating its `content` attribute. Empty content models are now signalled by an explicit empty content particle (see XML Representation of Model Group Schema Components (§4.3.6)), mixed content by specifying the value `true` for the `mixed` attribute on complexType or complexContent.

H.4 Simple type definitions changed

Both the abstract component and the XML representation for simple type definitions have been changed, the former to handle list type definitions more cleanly and to support union type definitions, the latter to give tighter content models and a better fit between the abstract component and its XML Representation.

H.5 Simple value normalization

All values governed by simple type definitions are now subject to normalization, as in XML 1.0 attribute value normalization.

H.6 Schema component redefinition

A new form of schema composition operation, similar to that provided byinclude but allowing constrained redefinition of the included components has been added, using aredefine element.

H.7 Element and attribute reference restricted

The ability to override properties of global element declarations when referencing them from complex type definitions has been removed. As a consequence of this attribute declarations no longer have max- and minOccurs properties.

H.8 Default values for `minOccurs` and `maxOccurs` attributes

The defaulting for these attributes of element has been simplified: it is now`1` in both cases, with no interdependencies.

H.9 Content model for Model Group definition

The content model for the group element when it occurs at the top level has been tightened, to allow only a single all, choice, group or sequence child.

H.10 XML Schema namespace URI

In recognition of the above changes, the namespace URI for XML Schema ahs been changed to `http://www.w3.org/2000/10/XMLSchema`. There has been no change to the XML Schema instance namespace URI, which remains`http://www.w3.org/2000/10/XMLSchema-instance`.

H.11 Error codes

A standard format for identifying validation failures and schema form errors is now provided, and a post-schema-validation infoset property specified which processors may, but need not, use to record validation failure codes.

H.12 DTD non-normative

To avoid potential confusion the DTD for schemas is no longer normative, but its use is still encouraged.

H.13 Abstract types in element declarations

These had inadvertently been disallowed: they are now allowed, since the use of either substitution groups or `xsi:type` may derive a non-abstract type in all actual occurrences.

H.14 Schema components as info items (optional)

Processors may, but need not, provide detailed information in the form of post-schema-validation infoset contributions of information items corresponding to element declaration and type definition components. They

may also do the same for schema components in general.

H.15 Ur-types

The structure and nature of the ur-type has been clarified, with names provided for both its complex and simple forms. Approximations to definitions at both the component and XML representation layer are now provided. The status of lists and unions wrt the type hierarchy has been regularised.

H.16 Type-related validation properties

In the post-schema-validation infoset, element and attribute items now always have type information, even when validation is lax. In this case the type information given is the simple or complex ur-type, as appropriate.

H.17 Facilitate online schema validation

The first `xsi:schemaLocation` for a namespace must not be preceded by any names from that namespace; otherwise it is an error.

H.18 Complex type definitions by extension

Complex type definitions which extended other complex type definitions whose base type definition is simple had inadvertently been disallowed: this is now allowed.

dummy