XML Schema Part 1: Structures Second Edition

W3C Recommendation 28 October 2004

This version:
http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/
Latest version:
http://www.w3.org/TR/xmlschema-1/
Previous version:
http://www.w3.org/TR/2004/PER-xmlschema-1-20040318/
Editors:
Henry S. Thompson, University of Edinburgh <ht@cogsci.ed.ac.uk>
David Beech, Oracle Corporation <David.Beech@oracle.com>
Murray Maloney, for Commerce One <murray@muzmo.com>
Noah Mendelsohn, Lotus Development Corporation <Noah_Mendelsohn@lotus.com>

Please refer to the **errata** for this document, which may include some normative corrections.

This document is also available in these non-normative formats: XML, XHTML with visible change markup, Independent copy of the schema for schema documents, and Independent copy of the DTD for schema documents. See also **translations**.

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, including those which exploit the XML Namespace facility. The schema language, which is itself represented in XML 1.0 and uses namespaces, substantially reconstructs and considerably extends 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 section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

This is a W3C Recommendation, which forms part of the Second Edition of XML Schema. This document has been reviewed by W3C Members and other interested parties and has been endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited as a normative reference from another document. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

This document has been produced by the W3C XML Schema Working Group as part of the W3C XML Activity. The goals of the XML Schema language are discussed in the XML Schema Requirements document. The authors of this document are the members of the XML Schema Working Group. Different parts of this specification have different editors.

This document was produced under the 24 January 2002 Current Patent Practice (CPP) as amended by the W3C Patent Policy Transition Procedure. The Working Group maintains a public list of patent disclosures relevant to this document; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) with respect to this specification should disclose the information in accordance with section 6 of the W3C Patent Policy.

The English version of this specification is the only normative version. Information about translations of this document is available at http://www.w3.org/2001/05/xmlschema-translations.

This second edition is *not* a new version, it merely incorporates the changes dictated by the corrections to errors found in the first edition as agreed by the XML Schema Working Group, as a convenience to readers. A separate list of all such corrections is available at http://www.w3.org/2001/05/xmlschema-errata.

The errata list for this second edition is available at http://www.w3.org/2004/03/xmlschema-errata.

Please report errors in this document to www-xml-schema-comments@w3.org (archive).

> **Note:** David Beech has retired since the publication of the first edition, and can be reached at davidbeech@earthlink.net.
>
> Murray Maloney is no longer affiliated with Commerce One; his contact details are unchanged.
>
> Noah Mendelsohn's affiliation has changed since the publication of the first edition. He is now at IBM, and can be contacted at noah_mendelsohn@us.ibm.com

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 an introduction to 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, the representation of each component in XML, with reference to a DTD and XML Schema for an XML Schema document type, along with a detailed mapping between the elements and attribute vocabulary of this representation and the components and properties of the abstract model.

Chapter 4 presents Schemas and Namespaces: Access and Composition (§4), including the connection between documents and schemas, the import, inclusion and redefinition of declarations and definitions and the foundations of schema-validity assessment.

Chapter 5 discusses Schemas and Schema-validity Assessment (§5), including the overall approach to schema-validity assessment of documents, and responsibilities of schema-aware processors.

The normative appendices include a Schema for Schemas (normative) (§A) for the XML representation of schemas and References (normative) (§B).

The non-normative appendices include the DTD for Schemas (non-normative) (§G) and a Glossary (non-normative) (§F).

This document is primarily intended as a language definition reference. As such, although it contains a few examples, it is *not* primarily designed to serve as a motivating introduction to the design and its features, or as a tutorial for new users. Rather it presents 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 normalization and defaulting of attribute and element values. 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 implemented 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: [XML-Infoset], [XML-Namespaces], [XPath], and [XML Schemas: Datatypes].

See Required Information Set Items and Properties (normative) (§D) for a tabulation of the information items and properties specified in [XML-Infoset] which this specification requires as a precondition to schema-aware processing.

1.3 Documentation Conventions and Terminology

The section introduces the highlighting and typography as used in this document to present technical material.

Special terms are defined at their point of introduction in the text. For example [Definition:] a **term** is something used with a special meaning. The definition is labeled as such and the term it defines is displayed in boldface. The end of the definition is not specially marked in the displayed or printed text. Uses of defined terms are links to their definitions, set off with middle dots, for instance ·term·.

Non-normative examples are set off in boxes and accompanied by a brief explanation:

Example

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

And an explanation of the example.

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 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. This is 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 context, are given. The property correspondences are normative, as are the illustrations of the XML representation element information items.

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. Where an attribute information item has a built-in simple type definition defined in [XML Schemas: Datatypes], a hyperlink to its definition therein is given.

The allowed content of the information item is shown as a grammar fragment, using the Kleene operators `?`, * and +. Each element name therein is a hyperlink to its own illustration.

Note: The illustrations are derived automatically from the Schema for Schemas (normative) (§A). In the case of apparent conflict, the Schema for Schemas (normative) (§A) takes precedence, as it, together with the ·Schema Representation Constraints·, provide the normative statement of the form of XML representations.

XML Representation Summary: `example` Element Information Item

```
<example
  count = integer
  size = (large | medium | small) : medium>
  Content: (all | any*)
</example>
```

Example Schema Component

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

References to elements in the text are links to the relevant illustration as exemplified above, set off with angle brackets, for instance <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].

Properties which this specification defines for information items are introduced as follows:

PSVI Contributions for example information items
[new property] The value the property gets.

References to properties of information items defined in this specification are notated as links to their introduction as exemplified above, set off with square brackets, for example [new property].

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

Note: General comments directed to all readers.

Following [XML 1.0 (Second Edition)], within normative prose in this specification, the words *may* and *must* are defined as follows:

may
Conforming documents and XML Schema-aware processors are permitted to but need not behave as described.
must
Conforming documents and XML Schema-aware processors are required to behave as described; otherwise they are in error.

Note however that this specification provides a definition of error and of conformant processors' responsibilities with respect to errors (see Schemas and Schema-validity Assessment (§5)) which is considerably more complex than that of [XML 1.0 (Second Edition)].

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, including 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] for a tutorial introduction, and only then consult the sub-sections of Schema Component Details (§3) named *XML Representation of ...* for the details.

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 and attribute 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 normalized and/or default values for attributes and elements and the types of element and attribute information items. [Definition:] We refer to the augmented infoset which results from conformant processing as defined in this specification as the **post-schema-validation infoset**, or PSVI.

Schema-validity assessment has two aspects:

1 Determining local schema-validity, that is whether an element or attribute information item satisfies the constraints embodied in the relevant components of an XML Schema;
2 Synthesizing an overall validation outcome for the item, combining local schema-validity with the results of schema-validity assessments of its descendants, if any, and adding appropriate augmentations to the infoset to record this outcome.

Throughout this specification, [Definition:] the word **valid** and its derivatives are used to refer to clause 1 above, the determination of local schema-validity.

Throughout this specification, [Definition:] the word **assessment** is used to refer to the overall process of local validation, schema-validity assessment and infoset augmentation.

2.2 XML Schema Abstract Data Model

This specification builds on [XML 1.0 (Second Edition)] 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 1.0 (Second Edition)]) and namespace conformance (as defined in [XML-Namespaces]) for all candidates for ·assessment· and for all ·schema documents·.

Just as [XML 1.0 (Second Edition)] 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 XML interchange format for schemas is provided.

[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 13 kinds of component in all, falling into three groups. The primary components, which may (type definitions) or must (element and attribute declarations) have names are as follows:

- Simple type definitions
- Complex type definitions

- Attribute declarations
- Element declarations

The secondary components, which must have names, are as follows:

- 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:

- Annotations
- Model groups
- Particles
- Wildcards
- Attribute Uses

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 name, 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 name 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 ·assessment· 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 components, in which each schema document contributes definitions and declarations to a single target namespace.

·Validation·, defined in detail in Schema Component Details (§3), is a relation between information items and schema components. For example, an attribute information item may ·validate· with respect to an attribute declaration, a list of element information items may ·validate· 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 ·validation·.

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. The subsections below first 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 complex type definition, the **ur-type definition**, whose name is *anyType* in the XML Schema namespace, is present in each ·XML Schema·, serving as the root of the type definition hierarchy for that schema.

[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 the values of attributes and the 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 type definitions, this is [Definition:] the **simple ur-type definition**, a special restriction of the ·ur-type definition·, whose name is **anySimpleType** in the XML Schema namespace. The ·simple ur-type definition· 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 mapping from lexical space to value space is unspecified for items whose type definition is the ·simple ur-type definition·. Accordingly this specification does not constrain processors' behaviour in areas where this mapping is implicated, for example checking such items against enumerations, constructing default attributes or elements whose declared type definition is the ·simple ur-type definition·, checking identity constraints involving such items.

Note: The Working Group expects to return to this area in a future version of this specification.

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. Such list and union simple type definitions are also restrictions of the ·simple ur-type definition·.

For detailed information on simple type definitions, see Simple Type Definitions (§3.14) and [XML Schemas: Datatypes]. The latter also defines an extensive inventory of pre-defined simple types.

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 (that is, to be empty), to be a string which belongs to a particular simple type or to contain a sequence of element information items which conforms to a particular model group, with or without character information items as well.

Each complex type definition other than the ·ur-type definition· is either

- a restriction of a complex ·base type definition·

or

- an ·extension· of a simple or complex ·base 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.

For detailed information on complex type definitions, see Complex Type Definitions (§3.4).

2.2.2 Declaration Components

There are three kinds of declaration component: element, attribute, and notation. Each is 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 (possibly empty) set of identity-constraint definitions. The association is either global or scoped to a containing complex type definition. A top-level element declaration with name 'A' is broadly comparable to a pair of DTD declarations as follows, where the associated type definition fills in the ellipses:

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

Element declarations contribute to ·validation· 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·.

For detailed information on element declarations, see Element Declarations (§3.3).

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 top-level element declaration can serve as the defining member, or head, for an element substitution group. Other top-level 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 other 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 Declarations (§3.3)).

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 ·validation· 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.

For detailed information on attribute declarations, see Attribute Declarations (§3.2).

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 ·valid· with respect to a `NOTATION` simple type definition, its value must have been declared with a notation declaration.

For detailed information on notation declarations, see Notation Declarations (§3.12).

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.

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).

For detailed information on model groups, see Model Groups (§3.8).

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 ·validation· as part of complex type definition ·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 constrain the ·validation· of 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 1.0 (Second Edition)] content models; unlike [XML 1.0 (Second Edition)], *XML Schema: Structures* applies ·content models· to the ·validation· of both mixed and element-only content.

For detailed information on particles, see Particles (§3.9).

2.2.3.3 Attribute Use

An attribute use plays a role similar to that of a particle, but for attribute declarations: an attribute declaration within a complex type definition is embedded within an attribute use, which specifies whether the declaration requires or merely allows its attribute, and whether it has a default or fixed value.

2.2.3.4 Wildcard

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

For detailed information on wildcards, see Wildcards (§3.10).

2.2.4 Identity-constraint Definition Components

An 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 ·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.

For detailed information on identity-constraint definitions, see Identity-constraint Definitions (§3.11).

2.2.5 Group Definition Components

There are two kinds of convenience definitions provided to enable the re-use of 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, enabling re-use of the same model group in several complex type definitions.

For detailed information on model group definitions, see Model Group Definitions (§3.7).

2.2.5.2 Attribute Group Definition

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

For detailed information on attribute group definitions, see Attribute Group Definitions (§3.6).

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.

For detailed information on annotations, see Annotations (§3.13).

2.3 Constraints and Validation Rules

The [XML 1.0 (Second Edition)] 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 focused on ·validation·, 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 ·validation· of information items:

Schema Component Constraint
: [Definition:] Constraints on the schema components themselves, i.e. conditions components must satisfy to be components at all. Located in the sixth sub-section of the per-component sections of Schema Component Details (§3) and tabulated in Schema Component Constraints (§C.4).

Schema Representation Constraint
: [Definition:] Constraints on the representation of schema components in XML beyond those which are expressed in Schema for Schemas (normative) (§A). Located in the third sub-section of the per-component sections of Schema Component Details (§3) and tabulated in Schema Representation Constraints (§C.3).

Validation Rules
: [Definition:] Contributions to ·validation· associated with schema components. Located in the fourth sub-section of the per-component sections of Schema Component Details (§3) and tabulated in Validation Rules (§C.1).

Schema Information Set Contribution
: [Definition:] Augmentations to ·post-schema-validation infoset·s expressed by schema components, which follow as a consequence of ·validation· and/or ·assessment·. Located in the fifth sub-section of the per-component sections of Schema Component Details (§3) and tabulated in Contributions to the post-schema-validation infoset (§C.2).

The last of these, schema information set contributions, are not as new as they might at first seem. 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 white space, 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 ·Schema Component Constraints·, ·Validation Rules·, and ·Schema Information Set Contributions· contained in this specification.

[Definition:] ·Minimally conforming· processors which accept schemas represented in the form of XML documents as described in Layer 2: Schema Documents, Namespaces and Composition (§4.2) 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 Schema Component Details (§3) for mapping the contents of such documents to ·schema components· for use in ·validation· and ·assessment·.

> **Note:** By separating the conformance requirements relating to the concrete syntax of XML schema documents, this specification admits processors which use schemas stored in optimized 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 are not only both ·minimally conforming· and ·in conformance to the XML Representation of Schemas·, but which additionally must 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 (§4.3.2). .

> **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 specification to define such additional levels of conformance.

See Schemas and Namespaces: Access and Composition (§4) 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·.

Therefore [Definition:] this specification introduces the term **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 Validated

The XML representation of schema components uses a vocabulary identified by the namespace name `http://www.w3.org/2001/XMLSchema`. For brevity, the text and examples in this specification use the prefix `xs:` to stand for this namespace; in practice, any prefix can be used.

XML Schema: Structures also defines several attributes for direct use in any XML documents. These attributes are in a different namespace, which has the namespace name `http://www.w3.org/2001/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. All schema processors have appropriate attribute declarations for these attributes built in, see Attribute Declaration for the 'type' attribute (§3.2.7), Attribute Declaration for the 'nil' attribute (§3.2.7), Attribute Declaration for the 'schemaLocation' attribute (§3.2.7) and Attribute Declaration for the 'noNamespaceSchemaLocation' attribute (§3.2.7).

2.6.1 xsi:type

The Simple Type Definition (§2.2.1.2) or Complex Type Definition (§2.2.1.3) used in ·validation· of an element is usually determined by reference to the appropriate schema components. An element information item in an instance may, however, explicitly assert its type using the attribute `xsi:type`. The value of this attribute is a ·QName·; see QName Interpretation (§3.15.3) for the means by which the ·QName· is associated with a type definition.

2.6.2 xsi:nil

XML Schema: Structures introduces a mechanism for signaling that an element should be accepted as ·valid· when it has no content despite a content type which does not require or even necessarily allow empty content. An element may be ·valid· without content if it has the attribute `xsi:nil` with the value `true`. An element so labeled 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 ·assessment·. See How schema definitions are located on the Web (§4.3.2) 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 XML documents (preferably of MIME type `application/xml` or `text/xml`, but see clause 1.1 of Inclusion Constraints and Semantics (§4.2.1)), conforming to the specifications in Layer 2: Schema Documents, Namespaces and Composition (§4.2). 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 (§4.3.1) and How schema definitions are located on the Web (§4.3.2).

3 Schema Component Details

3.1 Introduction

The following sections provide full details on the composition of all schema components, together with their XML representations and their contributions to ·assessment·. Each section is devoted to a single component, with separate subsections for

1. properties: their values and significance
2. XML representation and the mapping to properties
3. constraints on representation
4. validation rules
5. ·post-schema-validation infoset· contributions
6. constraints on the components themselves

The sub-sections immediately below introduce conventions and terminology used throughout the component sections.

3.1.1 Components and Properties

Components are defined in terms of their properties, and each property in turn is defined by giving its range, that is the values it may have. This can be understood as defining a schema as a labeled directed graph, where the root is a schema, every other vertex is a schema component or a literal (string, boolean, number) and every labeled edge is 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 defined as equality of names (including target namespaces) within symbol spaces.

> **Note:** A schema and its components as defined in this chapter are an idealization 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 below of language such as "properties . . . having . . . components as values".

[Definition:] Throughout this specification, the term **absent** is used as a distinguished property value denoting absence.

Any property not identified as optional is required to be present; optional properties which are not present 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 a 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 characters identified as legal XML characters in [XML 1.0 (Second Edition)].

3.1.2 XML Representations of 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, this specification provides 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 Schema for Schemas (normative) (§A) and DTD for Schemas (non-normative) (§G)) and schema components. All the element information items in the XML representation of a schema must be in the XML Schema namespace, that is their [namespace name] must be `http://www.w3.org/2001/XMLSchema`. Although a common way of creating the XML Infosets which are or contain ·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.

Two aspects of the XML representations of components presented in the following sections are constant across them all:

1. All of them allow attributes qualified with namespace names other than the XML Schema namespace itself: these appear as annotations in the corresponding schema component;
2. All of them allow an <annotation> as their first child, for human-readable documentation and/or machine-targeted information.

3.1.3 The Mapping between XML Representations and 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 Schema for Schemas (normative) (§A).

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.

In discussing the mapping from XML representations to schema components below, the value of a component property is often determined by the value of an attribute information item, one of the [attributes] of an element information item. Since schema documents are constrained by the Schema for Schemas (normative) (§A), there is always a simple type definition associated with any such attribute information item. [Definition:] The phrase **actual value** is used to refer to the member of the value space of the simple type definition associated with an attribute information item which corresponds to its ·normalized value·. This will often be a string, but may also be an integer, a boolean, a URI reference, etc. This term is also occasionally used with respect to element or attribute information items in a document being ·validated·.

Many properties are identified below as having other schema components or sets of 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 (§5.3): no mention of handling missing components will be found in the individual component descriptions below.

Forward reference to named definitions and declarations *is* allowed, both within and between ·schema documents·. By the time the component corresponding to an XML representation which contains a forward reference is actually needed for ·validation· an appropriately-named component may have become available to discharge the reference: see Schemas and Namespaces: Access and Composition (§4) for details.

3.1.4 White Space Normalization during Validation

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

The above definition means that comments and processing instructions, even in the midst of text, are ignored for all ·validation· purposes.

[Definition:] The **normalized value** of an element or attribute information item is an ·initial value· whose white space, if any, has been normalized according to the value of the whiteSpace facet of the simple type definition used in its ·validation·:

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

replace
: All occurrences of `#x9` (tab), `#xA` (line feed) and `#xD` (carriage return) are replaced with `#x20` (space).

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

If the simple type definition used in an item's ·validation· is the ·simple ur-type definition·, the ·normalized value· must be determined as in the **preserve** case above.

There are three alternative validation rules which may supply the necessary background for the above: Attribute Locally Valid (§3.2.4) (clause 3), Element Locally Valid (Type) (§3.3.4) (clause 3.1.3) or Element Locally Valid (Complex Type) (§3.4.4) (clause 2.2).

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 1.0 (Second Edition)] for the precedent for **replace** and **collapse** for attributes. Extending this processing to element content is necessary to ensure a consistent ·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 takes place, as for instance when character entity references in attribute values result in white space characters other than spaces in their ·initial value·s.

3.2 Attribute Declarations

Attribute declarations provide for:

- Local ·validation· of attribute information item values using a simple type definition;
- Specifying default or fixed values for attribute information items.

Example

```
<xs:attribute name="age" type="xs:positiveInteger" use="required"/>
```

The XML representation of an attribute declaration.

3.2.1 The Attribute Declaration Schema Component

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 name, as defined in [XML-Namespaces].
{type definition}
A simple type definition.
{scope}
Optional. Either *global* or a complex type definition.
{value constraint}
Optional. A pair consisting of a value and one of *default*, *fixed*.
{annotation}
Optional. An annotation.

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

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

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.

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 ·absent· in the case of declarations within attribute group definitions: their scope will be determined when they are used in the construction of complex type definitions.

{value constraint} reproduces the functions of XML 1.0 default and `#FIXED` attribute values. *default* specifies that the attribute is to appear unconditionally in the ·post-schema-validation infoset·, with the supplied value used whenever the attribute is not actually present; *fixed* indicates that the attribute value if present must equal the supplied constraint value, and if absent receives the supplied value as for *default*. Note that it is *values* that are supplied and/or checked, not strings.

See Annotations (§3.13) for information on the role of the {annotation} property.

> **Note:** A more complete and formal presentation of the semantics of {name}, {target namespace} and {value constraint} is provided in conjunction with other aspects of complex type ·validation· (see Element Locally Valid (Complex Type) (§3.4.4).)

[XML-Infoset] distinguishes attributes with names such as `xmlns` or `xmlns:xsl` from ordinary attributes, identifying them as [namespace 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 (§3.2.6). No means is provided in this specification to supply a default value for a namespace declaration.

3.2.2 XML Representation of Attribute Declaration Schema Components

The XML representation for an attribute declaration schema component is an <attribute> 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
  default = string
```

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

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

Attribute Declaration Schema Component

Property	Representation
{name}	The ·actual value· of the `name` [attribute]
{target namespace}	The ·actual value· of the `targetNamespace` [attribute] of the parent <schema> element information item, or ·absent· if there is none.
{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 ·actual value· of the `type` [attribute], if present, otherwise the ·simple ur-type definition·.
{scope}	*global*.
{value constraint}	If there is a `default` or a `fixed` [attribute], then a pair consisting of the ·actual value· (with respect to the {type definition}) of that [attribute] and either *default* or *fixed*, as appropriate, 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 has <complexType> or <attributeGroup> as an ancestor and the `ref` [attribute] is absent, it corresponds to an attribute use with properties as follows (unless `use='prohibited'`, in which case the item corresponds to nothing at all):

Attribute Use Schema Component

Property	Representation
{required}	*true* if the `use` [attribute] is present with ·actual value· `required`, otherwise *false*.
{attribute declaration}	See the Attribute Declaration mapping immediately below.
{value constraint}	If there is a `default` or a `fixed` [attribute], then a pair consisting of the ·actual value· (with respect to the {type definition} of the {attribute declaration}) of that [attribute] and either *default* or *fixed*, as appropriate, otherwise ·absent·.

Attribute Declaration Schema Component

Property	Representation
{name}	The ·actual value· of the `name` [attribute]
{target namespace}	If `form` is present and its ·actual value· is `qualified`, or if `form` is absent and the ·actual value· of `attributeFormDefault` on the <schema> ancestor is `qualified`, then the ·actual value· of the `targetNamespace` [attribute] of the parent <schema> element information item, or ·absent· if there is none, otherwise ·absent·.
{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 ·actual 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 an <attributeGroup> definition), ·absent·.
{value constraint}	·absent·.
{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 with properties as follows (unless `use='prohibited'`, in which case the item corresponds to nothing at all):

Attribute Use Schema Component

Property	Representation
{required}	*true* if the `use` [attribute] is present with ·actual value· `required`, otherwise *false*.
{attribute declaration}	The (top-level) attribute declaration ·resolved· to by the ·actual value· of the `ref` [attribute]
{value constraint}	If there is a `default` or a `fixed` [attribute], then a pair consisting of the ·actual value· (with respect to the {type definition} of the {attribute declaration}) of that [attribute] and either *default* or *fixed*, as appropriate, otherwise ·absent·.

Attribute declarations can appear at the top level of a schema document, or within complex type definitions, either as complete (local) declarations, or by reference to top-level declarations, or within attribute group definitions. 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 information items ·validated· by a top-level declaration must be qualified with the {target namespace} of that declaration (if this is ·absent·, the item must be unqualified). Control over whether attribute information items ·validated· by a local declaration must be similarly 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 reside in symbol spaces local to the type definition which contains them.

3.2.3 Constraints on XML Representations of Attribute Declarations

Schema Representation Constraint: Attribute Declaration Representation OK
In addition to the conditions imposed on <attribute> element information items by the schema for schemas, **all** of the following must be true:
1 `default` and `fixed` must not both be present.
2 If `default` and `use` are both present, `use` must have the ·actual value· `optional`.
3 If the item's parent is not <schema>, then **all** of the following must be true:
  3.1 One of `ref` or `name` must be present, but not both.
  3.2 If `ref` is present, then all of <simpleType>, `form` and `type` must be absent.
4 `type` and <simpleType> must not both be present.
5 The corresponding attribute declaration must satisfy the conditions set out in Constraints on Attribute Declaration Schema Components (§3.2.6).

3.2.4 Attribute Declaration Validation Rules

Validation Rule: Attribute Locally Valid
For an attribute information item to be locally ·valid· with respect to an attribute declaration **all** of the following must be true:
1 The declaration must not be ·absent· (see Missing Sub-components (§5.3) for how this can fail to be the case).
2 Its {type definition} must not be absent.
3 The item's ·normalized value· must be locally ·valid· with respect to that {type definition} as per String Valid (§3.14.4).
4 The item's ·actual value· must match the value of the {value constraint}, if it is present and *fixed*.

Validation Rule: Schema-Validity Assessment (Attribute)
The schema-validity assessment of an attribute information item depends on its ·validation· alone.

[Definition:] During ·validation·, associations between element and attribute information items among the [children] and [attributes] on the one hand, and element and attribute declarations on the other, are established as a side-effect. Such declarations are called the **context-determined declarations**. See clause 3.1 (in Element Locally Valid (Complex Type) (§3.4.4)) for attribute declarations, clause 2 (in Element Sequence Locally Valid (Particle) (§3.9.4)) for element declarations.

For an attribute information item's schema-validity to have been assessed **all** of the following must be true:

1 A non-·absent· attribute declaration must be known for it, namely **one** of the following:
  1.1 A declaration which has been established as its ·context-determined declaration·;
  1.2 A declaration resolved to by its [local name] and [namespace name] as defined by QName resolution (Instance) (§3.15.4), provided its ·context-determined declaration· is not *skip*.
2 Its ·validity· with respect to that declaration must have been evaluated as per Attribute Locally Valid (§3.2.4).
3 Both clause 1 and clause 2 of Attribute Locally Valid (§3.2.4) must be satisfied.

[Definition:] For attributes, there is no difference between assessment and strict assessment, so if the above holds, the attribute information item has been **strictly assessed**.

3.2.5 Attribute Declaration Information Set Contributions

Schema Information Set Contribution: Assessment Outcome (Attribute)
If the schema-validity of an attribute information item has been assessed as per Schema-Validity Assessment (Attribute) (§3.2.4), then in the ·post-schema-validation infoset· it has properties as follows:

PSVI Contributions for attribute information items

`[validation context]`
The nearest ancestor element information item with a [schema information] property.
`[validity]`
The appropriate **case** among the following:
1 **If** it was ·strictly assessed·, **then** the appropriate **case** among the following:
  1.1 **If** it was ·valid· as defined by Attribute Locally Valid (§3.2.4), **then** *valid*;
  1.2 **otherwise** *invalid*.
2 **otherwise** *notKnown*.
`[validation attempted]`
The appropriate **case** among the following:
1 **If** it was ·strictly assessed·, **then** *full*;
2 **otherwise** *none*.
`[schema specified]`
infoset. See Attribute Default Value (§3.4.5) for the other possible value.

Schema Information Set Contribution: Validation Failure (Attribute)
If the local ·validity·, as defined by Attribute Locally Valid (§3.2.4) above, of an attribute information item has been assessed, in the ·post-schema-validation infoset· the item has a property:

PSVI Contributions for attribute information items

[schema error code]
The appropriate **case** among the following:
1 **If** the item is not ·valid·, **then** a list. Applications wishing to provide information as to the reason(s) for the ·validation· failure are encouraged to record one or more error codes (see Outcome Tabulations (normative) (§C)) herein.
2 **otherwise** ·absent·.

Schema Information Set Contribution: Attribute Declaration
If an attribute information item is ·valid· with respect to an attribute declaration as per Attribute Locally Valid (§3.2.4) then in the ·post-schema-validation infoset· the attribute information item may, at processor option, have a property:

PSVI Contributions for attribute information items

[attribute declaration]
An ·item isomorphic· to the declaration component itself.

Schema Information Set Contribution: Attribute Validated by Type
If clause 3 of Attribute Locally Valid (§3.2.4) applies with respect to an attribute information item, in the ·post-schema-validation infoset· the attribute information item has a property:

PSVI Contributions for attribute information items

[schema normalized value]
The ·normalized value· of the item as ·validated·.

Furthermore, the item has one of the following alternative sets of properties:

Either

PSVI Contributions for attribute information items

[type definition]
An ·item isomorphic· to the relevant attribute declaration's {type definition} component.
[member type definition]
If and only if that type definition has {variety} *union*, then an ·item isomorphic· to that member of its {member type definitions} which actually ·validated· the attribute item's [normalized value].

or

PSVI Contributions for attribute information items

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

If the ·type definition· has {variety} *union*, then calling [Definition:] that member of the {member type definitions} which actually ·validated· the attribute item's ·normalized value· the **actual member type definition**, there are three additional properties:

PSVI Contributions for attribute information items

[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 definition.

The first (·item isomorphic·) alternative above is provided for applications such as query processors which need access to the full range of details about an item's ·assessment·, 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 has a property:

PSVI Contributions for attribute information items

[schema default]
The canonical lexical representation of the declaration's {value constraint} value.

If the attribute information item was not ·strictly assessed·, then instead of the values specified above,
1 The item's [schema normalized value] property has the ·initial value· of the item as its value;
2 The [type definition] and [member type definition] properties, or their alternatives, are based on the ·simple ur-type definition·.

3.2.6 Constraints on Attribute Declaration Schema Components

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

Schema Component Constraint: Attribute Declaration Properties Correct
All of the following must be true:
1 The values of the properties of an attribute declaration must be as described in the property tableau in The Attribute Declaration Schema Component (§3.2.1), modulo the impact of Missing Sub-components (§5.3).
2 if there is a {value constraint}, the canonical lexical representation of its value must be ·valid· with respect to the {type definition} as defined in String Valid (§3.14.4).
3 If the {type definition} is or is derived from ID then there must not be a {value constraint}.

Schema Component Constraint: 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:*`.

Schema Component Constraint: xsi: Not Allowed
The {target namespace} of an attribute declaration, whether local or top-level, must not match `http://www.w3.org/2001/XMLSchema-instance` (unless it is one of the four built-in declarations given in the next section).
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:nil`, which would be seriously misleading, as they would have no effect.

3.2.7 Built-in Attribute Declarations

There are four attribute declarations present in every schema by definition:

Attribute Declaration for the 'type' attribute

Property	Value
{name}	`type`
{target namespace}	`http://www.w3.org/2001/XMLSchema-instance`
{type definition}	The built-in QName simple type definition
{scope}	*global*
{value constraint}	·absent·
{annotation}	·absent·

Attribute Declaration for the 'nil' attribute

Property	Value
{name}	`nil`
{target namespace}	`http://www.w3.org/2001/XMLSchema-instance`
{type definition}	The built-in boolean simple type definition
{scope}	*global*
{value constraint}	·absent·
{annotation}	·absent·

Attribute Declaration for the 'schemaLocation' attribute

Property	Value
{name}	`schemaLocation`
{target namespace}	`http://www.w3.org/2001/XMLSchema-instance`
{type definition}	An anonymous simple type definition, as follows:

Property	Value
{name}	·absent·
{target namespace}	`http://www.w3.org/2001/XMLSchema-instance`
{base type definition}	The built in ·simple ur-type definition·
{facets}	·absent·
{variety}	*list*
{item type definition}	The built-in anyURI simple type definition
{annotation}	·absent·

{scope}	*global*
{value constraint}	·absent·
{annotation}	·absent·

Attribute Declaration for the 'noNamespaceSchemaLocation' attribute

Property	Value
{name}	`noNamespaceSchemaLocation`
{target namespace}	`http://www.w3.org/2001/XMLSchema-instance`

Attribute Declaration for the 'noNamespaceSchemaLocation' attribute	
Property	**Value**
{type definition}	The built-in anyURI simple type definition
{scope}	*global*
{value constraint}	·absent·
{annotation}	·absent·

3.3 Element Declarations

Element declarations provide for:

- Local ·validation· of element information item values using a type definition;
- Specifying default or fixed values for an element information items;
- 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·.

Example

```
<xs:element name="PurchaseOrder" type="PurchaseOrderType"/>

<xs:element name="gift">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="birthday" type="xs:date"/>
   <xs:element ref="PurchaseOrder"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

XML representations of several different types of element declaration

3.3.1 The Element Declaration Schema Component

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 name, as defined in [XML-Namespaces].
{type definition}
Either a simple type definition or a complex type definition.
{scope}
Optional. Either *global* or a complex type definition.
{value constraint}
Optional. A pair consisting of a value and one of *default*, *fixed*.
{nillable}
A boolean.
{identity-constraint definitions}
A set of constraint definitions.
{substitution group affiliation}
Optional. A top-level element definition.
{substitution group exclusions}
A subset of {*extension*, *restriction*}.
{disallowed substitutions}
A subset of {*substitution*, *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 declarations within named model groups: their scope is 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 ·valid· if it satisfies the {type definition}. For such an item, schema information set contributions appropriate to the {type definition} are added to the corresponding element information item in the ·post-schema-validation infoset·.

If {nillable} is *true*, then an element may also be ·valid· if it carries the namespace qualified attribute with [local name] `nil` from namespace `http://www.w3.org/2001/XMLSchema-instance` and value `true` (see xsi:nil (§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 Locally Valid (Element) (§3.3.4).

{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 canonical form of the supplied constraint value 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 its value must match the supplied constraint value.

> **Note:** The provision of defaults for elements goes beyond what is possible in XML 1.0 DTDs, and does not exactly correspond to defaults for attributes. In particular, an element with a non-empty {value constraint} whose simple type definition includes the empty string in its lexical space will nonetheless never receive that value, because the {value constraint} will override it.

{identity-constraint definitions} express constraints establishing uniquenesses and reference relationships among the values of related elements and attributes. See Identity-constraint Definitions (§3.11).

Element declarations are potential members of the substitution group, if any, identified by {substitution group affiliation}. Potential membership is transitive but not symmetric; an element declaration is a potential member of any group of which its {substitution group affiliation} is a potential member. Actual membership may be blocked by the effects of {substitution group exclusions} or {disallowed substitutions}, see below.

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 substitution group headed by the declared element. If {disallowed substitutions} is empty, then all derived types and substitution group members are allowed.

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 Annotations (§3.13) for information on the role of the {annotation} property.

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

If the <element> element information item has <schema> as its parent, the corresponding schema component is as follows:

Element Declaration Schema Component

Property	Representation
{name}	The ·actual value· of the `name` [attribute].
{target namespace}	The ·actual 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 ·actual value· of the `type` [attribute], otherwise the {type definition} of the element declaration ·resolved· to by the ·actual value· of the `substitutionGroup` [attribute], if present, otherwise the ·ur-type definition·.
{nillable}	The ·actual value· of the `nillable` [attribute], if present, otherwise *false*.
{value constraint}	If there is a `default` or a `fixed` [attribute], then a pair consisting of the ·actual 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

Element Declaration Schema Component

Property	Representation
	else with respect to the built-in string simple type definition) 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 ·actual value· of the `substitutionGroup` [attribute], if present, otherwise ·absent·.
{disallowed substitutions}	A set depending on the ·actual value· of the `block` [attribute], if present, otherwise on the ·actual value· of the `blockDefault` [attribute] of the ancestor <schema> element information item, if present, otherwise on the empty string. Call this the **EBV** (for effective block value). Then the value of this property is the appropriate **case** among the following: 1 **If** the **EBV** is the empty string, **then** the empty set; 2 **If** the **EBV** is `#all`, **then** {*extension*, *restriction*, *substitution*}; 3 **otherwise** a set with members drawn from the set above, each being present or absent depending on whether the ·actual value· (which is a list) contains an equivalently named item. **Note:** Although the `blockDefault` [attribute] of <schema> may include values other than *extension*, *restriction* or *substitution*, those values are ignored in the determination of {disallowed substitutions} for element declarations (they *are* used elsewhere).
{substitution group exclusions}	As for {disallowed substitutions} above, but using the `final` and `finalDefault` [attributes] in place of the `block` and `blockDefault` [attributes] and with the relevant set being {*extension*, *restriction*}.
{abstract}	The ·actual 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 ·actual value· of the `minOccurs` [attribute], if present, otherwise `1`.
{max occurs}	*unbounded*, if the `maxOccurs` [attribute] equals *unbounded*, otherwise the ·actual 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 ·actual value· is `qualified`, or if `form` is absent and the ·actual value· of `elementFormDefault` on the <schema> ancestor is `qualified`, then the ·actual 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 named <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 ·actual value· of the `minOccurs` [attribute], if present, otherwise `1`.
{max occurs}	*unbounded*, if the `maxOccurs` [attribute] equals *unbounded*, otherwise the ·actual value· of the `maxOccurs` [attribute], if present, otherwise `1`.
{term}	The (top-level) element declaration ·resolved· to by the ·actual 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.

<element>s within <schema> produce *global* element declarations; <element>s 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 information items ·validated· by a top-level declaration must be qualified with the {target namespace} of that declaration (if this is ·absent·, the item must be unqualified). Control over whether element information items ·validated· by a local declaration must be similarly 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}.

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 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·. This has the important consequence that the minimum valid element declaration, that is, one with only a `name` attribute and no contents, is also (nearly) the most general, validating any combination of text and element content and allowing any attributes, and providing for recursive validation where possible.

See below at XML Representation of Identity-constraint Definition Schema Components (§3.11.2) for <key>, <unique> and <keyref>.

Example

```
 <xs:element name="unconstrained"/>

 <xs:element name="emptyElt">
  <xs:complexType>
   <xs:attribute ...>. . .</xs:attribute>
  </xs:complexType>
 </xs:element>

 <xs:element name="contextOne">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="myLocalElement" type="myFirstType"/>
    <xs:element ref="globalElement"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:element name="contextTwo">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="myLocalElement" type="mySecondType"/>
    <xs:element ref="globalElement"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
```

The first example above declares an element whose type, by default, is the ·ur-type definition·. The second uses an embedded anonymous complex type definition.

The last two examples illustrate the use of local element declarations. 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" use="required"/>
   </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:sequence>
      <xs:element ref="annotation" minOccurs="0"/>
     </xs:sequence>
     <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:sequence>
      <xs:element ref="annotation" minOccurs="0"/>
     </xs:sequence>
     <xs:attribute name="value" type="xs:duration"/>
    </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" use="optional"/>
  . . .
 </xs:complexType>
```

An example from a previous version of the schema for 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).

3.3.3 Constraints on XML Representations of Element Declarations

Schema Representation Constraint: Element Declaration Representation OK

In addition to the conditions imposed on <element> element information items by the schema for schemas: **all** of the following must be true:

1 `default` and `fixed` must not both be present.

2 If the item's parent is not <schema>, then **all** of the following must be true:

2.1 One of `ref` or `name` must be present, but not both.

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

3 `type` and either <simpleType> or <complexType> are mutually exclusive.

4 The corresponding particle and/or element declarations must satisfy the conditions set out in Constraints on Element Declaration Schema Components (§3.3.6) and Constraints on Particle Schema Components (§3.9.6).

3.3.4 Element Declaration Validation Rules

Validation Rule: Element Locally Valid (Element)

For an element information item to be locally ·valid· with respect to an element declaration **all** of the following must be true:

1 The declaration must not be ·absent·.

2 Its {abstract} must be *false*.

3 The appropriate **case** among the following must be true:

3.1 **If** {nillable} is *false*, **then** there must be no attribute information item among the element information item's [attributes] whose [namespace name] is identical to `http://www.w3.org/2001/XMLSchema-instance` and whose [local name] is `nil`.

3.2 **If** {nillable} is *true* and there is such an attribute information item and its ·actual value· is `true` , **then all** of the following must be true:

3.2.1 The element information item must have no character or element information item [children].

3.2.2 There must be no *fixed* {value constraint}.

4 If there is an attribute information item among the element information item's [attributes] whose [namespace name] is identical to `http://www.w3.org/2001/XMLSchema-instance` and whose [local name] is `type`, then **all** of the following must be true:

4.1 The ·normalized value· of that attribute information item must be ·valid· with respect to the built-in QName simple type, as defined by String Valid (§3.14.4);

4.2 The ·local name· and ·namespace name· (as defined in QName Interpretation (§3.15.3)), of the ·actual value· of that attribute information item must resolve to a type definition, as defined in QName resolution (Instance) (§3.15.4) -- [Definition:] call this type definition the **local type definition**;

4.3 The ·local type definition· must be validly derived from the {type definition} given the union of the {disallowed substitutions} and the {type definition}'s {prohibited substitutions}, as defined in Type Derivation OK (Complex) (§3.4.6) (if it is a complex type definition), or given {disallowed substitutions} as defined in Type Derivation OK (Simple) (§3.14.6) (if it is a simple type definition).

[Definition:] The phrase **actual type definition** occurs below. If the above three clauses are satisfied, this should be understood as referring to the ·local type definition·, otherwise to the {type definition}.

5 The appropriate **case** among the following must be true:

5.1 **If** the declaration has a {value constraint}, the item has neither element nor character [children] and clause 3.2 has not applied, **then all** of the following must be true:

5.1.1 If the ·actual type definition· is a ·local type definition· then the canonical lexical representation of the {value constraint} value must be a valid default for the ·actual type definition· as defined in Element Default Valid (Immediate) (§3.3.6).

5.1.2 The element information item with the canonical lexical representation of the {value constraint} value used as its ·normalized value· must be ·valid· with respect to the ·actual type definition· as defined by Element Locally Valid (Type) (§3.3.4).

5.2 **If** the declaration has no {value constraint} or the item has either element or character [children] or clause 3.2 has applied, **then all** of the following must be true:

5.2.1 The element information item must be ·valid· with respect to the ·actual type definition· as defined by Element Locally Valid (Type) (§3.3.4).

5.2.2 If there is a *fixed* {value constraint} and clause 3.2 has not applied, **all** of the following must be true:

5.2.2.1 The element information item must have no element information item [children].

5.2.2.2 The appropriate **case** among the following must be true:

5.2.2.2.1 **If** the {content type} of the ·actual type definition· is *mixed*, **then** the ·initial value· of the item must match the canonical lexical representation of the {value constraint} value.

5.2.2.2.2 **If** the {content type} of the ·actual type definition· is a simple type definition, **then** the ·actual value· of the item must match the canonical lexical representation of the {value constraint} value.

6 The element information item must be ·valid· with respect to each of the {identity-constraint definitions} as per Identity-constraint Satisfied (§3.11.4).

7 If the element information item is the ·validation root·, it must be ·valid· per Validation Root Valid (ID/IDREF) (§3.3.4).

Validation Rule: Element Locally Valid (Type)

For an element information item to be locally ·valid· with respect to a type definition **all** of the following must be true:

1 The type definition must not be ·absent·;

2 It must not have {abstract} with value *true*.

3 The appropriate **case** among the following must be true:

3.1 **If** the type definition is a simple type definition, **then all** of the following must be true:

3.1.1 The element information item's [attributes] must be empty, excepting those whose [namespace name] is identical to `http://www.w3.org/2001/XMLSchema-instance` and whose [local name] is one of `type`, `nil`, `schemaLocation` or `noNamespaceSchemaLocation`.

3.1.2 The element information item must have no element information item [children].

3.1.3 If clause 3.2 of Element Locally Valid (Element) (§3.3.4) did not apply, then the ·normalized value· must be ·valid· with respect to the type definition as defined by String Valid (§3.14.4).

3.2 **If** the type definition is a complex type definition, **then** the element information item must be ·valid· with respect to the type definition as per Element Locally Valid (Complex Type) (§3.4.4);

Validation Rule: Validation Root Valid (ID/IDREF)

For an element information item which is the ·validation root· to be ·valid· **all** of the following must be true:

1 There must be no **ID/IDREF binding** in the item's [ID/IDREF table] whose [binding] is the empty set.
2 There must be no **ID/IDREF binding** in the item's [ID/IDREF table] whose [binding] has more than one member.

See ID/IDREF Table (§3.15.5) for the definition of **ID/IDREF binding**.

> **Note:** The first clause above applies when there is a reference to an undefined ID. The second applies when there is a multiply-defined ID. They are separated out to ensure that distinct error codes (see Outcome Tabulations (normative) (§C)) are associated with these two cases.
> **Note:** Although this rule applies at the ·validation root·, in practice processors, particularly streaming processors, may wish to detect and signal the clause 2 case as it arises.
> **Note:** This reconstruction of [XML 1.0 (Second Edition)]'s `ID/IDREF` functionality is imperfect in that if the ·validation root· is not the document element of an XML document, the results will not necessarily be the same as those a validating parser would give were the document to have a DTD with equivalent declarations.

Validation Rule: Schema-Validity Assessment (Element)
The schema-validity assessment of an element information item depends on its ·validation· and the ·assessment· of its element information item children and associated attribute information items, if any.

So for an element information item's schema-validity to be assessed **all** of the following must be true:

1 **One** of the following must be true:
  1.1 **All** of the following must be true:
    1.1.1 A non-·absent· element declaration must be known for it, because**one** of the following is true
      1.1.1.1 A declaration was stipulated by the processor (see Assessing Schema-Validity (§5.2)).
      1.1.1.2 A declaration has been established as its ·context-determined declaration·.
      1.1.1.3 **All** of the following must be true:
        1.1.1.3.1 Its ·context-determined declaration· is not *skip*.
        1.1.1.3.2 Its [local name] and [namespace name] resolve to an element declaration as defined by QName resolution (Instance) (§3.15.4).
    1.1.2 Its ·validity· with respect to that declaration must have been evaluated as per Element Locally Valid (Element) (§3.3.4).
    1.1.3 If that evaluation involved the evaluation of Element Locally Valid (Type) (§3.3.4), clause 1 thereof must be satisfied.
  1.2 **All** of the following must be true:
    1.2.1 A non-·absent· type definition is known for it because **one** of the following is true
      1.2.1.1 A type definition was stipulated by the processor (see Assessing Schema-Validity (§5.2)).
      1.2.1.2 **All** of the following must be true:
        1.2.1.2.1 There is an attribute information item among the element information item's [attributes] whose [namespace name] is identical to `http://www.w3.org/2001/XMLSchema-instance` and whose [local name] is `type`.
        1.2.1.2.2 The ·normalized value· of that attribute information item is ·valid· with respect to the built-in QName simple type, as defined by String Valid (§3.14.4).
        1.2.1.2.3 The ·local name· and ·namespace name· (as defined in QName Interpretation (§3.15.3)), of the ·actual value· of that attribute information item resolve to a type definition, as defined in QName resolution (Instance) (§3.15.4) -- [Definition:] call this type definition the **local type definition**.
        1.2.1.2.4 If there is also a processor-stipulated type definition, the ·local type definition· must be validly derived from that type definition given its {prohibited substitutions}, as defined in Type Derivation OK (Complex) (§3.4.6) (if it is a complex type definition), or given the empty set, as defined in Type Derivation OK (Simple) (§3.14.6) (if it is a simple type definition).
    1.2.2 The element information item's ·validity· with respect to the ·local type definition· (if present and validly derived) or the processor-stipulated type definition (if no ·local type definition· is present) has been evaluated as per Element Locally Valid (Type) (§3.3.4).
2 The schema-validity of all the element information items among its [children] has been assessed as per Schema-Validity Assessment (Element) (§3.3.4), and the schema-validity of all the attribute information items among its [attributes] has been assessed as per Schema-Validity Assessment (Attribute) (§3.2.4).

[Definition:] If either case of clause 1 above holds, the element information item has been **strictly assessed**.

If the item cannot be ·strictly assessed·, because neither clause 1.1 nor clause 1.2 above are satisfied, [Definition:] an element information item's schema validity may be **laxly assessed** if its ·context-determined declaration· is not *skip* by ·validating· with respect to the ·ur-type definition· as per Element Locally Valid (Type) (§3.3.4).

> **Note:** In general if clause 1.1 above holds clause 1.2 does not, and vice versa. When an `xsi:type` [attribute] is involved, however, clause 1.2 takes precedence, as is made clear in Element Locally Valid (Element) (§3.3.4).
> **Note:** The {name} and {target namespace} properties are not mentioned above because they are checked during particle ·validation·, as per Element Sequence Locally Valid (Particle) (§3.9.4).

3.3.5 Element Declaration Information Set Contributions

Schema Information Set Contribution: Assessment Outcome (Element)
If the schema-validity of an element information item has been assessed as per Schema-Validity Assessment (Element) (§3.3.4), then in the ·post-schema-validation infoset· it has properties as follows:

PSVI Contributions for element information items

`[validation context]`
The nearest ancestor element information item with a [schema information] property (or this element item itself if it has such a property).
`[validity]`
The appropriate **case** among the following:
1 **If** it was ·strictly assessed·, **then** the appropriate **case** among the following:
  1.1 **If all** of the following are true
    1.1.1
      1.1.1.1 clause 1.1 of Schema-Validity Assessment (Element) (§3.3.4) applied and the item was ·valid· as defined by Element Locally Valid (Element) (§3.3.4);
      1.1.1.2 clause 1.2 of Schema-Validity Assessment (Element) (§3.3.4) applied and the item was ·valid· as defined by Element Locally Valid (Type) (§3.3.4).

1.1.2 Neither its [children] nor its [attributes] contains an information item (element or attribute respectively) whose [validity] is *invalid*.
1.1.3 Neither its [children] nor its [attributes] contains an information item (element or attribute respectively) with a ·context-determined declaration· of *mustFind* whose [validity] is *notKnown*.
, **then** *valid*;
1.2 **otherwise** *invalid*..
2 **otherwise** *notKnown*.

[validation attempted]
The appropriate **case** among the following:
1 **If** it was ·strictly assessed· and neither its [children] nor its [attributes] contains an information item (element or attribute respectively) whose [validation attempted] is not *full*, **then** *full*;
2 **If** it was not ·strictly assessed· and neither its [children] nor its [attributes] contains an information item (element or attribute respectively) whose [validation attempted] is not *none*, **then** *none*;
3 **otherwise** *partial*.

Schema Information Set Contribution: Validation Failure (Element)
If the local ·validity·, as defined by Element Locally Valid (Element) (§3.3.4) above and/or Element Locally Valid (Type) (§3.3.4) below, of an element information item has been assessed, in the ·post-schema-validation infoset· the item has a property:

PSVI Contributions for element information items

[schema error code]
The appropriate **case** among the following:
1 **If** the item is not ·valid·, **then** a list. Applications wishing to provide information as to the reason(s) for the ·validation· failure are encouraged to record one or more error codes (see Outcome Tabulations (normative) (§C)) herein.
2 **otherwise** ·absent·.

Schema Information Set Contribution: Element Declaration
If an element information item is ·valid· with respect to an element declaration as per Element Locally Valid (Element) (§3.3.4) then in the ·post-schema-validation infoset· the element information item must, at processor option, have either:

PSVI Contributions for element information items

[element declaration]
an ·item isomorphic· to the declaration component itself

or

PSVI Contributions for element information items

[nil]
true if clause 3.2 of Element Locally Valid (Element) (§3.3.4) above is satisfied, otherwise *false*

Schema Information Set Contribution: Element Validated by Type
If an element information item is ·valid· with respect to a ·type definition· as per Element Locally Valid (Type) (§3.3.4), in the ·post-schema-validation infoset· the item has a property:

PSVI Contributions for element information items

[schema normalized value]
The appropriate **case** among the following:
1 **If** clause 3.2 of Element Locally Valid (Element) (§3.3.4) and Element Default Value (§3.3.5) above have *not* applied and either the ·type definition· is a simple type definition or its {content type} is a simple type definition, **then** the ·normalized value· of the item as ·validated·.
2 **otherwise** ·absent·.

Furthermore, the item has one of the following alternative sets of properties:

Either

PSVI Contributions for element information items

[type definition]
An ·item isomorphic· to the ·type definition· component itself.
[member type definition]
If and only if that type definition is a simple type definition with {variety} *union*, or a complex type definition whose {content type} is a simple type definition with {variety} *union*, then an ·item isomorphic· to that member of the union's {member type definitions} which actually ·validated· the element item's ·normalized value·.

or

PSVI Contributions for element information items

[type definition type]
simple or *complex*, depending on the ·type definition·.
[type definition namespace]
The {target namespace} of the ·type definition·.
[type definition anonymous]

true if the {name} of the ·type definition· is ·absent·, otherwise *false*.

[type definition name]
The {name} of the ·type definition·, if it is not ·absent·. If it is ·absent·, schema processors may, but need not, provide a value unique to the definition.

If the ·type definition· is a simple type definition or its {content type} is a simple type definition, and that type definition has {variety} *union*, then calling [Definition:] that member of the {member type definitions} which actually ·validated· the element item's ·normalized value· the **actual member type definition**, there are three additional properties:

PSVI Contributions for element information items

[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 definition.

The first (·item isomorphic·) alternative above is provided for applications such as query processors which need access to the full range of details about an item's ·assessment·, 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 has a property:

PSVI Contributions for element information items

[schema default]
The canonical lexical representation of the declaration's {value constraint} value.

Note that if an element is ·laxly assessed·, then the [type definition] and [member type definition] properties, or their alternatives, are based on the ·ur-type definition·.

Schema Information Set Contribution: Element Default Value

If the local ·validity·, as defined by Element Locally Valid (Element) (§3.3.4) above, of an element information item has been assessed, in the ·post-schema-validation infoset· the item has a property:

PSVI Contributions for element information items

[schema specified]
The appropriate **case** among the following:
1 **If** the item is ·valid· with respect to an element declaration as per Element Locally Valid (Element) (§3.3.4) and the {value constraint} is present, but clause 3.2 of Element Locally Valid (Element) (§3.3.4) above is not satisfied and the item has no element or character information item [children], **then** *schema*. Furthermore, the ·post-schema-validation infoset· has the canonical lexical representation of the {value constraint} value as the item's [schema normalized value] property.
2 **otherwise** *infoset*.

3.3.6 Constraints on Element Declaration Schema Components

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

Schema Component Constraint: Element Declaration Properties Correct

All of the following must be true:
1 The values of the properties of an element declaration must be as described in the property tableau in The Element Declaration Schema Component (§3.3.1), modulo the impact of Missing Sub-components (§5.3).
2 If there is a {value constraint}, the canonical lexical representation of its value must be ·valid· with respect to the {type definition} as defined in Element Default Valid (Immediate) (§3.3.6).
3 If there is a non-·absent· {substitution group affiliation}, then {scope} must be *global*.
4 If there is a {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) (§3.4.6) (if the {type definition} is complex) or as defined in Type Derivation OK (Simple) (§3.14.6) (if the {type definition} is simple).
5 If the {type definition} or {type definition}'s {content type} is or is derived from ID then there must not be a {value constraint}.
Note: The use of ID as a type definition for elements goes beyond XML 1.0, and should be avoided if backwards compatibility is desired.
6 Circular substitution groups are disallowed. That is, it must not be possible to return to an element declaration by repeatedly following the {substitution group affiliation} property.

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

Schema Component Constraint: Element Default Valid (Immediate)

For a string to be a valid default with respect to a type definition the appropriate **case** among the following must be true:
1 **If** the type definition is a simple type definition, **then** the string must be ·valid· with respect to that definition as defined by String Valid (§3.14.4).
2 **If** the type definition is a complex type definition, **then all** of the following must be true:
2.1 its {content type} must be a simple type definition or *mixed*.
2.2 The appropriate **case** among the following must be true:
2.2.1 **If** the {content type} is a simple type definition, **then** the string must be ·valid· with respect to that simple type definition as defined by String Valid (§3.14.4).
2.2.2 **If** the {content type} is *mixed*, **then** the {content type}'s particle must be ·emptiable· as defined by Particle Emptiable (§3.9.6).

Schema Component Constraint: Substitution Group OK (Transitive)

For an element declaration (call it **D**) to be validly substitutable for another element declaration (call it **C**) subject to a blocking constraint (a subset of {*substitution*, *extension*, *restriction*}, the value of a {disallowed substitutions}) **one** of the following must be true:

1 **D** and **C** are the same element declaration.

2 **All** of the following must be true:

2.1 The blocking constraint does not contain *substitution*.

2.2 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 . . .

2.3 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} (if **C** is complex, otherwise the empty set) and the {prohibited substitutions} (respectively the empty set) of any intermediate {type definition}s in the derivation of **D**'s {type definition} from **C**'s {type definition}.

Schema Component Constraint: Substitution Group

[Definition:] Every element declaration (call this **HEAD**) in the {element declarations} of a schema defines a **substitution group**, a subset of those {element declarations}, as follows:

Define **P**, the potential substitution group for **HEAD**, as follows:

1 The element declaration itself is in **P**;

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

HEAD's actual ·substitution group· is then the set consisting of each member of **P** such that **all** of the following must be true:

1 Its {abstract} is *false*.

2 It is validly substitutable for **HEAD** subject to **HEAD**'s {disallowed substitutions} as the blocking constraint, as defined in Substitution Group OK (Transitive) (§3.3.6).

3.4 Complex Type Definitions

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, or else constraining the character information item [children] to conform to a specified simple type definition.
- Using the mechanisms of Type Definition Hierarchy (§2.2.1.1) to derive a complex type from another simple or complex type.
- Specifying ·post-schema-validation infoset contributions· 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.

Example

```
<xs:complexType name="PurchaseOrderType">
  <xs:sequence>
   <xs:element name="shipTo" type="USAddress"/>
   <xs:element name="billTo" type="USAddress"/>
   <xs:element ref="comment" minOccurs="0"/>
   <xs:element name="items"  type="Items"/>
  </xs:sequence>
  <xs:attribute name="orderDate" type="xs:date"/>
 </xs:complexType>
```

The XML representation of a complex type definition.

3.4.1 The Complex Type Definition Schema Component

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 name, 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 uses}
A set of attribute uses.

{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 schema components (specifically in <element>). See References to schema components across namespaces (§4.2.3) 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 Declarations (§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 [Definition:] the complex type is said to be **final**, because no further derivations are possible. Finality is *not* inherited, that is, a type definition derived by restriction from a type definition which is final for extension is not itself, in the absence of any explicit `final` attribute of its own, final for anything.

Complex types for which {abstract} is *true* must not be used as the {type definition} for the ·validation· of element information items. It follows that they must not be referenced from an xsi:type (§2.6.1) attribute in an instance document. Abstract complex types can be used as {base type definition}s, or even as the {type definition}s of element declarations, provided in every case a concrete derived type definition is used for ·validation·, either via xsi:type (§2.6.1) or the operation of a substitution group.

{attribute uses} are a set of attribute uses. See Element Locally Valid (Complex Type) (§3.4.4) and Attribute Locally Valid (§3.2.4) for details of attribute ·validation·.

{attribute wildcard}s provide a more flexible specification for ·validation· of attributes not explicitly included in {attribute uses}. 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 names 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 name: [attributes] cannot include attributes from the specified namespace.
- *'not'* and ·absent·: [attributes] cannot include unqualified attributes.

See Element Locally Valid (Complex Type) (§3.4.4) and Wildcard allows Namespace Name (§3.10.4) for formal details of attribute wildcard ·validation·.

{content type} determines the ·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 [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 a complex type definition derived by *extension* or *restriction* from this definition, or element items in a substitution group whose type definition is similarly derived: If {prohibited substitutions} is empty, then all such substitutions are allowed, otherwise, the derivation method(s) it names are disallowed.

See Annotations (§3.13) for information on the role of the {annotations} property.

3.4.2 XML Representation of Complex Type Definitions

The XML representation for a complex type definition schema component is a <complexType> element information item.

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 | List of (extension | restriction))
  final = (#all | List 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 apply:

Complex Type Definition Schema Component

Property	Representation
{name}	The ·actual value· of the `name` [attribute] if present, otherwise ·absent·.
{target namespace}	The ·actual value· of the `targetNamespace` [attribute] of the <schema> ancestor element information item if present, otherwise ·absent·.
{abstract}	The ·actual value· of the `abstract` [attribute], if present, otherwise *false*.
{prohibited substitutions}	A set corresponding to the ·actual value· of the `block` [attribute], if present, otherwise on the ·actual value· of the `blockDefault` [attribute] of the ancestor <schema> element information item, if present, otherwise on the empty string. Call this the **EBV** (for effective block value). Then the value of this property is the appropriate **case** among the following: 1 **If** the **EBV** is the empty string, **then** the empty set; 2 **If** the **EBV** is `#all`, **then** {*extension*, *restriction*}; 3 **otherwise** a set with members drawn from the set above, each being present or absent depending on whether the ·actual value· (which is a list) contains an equivalently named item. **Note:** Although the `blockDefault` [attribute] of <schema> may include values other than *restriction* or*extension*, those values are ignored in the determination of {prohibited substitutions} for complex type definitions (they *are* used elsewhere).
{final}	As for {prohibited substitutions} above, but using the `final` and `finalDefault` [attributes] in place of the `block` and `blockDefault` [attributes].
{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?, (simpleType?, (minExclusive | minInclusive | maxExclusive | maxInclusive | totalDigits |
fractionDigits | length | minLength | maxLength | enumeration | whiteSpace | pattern)*)?, ((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 (anyURI | (##targetNamespace | ##local)) )  : ##any
  processContents = (lax | skip | 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 ·actual value· of the `base` [attribute]
{derivation method}	If the <restriction> alternative is chosen, then *restriction*, otherwise (the <extension> alternative is chosen) *extension*.
{attribute uses}	A union of sets of attribute uses as follows 1 The set of attribute uses corresponding to the <attribute> [children], if any. 2 The {attribute uses} of the attribute groups ·resolved· to by the ·actual value·s of the `ref` [attribute] of the <attributeGroup> [children], if any. 3 if the type definition ·resolved· to by the ·actual value· of the `base` [attribute] is a complex type definition, the {attribute uses} of that type definition, unless the <restriction> alternative is chosen, in which case some members of that type definition's {attribute uses} may not be included, namely those whose {attribute declaration}'s {name} and {target namespace} are the same as **one** of the following:

Complex Type Definition with simple content Schema Component

Property	Representation
	3.1 the {name} and {target namespace} of the {attribute declaration} of an attribute use in the set per clause 1 or clause 2 above; 3.2 what would have been the {name} and {target namespace} of the {attribute declaration} of an attribute use in the set per clause 1 above but for the ·actual value· of the `use` [attribute] of the relevant <attribute> among the [children] of <restriction> being *prohibited*.
{attribute wildcard}	1 [Definition:] Let the **local wildcard** be defined as the appropriate **case** among the following: 1.1 **If** there is an <anyAttribute> present, **then** a wildcard based on the ·actual value·s 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 (§3.10.2); 1.2 **otherwise** ·absent·. 2 [Definition:] Let the **complete wildcard** be defined as the appropriate **case** among the following: 2.1 **If** there are no <attributeGroup> [children] corresponding to attribute groups with non-·absent· {attribute wildcard}s, **then** the ·local wildcard·. 2.2 **If** there are one or more <attributeGroup> [children] corresponding to attribute groups with non-·absent· {attribute wildcard}s, **then** the appropriate **case** among the following: 2.2.1 **If** there is an <anyAttribute> present, **then** a wildcard whose {process contents} and {annotation} are those of the ·local wildcard·, and whose {namespace constraint} is the intensional intersection of the {namespace constraint} of the ·local wildcard· and of the {namespace constraint}s of all the non-·absent· {attribute wildcard}s of the attribute groups corresponding to the <attributeGroup> [children], as defined in Attribute Wildcard Intersection (§3.10.6). 2.2.2 **If** there is no <anyAttribute> present, **then** a wildcard whose properties are as follows: **{process contents}** The {process contents} of the first non-·absent· {attribute wildcard} of an attribute group among the attribute groups corresponding to the <attributeGroup> [children]. **{namespace constraint}** The intensional intersection of the {namespace constraint}s of all the non-·absent· {attribute wildcard}s of the attribute groups corresponding to the <attributeGroup> [children], as defined in Attribute Wildcard Intersection (§3.10.6). **{annotation}** ·absent·. 3 The value is then determined by the appropriate **case** among the following: 3.1 **If** the <restriction> alternative is chosen, **then** the ·complete wildcard·; 3.2 **If** the <extension> alternative is chosen, **then** 3.2.1 [Definition:] let the **base wildcard** be defined as the appropriate **case** among the following: 3.2.1.1 **If** the type definition ·resolved· to by the ·actual value· of the `base` [attribute] is a complex type definition with an {attribute wildcard}, **then** that {attribute wildcard}. 3.2.1.2 **otherwise** ·absent·. 3.2.2 The value is then determined by the appropriate **case** among the following: 3.2.2.1 **If** the ·base wildcard· is non-·absent·, **then** the appropriate **case** among the following: 3.2.2.1.1 **If** the ·complete wildcard· is ·absent·, **then** the ·base wildcard·. 3.2.2.1.2 **otherwise** a wildcard whose {process contents} and {annotation} are those of the ·complete wildcard·, and whose {namespace constraint} is the intensional union of the {namespace constraint} of the ·complete wildcard· and of the ·base wildcard·, as defined in Attribute Wildcard Union (§3.10.6). 3.2.2.2 **otherwise** (the ·base wildcard· is ·absent·) the ·complete wildcard·
{content type}	the appropriate **case** among the following: 1 **If** the type definition ·resolved· to by the ·actual value· of the `base` [attribute] is a complex type definition whose own {content type} is a simple type definition and the <restriction> alternative is chosen, **then** starting from either 1.1 the simple type definition corresponding to the <simpleType> among the [children] of <restriction> if there is one; 1.2 otherwise (<restriction> has no <simpleType> among its [children]), the simple type definition which is the {content type} of the type definition ·resolved· to by the ·actual value· of the `base` [attribute] a simple type definition which restricts the simple type definition identified in clause 1.1 or clause 1.2 with a set of facet components corresponding to the appropriate element information items among the <restriction>'s [children] (i.e. those which specify facets, if any), as defined in Simple Type Restriction (Facets) (§3.14.6); 2 **If** the type definition ·resolved· to by the ·actual value· of the `base` [attribute] is a complex type definition whose own {content type} is *mixed* and a particle which is ·emptiable·, as defined in Particle Emptiable (§3.9.6) and the <restriction> alternative is chosen, **then** starting from the simple type definition corresponding to the <simpleType> among the [children] of <restriction> (which must be present) a simple type definition which restricts that simple type definition with a set of facet components corresponding to the appropriate element information items among the <restriction>'s [children] (i.e. those which specify facets, if any), as defined in Simple Type Restriction (Facets) (§3.14.6); 3 **If** the type definition ·resolved· to by the ·actual value· of the `base` [attribute] is a complex type definition (whose own {content type} must be a simple type definition, see below) and the <extension> alternative is chosen, **then** the {content type} of that complex type definition; 4 **otherwise** (the type definition ·resolved· to by the ·actual value· of the `base` [attribute] is a simple type definition and 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 ·actual value· of the `base` [attribute]
{derivation method}	If the <restriction> alternative is chosen, then *restriction*, otherwise (the <extension> alternative is chosen) *extension*.
{attribute uses}	A union of sets of attribute uses as follows: 1 The set of attribute uses corresponding to the <attribute> [children], if any. 2 The {attribute uses} of the attribute groups ·resolved· to by the ·actual value·s of the `ref` [attribute] of the <attributeGroup> [children], if any. 3 The {attribute uses} of the type definition ·resolved· to by the ·actual value· of the `base` [attribute], unless the <restriction> alternative is chosen, in which case some members of that type definition's {attribute uses} may not be included, namely those whose {attribute declaration}'s {name} and {target namespace} are the same as **one** of the following: 3.1 The {name} and {target namespace} of the {attribute declaration} of an attribute use in the set per clause 1 or clause 2 above; 3.2 what would have been the {name} and {target namespace} of the {attribute declaration} of an attribute use in the set per clause 1 above but for the ·actual value· of the `use` [attribute] of the relevant <attribute> among the [children] of <restriction> being *prohibited*.
{attribute wildcard}	As above for the <simpleContent> alternative.
{content type}	1 [Definition:] Let the **effective mixed** be the appropriate **case** among the following: 1.1 **If** the `mixed` [attribute] is present on <complexContent>, **then** its ·actual value·; 1.2 **If** the `mixed` [attribute] is present on <complexType>, **then** its ·actual value·; 1.3 **otherwise** `false`. 2 [Definition:] Let the **effective content** be the appropriate **case** among the following: 2.1 **If one** of the following is true 2.1.1 There is no <group>, <all>, <choice> or <sequence> among the [children]; 2.1.2 There is an <all> or <sequence> among the [children] with no [children] of its own excluding <annotation>; 2.1.3 There is a <choice> among the [children] with no [children] of its own excluding <annotation> whose `minOccurs` [attribute] has the ·actual value· `0`; , **then** the appropriate **case** among the following: 2.1.4 **If** the ·effective mixed· is `true`, **then** A particle whose properties are as follows: **{min occurs}** 1 **{max occurs}** 1 **{term}** A model group whose {compositor} is *sequence* and whose {particles} is empty. . 2.1.5 **otherwise** *empty* 2.2 **otherwise** the particle corresponding to the <all>, <choice>, <group> or <sequence> among the [children]. 3 Then the value of the property is the appropriate **case** among the following: 3.1 **If** the <restriction> alternative is chosen, **then** the appropriate **case** among the following: 3.1.1 **If** the ·effective content· is *empty* , **then** *empty*; 3.1.2 **otherwise** a pair consisting of 3.1.2.1 *mixed* if the ·effective mixed· is `true`, otherwise *elementOnly* 3.1.2.2 The ·effective content·. 3.2 **If** the <extension> alternative is chosen, **then** the appropriate **case** among the following: 3.2.1 **If** the ·effective content· is *empty*, **then** the {content type} of the type definition ·resolved· to by the ·actual value· of the `base` [attribute] 3.2.2 **If** the type definition ·resolved· to by the ·actual value· of the `base` [attribute] has a {content type} of *empty*, **then** a pair as per clause 3.1.2 above; 3.2.3 **otherwise** a pair of *mixed* or *elementOnly* (determined as per clause 3.1.2.1 above) and a particle whose properties are as follows: **{min occurs}** 1 **{max occurs}** 1 **{term}**

Complex Type Definition with complex content Schema Component	
Property	**Representation**
	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 ·actual value· of the `base` [attribute] followed by the ·effective content·.

Note: Aside from the simple coherence requirements enforced above, constraining type definitions identified as restrictions to actually *be* restrictions, that is, to ·validate· a subset of the items which are ·validated· by their base type definition, is enforced in Constraints on Complex Type Definition Schema Components (§3.4.6).
Note: The *only* substantive function of the value *prohibited* for the `use` attribute of an <attribute> is in establishing the correspondence between a complex type defined by restriction and its XML representation. It serves to prevent inheritance of an identically named attribute use from the {base type definition}. Such an <attribute> does not correspond to any component, and hence there is no interaction with either explicit or inherited wildcards in the operation of Complex Type Definition Validation Rules (§3.4.4) or Constraints on Complex Type Definition Schema Components (§3.4.6).

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.

Example

```
<xs:complexType name="length1">
 <xs:simpleContent>
  <xs:extension base="xs:nonNegativeInteger">
   <xs:attribute name="unit" type="xs:NMTOKEN"/>
  </xs:extension>
 </xs:simpleContent>
</xs:complexType>

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

  <width unit="cm">25</width>

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

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

  <depth>
   <size>25</size><unit>cm</unit>
  </depth>

<xs:complexType name="length3">
 <xs:sequence>
  <xs:element name="size" type="xs:nonNegativeInteger"/>
  <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 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:element name="surname"/>
 </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="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:number"/>
</xs:complexType>
```

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

3.4.3 Constraints on XML Representations of Complex Type Definitions

Schema Representation Constraint: Complex Type Definition Representation OK

In addition to the conditions imposed on <complexType> element information items by the schema for schemas, **all** of the following must be true:

1 If the <complexContent> alternative is chosen, the type definition ·resolved· to by the ·actual value· of the `base` [attribute] must be a complex type definition;

2 If the <simpleContent> alternative is chosen, **all** of the following must be true:

2.1 The type definition ·resolved· to by the ·actual value· of the `base` [attribute] must be **one** of the following:

2.1.1 a complex type definition whose {content type} is a simple type definition;

2.1.2 only if the <restriction> alternative is also chosen, a complex type definition whose {content type} is *mixed* and a particle which is ·emptiable·, as defined in Particle Emptiable (§3.9.6);

2.1.3 only if the <extension> alternative is also chosen, a simple type definition.

2.2 If clause 2.1.2 above is satisfied, then there must be a <simpleType> among the [children] of <restriction>.

Note: Although not explicitly ruled out either here or in Schema for Schemas (normative) (§A), specifying `<xs:complexType . . .mixed='true'` when the <simpleContent> alternative is chosen has no effect on the corresponding component, and should be avoided. This may be ruled out in a subsequent version of this specification.

3 The corresponding complex type definition component must satisfy the conditions set out in Constraints on Complex Type Definition Schema Components (§3.4.6);

4 If clause 2.2.1 or clause 2.2.2 in the correspondence specification above for {attribute wildcard} is satisfied, the intensional intersection must be expressible, as defined in Attribute Wildcard Intersection (§3.10.6).

3.4.4 Complex Type Definition Validation Rules

Validation Rule: Element Locally Valid (Complex Type)

For an element information item to be locally ·valid· with respect to a complex type definition **all** of the following must be true:

1 {abstract} is *false*.

2 If clause 3.2 of Element Locally Valid (Element) (§3.3.4) did not apply, then the appropriate **case** among the following must be true:

2.1 **If** the {content type} is *empty*, **then** the element information item has no character or element information item [children].

2.2 **If** the {content type} is a simple type definition, **then** the element information item has no element information item [children], and the ·normalized value· of the element information item is ·valid· with respect to that simple type definition as defined by String Valid (§3.14.4).

2.3 **If** the {content type} is *element-only*, **then** the element information item has no character information item [children] other than those whose [character code] is defined as a white space in [XML 1.0 (Second Edition)].

2.4 **If** the {content type} is *element-only* or *mixed*, **then** the sequence of the element information item's element information item [children], if any, taken in order, is ·valid· with respect to the {content type}'s particle, as defined in Element Sequence Locally Valid (Particle) (§3.9.4).

3 For each attribute information item in the element information item's [attributes] excepting those whose [namespace name] is identical to `http://www.w3.org/2001/XMLSchema-instance` and whose [local name] is one of `type`, `nil`, `schemaLocation` or `noNamespaceSchemaLocation`, the appropriate **case** among the following must be true:

3.1 **If** there is among the {attribute uses} an attribute use 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 name] (where an ·absent· {target namespace} is taken to be identical to a [namespace name] with no value), **then** the attribute information must be ·valid· with respect to that attribute use as per Attribute Locally Valid (Use) (§3.5.4). In this case the {attribute declaration} of that attribute use is the ·context-determined declaration· for the attribute information item with respect to Schema-Validity Assessment (Attribute) (§3.2.4) and Assessment Outcome (Attribute) (§3.2.5).

3.2 **otherwise all** of the following must be true:

3.2.1 There must be an {attribute wildcard}.

3.2.2 The attribute information item must be ·valid· with respect to it as defined in Item Valid (Wildcard) (§3.10.4).

4 The {attribute declaration} of each attribute use in the {attribute uses} whose {required} is *true* matches one of the attribute information items in the element information item's [attributes] as per clause 3.1 above.
5 Let [Definition:] the **wild IDs** be the set of all attribute information item to which clause 3.2 applied and whose ·validation· resulted in a ·context-determined declaration· of *mustFind* or no ·context-determined declaration· at all, and whose [local name] and [namespace name] resolve (as defined by QName resolution (Instance) (§3.15.4)) to an attribute declaration whose {type definition} is or is derived from ID. Then **all** of the following must be true:
5.1 There must be no more than one item in ·wild IDs·.
5.2 If ·wild IDs· is non-empty, there must not be any attribute uses among the {attribute uses} whose {attribute declaration}'s {type definition} is or is derived from ID.
Note: This clause serves to ensure that even via attribute wildcards no element has more than one attribute of type ID, and that even when an element legitimately lacks a declared attribute of type ID, a wildcard-validated attribute must not supply it. That is, if an element has a type whose attribute declarations include one of type ID, it either has that attribute or no attribute of type ID.
Note: When an {attribute wildcard} is present, this does *not* introduce any ambiguity with respect to how attribute information items for which an attribute use is present amongst the {attribute uses} whose name and target namespace match are ·assessed·. In such cases the attribute use *always* takes precedence, and the ·assessment· of such items stands or falls entirely on the basis of the attribute use and its {attribute declaration}. This follows from the details of clause 3.

3.4.5 Complex Type Definition Information Set Contributions

Schema Information Set Contribution: Attribute Default Value
For each attribute use in the {attribute uses} whose {required} is *false* and whose {value constraint} is not ·absent· but whose {attribute declaration} does not match one of the attribute information items in the element information item's [attributes] as per clause 3.1 of Element Locally Valid (Complex Type) (§3.4.4) above, the ·post-schema-validation infoset· has an attribute information item whose properties are as below added to the [attributes] of the element information item.

[local name]
The {attribute declaration}'s {name}.
[namespace name]
The {attribute declaration}'s {target namespace}.
[schema normalized value]
The canonical lexical representation of the ·effective value constraint· value.
[schema default]
The canonical lexical representation of the ·effective value constraint· value.
[validation context]
The nearest ancestor element information item with a [schema information] property.
[validity]
valid.
[validation attempted]
full.
[schema specified]
schema.

The added items should also either have [type definition] (and [member type definition] if appropriate) properties, or their lighter-weight alternatives, as specified in Attribute Validated by Type (§3.2.5).

3.4.6 Constraints on Complex Type Definition Schema Components

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

Schema Component Constraint: Complex Type Definition Properties Correct
All of the following must be true:
1 The values of the properties of a complex type definition must be as described in the property tableau in The Complex Type Definition Schema Component (§3.4.1), modulo the impact of Missing Sub-components (§5.3).
2 If the {base type definition} is a simple type definition, the {derivation method} must be *extension*.
3 Circular definitions are disallowed, except for the ·ur-type definition·. That is, it must be possible to reach the ·ur-type definition· by repeatedly following the {base type definition}.
4 Two distinct attribute declarations in the {attribute uses} must not have identical {name}s and {target namespace}s.
5 Two distinct attribute declarations in the {attribute uses} must not have {type definition}s which are or are derived from ID.

Schema Component Constraint: Derivation Valid (Extension)
If the {derivation method} is *extension*, the appropriate **case** among the following must be true:
1 **If** the {base type definition} is a complex type definition, **then all** of the following must be true:
1.1 The {final} of the {base type definition} must not contain *extension*.
1.2 Its {attribute uses} must be a subset of the {attribute uses} of the complex type definition itself, that is, for every attribute use in the {attribute uses} of the {base type definition}, there must be an attribute use in the {attribute uses} of the complex type definition itself whose {attribute declaration} has the same {name}, {target namespace} and {type definition} as its attribute declaration.
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 (§3.10.6).
1.4 **One** of the following must be true:
1.4.1 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.
1.4.2 The {content type} of both the {base type definition} and the complex type definition itself must be *empty*.
1.4.3 **All** of the following must be true:
1.4.3.1 The {content type} of the complex type definition itself must specify a particle.
1.4.3.2 **One** of the following must be true:
1.4.3.2.1 The {content type} of the {base type definition} must be *empty*.
1.4.3.2.2 **All** of the following must be true:
1.4.3.2.2.1 Both {content type}s must be *mixed* or both must be *element-only*.
1.4.3.2.2.2 The particle of the complex type definition must be a ·valid extension· of the {base type definition}'s particle, as defined in Particle Valid (Extension) (§3.9.6).

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 trivial 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.

2 **If** the {base type definition} is a simple type definition, **then all** of the following must be true:
2.1 The {content type} must be the same simple type definition.
2.2 The {final} of the {base type definition} must not contain *extension*.

[Definition:] If this constraint Derivation Valid (Extension) (§3.4.6) holds of a complex type definition, it is a **valid extension** of its {base type definition}.

Schema Component Constraint: Derivation Valid (Restriction, Complex)

If the {derivation method} is *restriction* **all** of the following must be true:
1 The {base type definition} must be a complex type definition whose {final} does not contain *restriction*.
2 For each attribute use (call this **R**) in the {attribute uses} the appropriate **case** among the following must be true:
2.1 **If** there is an attribute use in the {attribute uses} of the {base type definition} (call this **B**) whose {attribute declaration} has the same {name} and {target namespace}, **then all** of the following must be true:
2.1.1 **one** of the following must be true:
2.1.1.1 **B**'s {required} is *false*.
2.1.1.2 **R**'s {required} is *true*.
2.1.2 **R**'s {attribute declaration}'s {type definition} must be validly derived from **B**'s {type definition} given the empty set as defined in Type Derivation OK (Simple) (§3.14.6).
2.1.3 [Definition:] Let the **effective value constraint** of an attribute use be its {value constraint}, if present, otherwise its {attribute declaration}'s {value constraint} . Then **one** of the following must be true:
2.1.3.1 **B**'s ·effective value constraint· is ·absent· or *default*.
2.1.3.2 **R**'s ·effective value constraint· is *fixed* with the same string as **B**'s.
2.2 **otherwise** the {base type definition} must have an {attribute wildcard} and the {target namespace} of the **R**'s {attribute declaration} must be ·valid· with respect to that wildcard, as defined in Wildcard allows Namespace Name (§3.10.4).
3 For each attribute use in the {attribute uses} of the {base type definition} whose {required} is *true*, there must be an attribute use with an {attribute declaration} with the same {name} and {target namespace} as its {attribute declaration} in the {attribute uses} of the complex type definition itself whose {required} is *true*.
4 If there is an {attribute wildcard}, **all** of the following must be true:
4.1 The {base type definition} must also have one.
4.2 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 (§3.10.6).
4.3 Unless the {base type definition} is the ·ur-type definition·, the complex type definition's {attribute wildcard}'s {process contents} must be identical to or stronger than the {base type definition}'s {attribute wildcard}'s {process contents}, where *strict* is stronger than *lax* is stronger than *skip*.
5 **One** of the following must be true:
5.1 The {base type definition} must be the ·ur-type definition·.
5.2 **All** of the following must be true:
5.2.1 The {content type} of the complex type definition must be a simple type definition
5.2.2 **One** of the following must be true:
5.2.2.1 The {content type} of the {base type definition} must be a simple type definition from which the {content type} is validly derived given the empty set as defined in Type Derivation OK (Simple) (§3.14.6).
5.2.2.2 The {base type definition} must be *mixed* and have a particle which is ·emptiable· as defined in Particle Emptiable (§3.9.6).
5.3 **All** of the following must be true:
5.3.1 The {content type} of the complex type itself must be *empty*
5.3.2 **One** of the following must be true:
5.3.2.1 The {content type} of the {base type definition} must also be *empty*.
5.3.2.2 The {content type} of the {base type definition} must be *elementOnly* or *mixed* and have a particle which is ·emptiable· as defined in Particle Emptiable (§3.9.6).
5.4 **All** of the following must be true:
5.4.1 **One** of the following must be true:
5.4.1.1 The {content type} of the complex type definition itself must be *element-only*
5.4.1.2 The {content type} of the complex type definition itself and of the {base type definition} must be *mixed*
5.4.2 The particle of the complex type definition itself must be a ·valid restriction· of the particle of the {content type} of the {base type definition} as defined in Particle Valid (Restriction) (§3.9.6).

Note: Attempts to derive complex type definitions whose {content type} is *element-only* by restricting a {base type definition} whose {content type} is *empty* are not ruled out by this clause. However if the complex type definition itself has a non-pointless particle it will fail to satisfy Particle Valid (Restriction) (§3.9.6). On the other hand some type definitions with pointless *element-only* content, for example an empty <sequence>, will satisfy Particle Valid (Restriction) (§3.9.6) with respect to an *empty* {base type definition}, and so be valid restrictions.

[Definition:] If this constraint Derivation Valid (Restriction, Complex) (§3.4.6) holds of a complex type definition, it is a **valid restriction** of its {base type definition}.

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.

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

Schema Component Constraint: Type Derivation OK (Complex)

For a complex type definition (call it **D**, for derived) to be validly derived from a type definition (call this **B**, for base) given a subset of {*extension*, *restriction*} **all** of the following must be true:
1 If **B** and **D** are not the same type definition, then the {derivation method} of **D** must not be in the subset.
2 **One** of the following must be true:
2.1 **B** and **D** must be the same type definition.
2.2 **B** must be **D**'s {base type definition}.
2.3 **All** of the following must be true:
2.3.1 **D**'s {base type definition} must not be the ·ur-type definition·.

2.3.2 The appropriate **case** among the following must be true:
2.3.2.1 **If D**'s {base type definition} is complex, **then** it must be validly derived from **B** given the subset as defined by this constraint.
2.3.2.2 **If D**'s {base type definition} is simple, **then** it must be validly derived from **B** given the subset as defined in Type Derivation OK (Simple) (§3.14.6).

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.

Note:

The wording of clause 2.1 above appeals to a notion of component identity which is only incompletely defined by this version of this specification. In some cases, the wording of this specification does make clear the rules for component identity. These cases include:

- When they are both top-level components with the same component type, namespace name, and local name;
- When they are necessarily the same type definition (for example, when the two types definitions in question are the type definitions associated with two attribute or element declarations, which are discovered to be the same declaration);
- When they are the same by construction (for example, when an element's type definition defaults to being the same type definition as that of its substitution-group head or when a complex type definition inherits an attribute declaration from its base type definition).

In other cases two conforming implementations may disagree as to whether components are identical.

3.4.7 Built-in Complex Type Definition

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/2001/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 the following properties:

Property	**Value**
{namespace constraint}	*any*
{process contents}	*lax*

{attribute uses}	The empty set
{attribute wildcard}	a wildcard with the following properties::

Property	**Value**
{namespace constraint}	*any*
{process contents}	*lax*

{final}	The empty set
{prohibited substitutions}	The empty set
{abstract}	*false*

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

Note: This specification does not provide an inventory of built-in complex type definitions for use in user schemas. A preliminary library of complex type definitions is available which includes both mathematical (e.g. `rational`) and utility (e.g. `array`) type definitions. In particular, there is a `text` type definition which is recommended for use as the type definition in element declarations intended for general text content, as it makes sensible provision for various aspects of internationalization. For more details, see the schema document for the type library at its namespace name: http://www.w3.org/2001/03/XMLSchema/TypeLibrary.xsd.

3.5 AttributeUses

3.5.1 The Attribute Use Schema Component
3.5.2 XML Representation of Attribute Use Components

An attribute use is a utility component which controls the occurrence and defaulting behavior of attribute declarations. It plays the same role for attribute declarations in complex types that particles play for element declarations.

Example

```
<xs:complexType>
 . . .
 <xs:attribute ref="xml:lang" use="required"/>
 <xs:attribute ref="xml:space" default="preserve"/>
 <xs:attribute name="version" type="xs:number" fixed="1.0"/>
</xs:complexType>
```

XML representations which all involve attribute uses, illustrating some of the possibilities for controlling occurrence.

3.5.1 The Attribute Use Schema Component

The attribute use schema component has the following properties:

Schema Component: Attribute Use

{required}
A boolean.
{attribute declaration}
An attribute declaration.
{value constraint}
Optional. A pair consisting of a value and one of *default*, *fixed*.

{required} determines whether this use of an attribute declaration requires an appropriate attribute information item to be present, or merely allows it.

{attribute declaration} provides the attribute declaration itself, which will in turn determine the simple type definition used.

{value constraint} allows for local specification of a default or fixed value. This must be consistent with that of the {attribute declaration}, in that if the {attribute declaration} specifies a fixed value, the only allowed {value constraint} is the same fixed value.

3.5.2 XML Representation of Attribute Use Components

Attribute uses correspond to all uses of <attribute> which allow a `use` attribute. These in turn correspond to *two* components in each case, an attribute use and its {attribute declaration} (although note the latter is not new when the attribute use is a reference to a top-level attribute declaration). The appropriate mapping is described in XML Representation of Attribute Declaration Schema Components (§3.2.2).

3.5.3 Constraints on XML Representations of Attribute Uses

None as such.

3.5.4 Attribute Use Validation Rules

Validation Rule: Attribute Locally Valid (Use)
For an attribute information item to be ·valid· with respect to an attribute use its ·normalized value· must match the canonical lexical representation of the attribute use's {value constraint} value, if it is present and *fixed*.

3.5.5 Attribute Use Information Set Contributions

None as such.

3.5.6 Constraints on Attribute Use Schema Components

All attribute uses (see AttributeUses (§3.5)) must satisfy the following constraints.

Schema Component Constraint: Attribute Use Correct
All of the following must be true:
1 The values of the properties of an attribute use must be as described in the property tableau in The Attribute Use Schema Component (§3.5.1), modulo the impact of Missing Sub-components (§5.3).
2 If the {attribute declaration} has a *fixed* {value constraint}, then if the attribute use itself has a {value constraint}, it must also be *fixed* and its value must match that of the {attribute declaration}'s {value constraint}.

3.6 Attribute Group Definitions

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 ·validation· as such, but the {attribute uses} 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 entity facility. Attribute group definitions are provided primarily for reference from the XML representation of schema components (see <complexType> and <attributeGroup>).

Example

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

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

XML representations for attribute group definitions. The effect is as if the attribute declarations in the group were present in the type definition.

3.6.1 The Attribute Group Definition Schema Component

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 name, as defined in [XML-Namespaces].
{attribute uses}
A set of attribute uses.
{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 (§4.2.3) for the use of component identifiers when importing one schema into another.

{attribute uses} is a set attribute uses, allowing for local specification of occurrence and default or fixed values.

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

See Annotations (§3.13) for information on the role of the {annotation} property.

3.6.2 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> or <redefine>, 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 ·actual value· of the `name` [attribute]
{target namespace}	The ·actual value· of the `targetNamespace` [attribute] of the parent `schema` element information item.
{attribute uses}	The union of the set of attribute uses corresponding to the <attribute> [children], if any, with the {attribute uses} of the attribute groups ·resolved· to by the ·actual value·s of the `ref` [attribute] of the <attributeGroup> [children], if any.
{attribute wildcard}	As for the ·complete wildcard· as described in XML Representation of Complex Type Definitions (§3.4.2).
{annotation}	The annotation corresponding to the <annotation> element information item in the [children], if present, otherwise ·absent·.

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.

3.6.3 Constraints on XML Representations of Attribute Group Definitions

Schema Representation Constraint: Attribute Group Definition Representation OK
In addition to the conditions imposed on <attributeGroup> element information items by the schema for schemas, **all** of the following must be true:
1 The corresponding attribute group definition, if any, must satisfy the conditions set out in Constraints on Attribute Group Definition Schema Components (§3.6.6).
2 If clause 2.2.1 or clause 2.2.2 in the correspondence specification in XML Representation of Complex Type Definitions (§3.4.2) for {attribute wildcard}, as referenced above, is satisfied, the intensional intersection must be expressible, as defined in Attribute Wildcard Intersection (§3.10.6).
3 Circular group reference is disallowed outside <redefine>. That is, unless this element information item's parent is <redefine>, then among the [children], if any, there must not be an <attributeGroup> with `ref` [attribute] which resolves to the component corresponding to this <attributeGroup>. Indirect circularity is also ruled out. That is, when QName resolution (Schema Document) (§3.15.3) is applied to a ·QName· arising from any <attributeGroup>s with a `ref` [attribute] among the [children], it must not be the case that a ·QName· is encountered at any depth which resolves to the component corresponding to this <attributeGroup>.

3.6.4 Attribute Group Definition Validation Rules

None as such.

3.6.5 Attribute Group Definition Information Set Contributions

None as such.

3.6.6 Constraints on Attribute Group Definition Schema Components

All attribute group definitions (see Attribute Group Definitions (§3.6)) must satisfy the following constraint.

Schema Component Constraint: Attribute Group Definition Properties Correct
All of the following must be true:
1 The values of the properties of an attribute group definition must be as described in the property tableau in The Attribute Group Definition Schema Component (§3.6.1), modulo the impact of Missing Sub-components (§5.3);
2 Two distinct members of the {attribute uses} must not have {attribute declaration}s both of whose {name}s match and whose {target namespace}s are identical.
3 Two distinct members of the {attribute uses} must not have {attribute declaration}s both of whose {type definition}s are or are derived from ID.

3.7 Model Group Definitions

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 Definitions (§3.4.2) (see <complexType> and <group>). Thus, model group definitions provide a replacement for some uses of XML's parameter entity facility.

Example

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

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

3.7.1 The Model Group Definition Schema Component

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 name, 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 (§4.2.3) for the use of component identifiers when importing one schema into another.

Model group definitions *per se* do not participate in ·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 Annotations (§3.13) for information on the role of the {annotation} property.

3.7.2 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 = (nonNegativeInteger | unbounded)  : 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> or <redefine> 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 ·actual value· of the `name` [attribute]
{target namespace}	The ·actual 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 a `ref` [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 ·actual value· of the `minOccurs` [attribute], if present, otherwise `1`.
{max occurs}	*unbounded*, if the `maxOccurs` [attribute] equals *unbounded*, otherwise the ·actual value· of the `maxOccurs` [attribute], if present, otherwise `1`.
{term}	The {model group} of the model group definition ·resolved· to by the ·actual value· of the `ref` [attribute]

The name of this section is slightly misleading, in that the second, un-named, case above (with a `ref` and no `name`) is not really a named model group at all, but a reference to one. Also note that in the first (named) case above no reference is made to `minOccurs` or `maxOccurs`: this is because the schema for schemas does not allow them on the child of <group> when it is named. This in turn is because the {min occurs} and {max occurs} of the particles which *refer* to the definition are what count.

Given the constraints on its appearance in content models, an <all> should only occur as the only item in the [children] of a named model group definition or a content model: see Constraints on Model Group Schema Components (§3.8.6).

3.7.3 Constraints on XML Representations of Model Group Definitions

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 corresponding model group definition, if any, must satisfy the conditions set out in Constraints on Model Group Schema Components (§3.8.6).

3.7.4 Model Group Definition Validation Rules

None as such.

3.7.5 Model Group Definition Information Set Contributions

None as such.

3.7.6 Constraints on Model Group Definition Schema Components

All model group definitions (see Model Group Definitions (§3.7)) must satisfy the following constraint.

Schema Component Constraint: Model Group Definition Properties Correct
The values of the properties of a model group definition must be as described in the property tableau in The Model Group Definition Schema Component (§3.7.1), modulo the impact of Missing Sub-components (§5.3).

3.8 Model Groups

When the [children] of element information items are not constrained to be *empty* or by reference to a simple type definition (Simple Type Definitions (§3.14)), 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.

Example

```
<xs:all>
 <xs:element ref="cats"/>
 <xs:element ref="dogs"/>
</xs:all>

<xs:sequence>
 <xs:choice>
  <xs:element ref="left"/>
  <xs:element ref="right"/>
 </xs:choice>
 <xs:element ref="landmark"/>
</xs:sequence>
```

XML representations for the three kinds of model group, the third nested inside the second.

3.8.1 The Model Group Schema Component

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.

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*) contain all and only exactly zero or one of each element specified in {particles}. The elements can occur in any order. In this case, to reduce implementation complexity, {particles} is restricted to contain local and top-level element declarations only, with {min occurs}=0 or 1, {max occurs}=1.

When two or more particles contained directly or indirectly in the {particles} of a model group have identically named element declarations as their {term}, the type definitions of those declarations must be the same. By 'indirectly' is meant particles within the {particles} of a group which is itself the {term} of a directly contained particle, and so on recursively.

See Annotations (§3.13) for information on the role of the {annotation} property.

3.8.2 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 = 1 : 1
  minOccurs = (0 | 1) : 1
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, element*)
</all>
```

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

```
<sequence
  id = ID
  maxOccurs = (nonNegativeInteger | unbounded)  : 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 ·actual value· of the `minOccurs` [attribute], if present, otherwise `1`.
{max occurs}	*unbounded*, if the `maxOccurs` [attribute] equals *unbounded*, otherwise the ·actual 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·.

3.8.3 Constraints on XML Representations of Model Groups

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 corresponding particle and model group must satisfy the conditions set out in Constraints on Model Group Schema Components (§3.8.6) and Constraints on Particle Schema Components (§3.9.6).

3.8.4 Model Group Validation Rules

Validation Rule: Element Sequence Valid

[Definition:] 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.

For a sequence (possibly empty) of element information items to be locally ·valid· with respect to a model group the appropriate **case** among the following must be true:

1 **If** the {compositor} is *sequence*, **then** there must be 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 ·valid· with respect to the corresponding particle in the {particles} as defined in Element Sequence Locally Valid (Particle) (§3.9.4).
2 **If** the {compositor} is *choice*, **then** there must be a particle among the {particles} such that the sequence is ·valid· with respect to that particle as defined in Element Sequence Locally Valid (Particle) (§3.9.4).
3 **If** the {compositor} is *all*, **then** there must be 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 ·valid· with respect to the corresponding particle as defined in Element Sequence Locally Valid (Particle) (§3.9.4).

Nothing in the above should be understood as ruling out groups whose {particles} is empty: although no sequence can be ·valid· with respect to such a group whose {compositor} is *choice*, the empty sequence *is* ·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 top-level element declarations (see Constraints on Model Group Schema Components (§3.8.6)). 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.

3.8.5 Model Group Information Set Contributions

None as such.

3.8.6 Constraints on Model Group Schema Components

All model groups (see Model Groups (§3.8)) must satisfy the following constraints.

Schema Component Constraint: Model Group Correct
All of the following must be true:
1 The values of the properties of a model group must be as described in the property tableau in The Model Group Schema Component (§3.8.1), modulo the impact of Missing Sub-components (§5.3).
2 Circular groups are disallowed. That is, within the {particles} of a group there must not be at any depth a particle whose {term} is the group itself.

Schema Component Constraint: All Group Limited
When a model group has {compositor} *all*, then **all** of the following must be true:
1 It appears only as the value of one or both of the following properties:
1.1 the {model group} property of a model group definition.
1.2 the {term} property of a particle with {max occurs}=1which is part of a pair which constitutes the {content type} of a complex type definition.
2 The {max occurs} of all the particles in the {particles} of the group must be `0` or `1`.

Schema Component Constraint: 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}, then all their type definitions must be the same top-level definition, that is, **all** of the following must be true:
1 all their {type definition}s must have a non-·absent· {name}.
2 all their {type definition}s must have the same {name}.
3 all their {type definition}s must have the same {target namespace}.

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

Schema Component Constraint: Unique Particle Attribution
A content model must be formed such that during ·validation· of an element information item sequence, the particle component contained directly, indirectly or ·implicitly· therein with which to attempt to ·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 1.0 (Second Edition)] and SGML. Given the presence of element substitution groups and wildcards, the concise expression of this constraint is difficult, see Analysis of the Unique Particle Attribution Constraint (non-normative) (§H) for further discussion.

Since this constraint is expressed at the component level, it applies to content models whose origins (e.g. via type derivation and references to named model groups) are no longer evident. So particles at different points in the content model are always distinct from one another, even if they originated from the same named model group.

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.

Schema Component Constraint: 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}).

Schema Component Constraint: 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}).

3.9 Particles

As described in Model Groups (§3.8), particles contribute to the definition of content models.

Example

```
<xs:element ref="egg" minOccurs="12" maxOccurs="12"/>

<xs:group ref="omelette" minOccurs="0"/>

<xs:any maxOccurs="unbounded"/>
```

XML representations which all involve particles, illustrating some of the possibilities for controlling occurrence.

3.9.1 The Particle Schema Component

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.

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. When the {term} is an element declaration or wildcard, {min occurs} determines the minimum number of such element [children] that can 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.

Again, when the {term} is an element declaration or wildcard, 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.

When the {term} is a model group, the permitted occurrence range is determined by a combination of {min occurs} and {max occurs} and the occurrence ranges of the {term}'s {particles}.

3.9.2 XML Representation of Particle Components

Particles correspond to all three elements (<element> not immediately within <schema>, <group> not immediately within <schema> and <any>) which allow `minOccurs` and `maxOccurs` attributes. These in turn correspond to *two* components in each case, a particle and its {term}. The appropriate mapping is described in XML Representation of Element Declaration Schema Components (§3.3.2), XML Representation of Model Group Schema Components (§3.8.2) and XML Representation of Wildcard Schema Components (§3.10.2) respectively.

3.9.3 Constraints on XML Representations of Particles

None as such.

3.9.4 Particle Validation Rules

Validation Rule: Element Sequence Locally Valid (Particle)
For a sequence (possibly empty) of element information items to be locally ·valid· with respect to a particle the appropriate **case** among the following must be true:
1 **If** the {term} is a wildcard, **then all** of the following must be true:
1.1 The length of the sequence must be greater than or equal to the {min occurs}.
1.2 If {max occurs} is a number, the length of the sequence must be less than or equal to the {max occurs}.
1.3 Each element information item in the sequence must be ·valid· with respect to the wildcard as defined by Item Valid (Wildcard) (§3.10.4).
2 **If** the {term} is an element declaration, **then all** of the following must be true:
2.1 The length of the sequence must be greater than or equal to the {min occurs}.
2.2 If {max occurs} is a number, the length of the sequence must be less than or equal to the {max occurs}.
2.3 For each element information item in the sequence **one** of the following must be true:
2.3.1 The element declaration is local (i.e. its {scope} must not be *global*), its {abstract} is *false*, the element information item's [namespace name] is identical to the element declaration's {target namespace} (where an ·absent· {target namespace} is taken to be identical to a [namespace name] with no value) and the element information item's [local name] matches the element declaration's {name}.

In this case the element declaration is the ·context-determined declaration· for the element information item with respect to Schema-Validity Assessment (Element) (§3.3.4) and Assessment Outcome (Element) (§3.3.5).

2.3.2 The element declaration is top-level (i.e. its {scope} is *global*), {abstract} is *false*, the element information item's [namespace name] is identical to the element declaration's {target namespace} (where an ·absent· {target namespace} is taken to be identical to a [namespace name] with no value) and the element information item's [local name] matches the element declaration's {name}.

In this case the element declaration is the ·context-determined declaration· for the element information item with respect to Schema-Validity Assessment (Element) (§3.3.4) and Assessment Outcome (Element) (§3.3.5).

2.3.3 The element declaration is top-level (i.e. its {scope} is *global*), its {disallowed substitutions} does not contain *substitution*, the [local] and [namespace name] of the element information item resolve to an element declaration, as defined in QName resolution (Instance)

(§3.15.4) -- [Definition:] call this declaration the **substituting declaration** and 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) (§3.3.6).

In this case the ·substituting declaration· is the ·context-determined declaration· for the element information item with respect to Schema-Validity Assessment (Element) (§3.3.4) and Assessment Outcome (Element) (§3.3.5).

3 **If** the {term} is a model group, **then all** of the following must be true:
3.1 There is a ·partition· of the sequence into n sub-sequences such that n is greater than or equal to {min occurs}.
3.2 If {max occurs} is a number, n must be less than or equal to {max occurs}.
3.3 Each sub-sequence in the ·partition· is ·valid· with respect to that model group as defined in Element Sequence Valid (§3.8.4).

Note: Clauses clause 1 and clause 2.3.3 do not interact: an element information item validatable by a declaration with a substitution group head in a different namespace is *not* validatable by a wildcard which accepts the head's namespace but not its own.

3.9.5 Particle Information Set Contributions

None as such.

3.9.6 Constraints on Particle Schema Components

All particles (see Particles (§3.9)) must satisfy the following constraints.

Schema Component Constraint: Particle Correct
All of the following must be true:
1 The values of the properties of a particle must be as described in the property tableau in The Particle Schema Component (§3.9.1), modulo the impact of Missing Sub-components (§5.3).
2 If {max occurs} is not *unbounded*, that is, it has a numeric value, then **all** of the following must be true:
2.1 {min occurs} must not be greater than {max occurs}.
2.2 {max occurs} must be greater than or equal to 1.

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

Schema Component Constraint: Particle Valid (Extension)
[Definition:] For a particle (call it **E**, for extension) to be a **valid extension** of another particle (call it **B**, for base) **one** of the following must be true:
1 They are the same particle.
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.

The approach to defining a type by restricting another type definition set out here is designed to ensure that types defined in this way are guaranteed to be a subset of the type they restrict. This is accomplished by requiring a clear mapping between the components of the base type definition and the restricting type definition. Permissible mappings are set out below via a set of recursive definitions, bottoming out in the obvious cases, e.g. where an (restricted) element declaration corresponds to another (base) element declaration with the same name and type but the same or wider range of occurrence.

Note: The structural correspondence approach to guaranteeing the subset relation set out here is necessarily verbose, but has the advantage of being checkable in a straightforward way. The working group solicits feedback on how difficult this is in practice, and on whether other approaches are found to be viable.

Schema Component Constraint: Particle Valid (Restriction)
[Definition:] For a particle (call it **R**, for restriction) to be a **valid restriction** of another particle (call it **B**, for base) **one** of the following must be true:
1 They are the same particle.
2 depending on the kind of particle, per the table below, with the qualifications that **all** of the following must be true:
2.1 Any top-level element declaration particle (in **R** or **B**) which is the {substitution group affiliation} of one or more other element declarations and whose ·substitution group· contains at least one element declaration other than itself is treated as if it were a *choice* group whose {min occurs} and {max occurs} are those of the particle, and whose {particles} consists of one particle with {min occurs} and {max occurs} of 1 for each of the declarations in its ·substitution group·.
2.2 Any pointless occurrences of <sequence>, <choice> or <all> are ignored, where pointlessness is understood as follows:

<sequence>
One of the following must be true:
2.2.1 {particles} is empty.
2.2.2 **All** of the following must be true:
2.2.2.1 The particle within which this <sequence> appears has {max occurs} and {min occurs} of 1.
2.2.2.2 **One** of the following must be true:
2.2.2.2.1 The <sequence>'s {particles} has only one member.
2.2.2.2.2 The particle within which this <sequence> appears is itself among the {particles} of a <sequence>.

<all>
One of the following must be true:
2.2.1 {particles} is empty.
2.2.2 {particles} has only one member.

<choice>
One of the following must be true:
2.2.1 {particles} is empty and the particle within which this <choice> appears has {min occurs} of 0.
2.2.2 **All** of the following must be true:
2.2.2.1 The particle within which this <choice> appears has {max occurs} and {min occurs} of 1.
2.2.2.2 **One** of the following must be true:
2.2.2.2.1 The <choice>'s {particles} has only one member.
2.2.2.2.2 The particle within which this <choice> appears is itself among the {particles} of a <choice>.

	Base Particle				

		Base Particle				
		elt	**any**	**all**	**choice**	**sequence**
Derived Particle	**elt**	NameAnd- TypeOK	NSCompat	Recurse- AsIfGroup	Recurse- AsIfGroup	RecurseAs- IfGroup
	any	Forbidden	NSSubset	Forbidden	Forbidden	Forbidden
	all	Forbidden	NSRecurse- CheckCardinality	Recurse	Forbidden	Forbidden
	choice	Forbidden	NSRecurse- CheckCardinality	Forbidden	RecurseLax	Forbidden
	seq- uence	Forbidden	NSRecurse- CheckCardinality	Recurse- Unordered	MapAndSum	Recurse

Schema Component Constraint: Occurrence Range OK

For a particle's occurrence range to be a valid restriction of another's occurrence range **all** of the following must be true:

1 Its {min occurs} is greater than or equal to the other's {min occurs}.

2 **one** of the following must be true:

2.1 The other's {max occurs} is *unbounded*.

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

Schema Component Constraint: Particle Restriction OK (Elt:Elt -- NameAndTypeOK)

For an element declaration particle to be a ·valid restriction· of another element declaration particle **all** of the following must be true:

1 The declarations' {name}s and {target namespace}s are the same.

2 **R**'s occurrence range is a valid restriction of **B**'s occurrence range as defined by Occurrence Range OK (§3.9.6).

3 **One** of the following must be true:

3.1 Both **B**'s declaration's {scope} and **R**'s declaration's {scope} are *global*.

3.2 **All** of the following must be true:

3.2.1 Either **B**'s {nillable} is *true* or **R**'s {nillable} is *false*.

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

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

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

3.2.5 **R**'s {type definition} is validly derived given {*extension*, *list*, *union*} from **B**'s {type definition} as defined by Type Derivation OK (Complex) (§3.4.6) or Type Derivation OK (Simple) (§3.14.6), as appropriate.

Note: The above constraint on {type definition} means that in deriving a type by restriction, any contained type definitions must themselves be explicitly derived by restriction from the corresponding type definitions in the base definition, or be one of the member types of a corresponding union..

Schema Component Constraint: Particle Derivation OK (Elt:Any -- NSCompat)

For an element declaration particle to be a ·valid restriction· of a wildcard particle **all** of the following must be true:

1 The element declaration's {target namespace} is ·valid· with respect to the wildcard's {namespace constraint} as defined by Wildcard allows Namespace Name (§3.10.4).

2 **R**'s occurrence range is a valid restriction of **B**'s occurrence range as defined by Occurrence Range OK (§3.9.6).

Schema Component Constraint: Particle Derivation OK (Elt:All/Choice/Sequence -- RecurseAsIfGroup)

For an element declaration particle to be a ·valid restriction· of a group particle (*all*, *choice* or *sequence*) 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 must be a ·valid restriction· of the group as defined by Particle Derivation OK (All:All,Sequence:Sequence -- Recurse) (§3.9.6), Particle Derivation OK (Choice:Choice -- RecurseLax) (§3.9.6) or Particle Derivation OK (All:All,Sequence:Sequence -- Recurse) (§3.9.6), depending on whether the group is *all*, *choice* or *sequence*.

Schema Component Constraint: Particle Derivation OK (Any:Any -- NSSubset)

For a wildcard particle to be a ·valid restriction· of another wildcard particle **all** of the following must be true:

1 **R**'s occurrence range must be a valid restriction of **B**'s occurrence range as defined by Occurrence Range OK (§3.9.6).

2 **R**'s {namespace constraint} must be an intensional subset of **B**'s {namespace constraint} as defined by Wildcard Subset (§3.10.6).

3 Unless **B** is the content model wildcard of the ·ur-type definition·, **R**'s {process contents} must be identical to or stronger than **B**'s {process contents}, where *strict* is stronger than *lax* is stronger than *skip*.

Note:

The exception to the third clause above for derivations from the ·ur-type definition· is necessary as its wildcards have a {process contents} of *lax*, so without this exception, no use of wildcards with {process contents} of *skip* would be possible.

Schema Component Constraint: Particle Derivation OK (All/Choice/Sequence:Any -- NSRecurseCheckCardinality)

For a group particle to be a ·valid restriction· of a wildcard particle **all** of the following must be true:

1 Every member of the {particles} of the group is a ·valid restriction· of the wildcard as defined by Particle Valid (Restriction) (§3.9.6).

2 The effective total range of the group, as defined by Effective Total Range (all and sequence) (§3.8.6) (if the group is *all* or *sequence*) or Effective Total Range (choice) (§3.8.6) (if it is *choice*) is a valid restriction of **B**'s occurrence range as defined by Occurrence Range OK (§3.9.6).

Schema Component Constraint: Particle Derivation OK (All:All,Sequence:Sequence -- Recurse)

For an *all* or *sequence* group particle to be a ·valid restriction· of another group particle with the same {compositor} **all** of the following must be true:

1 **R**'s occurrence range is a valid restriction of **B**'s occurrence range as defined by Occurrence Range OK (§3.9.6).

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 **all** of the following must be true:

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) (§3.9.6).

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 (§3.9.6).

Note: Although the ·validation· semantics of an *all* group does not depend on the order of its particles, derived *all* groups are required to match the order of their base in order 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**.

Schema Component Constraint: Particle Derivation OK (Choice:Choice -- RecurseLax)

For a *choice* group particle to be a ·valid restriction· of another *choice* group particle **all** of the following must be true:

1 **R**'s occurrence range is a valid restriction of **B**'s occurrence range as defined by Occurrence Range OK (§3.9.6);

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) (§3.9.6).

Note: Although the ·validation· semantics of a *choice* group does not depend on the order of its particles, derived *choice* groups are required to match the order of their base in order to simplify checking that the derivation is OK.

Schema Component Constraint: Particle Derivation OK (Sequence:All -- RecurseUnordered)

For a *sequence* group particle to be a ·valid restriction· of an *all* group particle **all** of the following must be true:

1 **R**'s occurrence range is a valid restriction of **B**'s occurrence range as defined by Occurrence Range OK (§3.9.6).

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 **all** of the following must be true:

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

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) (§3.9.6);

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 (§3.9.6).

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

Schema Component Constraint: Particle Derivation OK (Sequence:Choice -- MapAndSum)

For a *sequence* group particle to be a ·valid restriction· of a *choice* group particle **all** of the following must be true:

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) (§3.9.6).

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 (§3.9.6).

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.

Schema Component Constraint: Particle Emptiable

[Definition:] For a particle to be **emptiable one** of the following must be true:

1 Its {min occurs} is `0`.

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) (§3.8.6) (if the group is *all* or *sequence*) or Effective Total Range (choice) (§3.8.6) (if it is *choice*), is `0`.

3.10 Wildcards

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 name, but independently of their local name.

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"/>
```

XML representations of the four basic types of wildcard, plus one attribute wildcard.

3.10.1 The Wildcard Schema Component

The wildcard schema component has the following properties:

Schema Component: Wildcard

{namespace constraint}
One of *any*; a pair of *not* and a namespace name or ·absent·; or a set whose members are either namespace names or ·absent·.

{process contents}
One of *skip*, *lax* or *strict*.

{annotation}
Optional. An annotation.

{namespace constraint} provides for ·validation· of attribute and element items that:

1. (*any*) have any namespace or are not namespace-qualified;
2. (*not* and a namespace name) are namespace-qualified with a namespace other than the specified namespace name;

3. (*not* and ·absent·) are namespace-qualified;
4. (a set whose members are either namespace names 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 ·assessment· of the information items allowed by wildcards, as follows:

strict
There must be a top-level declaration for the item available, or the item must have an `xsi:type`, and the item must be ·valid· as appropriate.

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

lax
If the item has a uniquely determined declaration available, it must be ·valid· with respect to that definition, that is, ·validate· if you can, don't worry if you can't.

See Annotations (§3.13) for information on the role of the {annotation} property.

3.10.2 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 = (nonNegativeInteger | unbounded)  : 1
  minOccurs = nonNegativeInteger : 1
  namespace = ((##any | ##other) | List of (anyURI | (##targetNamespace | ##local)) )  : ##any
  processContents = (lax | skip | 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 ·actual value· of the `minOccurs` [attribute], if present, otherwise `1`.
{max occurs}	*unbounded*, if the `maxOccurs` [attribute] equals *unbounded*, otherwise the ·actual 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 ·actual value· of the `namespace` [attribute]: if absent, then *any*, otherwise as follows: **##any** *any* **##other** a pair of *not* and the ·actual value· of the `targetNamespace` [attribute] of the <schema> ancestor element information item if present, otherwise ·absent·. ***otherwise*** a set whose members are namespace names corresponding to the space-delimited substrings of the string, except 1 if one such substring is `##targetNamespace`, the corresponding member is the ·actual 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}	The ·actual 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 (§3.8.6)) 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.

3.10.3 Constraints on XML Representations of Wildcards

Schema Representation Constraint: Wildcard Representation OK
In addition to the conditions imposed on <any> element information items by the schema for schemas, the corresponding particle and model group must satisfy the conditions set out in Constraints on Model Group Schema Components (§3.8.6) and Constraints on Particle Schema Components (§3.9.6).

3.10.4 Wildcard Validation Rules

Validation Rule: Item Valid (Wildcard)
For an element or attribute information item to be locally ·valid· with respect to a wildcard constraint its [namespace name] must be ·valid· with respect to the wildcard constraint, as defined in Wildcard allows Namespace Name (§3.10.4).

When this constraint applies the appropriate **case** among the following must be true:

1 **If** {process contents} is *lax*, **then** the item has no ·context-determined declaration· with respect to Assessment Outcome (Element) (§3.3.5), Schema-Validity Assessment (Element) (§3.3.4) and Schema-Validity Assessment (Attribute) (§3.2.4).
2 **If** {process contents} is *strict*, **then** the item's ·context-determined declaration· is *mustFind*.
3 **If** {process contents} is *skip*, **then** the item's ·context-determined declaration· is *skip*.

Validation Rule: Wildcard allows Namespace Name
For a value which is either a namespace name or ·absent· to be ·valid· with respect to a wildcard constraint (the value of a {namespace constraint}) **one** of the following must be true:
1 The constraint must be *any*.
2 **All** of the following must be true:
  2.1 The constraint is a pair of *not* and a namespace name or ·absent· ([Definition:] call this the **namespace test**).
  2.2 The value must not be identical to the ·namespace test·.
  2.3 The value must not be ·absent·.
3 The constraint is a set, and the value is identical to one of the members of the set.

3.10.5 Wildcard Information Set Contributions

None as such.

3.10.6 Constraints on Wildcard Schema Components

All wildcards (see Wildcards (§3.10)) must satisfy the following constraint.

Schema Component Constraint: Wildcard Properties Correct
The values of the properties of a wildcard must be as described in the property tableau in The Wildcard Schema Component (§3.10.1), modulo the impact of Missing Sub-components (§5.3).

The following constraints define a relation appealed to elsewhere in this specification.

Schema Component Constraint: Wildcard Subset
For a namespace constraint (call it **sub**) to be an intensional subset of another namespace constraint (call it **super**) **one** of the following must be true:
1 **super** must be *any*.
2 **All** of the following must be true:
  2.1 **sub** must be a pair of *not* and a value (a namespace name or ·absent·).
  2.2 **super** must be a pair of *not* and the same value.
3 **All** of the following must be true:
  3.1 **sub** must be a set whose members are either namespace names or ·absent·.
  3.2 **One** of the following must be true:
    3.2.1 **super** must be the same set or a superset thereof.
    3.2.2 **super** must be a pair of *not* and a value (a namespace name or ·absent·) and neither that value nor ·absent· must be in **sub**'s set.

Schema Component Constraint: Attribute Wildcard Union
For a wildcard's {namespace constraint} value to be the intensional union of two other such values (call them **O1** and **O2**): the appropriate **case** among the following must be true:
1 **If O1** and **O2** are the same value, **then** that value must be the value.
2 **If** either **O1** or **O2** is *any*, **then** *any* must be the value.
3 **If** both **O1** and **O2** are sets of (namespace names or ·absent·), **then** the union of those sets must be the value.
4 **If** the two are negations of different values (namespace names or ·absent·), **then** a pair of *not* and ·absent· must be the value.
5 **If** either **O1** or **O2** is a pair of *not* and a namespace name and the other is a set of (namespace names or ·absent·) (call this set **S**), **then** The appropriate **case** among the following must be true:
  5.1 **If** the set **S** includes both the negated namespace name and ·absent·, **then** *any* must be the value.
  5.2 **If** the set **S** includes the negated namespace name but not ·absent·, **then** a pair of *not* and ·absent· must be the value.
  5.3 **If** the set **S** includes ·absent· but not the negated namespace name, **then** the union is not expressible.
  5.4 **If** the set **S** does not include either the negated namespace name or ·absent·, **then** whichever of **O1** or **O2** is a pair of *not* and a namespace name must be the value.
6 **If** either **O1** or **O2** is a pair of *not* and ·absent· and the other is a set of (namespace names or ·absent·) (again, call this set **S**), **then** The appropriate **case** among the following must be true:
  6.1 **If** the set **S** includes ·absent·, **then** *any* must be the value.
  6.2 **If** the set **S** does not include ·absent·, **then** a pair of *not* and ·absent· must be the value.
In the case where there are more than two values, the intensional union is determined by identifying the intensional union of two of the values as above, then the intensional union of that value with the third (providing the first union was expressible), and so on as required.

Schema Component Constraint: Attribute Wildcard Intersection
For a wildcard's {namespace constraint} value to be the intensional intersection of two other such values (call them **O1** and **O2**): the appropriate **case** among the following must be true:
1 **If O1** and **O2** are the same value, **then** that value must be the value.
2 **If** either **O1** or **O2** is *any*, **then** the other must be the value.
3 **If** either **O1** or **O2** is a pair of *not* and a value (a namespace name or ·absent·) and the other is a set of (namespace names or ·absent·), **then** that set, minus the negated value if it was in the set, minus ·absent· if it was in the set, must be the value.
4 **If** both **O1** and **O2** are sets of (namespace names or ·absent·), **then** the intersection of those sets must be the value.
5 **If** the two are negations of different namespace names, **then** the intersection is not expressible.
6 **If** the one is a negation of a namespace name and the other is a negation of ·absent·, **then** the one which is the negation of a namespace name must be the value.

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.

3.11 Identity-constraint Definitions

Identity-constraint definition components provide for uniqueness and reference constraints with respect to the contents of multiple elements and attributes.

Example

```
<xs:key name="fullName">
 <xs:selector xpath=".//person"/>
 <xs:field xpath="forename"/>
 <xs:field xpath="surname"/>
</xs:key>

<xs:keyref name="personRef" refer="fullName">
 <xs:selector xpath=".//personPointer"/>
 <xs:field xpath="@first"/>
 <xs:field xpath="@last"/>
</xs:keyref>

<xs:unique name="nearlyID">
 <xs:selector xpath=".//*"/>
 <xs:field xpath="@id"/>
</xs:unique>
```

XML representations for the three kinds of identity-constraint definitions.

3.11.1 The Identity-constraint Definition Schema Component

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 name, as defined in [XML-Namespaces].
{identity-constraint category}
One of *key*, *keyref* or *unique*.
{selector}
A restricted XPath ([XPath]) expression.
{fields}
A non-empty list of restricted XPath ([XPath]) expressions.
{referenced key}
Required if {identity-constraint category} is *keyref*, forbidden otherwise. An identity-constraint definition with {identity-constraint category} equal to *key* or *unique*.
{annotation}
Optional. A set of annotations.

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 (§4.2.3) 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 along side the specification of types for the attributes and elements involved, i.e. something declared as of type integer may also serve as a key. 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 apply to the *value* of the fields selected, so that for example `3.0` and `3` would be conflicting keys if they were both number, but non-conflicting if they were both strings, or one was a string and one a number. Values of differing type can only be equal if one type is derived from the other, and the value is in the value space of both.

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

- Functioning as a part of an identity-constraint is in addition to, not instead of, having a type;
- Not just attribute values, but also element content and combinations of values and content can be declared to be unique;
- Identity-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-nillable;
- The comparison between *keyref* {fields} and *key* or *unique* {fields} is by value equality, not by string equality.

{selector} specifies a restricted XPath ([XPath]) expression relative to instances of the element being declared. This must identify a node set of subordinate elements (i.e. 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) whose content or value, which must be of 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.

In order to reduce the burden on implementers, in particular implementers of streaming processors, only restricted subsets of XPath expressions are allowed in {selector} and {fields}. The details are given in Constraints on Identity-constraint Definition Schema Components (§3.11.6).

Note: Provision for multi-field keys etc. goes beyond what is supported by `xsl:key`.

See Annotations (§3.13) for information on the role of the {annotation} property.

3.11.2 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 = a subset of XPath expression, see below
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</selector>
```

```
<field
  id = ID
  xpath = a subset of XPath expression, see below
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</field>
```

Identity-constraint Definition Schema Component

Property	Representation
{name}	The ·actual value· of the `name` [attribute]
{target namespace}	The ·actual 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}	A restricted XPath expression corresponding to the ·actual value· of the `xpath` [attribute] of the <selector> element information item among the [children]
{fields}	A sequence of XPath expressions, corresponding to the ·actual value·s of the `xpath` [attribute]s 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 ·actual value· of the `refer` [attribute], otherwise ·absent·.
{annotation}	The annotations corresponding to the <annotation> element information item in the [children], if present, and in the <selector> and <field> [children], if present, otherwise ·absent·.

Example

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

```
<xs:element name="state">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="code" type="twoLetterCode"/>
   <xs:element ref="vehicle" maxOccurs="unbounded"/>
   <xs:element ref="person" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>

 <xs:key name="reg"> <!-- vehicles are keyed by their plate within states -->
  <xs:selector xpath=".//vehicle"/>
  <xs:field xpath="@plateNumber"/>
 </xs:key>
</xs:element>

<xs:element name="root">
 <xs:complexType>
  <xs:sequence>
   . . .
   <xs:element ref="state" maxOccurs="unbounded"/>
   . . .
  </xs:sequence>
 </xs:complexType>

 <xs:key name="state"> <!-- states are keyed by their code -->
  <xs:selector xpath=".//state"/>
  <xs:field xpath="code"/>
 </xs:key>

 <xs:keyref name="vehicleState" refer="state">
  <!-- every vehicle refers to its state -->
  <xs:selector xpath=".//vehicle"/>
  <xs:field xpath="@state"/>
 </xs:keyref>

 <xs:key name="regKey"> <!-- vehicles are keyed by a pair of state and plate -->
  <xs:selector xpath=".//vehicle"/>
  <xs:field xpath="@state"/>
  <xs:field xpath="@plateNumber"/>
 </xs:key>

 <xs:keyref name="carRef" refer="regKey"> <!-- people's cars are a reference -->
  <xs:selector xpath=".//car"/>
  <xs:field xpath="@regState"/>
  <xs:field xpath="@regPlate"/>
 </xs:keyref>

</xs:element>

<xs:element name="person">
 <xs:complexType>
  <xs:sequence>
   . . .
   <xs:element name="car">
    <xs:complexType>
     <xs:attribute name="regState" type="twoLetterCode"/>
     <xs:attribute name="regPlate" type="xs:integer"/>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

A `state` element is defined, which contains a `code` child and some `vehicle` and `person` children. A `vehicle` in turn has a `plateNumber` attribute, which is an integer, and a `state` attribute. State's `code`s are a key for them within the document. Vehicle's `plateNumber`s are a key for them within states, and `state` and `plateNumber` is asserted to be a *key* for `vehicle` within the document as a whole. Furthermore, a `person` element has an empty `car` child, with `regState` and `regPlate` attributes, which are then asserted together to refer to `vehicle`s via the `carRef` constraint. The requirement that a `vehicle`'s `state` match its containing `state`'s `code` is not expressed here.

3.11.3 Constraints on XML Representations of Identity-constraint Definitions

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 corresponding identity-constraint definition must satisfy the conditions set out in Constraints on Identity-constraint Definition Schema Components (§3.11.6).

3.11.4 Identity-constraint Definition Validation Rules

Validation Rule: Identity-constraint Satisfied

For an element information item to be locally ·valid· with respect to an identity-constraint **all** of the following must be true:

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**.

2 Each node in the ·target node set· is either the context node oran element node among its descendants.

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, which must have a simple type. [Definition:] Call the sequence of the type-determined values (as defined in [XML Schemas: Datatypes]) of the [schema normalized value] of the element and/or attribute information items in those node-sets in order the **key-sequence** of the node.

4 [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 with a simple type the **qualified node set**. The appropriate **case** among the following must be true:

4.1 **If** the {identity-constraint category} is *unique*, **then** no two members of the ·qualified node set· have ·key-sequences· whose members are pairwise equal, as defined by Equal in [XML Schemas: Datatypes].
4.2 **If** the {identity-constraint category} is *key*, **then all** of the following must be true:
4.2.1 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*.
4.2.2 No two members of the ·qualified node set· have ·key-sequences· whose members are pairwise equal, as defined by Equal in [XML Schemas: Datatypes].
4.2.3 No element member of the ·key-sequence· of any member of the ·qualified node set· was assessed as ·valid· by reference to an element declaration whose {nillable} is *true*.
4.3 **If** the {identity-constraint category} is *keyref*, **then** for each member of the ·qualified node set· (call this the **keyref member**), there must be a ·node table· associated with the {referenced key} in the [identity-constraint table] of the element information item (see Identity-constraint Table (§3.11.5)), which must be understood as logically prior to this clause of this constraint, below) and there must be an entry in that table whose ·key-sequence· is equal to the **keyref member's** ·key-sequence· member for member, as defined by Equal in [XML Schemas: Datatypes].
Note: The use of [schema normalized value] in the definition of ·key sequence· above means that *default* or *fixed* value constraints may play a part in ·key sequence·s.
Note: Because the validation of *keyref* (see clause 4.3) depends on finding appropriate entries in a element information item's ·node table·, and ·node tables· are assembled strictly recursively from the node tables of descendants, only element information items within the sub-tree rooted at the element information item being ·validated· can be referenced successfully.
Note: Although this specification defines a ·post-schema-validation infoset· contribution which would enable schema-aware processors to implement clause 4.2.3 above (Element Declaration (§3.3.5)), processors are not required to provide it. This clause can be read as if in the absence of this infoset contribution, the value of the relevant {nillable} property must be available.

3.11.5 Identity-constraint Definition Information Set Contributions

Schema Information Set Contribution: Identity-constraint Table
[Definition:] An **eligible identity-constraint** of an element information item is one such that clause 4.1 or clause 4.2 of Identity-constraint Satisfied (§3.11.4) is satisfied with respect to that item and that constraint, or such that any of the element information item [children] of that item have an [identity-constraint table] property whose value has 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 identity-constraints·, in the ·post-schema-validation infoset· that element information item has a property as follows:

PSVI Contributions for element information items

[identity-constraint table]
one **Identity-constraint Binding** information item for each ·eligible identity-constraint·, with properties as follows:

PSVI Contributions for Identity-constraint Binding information items

[definition]
The ·eligible identity-constraint·.
[node table]
A ·node table· with one entry for every ·key-sequence· (call it **k**) and node (call it **n**) such that **one** of the following must be true:
1 There is an entry in one of the ·node tables· associated with the [definition] in an **Identity-constraint Binding** information item in at least one of the [identity-constraint table]s of the element information item [children] of the element information item whose ·key-sequence· is **k** and whose node is **n**;
2 **n** appears with ·key-sequence· **k** in the ·qualified node set· for the [definition].
provided no two entries have the same ·key-sequence· but distinct nodes. Potential conflicts are resolved by not including any conflicting entries which would have owed their inclusion to clause 1 above. Note that if all the conflicting entries arose under clause 1 above, this means no entry at all will appear for the offending ·key-sequence·.

Note: The complexity of the above arises from the fact that *keyref* identity-constraints may be defined on domains distinct from the embedded domain of the identity-constraint they reference, or the domains may be the same but self-embedding at some depth. In either case the ·node table· for the referenced identity-constraint needs to propagate upwards, with conflict resolution.

The **Identity-constraint Binding** information item, unlike others in this specification, is essentially an internal bookkeeping mechanism. It is introduced to support the definition of Identity-constraint Satisfied (§3.11.4) above. Accordingly, conformant processors may, but are *not* required to, expose them via [identity-constraint table] properties 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 infoset items existed.

3.11.6 Constraints on Identity-constraint Definition Schema Components

All identity-constraint definitions (see Identity-constraint Definitions (§3.11)) must satisfy the following constraint.

Schema Component Constraint: Identity-constraint Definition Properties Correct
All of the following must be true:
1 The values of the properties of an identity-constraint definition must be as described in the property tableau in The Identity-constraint Definition Schema Component (§3.11.1), modulo the impact of Missing Sub-components (§5.3).
2 If the {identity-constraint category} is *keyref*, the cardinality of the {fields} must equal that of the {fields} of the {referenced key}.

Schema Component Constraint: Selector Value OK
All of the following must be true:
1 The {selector} must be a valid XPath expression, as defined in [XPath].
2 **One** of the following must be true:
2.1 It must conform to the following extended BNF:

Selector XPath expressions

```
[1]  Selector  ::=  Path ( '|' Path )*
[2]  Path      ::=  ('.//')? Step ( '/' Step )*
[3]  Step      ::=  '.' | NameTest
[4]  NameTest  ::=  QName | '*' | NCName ':' '*'
```

2.2 It must be an XPath expression involving the `child` axis whose abbreviated form is as given above.

For readability, whitespace may be used in selector XPath expressions even though not explicitly allowed by the grammar: whitespace may be freely added within patterns before or after any token.

Lexical productions

```
[5]  token       ::=  '.' | '/' | '//' | '|' | '@' | NameTest
[6]  whitespace  ::=  S
```

When tokenizing, the longest possible token is always returned.

Schema Component Constraint: Fields Value OK

All of the following must be true:

1 Each member of the {fields} must be a valid XPath expression, as defined in [XPath].

2 **One** of the following must be true:

2.1 It must conform to the extended BNF given above for Selector, with the following modification:

Path in Field XPath expressions

```
[7]  Path  ::=  ('.//')? ( Step '/' )* ( Step | '@' NameTest )
```

This production differs from the one above in allowing the final step to match an attribute node.

2.2 It must be an XPath expression involving the `child` and/or `attribute` axes whose abbreviated form is as given above.

For readability, whitespace may be used in field XPath expressions even though not explicitly allowed by the grammar: whitespace may be freely added within patterns before or after any token.

When tokenizing, the longest possible token is always returned.

3.12 Notation Declarations

Notation declarations reconstruct XML 1.0 NOTATION declarations.

Example

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

The XML representation of a notation declaration.

3.12.1 The Notation Declaration Schema Component

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 name, 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 1.0 (Second Edition)].

{annotation}
Optional. An annotation.

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

See Annotations (§3.13) for information on the role of the {annotation} property.

3.12.2 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 = token
  system = anyURI
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</notation>
```

Notation Declaration Schema Component

Property	Representation
{name}	The ·actual value· of the `name` [attribute]
{target namespace}	The ·actual value· of the `targetNamespace` [attribute] of the parent `schema` element information item.
{system identifier}	The ·actual value· of the `system` [attribute], if present, otherwise ·absent·.
{public identifier}	The ·actual 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>
  <xs:simpleContent>
   <xs:extension base="xs:hexBinary">
    <xs:attribute name="pictype">
     <xs:simpleType>
      <xs:restriction base="xs:NOTATION">
       <xs:enumeration value="jpeg"/>
       <xs:enumeration value="png"/>
       . . .
      </xs:restriction>
     </xs:simpleType>
    </xs:attribute>
   </xs:extension>
  </xs:simpleContent>
 </xs:complexType>
</xs:element>

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

3.12.3 Constraints on XML Representations of Notation Declarations

Schema Representation Constraint: Notation Definition Representation OK
In addition to the conditions imposed on <notation> element information items by the schema for schemas, the corresponding notation definition must satisfy the conditions set out in Constraints on Notation Declaration Schema Components (§3.12.6).

3.12.4 Notation Declaration Validation Rules

None as such.

3.12.5 Notation Declaration Information Set Contributions

Schema Information Set Contribution: Validated with Notation
Whenever an attribute information item is ·valid· with respect to a NOTATION, in the ·post-schema-validation infoset· its parent element information item either has a property as follows:

PSVI Contributions for element information items

[notation]
An ·item isomorphic· to the notation declaration whose {name} and {target namespace} match the ·local name· and ·namespace name· (as defined in QName Interpretation (§3.15.3)) of the attribute item's ·actual value·

or has a pair of properties as follows:

PSVI Contributions for element information items

[notation system]
The value of the {system identifier} of that notation declaration.
[notation public]
The value of the {public identifier} of that notation declaration.

Note: For compatibility, only one such attribute should appear on any given element. If more than one such attribute *does* appear, which one supplies the infoset property or properties above is not defined.

3.12.6 Constraints on Notation Declaration Schema Components

All notation declarations (see Notation Declarations (§3.12)) must satisfy the following constraint.

Schema Component Constraint: Notation Declaration Correct

The values of the properties of a notation declaration must be as described in the property tableau in The Notation Declaration Schema Component (§3.12.1), modulo the impact of Missing Sub-components (§5.3).

3.13 Annotations

Annotations provide for human- and machine-targeted annotations of schema components.

Example

```
<xs:simpleType fn:note="special">
  <xs:annotation>
   <xs:documentation>A type for experts only</xs:documentation>
   <xs:appinfo>
    <fn:specialHandling>checkForPrimes</fn:specialHandling>
   </xs:appinfo>
  </xs:annotation>
```

XML representations of three kinds of annotation.

3.13.1 The Annotation Schema Component

The annotation schema component has the following properties:

Schema Component: Annotation

{application information}
A sequence of element information items.
{user information}
A sequence of element information items.
{attributes}
A sequence of attribute 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, as the value of the `source` attribute of the respective element information items. ·Validation· does *not* involve dereferencing these URIs, when present. In the case of {user information}, indication should be given as to the identity of the (human) language used in the contents, using the `xml:lang` attribute.

{attributes} ensures that when schema authors take advantage of the provision for adding attributes from namespaces other than the XML Schema namespace to schema documents, they are available within the components corresponding to the element items where such attributes appear.

Annotations do not participate in ·validation· as such. Provided an annotation itself satisfies all relevant ·Schema Component Constraints· it *cannot* affect the ·validation· of element information items.

3.13.2 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
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (appinfo | documentation)*
</annotation>
```

```
<appinfo
   source = anyURI
   {any attributes with non-schema namespace . . .}>
   Content: ({any})*
</appinfo>
```

```
<documentation
   source = anyURI
   xml:lang = language
   {any attributes with non-schema namespace . . .}>
   Content: ({any})*
</documentation>
```

Annotation Schema Component

Property	Representation

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.
{attributes}	A sequence of attribute information items, namely those allowed by the attribute wildcard in the type definition for the <annotation> item itself or for the enclosing items which correspond to the component within which the annotation component is located.

The annotation component corresponding to the <annotation> element in the example above will have one element item in each of its {user information} and {application information} and one attribute item in its {attributes}.

3.13.3 Constraints on XML Representations of Annotations

Schema Representation Constraint: Annotation Definition Representation OK
In addition to the conditions imposed on <annotation> element information items by the schema for schemas, the corresponding annotation must satisfy the conditions set out in Constraints on Annotation Schema Components (§3.13.6).

3.13.4 Annotation Validation Rules

None as such.

3.13.5 Annotation Information Set Contributions

None as such: the addition of annotations to the ·post-schema-validation infoset· is covered by the ·post-schema-validation infoset· contributions of the enclosing components.

3.13.6 Constraints on Annotation Schema Components

All annotations (see Annotations (§3.13)) must satisfy the following constraint.

Schema Component Constraint: Annotation Correct
The values of the properties of an annotation must be as described in the property tableau in The Annotation Schema Component (§3.13.1), modulo the impact of Missing Sub-components (§5.3).

3.14 Simple Type Definitions

Note: This section consists of a combination of non-normative versions of normative material from [XML Schemas: Datatypes], for local cross-reference purposes, and normative material relating to the interface between schema components defined in this specification and the simple type definition component.

Simple type definitions provide for constraining character information item [children] of element and attribute information items.

Example

```
<xs:simpleType name="fahrenheitWaterTemp">
 <xs:restriction base="xs:number">
  <xs:fractionDigits value="2"/>
  <xs:minExclusive value="0.00"/>
  <xs:maxExclusive value="100.00"/>
 </xs:restriction>
</xs:simpleType>
```

The XML representation of a simple type definition.

3.14.1 (non-normative) The Simple Type Definition Schema Component

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 name, as defined in [XML-Namespaces].
{base type definition}
A simple type definition, which may be the ·simple ur-type definition·.
{facets}
A set of constraining facets.
{fundamental facets}
A set of fundamental facets.
{final}

A subset of {*extension*, *list*, *restriction*, *union*}.

{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.

list

{item type definition}

A simple type definition.

union

{member type definitions}

A non-empty sequence of simple type definitions.

{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 schema components (specifically in <element> and <attribute>). See References to schema components across namespaces (§4.2.3) 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 Declarations (§3.3) and Attribute Declarations (§3.2).

A simple type definition with an empty specification for {final} can be used as the {base type definition} for other types derived by either of extension or restriction, or as the {item type definition} in the definition of a list, or in the {member type definitions} of a union; the explicit values *extension*, *restriction*, *list* and *union* prevent further derivations by extension (to yield a complex type) and restriction (to yield a simple type) and use in constructing lists and unions respectively.

{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 which is not itself derived by restriction from a list or union respectively. Each *atomic* type is ultimately a restriction of exactly one such built-in primitive datatype, which is its {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. what is ·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 item type 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.

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

See Annotations (§3.13) for information on the role of the {annotation} property.

3.14.2 (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
  final = (#all | List of (list | union | restriction))
  id = ID
  name = NCName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (restriction | list | union))
</simpleType>
```

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

```
<list
  id = ID
  itemType = QName
  {any attributes with non-schema namespace . . .}>
```

```
  Content: (annotation?, simpleType?)
</list>
```

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

Simple Type Definition Schema Component

Property	Representation
{name}	The ·actual value· of the `name` [attribute] if present, otherwise ·absent·.
{target namespace}	The ·actual value· of the `targetNamespace` [attribute] of the <schema> ancestor element information item if present, otherwise ·absent·.
{base type definition}	The appropriate **case** among the following: 1 **If** the <restriction> alternative is chosen, **then** the type definition ·resolved· to by the ·actual value· of the `base` [attribute] of <restriction>, if present, otherwise the type definition corresponding to the <simpleType> among the [children] of <restriction>. 2 **If** the <list> or <union> alternative is chosen, **then** the ·simple ur-type definition·.
{final}	As for the {prohibited substitutions} property of complex type definitions, but using the `final` and `finalDefault` [attributes] in place of the `block` and `blockDefault` [attributes] and with the relevant set being {*extension*, *restriction*, *list*, *union*}.
{variety}	If the <list> alternative is chosen, then *list*, otherwise if the <union> alternative is chosen, then *union*, otherwise (the <restriction> alternative is chosen), 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 ·constituting a restriction· of the {facets} of the {base type definition} with respect to a set of facet components corresponding to the appropriate element information items among the [children] of <restriction> (i.e. those which specify facets, if any), as defined in Simple Type Restriction (Facets) (§3.14.6).

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

List Simple Type Definition Schema Component

Property	Representation
{item type definition}	The appropriate **case** among the following: 1 **If** the <list> alternative is chosen, **then** the type definition ·resolved· to by the ·actual value· of the `itemType` [attribute] of <list>, if present, otherwise the type definition corresponding to the <simpleType> among the [children] of <list>. 2 **If** the <restriction> option is chosen, **then** the {item type definition} of the {base type definition}.
{facets}	If the <restriction> alternative is chosen, a set of facet components ·constituting a restriction· of the {facets} of the {base type definition} with respect to a set of facet components corresponding to the appropriate element information items among the [children] of <restriction> (i.e. those which specify facets, if any), as defined in Simple Type Restriction (Facets) (§3.14.6), 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}	The appropriate **case** among the following: 1 **If** the <union> alternative is chosen, **then** [Definition:] define the **explicit members** as the type definitions ·resolved· to by the items in the ·actual value· of the `memberTypes` [attribute], if any, followed by the type definitions corresponding to the <simpleType>s among the [children] of <union>, if any. The actual value is then formed by replacing any union type definition in the ·explicit members· with the members of their {member type definitions}, in order. 2 **If** the <restriction> option is chosen, **then** the {member type definitions} of the {base type definition}.
{facets}	If the <restriction> alternative is chosen, a set of facet components ·constituting a restriction· of the {facets} of the {base type definition} with respect to a set of facet components corresponding to the appropriate element information items among the [children] of <restriction> (i.e. those which specify facets, if any), as defined in Simple Type Restriction (Facets) (§3.14.6), otherwise the empty set.

3.14.3 Constraints on XML Representations of Simple Type Definitions

Schema Representation Constraint: Simple Type Definition Representation OK

In addition to the conditions imposed on <simpleType> element information items by the schema for schemas, **all** of the following must be true:

1 The corresponding simple type definition, if any, must satisfy the conditions set out in Constraints on Simple Type Definition Schema Components (§3.14.6).
2 If the <restriction> alternative is chosen, either it must have a `base` [attribute] or a <simpleType> among its [children], but not both.
3 If the <list> alternative is chosen, either it must have an `itemType` [attribute] or a <simpleType> among its [children], but not both.
4 Circular union type definition is disallowed. That is, if the <union> alternative is chosen, there must not be any entries in the `memberTypes` [attribute] at any depth which resolve to the component corresponding to the <simpleType>.

3.14.4 Simple Type Definition Validation Rules

Validation Rule: String Valid

For a string to be locally ·valid· with respect to a simple type definition **all** of the following must be true:

1 It is schema-valid with respect to that definition as defined by Datatype Valid in [XML Schemas: Datatypes].

2 The appropriate **case** among the following must be true:

2.1 **If** The definition is ENTITY or is validly derived from ENTITY given the empty set, as defined in Type Derivation OK (Simple) (§3.14.6), **then** the string must be a ·declared entity name·.

2.2 **If** The definition is ENTITIES or is validly derived from ENTITIES given the empty set, as defined in Type Derivation OK (Simple) (§3.14.6), **then** every whitespace-delimited substring of the string must be a ·declared entity name·.

2.3 **otherwise** no further condition applies.

[Definition:] A string is a **declared entity name** if it is equal to the [name] of some unparsed entity information item in the value of the [unparsedEntities] property of the document information item at the root of the infoset containing the element or attribute information item whose ·normalized value· the string is.

3.14.5 Simple Type Definition Information Set Contributions

None as such.

3.14.6 Constraints on Simple Type Definition Schema Components

All simple type definitions other than the ·simple ur-type definition· and the built-in primitive datatype definitions (see Simple Type Definitions (§3.14)) must satisfy both the following constraints.

Schema Component Constraint: Simple Type Definition Properties Correct

All of the following must be true:

1 The values of the properties of a simple type definition must be as described in the property tableau in Datatype definition, modulo the impact of Missing Sub-components (§5.3).

2 All simple type definitions must be derived ultimately from the ·simple ur-type definition (so· circular definitions are disallowed). That is, it must be possible to reach a built-in primitive datatype or the ·simple ur-type definition· by repeatedly following the {base type definition}.

3 The {final} of the {base type definition} must not contain *restriction*.

Schema Component Constraint: Derivation Valid (Restriction, Simple)

The appropriate **case** among the following must be true:

1 **If** the {variety} is *atomic*, **then all** of the following must be true:

1.1 The {base type definition} must be an atomic simple type definition or a built-in primitive datatype.

1.2 The {final} of the {base type definition} must not contain *restriction*.

1.3 For each facet in the {facets} (call this **DF**) **all** of the following must be true:

1.3.1 **DF** must be an allowed constraining facet for the {primitive type definition}, as specified in the appropriate subsection of 3.2 Primitive datatypes.

1.3.2 If there is a facet of the same kind in the {facets} of the {base type definition} (call this **BF**),then the **DF**'s {value} must be a valid restriction of **BF**'s {value} as defined in [XML Schemas: Datatypes].

2 **If** the {variety} is *list*, **then all** of the following must be true:

2.1 The {item type definition} must have a {variety} of *atomic* or *union* (in which case all the {member type definitions} must be *atomic*).

2.2

2.3 The appropriate **case** among the following must be true:

2.3.1 **If** the {base type definition} is the ·simple ur-type definition· , **then all** of the following must be true:

2.3.1.1 The {final} of the {item type definition} must not contain *list*.

2.3.1.2 The {facets} must only contain the *whiteSpace* facet component.

2.3.2 **otherwise all** of the following must be true:

2.3.2.1 The {base type definition} must have a {variety} of *list*.

2.3.2.2 The {final} of the {base type definition} must not contain *restriction*.

2.3.2.3 The {item type definition} must be validly derived from the {base type definition}'s {item type definition} given the empty set, as defined in Type Derivation OK (Simple) (§3.14.6).

2.3.2.4 Only *length*, *minLength*, *maxLength*, *whiteSpace*, *pattern* and *enumeration* facet components are allowed among the {facets}.

2.3.2.5 For each facet in the {facets} (call this **DF**), if there is a facet of the same kind in the {facets} of the {base type definition} (call this **BF**),then the **DF**'s {value} must be a valid restriction of **BF**'s {value} as defined in [XML Schemas: Datatypes].

The first case above will apply when a list is derived by specifying an item type, the second when derived by restriction from another list.

3 **If** the {variety} is *union*, **then all** of the following must be true:

3.1 The {member type definitions} must all have {variety} of *atomic* or *list*.

3.2

3.3 The appropriate **case** among the following must be true:

3.3.1 **If** the {base type definition} is the ·simple ur-type definition· , **then all** of the following must be true:

3.3.1.1 All of the {member type definitions} must have a {final} which does not contain *union*.

3.3.1.2 The {facets} must be empty.

3.3.2 **otherwise all** of the following must be true:

3.3.2.1 The {base type definition} must have a {variety} of *union*.

3.3.2.2 The {final} of the {base type definition} must not contain *restriction*.

3.3.2.3 The {member type definitions}, in order, must be validly derived from the corresponding type definitions in the {base type definition}'s {member type definitions} given the empty set, as defined in Type Derivation OK (Simple) (§3.14.6).

3.3.2.4 Only *pattern* and *enumeration* facet components are allowed among the {facets}.

3.3.2.5 For each facet in the {facets} (call this **DF**), if there is a facet of the same kind in the {facets} of the {base type definition} (call this **BF**),then the **DF**'s {value} must be a valid restriction of **BF**'s {value} as defined in [XML Schemas: Datatypes].

The first case above will apply when a union is derived by specifying one or more member types, the second when derived by restriction from another union.

[Definition:] If this constraint Derivation Valid (Restriction, Simple) (§3.14.6) holds of a simple type definition, it is a **valid restriction** of its ·base type definition·.

The following constraint defines relations appealed to elsewhere in this specification.

Schema Component Constraint: Type Derivation OK (Simple)

For a simple type definition (call it **D**, for derived) to be validly derived from a type definition (call this **B**, for base) given a subset of {*extension*, *restriction*, *list*, *union*} (of which only *restriction* is actually relevant) **one** of the following must be true:
1 They are the same type definition.
2 **All** of the following must be true:
  2.1 *restriction* is not in the subset, or in the {final} of its own {base type definition};
  2.2 **One** of the following must be true:
    2.2.1 **D**'s ·base type definition· is **B**.
    2.2.2 **D**'s ·base type definition· is not the ·ur-type definition· and is validly derived from **B** given the subset, as defined by this constraint.
    2.2.3 **D**'s {variety} is *list* or *union* and **B** is the ·simple ur-type definition·.
    2.2.4 **B**'s {variety} is *union* and **D** is validly derived from a type definition in **B**'s {member type definitions} given the subset, as defined by this constraint.
  Note: With respect to clause 1, see the Note on identity at the end of (§3.4.6) above.

Schema Component Constraint: Simple Type Restriction (Facets)
For a simple type definition (call it **R**) to restrict another simple type definition (call it **B**) with a set of facets (call this **S**) **all** of the following must be true:
1 The {variety} of **R** is the same as that of **B**.
2 If {variety} is *atomic*, the {primitive type definition} of **R** is the same as that of **B**.
3 The {facets} of **R** are the union of **S** and the {facets} of **B**, eliminating duplicates. To eliminate duplicates, when a facet of the same kind occurs in both **S** and the {facets} of **B**, the one in the {facets} of **B** is not included, with the exception of enumeration and pattern facets, for which multiple occurrences with distinct values are allowed.

Additional constraint(s) may apply depending on the kind of facet, see the appropriate sub-section of 4.3 Constraining Facets

[Definition:] If clause 3 above holds, the {facets} of **R constitute a restriction** of the {facets} of **B** with respect to **S**.

3.14.7 Built-in Simple Type Definition

There is a simple type definition nearly equivalent to the ·simple 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/2001/XMLSchema
{base type definition}	·the ur-type definition·
{final}	The empty set
{variety}	·absent·

The ·simple ur-type definition· is the root of the simple type definition hierarchy, and as such mediates between the other simple type definitions, which all eventually trace back to it via their {base type definition} properties, and the ·ur-type definition·, which is *its* {base type definition}. This is why the ·simple ur-type definition· is exempted from the first clause of Simple Type Definition Properties Correct (§3.14.6), which would otherwise bar it because of its derivation from a complex type definition and absence of {variety}.

Simple type definitions for all the built-in primitive datatypes, namely *string*, *boolean*, *float*, *double*, *number*, *dateTime*, *duration*, *time*, *date*, *gMonth*, *gMonthDay*, *gDay*, *gYear*, *gYearMonth*, *hexBinary*, *base64Binary*, *anyURI* (see the Primitive Datatypes section of [XML Schemas: Datatypes]) are present by definition in every schema. All are in the XML Schema {target namespace} (namespace name `http://www.w3.org/2001/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 Schema for Schemas (normative) (§A).

3.15 Schemas as a Whole

A schema consists of a set of schema components.

Example

```
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.example.com/example">
  . . .
</xs:schema>
```

The XML representation of the skeleton of a schema.

3.15.1 The Schema Itself

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 (top-level) attribute declarations.
{element declarations}
A set of named (top-level) 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.

3.15.2 XML Representations of Schemas

A schema is represented in XML by one or more ·schema documents·, that is, one or more <schema> element information items. 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 (§4.2.3).

XML Representation Summary: `schema` Element Information Item

```
<schema
  attributeFormDefault = (qualified | unqualified) : unqualified
  blockDefault = (#all | List of (extension | restriction | substitution))  : ''
  elementFormDefault = (qualified | unqualified) : unqualified
  finalDefault = (#all | List of (extension | restriction | list | union))  : ''
  id = ID
  targetNamespace = anyURI
  version = token
  xml:lang = language
  {any attributes with non-schema namespace . . .}>
  Content: ((include | import | redefine | annotation)*, (((simpleType | complexType | group | attributeGroup) | element
| attribute | notation), 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 (§4.2.1) and References to schema components across namespaces (§4.2.3).
{attribute declarations}	The (top-level) 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 (§4.2.1) and References to schema components across namespaces (§4.2.3).
{element declarations}	The (top-level) 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 (§4.2.1) and References to schema components across namespaces (§4.2.3).
{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 (§4.2.1) and References to schema components across namespaces (§4.2.3).
{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 (§4.2.1) and References to schema components across namespaces (§4.2.3).
{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 (§4.2.1) and References to schema components across namespaces (§4.2.3).
{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 above, 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.

The definition of the schema abstract data model in XML 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 not a legal namespace name, supplying an empty string for `targetNamespace` is incoherent, and is *not* the same as not specifying it at all. The appropriate form of schema document corresponding to a ·schema· 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, the specification applies the target namespace facility uniformly to all schema components, i.e. not only declarations but also definitions have a {target namespace}.

Although the example schema at the beginning of this section 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 of <schema> corresponds to a schema component, and all except <annotation> are named. The sections below present each such item in turn, setting out the components to which it may correspond.

3.15.2.1 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 (§4.2.3)) from an external schema (which may, but need not, correspond to an actual schema document). The form of all such references is a ·QName·.

[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 or references to them, this refers to the simple type QName as defined in [XML Schemas: Datatypes].

[Definition:] An **NCName** is a name with no colon, 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 NCName as defined in [XML Schemas: Datatypes].

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/2001/XMLSchema"
           xmlns:xhtml="http://www.w3.org/1999/xhtml"
           xmlns="http://www.example.com"
           targetNamespace="http://www.example.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 (§4.2.3) for a discussion of importing.

3.15.2.2 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, [XPointer] provides a mechanism which maps well onto the 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 top-level 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.

3.15.3 Constraints on XML Representations of Schemas

Schema Representation Constraint: QName Interpretation

Where the type of an attribute information item in a document involved in ·validation· is identified as ·QName·, its ·actual value· is composed of a [Definition:] **local name** and a [Definition:] **namespace name**. Its ·actual value· is determined based on its ·normalized value· and the containing element information item's [in-scope namespaces] following [XML-Namespaces]:

The appropriate **case** among the following must be true:

1 **If** its ·normalized value· is prefixed, **then all** of the following must be true:
  1.1 There must be a namespace in the [in-scope namespaces] whose [prefix] matches the prefix.
  1.2 its ·namespace name· is the [namespace name] of that namespace.
  1.3 Its ·local name· is the portion of its ·normalized value· after the colon (`':'`).
2 **otherwise** (its ·normalized value· is unprefixed) **all** of the following must be true:

2.1 its ·local name· is its ·normalized value·.
2.2 The appropriate **case** among the following must be true:
2.2.1 **If** there is a namespace in the [in-scope namespaces] whose [prefix] has no value, **then** its ·namespace name· is the [namespace name] of that namespace.
2.2.2 **otherwise** its ·namespace name· is ·absent·.

In the absence of the [in-scope namespaces] property in the infoset for the schema document in question, processors must reconstruct equivalent information as necessary, using the [namespace attributes] of the containing element information item and its ancestors.

[Definition:] Whenever the word **resolve** in any form is used in this chapter in connection with a ·QName· in a schema document, the following definition QName resolution (Schema Document) (§3.15.3) should be understood:

Schema Representation Constraint: QName resolution (Schema Document)
For a ·QName· to resolve to a schema component of a specified kind **all** of the following must be true:
1 That component is a member of the value of the appropriate property of the schema which corresponds to the schema document within which the ·QName· appears, that is the appropriate **case** among the following must be true:
1.1 **If** the kind specified is simple or complex type definition, **then** the property is the {type definitions}.
1.2 **If** the kind specified is attribute declaration, **then** the property is the {attribute declarations}.
1.3 **If** the kind specified is element declaration, **then** the property is the {element declarations}.
1.4 **If** the kind specified is attribute group, **then** the property is the {attribute group definitions}.
1.5 **If** the kind specified is model group, **then** the property is the {model group definitions}.
1.6 **If** the kind specified is notation declaration, **then** the property is the {notation declarations}.
2 The component's {name} matches the ·local name· of the ·QName·;
3 The component's {target namespace} is identical to the ·namespace name· of the ·QName·;
4 The appropriate **case** among the following must be true:
4.1 **If** the ·namespace name· of the ·QName· is ·absent·, **then one** of the following must be true:
4.1.1 The <schema> element information item of the schema document containing the ·QName· has no `targetNamespace` [attribute].
4.1.2 The <schema> element information item of the that schema document contains an <import> element information item with no `namespace` [attribute].
4.2 **otherwise** the ·namespace name· of the ·QName· is the same as **one** of the following:
4.2.1 The ·actual value· of the `targetNamespace` [attribute] of the <schema> element information item of the schema document containing the ·QName·.
4.2.2 The ·actual value· of the `namespace` [attribute] of some <import> element information item contained in the <schema> element information item of that schema document.
.

3.15.4 Validation Rules for Schemas as a Whole

As the discussion above at Schema Component Details (§3) makes clear, at the level of schema components and ·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 Rule: QName resolution (Instance)
A pair of a local name and a namespace name (or ·absent·) resolve to a schema component of a specified kind in the context of ·validation· by appeal to the appropriate property of the schema being used for the ·assessment·. Each such property indexes components by name. The property to use is determined by the kind of component specified, that is, the appropriate **case** among the following must be true:
1 **If** the kind specified is simple or complex type definition, **then** the property is the {type definitions}.
2 **If** the kind specified is attribute declaration, **then** the property is the {attribute declarations}.
3 **If** the kind specified is element declaration, **then** the property is the {element declarations}.
4 **If** the kind specified is attribute group, **then** the property is the {attribute group definitions}.
5 **If** the kind specified is model group, **then** the property is the {model group definitions}.
6 **If** the kind specified is notation declaration, **then** the property is the {notation declarations}.
The component resolved to is the entry in the table whose {name} matches the local name of the pair and whose {target namespace} is identical to the namespace name of the pair.

3.15.5 Schema Information Set Contributions

Schema Information Set Contribution: Schema Information
Schema components provide a wealth of information about the basis of ·assessment·, which may well be of relevance to subsequent processing. Reflecting component structure into a form suitable for inclusion in the ·post-schema-validation infoset· is the way this specification provides for making this information available.

Accordingly, [Definition:] by an **item isomorphic** to a component is meant an information item whose type is equivalent to the component's, 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.

Processors must add a property in the ·post-schema-validation infoset· to the element information item at which ·assessment· began, as follows:

PSVI Contributions for element information items

`[schema information]`
A set of **namespace schema information** information items, one for each namespace name which appears as the {target namespace} of any schema component in the schema used for that assessment, and one for ·absent· if any schema component in the schema had no {target namespace}. Each **namespace schema information** information item has the following properties and values:

PSVI Contributions for namespace schema information information items

`[schema namespace]`
A namespace name or ·absent·.
`[schema components]`

A (possibly empty) set of schema component information items, each one an ·item 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) set of **schema document** information items, with properties and values as follows, for each schema document which contributed components to the schema, and whose `targetNamespace` matches the sibling [schema namespace] property above (or whose `targetNamespace` was ·absent· but that contributed components to that namespace by being <include>d by a schema document with that `targetNamespace` as per Assembling a schema for a single target namespace from multiple schema definition documents (§4.2.1)):

PSVI Contributions for schema document information items

[document location]
Either a URI reference, if available, otherwise ·absent·

[document]
A document information item, if available, otherwise ·absent·.

The {schema components} property is provided for processors which wish to provide a single access point to the components of the schema which was used during ·assessment·. Lightweight processors are free to leave it empty, but if it *is* provided, it must contain at a minimum all the top-level (i.e. named) components which actually figured in the ·assessment·, either directly or (because an anonymous component which figured is contained within) indirectly.

Schema Information Set Contribution: ID/IDREF Table

In the ·post-schema-validation infoset· a set of **ID/IDREF binding** information items is associated with the ·validation root· element information item:

PSVI Contributions for element information items

[ID/IDREF table]
A (possibly empty) set of **ID/IDREF binding** information items, as specified below.

[Definition:] Let the **eligible item set** be the set of consisting of every attribute or element information item for which **all** of the following are true

1 its [validation context] is the ·validation root·;
2 it was successfully ·validated· with respect to an attribute declaration as per Attribute Locally Valid (§3.2.4) or element declaration as per Element Locally Valid (Element) (§3.3.4) (as appropriate) whose attribute {type definition} or element {type definition} (respectively) is the built-in ID, IDREF or IDREFS simple type definition or a type derived from one of them.

Then there is one **ID/IDREF binding** in the [ID/IDREF table] for every distinct string which is**one** of the following:

1 the ·actual value· of a member of the ·eligible item set· whose type definition is or is derived from ID or IDREF;
2 one of the items in the ·actual value· of a member of the ·eligible item set· whose type definition is or is derived from IDREFS.

Each **ID/IDREF binding** has properties as follows:

PSVI Contributions for ID/IDREF binding information items

[id]
The string identified above.

[binding]
A set consisting of every element information item for which **all** of the following are true
1 its [validation context] is the ·validation root·;
2 it has an attribute information item in its [attributes] or an element information item in its [children] which was ·validated· by the built-in ID simple type definition or a type derived from it whose [schema normalized value] is the [id] of this **ID/IDREF binding**.

The net effect of the above is to have one entry for every string used as an id, whether by declaration or by reference, associated with those elements, if any, which actually purport to have that id. See Validation Root Valid (ID/IDREF) (§3.3.4) above for the validation rule which actually checks for errors here.

Note: The **ID/IDREF binding** information item, unlike most other aspects of this specification, is essentially an internal bookkeeping mechanism. It is introduced to support the definition of Validation Root Valid (ID/IDREF) (§3.3.4) above. Accordingly, conformant processors may, but are *not* required to, expose it in the ·post-schema-validation infoset·. In other words, the above constraint may be read as saying ·assessment· proceeds *as if* such an infoset item existed.

3.15.6 Constraints on Schemas as a Whole

All schemas (see Schemas as a Whole (§3.15)) must satisfy the following constraint.

Schema Component Constraint: Schema Properties Correct

All of the following must be true:

1 The values of the properties of a schema must be as described in the property tableau in The Schema Itself (§3.15.1), modulo the impact of Missing Sub-components (§5.3);
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}.

4 Schemas and Namespaces: Access and Composition

This chapter defines the mechanisms by which this specification establishes the necessary precondition for ·assessment·, namely access to one or more schemas. This chapter also sets out in detail the relationship between schemas and namespaces, as well as mechanisms for

modularization of schemas, including provision for incorporating definitions and declarations from one schema in another, possibly with modifications.

Conformance (§2.4) describes three levels of conformance for schema processors, and Schemas and Schema-validity Assessment (§5) provides a formal definition of ·assessment·. This section sets out in detail the 3-layer architecture implied by the three conformance levels. The layers are:

1. The ·assessment· 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 in the ·assessment· of an instance element information item. Layer 2 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.

4.1 Layer 1: Summary of the Schema-validity Assessment Core

The fundamental purpose of the ·assessment· core is to define ·assessment· 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.

·assessment· 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 ·assessment·. This is not a circular definition, but rather a *post facto* observation: no element information item can be fully assessed unless all the components required by any aspect of its (potentially recursive) ·assessment· 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 optimize or pre-compile) the entire schema prior to the start of an ·assessment· 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 an ·assessment· (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 of such dynamic acquisition that would cause the results of ·assessment· to differ from that which would have been obtained from the same schema components acquired in bulk.

> **Note:** the ·assessment· 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 ·assessment· core is concerned is to implement one or more of the options for ·assessment· given below in Assessing Schema-Validity (§5.2). Neither the choice of element information item for that ·assessment·, nor which of the means of initiating ·assessment· are used, is within the scope of this specification.

Although ·assessment· 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 an ·assessment· outcome that is the *same* as would be given if ·assessment· was undertaken again with the final, fully assembled schema.

4.2 Layer 2: Schema Documents, Namespaces and Composition

The sub-sections of Schema Component Details (§3) define 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 ·assessment· 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 ·assessment· 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), the precise mechanisms for resolving such references are expected to evolve over time. In support of such evolution, this specification observes the design principle that references from one schema document to a 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, it is documented with the layer 2 mechanisms of import and include, with which it is closely associated.

4.2.1 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 = anyURI
  {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 corresponding to its definition and declaration [children], but also all the components of all the ·XML Schemas· corresponding to any <include>d schema documents. Such included schema documents must either (a) have the same `targetNamespace` as the <include>ing schema document, or (b) no `targetNamespace` at all, in which case the <include>d schema document is converted to the <include>ing 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, **all** of the following must be true:

1 If the ·actual value· of the `schemaLocation` [attribute] successfully resolves **one** of the following must be true:
  1.1 It resolves to (a fragment of) a resource which is an XML document (of type `application/xml` or `text/xml` with an XML declaration for preference, but this is not required), which in turn corresponds to a <schema> element information item in a well-formed information set, which in turn corresponds to a valid schema.
  1.2 It resolves 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 <include>d <schema> item **SII**, the valid schema **I** and the <include>ing item's parent <schema> item **SII'**.
2 **One** of the following must be true:
  2.1 **SII** has a `targetNamespace` [attribute], and its ·actual value· is identical to the ·actual value· of the `targetNamespace` [attribute] of **SII'** (which must have such an [attribute]).
  2.2 Neither **SII** nor **SII'** have a `targetNamespace` [attribute].
  2.3 **SII** has no `targetNamespace` [attribute] (but **SII'** does).
3 The appropriate **case** among the following must be true:
  3.1 **If** clause 2.1 or clause 2.2 above is satisfied, **then** the schema corresponding to **SII'** must include not only definitions or declarations corresponding to the appropriate members of its own [children], but also components identical to all the ·schema components· of **I**.
  3.2 **If** clause 2.3 above is satisfied, **then** the schema corresponding to the <include>d item's parent <schema> must include not only definitions or declarations corresponding to the appropriate members of its own [children], but also components identical to all the ·schema components· of **I**, except that anywhere the ·absent· target namespace name would have appeared, the ·actual value· of the `targetNamespace` [attribute] of **SII'** is used. In particular, it replaces ·absent· in the following places:
    3.2.1 The {target namespace} of named schema components, both at the top level and (in the case of nested type definitions and nested attribute and element declarations whose `code` was *qualified*) nested within definitions;
    3.2.2 The {namespace constraint} of a wildcard, whether negated or not;

It is *not* an error for the ·actual 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 above to fail to be satisfied. Failure to resolve may well cause less than complete ·assessment· outcomes, of course.

As discussed in Missing Sub-components (§5.3), ·QName·s in XML representations may fail to ·resolve·, rendering components incomplete and unusable because of missing subcomponents. During schema construction, implementations must retain ·QName· values for such references, in case an appropriately-named component becomes available to discharge the reference by the time it is actually needed. ·Absent· target ·namespace name·s of such as-yet unresolved reference ·QName·s in <include>d components must also be converted if clause 3.2 is satisfied.

> **Note:** The above is carefully worded so that multiple <include>ing of the same schema document will not constitute a violation of clause 2 of Schema Properties Correct (§3.15.6), but applications are allowed, indeed encouraged, to avoid <include>ing the same schema document more than once to forestall the necessity of establishing identity component by component.

4.2.2 Including modified component definitions

In order 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.

XML Representation Summary: `redefine` Element Information Item

```
<redefine
  id = ID
  schemaLocation = anyURI
  {any attributes with non-schema namespace . . .}>
  Content: (annotation | (simpleType | complexType | group | attributeGroup))*
</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> contains not only the components corresponding to its definition and declaration [children], but also all the components of all the ·XML Schemas· corresponding to any <redefine>d schema documents. Such schema documents must either (a) have the same `targetNamespace` as the <redefine>ing schema document, or (b) no `targetNamespace` at all, in which case the <redefine>d schema document is converted to the <redefine>ing schema document's `targetNamespace`.

The definitions within the <redefine> element itself are restricted to be redefinitions of components from the <redefine>d schema document, *in terms of themselves*. That is,

- Type definitions must use themselves as their base type definition;
- Attribute group definitions and model group definitions must be supersets or subsets of their original definitions, either by including exactly one reference to themselves or by containing only (possibly restricted) components which appear in a corresponding way in their <redefine>d selves.

Not all the components of the <redefine>d 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 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.

> **Note:** The pervasive impact of redefinition reinforces the need for implementations to adopt some form of lazy or 'just-in-time' approach to component construction, which is also called for in order to avoid inappropriate dependencies on the order in which definitions and references appear in (collections of) schema documents.

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 **all** of the following must be true:

1 If there are any element information items among the [children] other than <annotation> then the ·actual value· of the `schemaLocation` [attribute] must successfully resolve.

2 If the ·actual value· of the `schemaLocation` [attribute] successfully resolves **one** of the following must be true:

2.1 it resolves to (a fragment of) a resource which is an XML document (see clause 1.1), which in turn corresponds to a <schema> element information item in a well-formed information set, which in turn corresponds to a valid schema.

2.2 It resolves 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 <redefine>d <schema> item **SII**, the valid schema **I** and the <redefine>ing item's parent <schema> item **SII'**.

3 **One** of the following must be true:

3.1 **SII** has a `targetNamespace` [attribute], and its ·actual value· is identical to the ·actual value· of the `targetNamespace` [attribute] of **SII'** (which must have such an [attribute]).

3.2 Neither **SII** nor **SII'** have a `targetNamespace` [attribute].

3.3 **SII** has no `targetNamespace` [attribute] (but **SII'** does).

4 The appropriate **case** among the following must be true:

4.1 **If** clause 3.1 or clause 3.2 above is satisfied, **then** the schema corresponding to **SII'** must include not only definitions or declarations corresponding to the appropriate members of its own [children], but also components identical to all the ·schema components· of **I**, with the exception of those explicitly redefined (see Individual Component Redefinition (§4.2.2) below).

4.2 **If** clause 3.3 above is satisfied, **then** the schema corresponding to **SII'** must include not only definitions or declarations corresponding to the appropriate members of its own [children], but also components identical to all the ·schema components· of **I**, with the exception of those explicitly redefined (see Individual Component Redefinition (§4.2.2) below), except that anywhere the ·absent· target namespace name would have appeared, the ·actual value· of the `targetNamespace` [attribute] of **SII'** is used (see clause 3.2 in Inclusion Constraints and Semantics (§4.2.1) for details).

5 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 ·actual value· of whose `base` [attribute] must be the same as the ·actual value· of its own `name` attribute plus target namespace;

6 Within the [children], for each <group> the appropriate **case** among the following must be true:

6.1 **If** it has a <group> among its contents at some level the ·actual value· of whose `ref` [attribute] is the same as the ·actual value· of its own `name` attribute plus target namespace, **then all** of the following must be true:

6.1.1 It must have exactly one such group.

6.1.2 The ·actual value· of both that group's `minOccurs` and `maxOccurs` [attribute] must be `1` (or ·absent·).

6.2 **If** it has no such self-reference, **then all** of the following must be true:

6.2.1 The ·actual value· of its own `name` attribute plus target namespace must successfully ·resolve· to a model group definition in **I**.

6.2.2 The {model group} of the model group definition which corresponds to it per XML Representation of Model Group Definition Schema Components (§3.7.2) must be a ·valid restriction· of the {model group} of that model group definition in **I**, as defined in Particle Valid (Restriction) (§3.9.6).

7 Within the [children], for each <attributeGroup> the appropriate **case** among the following must be true:

7.1 **If** it has an <attributeGroup> among its contents the ·actual value· of whose `ref` [attribute] is the same as the ·actual value· of its own `name` attribute plus target namespace, **then** it must have exactly one such group.

7.2 **If** it has no such self-reference, **then all** of the following must be true:

7.2.1 The ·actual value· of its own `name` attribute plus target namespace must successfully ·resolve· to an attribute group definition in **I**.

7.2.2 The {attribute uses} and {attribute wildcard} of the attribute group definition which corresponds to it per XML Representation of Attribute Group Definition Schema Components (§3.6.2) must be ·valid restrictions· of the {attribute uses} and {attribute wildcard} of that attribute group definition in **I**, as defined in clause 2, clause 3 and clause 4 of Derivation Valid (Restriction, Complex) (§3.4.6) (where references to the base type definition are understood as references to the attribute group definition in **I**).

Note: An attribute group restrictively redefined per clause 7.2 corresponds to an attribute group whose {attribute uses} consist all and only of those attribute uses corresponding to <attribute>s explicitly present among the [children] of the <redefine>ing <attributeGroup>. No inheritance from the <redefine>d attribute group occurs. Its {attribute wildcard} is similarly based purely on an explicit <anyAttribute>, if present.

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 <redefine>ing 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 <redefine>d schema document, as defined in Schema Component Details (§3), except that its {name} is ·absent·;

1.2 One component which corresponds to the information item itself, as defined in Schema Component Details (§3), except that its {base type definition} is the component defined in 1.1 above.

This pairing ensures 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 <redefine>ing and <redefine>d 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 Schema Component Details (§3), except that if and when a self-reference based on a `ref` [attribute] whose ·actual 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 **I** is used.

In all cases there must be a top-level definition item of the appropriate name and kind in the <redefine>d schema document.

Note: The above is carefully worded so that multiple equivalent <redefine>ing of the same schema document will not constitute a violation of clause 2 of Schema Properties Correct (§3.15.6), but applications are allowed, indeed encouraged, to avoid <redefine>ing 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).

4.2.3 References to schema components across namespaces

As described in XML Schema Abstract Data Model (§2.2), every top-level schema component is associated with a target namespace (or, explicitly, with none). This section sets 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.

Two things are required: 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 = anyURI
  schemaLocation = anyURI
  {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 ·actual 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 ·actual value· of the `schemaLocation`, if present, gives a hint as to where a serialization of 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: Schema Document Access and Web-interoperability (§4.3). When a `schemaLocation` is present, it must contain a single URI reference which the schema author warrants will resolve to a serialization of a ·schema document· containing the component(s) in the <import>ed 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 allowed. This simply allows unqualified reference to foreign components with no target namespace without giving any hints as to where to find them.

Example

The same namespace may be used both for real work, and in the course of defining schema components in terms of foreign components:

```
<schema xmlns="http://www.w3.org/2001/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">
  <sequence>
   <element ref="html:p" minOccurs="0"/>
  </sequence>
  . . .
 </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 **all** of the following must be true:

1 The appropriate **case** among the following must be true:
  1.1 **If** the `namespace` [attribute] is present, **then** its ·actual value· must not match the ·actual value· of the enclosing <schema>'s `targetNamespace` [attribute].
  1.2 **If** the `namespace` [attribute] is not present, **then** the enclosing <schema> must have a `targetNamespace` [attribute]

2 If the application schema reference strategy using the ·actual value·s of the `schemaLocation` and `namespace` [attributes], provides a referent, as defined by Schema Document Location Strategy (§4.3.2), **one** of the following must be true:
  2.1 The referent is (a fragment of) a resource which is an XML document (see clause 1.1), which in turn corresponds to a <schema> element information item in a well-formed information set, which in turn corresponds to a valid schema.
  2.2 The referent is 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**.

3 The appropriate **case** among the following must be true:
  3.1 **If** there is a `namespace` [attribute], **then** its ·actual value· must be identical to the ·actual value· of the `targetNamespace` [attribute] of **SII**.
  3.2 **If** there is no `namespace` [attribute], **then SII** must have no `targetNamespace` [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 2 above to fail to be satisfied. Failure to find a referent may well cause less than complete ·assessment· 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 2 above is satisfied, a set of ·schema components· identical to all the ·schema components· of **I**.

> **Note:** The above is carefully worded so that multiple <import>ing of the same schema document will not constitute a violation of clause 2 of Schema Properties Correct (§3.15.6), but applications are allowed, indeed encouraged, to avoid <import>ing 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 ·actual value· of `schemaLocation`, but such a strategy risks missing useful information when new `schemaLocation`s are offered.

4.3 Layer 3: Schema Document Access and Web-interoperability

Layers 1 and 2 provide a framework for ·assessment· and XML definition of schemas in a broad variety of environments. Over time, a range of standards and conventions may well evolve to support interoperability of XML Schema implementations on the World Wide Web. Layer 3 defines the minimum level of 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.

4.3.1 Standards for representation of schemas and retrieval of schema documents on the Web

For interoperability, serialized ·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.) Such documents on the Web must be part of XML documents (see clause 1.1), 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.
> **Note:** The variations among server software and web site administration policies make it difficult to recommend any particular approach to retrieval requests intended to retrieve serialized ·schema documents·. An `Accept` header of `application/xml, text/xml; q=0.9, */*` is perhaps a reasonable starting point.

4.3.2 How schema definitions are located on the Web

As described in Layer 1: Summary of the Schema-validity Assessment Core (§4.1), processors are responsible for providing the schema components (definitions and declarations) needed for ·assessment·. This section introduces a set of normative conventions to facilitate interoperability for instance and schema documents retrieved and processed from the Web.

> **Note:** As discussed above in Layer 2: Schema Documents, Namespaces and Composition (§4.2), other non-Web mechanisms for delivering schemas for ·assessment· may exist, but are outside the scope of this specification.

Processors on the Web are free to undertake ·assessment· against arbitrary schemas in any of the ways set out in Assessing Schema-Validity (§5.2). However, it is useful to have a common convention for determining the schema to use. Accordingly, general-purpose schema-aware processors (i.e. those not specialized to one or a fixed set of pre-determined schemas) undertaking ·assessment· of a document on the web must behave as follows:

- unless directed otherwise by the user, ·assessment· is undertaken on the document element information item of the specified document;
- unless directed otherwise by the user, the processor is required to construct a schema corresponding to a schema document whose `targetNamespace` is identical to the namespace name, if any, of the element information item on which ·assessment· is undertaken.

The composition of the complete schema for use in ·assessment· is discussed in Layer 2: Schema Documents, Namespaces and Composition (§4.2) 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 (§4.3.1);
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 name. Accordingly whether a processor's default behavior 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 namespace names 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 specification allows but does not require particular communities to establish and implement such conventions. Users are always free to supply namespace names 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 conforms to that schema, the `schemaLocation` and `noNamespaceSchemaLocation` [attributes] (in the XML Schema instance namespace, that is, `http://www.w3.org/2001/XMLSchema-instance`) (hereafter `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation`) are provided. The first records the author's warrant with pairs of URI references (one for the namespace name, and one for a hint as to the location of a schema document defining names for that namespace name). 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 ·actual 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 an element information item initially ·validated· whose [namespace name] it addresses. According to the rules of Layer 1: Summary of the Schema-validity Assessment Core (§4.1), the corresponding schema may be lazily assembled, but is otherwise stable throughout ·assessment·. Although schema location attributes can occur on any element, and can be processed incrementally as discovered, their effect is essentially global to the ·assessment·. 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/2001/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 names 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 name (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 Do nothing, for instance because a schema containing components for the given namespace name 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);

2 Based on the location URI, identify an existing schema document, either as a resource which is an XML document or a <schema> element information item, in some local schema repository;

3 Based on the namespace name, identify an existing schema document, either as a resource which is an XML document or a <schema> element information item, in some local schema repository;

4 Attempt to resolve the location URI, to locate a resource on the web which is or contains or references a <schema> element;

5 Attempt to resolve the namespace name to locate such a resource.

Whenever possible configuration and/or invocation options for selecting and/or ordering the implemented strategies should be provided.

Improved or alternative conventions for Web interoperability can be standardized in the future without reopening this specification. For example, the W3C is currently considering initiatives to standardize 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.

5 Schemas and Schema-validity Assessment

The architecture of schema-aware processing allows for a rich characterization of XML documents: schema validity is not a binary predicate.

This specification distinguishes between errors in schema construction and structure, on the one hand, and schema validation outcomes, on the other.

5.1 Errors in Schema Construction and Structure

Before ·assessment· can be attempted, a schema is required. Special-purpose applications are free to determine a schema for use in ·assessment· by whatever means are appropriate, but general purpose processors should implement the strategy set out in Schema Document Location Strategy (§4.3.2), starting with the namespaces declared in the document whose ·assessment· is being undertaken, and the ·actual value·s of the `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` [attributes] thereof, if any, along with any other information about schema identity or schema document location provided by users in application-specific ways, 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 the last section of each of the subsections of Schema Component Details (§3).

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 Schema Component Details (§3) and Schemas and Namespaces: Access and Composition (§4), two additional conditions hold:

- It is an error if any such schema document would not be fully valid with respect to a schema corresponding to the Schema for Schemas (normative) (§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 *valid*.
- 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 Schema Representation Constraints (§C.3).

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 ·assessment· with respect to schema-like entities which do *not* satisfy all the above conditions is incoherent. Accordingly, conformant processors must not attempt to undertake ·assessment· using such non-schemas.

5.2 Assessing Schema-Validity

With a schema which satisfies the conditions expressed in Errors in Schema Construction and Structure (§5.1) 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 appeals to Schema-Validity Assessment (Element) (§3.3.4) (clause 1.2);
2 The user or application identifies a element declaration from among the {element declarations} of the schema, checks that its {name} and {target namespace} match the [local name] and [namespace name] of the item, and appeals to Schema-Validity Assessment (Element) (§3.3.4) (clause 1.1);
3 The processor starts from Schema-Validity Assessment (Element) (§3.3.4) with no stipulated declaration or definition, and either ·strict· or ·lax· assessment ensues, depending on whether or not the element information and the schema determine either an element declaration (by name) or a type definition (via `xsi:type`) or not.

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 Assessment Outcome (Element) (§3.3.5) and Assessment Outcome (Attribute) (§3.2.5). It is up to applications to decide what constitutes a successful outcome.

Note that every element and attribute information item participating in the ·assessment· will also have a [validation context] property which refers back to the element information item at which ·assessment· began. [Definition:] This item, that is the element information item at which ·assessment· began, is called the **validation root**.

> **Note:** This specification does not reconstruct the XML 1.0 notion of *root* in either schemas or instances. Equivalent functionality is provided for at ·assessment· invocation, via clause 2 above.
> **Note:** This specification has nothing normative to say about multiple ·assessment· episodes. It should however be clear from the above that if a processor restarts ·assessment· with respect to a ·post-schema-validation infoset· some ·post-schema-validation infoset· contributions from the previous ·assessment· may be overwritten. Restarting nonetheless may be useful, particularly at a node whose [validation attempted] property is *none*, in which case there are three obvious cases in which additional useful information may result:
>
> - ·assessment· was not attempted because of a ·validation· failure, but declarations and/or definitions are available for at least some of the [children] or [attributes];
> - ·assessment· was not attempted because a named definition or declaration was missing, but after further effort the processor has retrieved it.
> - ·assessment· was not attempted because it was *skip*ped, but the processor has at least some declarations and/or definitions available for at least some of the [children] or [attributes].

5.3 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 ·assessment·, an element or attribute information item is being ·validated· with respect to a component of any kind any of whose properties has or contains such an ·absent· value, the ·validation· is modified, as following:

- In the case of attribute information items, the effect is as if clause 1 of Attribute Locally Valid (§3.2.4) had failed;
- In the case of element information items, the effect is as if clause 1 of Element Locally Valid (Element) (§3.3.4) had failed;
- In the case of element information items, processors may choose to continue ·assessment·: see ·lax assessment·.

Because of the value specification for [validation attempted] in Assessment Outcome (Element) (§3.3.5), if this situation ever arises, the document as a whole cannot show a [validation attempted] of *full*.

5.4 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 Schema for Schemas (normative)

The XML representation of the schema for schema documents is presented here as a normative part of the specification, and as an illustrative example of how the XML Schema language can define itself using its own constructs. 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 a schema document 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.

```
<!DOCTYPE xs:schema PUBLIC "-//W3C//DTD XMLSCHEMA 200102//EN" "XMLSchema.dtd" [

<!-- provide ID type information even for parsers which only read the
     internal subset -->
<!ATTLIST xs:schema          id  ID  #IMPLIED>
<!ATTLIST xs:complexType     id  ID  #IMPLIED>
<!ATTLIST xs:complexContent  id  ID  #IMPLIED>
<!ATTLIST xs:simpleContent   id  ID  #IMPLIED>
<!ATTLIST xs:extension       id  ID  #IMPLIED>
<!ATTLIST xs:element         id  ID  #IMPLIED>
<!ATTLIST xs:group           id  ID  #IMPLIED>
<!ATTLIST xs:all             id  ID  #IMPLIED>
<!ATTLIST xs:choice          id  ID  #IMPLIED>
<!ATTLIST xs:sequence        id  ID  #IMPLIED>
<!ATTLIST xs:any             id  ID  #IMPLIED>
<!ATTLIST xs:anyAttribute    id  ID  #IMPLIED>
<!ATTLIST xs:attribute       id  ID  #IMPLIED>
<!ATTLIST xs:attributeGroup  id  ID  #IMPLIED>
<!ATTLIST xs:unique          id  ID  #IMPLIED>
<!ATTLIST xs:key             id  ID  #IMPLIED>
<!ATTLIST xs:keyref          id  ID  #IMPLIED>
<!ATTLIST xs:selector        id  ID  #IMPLIED>
<!ATTLIST xs:field           id  ID  #IMPLIED>
<!ATTLIST xs:include         id  ID  #IMPLIED>
<!ATTLIST xs:import          id  ID  #IMPLIED>
<!ATTLIST xs:redefine        id  ID  #IMPLIED>
<!ATTLIST xs:notation        id  ID  #IMPLIED>
]>

<?xml version='1.0'?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" blockDefault="#all"
           elementFormDefault="qualified" xml:lang="EN"
           targetNamespace="http://www.w3.org/2001/XMLSchema"
           version="Id: structures.xsd,v 1.2 2004/01/15 11:34:25 ht Exp ">
  <xs:annotation>
    <xs:documentation source="../structures/structures-with-errata.html.html">
   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</xs:documentation>
  </xs:annotation>
  <xs:annotation>
    <xs:documentation>
   The simpleType element and all of its members are defined
   in datatypes.xsd</xs:documentation>
  </xs:annotation>
  <xs:include schemaLocation="datatypes.xsd"/>
  <xs:import namespace="http://www.w3.org/XML/1998/namespace"
             schemaLocation="http://www.w3.org/2001/xml.xsd">
    <xs:annotation>
      <xs:documentation>
       Get access to the xml: attribute groups for xml:lang
       as declared on 'schema' and 'documentation' below
     </xs:documentation>
    </xs:annotation>
  </xs:import>
  <xs:complexType name="openAttrs">
    <xs:annotation>
      <xs:documentation>
       This type is extended by almost all schema types
       to allow attributes from other namespaces to be
       added to user schemas.
     </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
      <xs:restriction base="xs:anyType">
        <xs:anyAttribute namespace="##other" processContents="lax"/>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="annotated">
    <xs:annotation>
      <xs:documentation>
       This type is extended by all types which allow annotation
       other than &lt;schema> itself
     </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
      <xs:extension base="xs:openAttrs">
        <xs:sequence>
          <xs:element ref="xs:annotation" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:ID"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:group name="schemaTop">
    <xs:annotation>
      <xs:documentation>
   This group is for the
   elements which occur freely at the top level of schemas.
   All of their types are based on the "annotated" type by extension.</xs:documentation>
    </xs:annotation>
    <xs:choice>
      <xs:group ref="xs:redefinable"/>
      <xs:element ref="xs:element"/>
```

```
    <xs:element ref="xs:attribute"/>
    <xs:element ref="xs:notation"/>
  </xs:choice>
</xs:group>
<xs:group name="redefinable">
  <xs:annotation>
    <xs:documentation>
 This group is for the
 elements which can self-redefine (see &lt;redefine> below).</xs:documentation>
  </xs:annotation>
  <xs:choice>
    <xs:element ref="xs:simpleType"/>
    <xs:element ref="xs:complexType"/>
    <xs:element ref="xs:group"/>
    <xs:element ref="xs:attributeGroup"/>
  </xs:choice>
</xs:group>
<xs:simpleType name="formChoice">
  <xs:annotation>
    <xs:documentation>
 A utility type, not for public use</xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:NMTOKEN">
    <xs:enumeration value="qualified"/>
    <xs:enumeration value="unqualified"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="reducedDerivationControl">
  <xs:annotation>
    <xs:documentation>
 A utility type, not for public use</xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:derivationControl">
    <xs:enumeration value="extension"/>
    <xs:enumeration value="restriction"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="derivationSet">
  <xs:annotation>
    <xs:documentation>
 A utility type, not for public use</xs:documentation>
    <xs:documentation>
 #all or (possibly empty) subset of {extension, restriction}</xs:documentation>
  </xs:annotation>
  <xs:union>
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="#all"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType>
      <xs:list itemType="xs:reducedDerivationControl"/>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>
<xs:simpleType name="typeDerivationControl">
  <xs:annotation>
    <xs:documentation>
 A utility type, not for public use</xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:derivationControl">
    <xs:enumeration value="extension"/>
    <xs:enumeration value="restriction"/>
    <xs:enumeration value="list"/>
    <xs:enumeration value="union"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="fullDerivationSet">
  <xs:annotation>
    <xs:documentation>
 A utility type, not for public use</xs:documentation>
    <xs:documentation>
 #all or (possibly empty) subset of {extension, restriction, list, union}</xs:documentation>
  </xs:annotation>
  <xs:union>
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="#all"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType>
      <xs:list itemType="xs:typeDerivationControl"/>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>
<xs:element name="schema" id="schema">
  <xs:annotation>
    <xs:documentation
         source="http://www.w3.org/TR/xmlschema-1/#element-schema"/>
  </xs:annotation>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xs:openAttrs">
        <xs:sequence>
          <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="xs:include"/>
            <xs:element ref="xs:import"/>
```

```
            <xs:element ref="xs:redefine"/>
            <xs:element ref="xs:annotation"/>
          </xs:choice>
          <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:group ref="xs:schemaTop"/>
            <xs:element ref="xs:annotation" minOccurs="0"
                        maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:sequence>
        <xs:attribute name="targetNamespace" type="xs:anyURI"/>
        <xs:attribute name="version" type="xs:token"/>
        <xs:attribute name="finalDefault" type="xs:fullDerivationSet"
                      default="" use="optional"/>
        <xs:attribute name="blockDefault" type="xs:blockSet" default=""
                      use="optional"/>
        <xs:attribute name="attributeFormDefault" type="xs:formChoice"
                      default="unqualified" use="optional"/>
        <xs:attribute name="elementFormDefault" type="xs:formChoice"
                      default="unqualified" use="optional"/>
        <xs:attribute name="id" type="xs:ID"/>
        <xs:attribute ref="xml:lang"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:key name="element">
    <xs:selector xpath="xs:element"/>
    <xs:field xpath="@name"/>
  </xs:key>
  <xs:key name="attribute">
    <xs:selector xpath="xs:attribute"/>
    <xs:field xpath="@name"/>
  </xs:key>
  <xs:key name="type">
    <xs:selector xpath="xs:complexType|xs:simpleType"/>
    <xs:field xpath="@name"/>
  </xs:key>
  <xs:key name="group">
    <xs:selector xpath="xs:group"/>
    <xs:field xpath="@name"/>
  </xs:key>
  <xs:key name="attributeGroup">
    <xs:selector xpath="xs:attributeGroup"/>
    <xs:field xpath="@name"/>
  </xs:key>
  <xs:key name="notation">
    <xs:selector xpath="xs:notation"/>
    <xs:field xpath="@name"/>
  </xs:key>
  <xs:key name="identityConstraint">
    <xs:selector xpath=".//xs:key|.//xs:unique|.//xs:keyref"/>
    <xs:field xpath="@name"/>
  </xs:key>
</xs:element>
<xs:simpleType name="allNNI">
  <xs:annotation>
    <xs:documentation>
 for maxOccurs</xs:documentation>
  </xs:annotation>
  <xs:union memberTypes="xs:nonNegativeInteger">
    <xs:simpleType>
      <xs:restriction base="xs:NMTOKEN">
        <xs:enumeration value="unbounded"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>
<xs:attributeGroup name="occurs">
  <xs:annotation>
    <xs:documentation>
 for all particles</xs:documentation>
  </xs:annotation>
  <xs:attribute name="minOccurs" type="xs:nonNegativeInteger" default="1"
               use="optional"/>
  <xs:attribute name="maxOccurs" type="xs:allNNI" default="1" use="optional"/>
</xs:attributeGroup>
<xs:attributeGroup name="defRef">
  <xs:annotation>
    <xs:documentation>
 for element, group and attributeGroup,
 which both define and reference</xs:documentation>
  </xs:annotation>
  <xs:attribute name="name" type="xs:NCName"/>
  <xs:attribute name="ref" type="xs:QName"/>
</xs:attributeGroup>
<xs:group name="typeDefParticle">
  <xs:annotation>
    <xs:documentation>
 'complexType' uses this</xs:documentation>
  </xs:annotation>
  <xs:choice>
    <xs:element name="group" type="xs:groupRef"/>
    <xs:element ref="xs:all"/>
    <xs:element ref="xs:choice"/>
    <xs:element ref="xs:sequence"/>
  </xs:choice>
</xs:group>
<xs:group name="nestedParticle">
```

```
  <xs:choice>
    <xs:element name="element" type="xs:localElement"/>
    <xs:element name="group" type="xs:groupRef"/>
    <xs:element ref="xs:choice"/>
    <xs:element ref="xs:sequence"/>
    <xs:element ref="xs:any"/>
  </xs:choice>
</xs:group>
<xs:group name="particle">
  <xs:choice>
    <xs:element name="element" type="xs:localElement"/>
    <xs:element name="group" type="xs:groupRef"/>
    <xs:element ref="xs:all"/>
    <xs:element ref="xs:choice"/>
    <xs:element ref="xs:sequence"/>
    <xs:element ref="xs:any"/>
  </xs:choice>
</xs:group>
<xs:complexType name="attribute">
  <xs:complexContent>
    <xs:extension base="xs:annotated">
      <xs:sequence>
        <xs:element name="simpleType" type="xs:localSimpleType" minOccurs="0"/>
      </xs:sequence>
      <xs:attributeGroup ref="xs:defRef"/>
      <xs:attribute name="type" type="xs:QName"/>
      <xs:attribute name="use" default="optional" use="optional">
        <xs:simpleType>
          <xs:restriction base="xs:NMTOKEN">
            <xs:enumeration value="prohibited"/>
            <xs:enumeration value="optional"/>
            <xs:enumeration value="required"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="default" type="xs:string"/>
      <xs:attribute name="fixed" type="xs:string"/>
      <xs:attribute name="form" type="xs:formChoice"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="topLevelAttribute">
  <xs:complexContent>
    <xs:restriction base="xs:attribute">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0"/>
        <xs:element name="simpleType" type="xs:localSimpleType" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="ref" use="prohibited"/>
      <xs:attribute name="form" use="prohibited"/>
      <xs:attribute name="use" use="prohibited"/>
      <xs:attribute name="name" type="xs:NCName" use="required"/>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:group name="attrDecls">
  <xs:sequence>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="attribute" type="xs:attribute"/>
      <xs:element name="attributeGroup" type="xs:attributeGroupRef"/>
    </xs:choice>
    <xs:element ref="xs:anyAttribute" minOccurs="0"/>
  </xs:sequence>
</xs:group>
<xs:element name="anyAttribute" type="xs:wildcard" id="anyAttribute">
  <xs:annotation>
    <xs:documentation
         source="http://www.w3.org/TR/xmlschema-1/#element-anyAttribute"/>
  </xs:annotation>
</xs:element>
<xs:group name="complexTypeModel">
  <xs:choice>
    <xs:element ref="xs:simpleContent"/>
    <xs:element ref="xs:complexContent"/>
    <xs:sequence>
      <xs:annotation>
        <xs:documentation>
 This branch is short for
 &lt;complexContent>
 &lt;restriction base="xs:anyType">
 ...
 &lt;/restriction>
 &lt;/complexContent></xs:documentation>
      </xs:annotation>
      <xs:group ref="xs:typeDefParticle" minOccurs="0"/>
      <xs:group ref="xs:attrDecls"/>
    </xs:sequence>
  </xs:choice>
</xs:group>
<xs:complexType name="complexType" abstract="true">
  <xs:complexContent>
    <xs:extension base="xs:annotated">
      <xs:group ref="xs:complexTypeModel"/>
      <xs:attribute name="name" type="xs:NCName">
        <xs:annotation>
          <xs:documentation>
```

```
    Will be restricted to required or forbidden</xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name="mixed" type="xs:boolean" default="false"
                    use="optional">
        <xs:annotation>
          <xs:documentation>
    Not allowed if simpleContent child is chosen.
    May be overriden by setting on complexContent child.</xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name="abstract" type="xs:boolean" default="false"
                    use="optional"/>
      <xs:attribute name="final" type="xs:derivationSet"/>
      <xs:attribute name="block" type="xs:derivationSet"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="topLevelComplexType">
  <xs:complexContent>
    <xs:restriction base="xs:complexType">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0"/>
        <xs:group ref="xs:complexTypeModel"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:NCName" use="required"/>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="localComplexType">
  <xs:complexContent>
    <xs:restriction base="xs:complexType">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0"/>
        <xs:group ref="xs:complexTypeModel"/>
      </xs:sequence>
      <xs:attribute name="name" use="prohibited"/>
      <xs:attribute name="abstract" use="prohibited"/>
      <xs:attribute name="final" use="prohibited"/>
      <xs:attribute name="block" use="prohibited"/>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="restrictionType">
  <xs:complexContent>
    <xs:extension base="xs:annotated">
      <xs:sequence>
        <xs:choice minOccurs="0">
          <xs:group ref="xs:typeDefParticle"/>
          <xs:group ref="xs:simpleRestrictionModel"/>
        </xs:choice>
        <xs:group ref="xs:attrDecls"/>
      </xs:sequence>
      <xs:attribute name="base" type="xs:QName" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="complexRestrictionType">
  <xs:complexContent>
    <xs:restriction base="xs:restrictionType">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0"/>
        <xs:choice minOccurs="0">
          <xs:annotation>
            <xs:documentation>This choice is added simply to
                 make this a valid restriction per the REC</xs:documentation>
          </xs:annotation>
          <xs:group ref="xs:typeDefParticle"/>
        </xs:choice>
        <xs:group ref="xs:attrDecls"/>
      </xs:sequence>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="extensionType">
  <xs:complexContent>
    <xs:extension base="xs:annotated">
      <xs:sequence>
        <xs:group ref="xs:typeDefParticle" minOccurs="0"/>
        <xs:group ref="xs:attrDecls"/>
      </xs:sequence>
      <xs:attribute name="base" type="xs:QName" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="complexContent" id="complexContent">
  <xs:annotation>
    <xs:documentation
         source="http://www.w3.org/TR/xmlschema-1/#element-complexContent"/>
  </xs:annotation>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xs:annotated">
        <xs:choice>
```

```
          <xs:element name="restriction" type="xs:complexRestrictionType"/>
          <xs:element name="extension" type="xs:extensionType"/>
        </xs:choice>
        <xs:attribute name="mixed" type="xs:boolean">
          <xs:annotation>
            <xs:documentation>
     Overrides any setting on complexType parent.</xs:documentation>
          </xs:annotation>
        </xs:attribute>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:complexType name="simpleRestrictionType">
  <xs:complexContent>
    <xs:restriction base="xs:restrictionType">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0"/>
        <xs:choice minOccurs="0">
          <xs:annotation>
            <xs:documentation>This choice is added simply to
                 make this a valid restriction per the REC</xs:documentation>
          </xs:annotation>
          <xs:group ref="xs:simpleRestrictionModel"/>
        </xs:choice>
        <xs:group ref="xs:attrDecls"/>
      </xs:sequence>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="simpleExtensionType">
  <xs:complexContent>
    <xs:restriction base="xs:extensionType">
      <xs:sequence>
        <xs:annotation>
          <xs:documentation>
    No typeDefParticle group reference</xs:documentation>
        </xs:annotation>
        <xs:element ref="xs:annotation" minOccurs="0"/>
        <xs:group ref="xs:attrDecls"/>
      </xs:sequence>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:element name="simpleContent" id="simpleContent">
  <xs:annotation>
    <xs:documentation
         source="http://www.w3.org/TR/xmlschema-1/#element-simpleContent"/>
  </xs:annotation>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xs:annotated">
        <xs:choice>
          <xs:element name="restriction" type="xs:simpleRestrictionType"/>
          <xs:element name="extension" type="xs:simpleExtensionType"/>
        </xs:choice>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:element name="complexType" type="xs:topLevelComplexType" id="complexType">
  <xs:annotation>
    <xs:documentation
         source="http://www.w3.org/TR/xmlschema-1/#element-complexType"/>
  </xs:annotation>
</xs:element>
<xs:simpleType name="blockSet">
  <xs:annotation>
    <xs:documentation>
  A utility type, not for public use</xs:documentation>
    <xs:documentation>
  #all or (possibly empty) subset of {substitution, extension,
  restriction}</xs:documentation>
  </xs:annotation>
  <xs:union>
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="#all"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType>
      <xs:list>
        <xs:simpleType>
          <xs:restriction base="xs:derivationControl">
            <xs:enumeration value="extension"/>
            <xs:enumeration value="restriction"/>
            <xs:enumeration value="substitution"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:list>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>
<xs:complexType name="element" abstract="true">
  <xs:annotation>
```

```
    <xs:documentation>
 The element element can be used either
 at the top level 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.</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="xs:annotated">
      <xs:sequence>
        <xs:choice minOccurs="0">
          <xs:element name="simpleType" type="xs:localSimpleType"/>
          <xs:element name="complexType" type="xs:localComplexType"/>
        </xs:choice>
        <xs:group ref="xs:identityConstraint" minOccurs="0"
                  maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attributeGroup ref="xs:defRef"/>
      <xs:attribute name="type" type="xs:QName"/>
      <xs:attribute name="substitutionGroup" type="xs:QName"/>
      <xs:attributeGroup ref="xs:occurs"/>
      <xs:attribute name="default" type="xs:string"/>
      <xs:attribute name="fixed" type="xs:string"/>
      <xs:attribute name="nillable" type="xs:boolean" default="false"
                    use="optional"/>
      <xs:attribute name="abstract" type="xs:boolean" default="false"
                    use="optional"/>
      <xs:attribute name="final" type="xs:derivationSet"/>
      <xs:attribute name="block" type="xs:blockSet"/>
      <xs:attribute name="form" type="xs:formChoice"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="topLevelElement">
  <xs:complexContent>
    <xs:restriction base="xs:element">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0"/>
        <xs:choice minOccurs="0">
          <xs:element name="simpleType" type="xs:localSimpleType"/>
          <xs:element name="complexType" type="xs:localComplexType"/>
        </xs:choice>
        <xs:group ref="xs:identityConstraint" minOccurs="0"
                  maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="ref" use="prohibited"/>
      <xs:attribute name="form" use="prohibited"/>
      <xs:attribute name="minOccurs" use="prohibited"/>
      <xs:attribute name="maxOccurs" use="prohibited"/>
      <xs:attribute name="name" type="xs:NCName" use="required"/>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="localElement">
  <xs:complexContent>
    <xs:restriction base="xs:element">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0"/>
        <xs:choice minOccurs="0">
          <xs:element name="simpleType" type="xs:localSimpleType"/>
          <xs:element name="complexType" type="xs:localComplexType"/>
        </xs:choice>
        <xs:group ref="xs:identityConstraint" minOccurs="0"
                  maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="substitutionGroup" use="prohibited"/>
      <xs:attribute name="final" use="prohibited"/>
      <xs:attribute name="abstract" use="prohibited"/>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:element name="element" type="xs:topLevelElement" id="element">
  <xs:annotation>
    <xs:documentation
         source="http://www.w3.org/TR/xmlschema-1/#element-element"/>
  </xs:annotation>
</xs:element>
<xs:complexType name="group" abstract="true">
  <xs:annotation>
    <xs:documentation>
 group type for explicit groups, named top-level groups and
 group references</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="xs:annotated">
      <xs:group ref="xs:particle" minOccurs="0" maxOccurs="unbounded"/>
      <xs:attributeGroup ref="xs:defRef"/>
      <xs:attributeGroup ref="xs:occurs"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="realGroup">
  <xs:complexContent>
    <xs:restriction base="xs:group">
      <xs:sequence>
```

```
        <xs:element ref="xs:annotation" minOccurs="0"/>
        <xs:choice minOccurs="0" maxOccurs="1">
          <xs:element ref="xs:all"/>
          <xs:element ref="xs:choice"/>
          <xs:element ref="xs:sequence"/>
        </xs:choice>
      </xs:sequence>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="namedGroup">
  <xs:complexContent>
    <xs:restriction base="xs:realGroup">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0"/>
        <xs:choice minOccurs="1" maxOccurs="1">
          <xs:element name="all">
            <xs:complexType>
              <xs:complexContent>
                <xs:restriction base="xs:all">
                  <xs:group ref="xs:allModel"/>
                  <xs:attribute name="minOccurs" use="prohibited"/>
                  <xs:attribute name="maxOccurs" use="prohibited"/>
                  <xs:anyAttribute namespace="##other" processContents="lax"/>
                </xs:restriction>
              </xs:complexContent>
            </xs:complexType>
          </xs:element>
          <xs:element name="choice" type="xs:simpleExplicitGroup"/>
          <xs:element name="sequence" type="xs:simpleExplicitGroup"/>
        </xs:choice>
      </xs:sequence>
      <xs:attribute name="name" type="xs:NCName" use="required"/>
      <xs:attribute name="ref" use="prohibited"/>
      <xs:attribute name="minOccurs" use="prohibited"/>
      <xs:attribute name="maxOccurs" use="prohibited"/>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="groupRef">
  <xs:complexContent>
    <xs:restriction base="xs:realGroup">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="ref" type="xs:QName" use="required"/>
      <xs:attribute name="name" use="prohibited"/>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="explicitGroup">
  <xs:annotation>
    <xs:documentation>
 group type for the three kinds of group</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:restriction base="xs:group">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0"/>
        <xs:group ref="xs:nestedParticle" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:NCName" use="prohibited"/>
      <xs:attribute name="ref" type="xs:QName" use="prohibited"/>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="simpleExplicitGroup">
  <xs:complexContent>
    <xs:restriction base="xs:explicitGroup">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0"/>
        <xs:group ref="xs:nestedParticle" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="minOccurs" use="prohibited"/>
      <xs:attribute name="maxOccurs" use="prohibited"/>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:group name="allModel">
  <xs:sequence>
    <xs:element ref="xs:annotation" minOccurs="0"/>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:annotation>
        <xs:documentation>This choice with min/max is here to
                        avoid a pblm with the Elt:All/Choice/Seq
                        Particle derivation constraint</xs:documentation>
      </xs:annotation>
      <xs:element name="element" type="xs:narrowMaxMin"/>
    </xs:choice>
  </xs:sequence>
</xs:group>
<xs:complexType name="narrowMaxMin">
```

```
  <xs:annotation>
    <xs:documentation>restricted max/min</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:restriction base="xs:localElement">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0"/>
        <xs:choice minOccurs="0">
          <xs:element name="simpleType" type="xs:localSimpleType"/>
          <xs:element name="complexType" type="xs:localComplexType"/>
        </xs:choice>
        <xs:group ref="xs:identityConstraint" minOccurs="0"
                  maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="minOccurs" default="1" use="optional">
        <xs:simpleType>
          <xs:restriction base="xs:nonNegativeInteger">
            <xs:enumeration value="0"/>
            <xs:enumeration value="1"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="maxOccurs" default="1" use="optional">
        <xs:simpleType>
          <xs:restriction base="xs:allNNI">
            <xs:enumeration value="0"/>
            <xs:enumeration value="1"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="all">
  <xs:annotation>
    <xs:documentation>
 Only elements allowed inside</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:restriction base="xs:explicitGroup">
      <xs:group ref="xs:allModel"/>
      <xs:attribute name="minOccurs" default="1" use="optional">
        <xs:simpleType>
          <xs:restriction base="xs:nonNegativeInteger">
            <xs:enumeration value="0"/>
            <xs:enumeration value="1"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="maxOccurs" default="1" use="optional">
        <xs:simpleType>
          <xs:restriction base="xs:allNNI">
            <xs:enumeration value="1"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:element name="all" type="xs:all" id="all">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-1/#element-all"/>
  </xs:annotation>
</xs:element>
<xs:element name="choice" type="xs:explicitGroup" id="choice">
  <xs:annotation>
    <xs:documentation
         source="http://www.w3.org/TR/xmlschema-1/#element-choice"/>
  </xs:annotation>
</xs:element>
<xs:element name="sequence" type="xs:explicitGroup" id="sequence">
  <xs:annotation>
    <xs:documentation
         source="http://www.w3.org/TR/xmlschema-1/#element-sequence"/>
  </xs:annotation>
</xs:element>
<xs:element name="group" type="xs:namedGroup" id="group">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-1/#element-group"/>
  </xs:annotation>
</xs:element>
<xs:complexType name="wildcard">
  <xs:complexContent>
    <xs:extension base="xs:annotated">
      <xs:attribute name="namespace" type="xs:namespaceList" default="##any"
                    use="optional"/>
      <xs:attribute name="processContents" default="strict" use="optional">
        <xs:simpleType>
          <xs:restriction base="xs:NMTOKEN">
            <xs:enumeration value="skip"/>
            <xs:enumeration value="lax"/>
            <xs:enumeration value="strict"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
```

```
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="any" id="any">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-1/#element-any"/>
  </xs:annotation>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xs:wildcard">
        <xs:attributeGroup ref="xs:occurs"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:annotation>
  <xs:documentation>
 simple type for the value of the 'namespace' attr of
 'any' and 'anyAttribute'</xs:documentation>
</xs:annotation>
<xs:annotation>
  <xs: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</xs:documentation>
</xs:annotation>
<xs:simpleType name="namespaceList">
  <xs:annotation>
    <xs:documentation>
 A utility type, not for public use</xs:documentation>
  </xs:annotation>
  <xs:union>
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="##any"/>
        <xs:enumeration value="##other"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType>
      <xs:list>
        <xs:simpleType>
          <xs:union memberTypes="xs:anyURI">
            <xs:simpleType>
              <xs:restriction base="xs:token">
                <xs:enumeration value="##targetNamespace"/>
                <xs:enumeration value="##local"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:union>
        </xs:simpleType>
      </xs:list>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>
<xs:element name="attribute" type="xs:topLevelAttribute" id="attribute">
  <xs:annotation>
    <xs:documentation
         source="http://www.w3.org/TR/xmlschema-1/#element-attribute"/>
  </xs:annotation>
</xs:element>
<xs:complexType name="attributeGroup" abstract="true">
  <xs:complexContent>
    <xs:extension base="xs:annotated">
      <xs:group ref="xs:attrDecls"/>
      <xs:attributeGroup ref="xs:defRef"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="namedAttributeGroup">
  <xs:complexContent>
    <xs:restriction base="xs:attributeGroup">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0"/>
        <xs:group ref="xs:attrDecls"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:NCName" use="required"/>
      <xs:attribute name="ref" use="prohibited"/>
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="attributeGroupRef">
  <xs:complexContent>
    <xs:restriction base="xs:attributeGroup">
```

```
        <xs:sequence>
          <xs:element ref="xs:annotation" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="ref" type="xs:QName" use="required"/>
        <xs:attribute name="name" use="prohibited"/>
        <xs:anyAttribute namespace="##other" processContents="lax"/>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="attributeGroup" type="xs:namedAttributeGroup"
              id="attributeGroup">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema-1/#element-attributeGroup"/>
    </xs:annotation>
  </xs:element>
  <xs:element name="include" id="include">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema-1/#element-include"/>
    </xs:annotation>
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="xs:annotated">
          <xs:attribute name="schemaLocation" type="xs:anyURI" use="required"/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="redefine" id="redefine">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema-1/#element-redefine"/>
    </xs:annotation>
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="xs:openAttrs">
          <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="xs:annotation"/>
            <xs:group ref="xs:redefinable"/>
          </xs:choice>
          <xs:attribute name="schemaLocation" type="xs:anyURI" use="required"/>
          <xs:attribute name="id" type="xs:ID"/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="import" id="import">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema-1/#element-import"/>
    </xs:annotation>
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="xs:annotated">
          <xs:attribute name="namespace" type="xs:anyURI"/>
          <xs:attribute name="schemaLocation" type="xs:anyURI"/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="selector" id="selector">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema-1/#element-selector"/>
    </xs:annotation>
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="xs:annotated">
          <xs:attribute name="xpath" use="required">
            <xs:simpleType>
              <xs:annotation>
                <xs:documentation>A subset of XPath expressions for use
in selectors</xs:documentation>
                <xs:documentation>A utility type, not for public
use</xs:documentation>
              </xs:annotation>
              <xs:restriction base="xs:token">
                <xs:annotation>
                  <xs:documentation>The following pattern is intended to allow XPath
                       expressions per the following EBNF:
          Selector    ::=    Path ( '|' Path )*
          Path    ::=    ('.//')? Step ( '/' Step )*
          Step    ::=    '.' | NameTest
          NameTest    ::=    QName | '*' | NCName ':' '*'
                       child:: is also allowed
         </xs:documentation>
                </xs:annotation>
                <xs:pattern
                     value="(\.//)?(((child::)?((\i\c*:)?(\i\c*|\*)))|\.)(/(((child::)?((\i\c*:)?(\i\c*|\*)))|\.))*(\|(\.//)?(((child::)?((\
              </xs:restriction>
            </xs:simpleType>
          </xs:attribute>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
```

```
  <xs:element name="field" id="field">
    <xs:annotation>
      <xs:documentation source="http://www.w3.org/TR/xmlschema-1/#element-field"/>
    </xs:annotation>
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="xs:annotated">
          <xs:attribute name="xpath" use="required">
            <xs:simpleType>
              <xs:annotation>
                <xs:documentation>A subset of XPath expressions for use
in fields</xs:documentation>
                <xs:documentation>A utility type, not for public
use</xs:documentation>
              </xs:annotation>
              <xs:restriction base="xs:token">
                <xs:annotation>
                  <xs:documentation>The following pattern is intended to allow XPath
                           expressions per the same EBNF as for selector,
                           with the following change:
          Path    ::=    ('.//')? ( Step '/' )* ( Step | '@' NameTest )
         </xs:documentation>
                </xs:annotation>
                <xs:pattern
                    value="(\.//)?((((child::)?((\i\c*:)?(\i\c*|\*)))|\.)/)*((((child::)?((\i\c*:)?(\i\c*|\*)))|\.)|((attribute::|@)((\i\c*
                </xs:restriction>
            </xs:simpleType>
          </xs:attribute>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="keybase">
    <xs:complexContent>
      <xs:extension base="xs:annotated">
        <xs:sequence>
          <xs:element ref="xs:selector"/>
          <xs:element ref="xs:field" minOccurs="1" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="name" type="xs:NCName" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:group name="identityConstraint">
    <xs:annotation>
      <xs:documentation>The three kinds of identity constraints, all with
                    type of or derived from 'keybase'.
   </xs:documentation>
    </xs:annotation>
    <xs:choice>
      <xs:element ref="xs:unique"/>
      <xs:element ref="xs:key"/>
      <xs:element ref="xs:keyref"/>
    </xs:choice>
  </xs:group>
  <xs:element name="unique" type="xs:keybase" id="unique">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema-1/#element-unique"/>
    </xs:annotation>
  </xs:element>
  <xs:element name="key" type="xs:keybase" id="key">
    <xs:annotation>
      <xs:documentation source="http://www.w3.org/TR/xmlschema-1/#element-key"/>
    </xs:annotation>
  </xs:element>
  <xs:element name="keyref" id="keyref">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema-1/#element-keyref"/>
    </xs:annotation>
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="xs:keybase">
          <xs:attribute name="refer" type="xs:QName" use="required"/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="notation" id="notation">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema-1/#element-notation"/>
    </xs:annotation>
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="xs:annotated">
          <xs:attribute name="name" type="xs:NCName" use="required"/>
          <xs:attribute name="public" type="xs:public"/>
          <xs:attribute name="system" type="xs:anyURI"/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:simpleType name="public">
    <xs:annotation>
      <xs:documentation>
```

```
 A utility type, not for public use</xs:documentation>
    <xs:documentation>
 A public identifier, per ISO 8879</xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:token"/>
</xs:simpleType>
<xs:element name="appinfo" id="appinfo">
  <xs:annotation>
    <xs:documentation
         source="http://www.w3.org/TR/xmlschema-1/#element-appinfo"/>
  </xs:annotation>
  <xs:complexType mixed="true">
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:any processContents="lax"/>
    </xs:sequence>
    <xs:attribute name="source" type="xs:anyURI"/>
    <xs:anyAttribute namespace="##other" processContents="lax"/>
  </xs:complexType>
</xs:element>
<xs:element name="documentation" id="documentation">
  <xs:annotation>
    <xs:documentation
         source="http://www.w3.org/TR/xmlschema-1/#element-documentation"/>
  </xs:annotation>
  <xs:complexType mixed="true">
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:any processContents="lax"/>
    </xs:sequence>
    <xs:attribute name="source" type="xs:anyURI"/>
    <xs:attribute ref="xml:lang"/>
    <xs:anyAttribute namespace="##other" processContents="lax"/>
  </xs:complexType>
</xs:element>
<xs:element name="annotation" id="annotation">
  <xs:annotation>
    <xs:documentation
         source="http://www.w3.org/TR/xmlschema-1/#element-annotation"/>
  </xs:annotation>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xs:openAttrs">
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element ref="xs:appinfo"/>
          <xs:element ref="xs:documentation"/>
        </xs:choice>
        <xs:attribute name="id" type="xs:ID"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:annotation>
  <xs:documentation>
 notations for use within XML Schema schemas</xs:documentation>
</xs:annotation>
<xs:notation name="XMLSchemaStructures" public="structures"
             system="http://www.w3.org/2000/08/XMLSchema.xsd"/>
<xs:notation name="XML" public="REC-xml-19980210"
             system="http://www.w3.org/TR/1998/REC-xml-19980210"/>
<xs:complexType name="anyType" mixed="true">
  <xs:annotation>
    <xs:documentation>
 Not the real urType, but as close an approximation as we can
 get in the XML representation</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:any minOccurs="0" maxOccurs="unbounded" processContents="lax"/>
  </xs:sequence>
  <xs:anyAttribute processContents="lax"/>
</xs:complexType>
</xs:schema>
```

Note: And that is the end of the schema for schema documents.

B References (normative)

XML 1.0 (Second Edition)
Extensible Markup Language (XML) 1.0, Second Edition, Tim Bray et al., eds., W3C, 6 October 2000. See http://www.w3.org/TR/2000/REC-xml-20001006

XML Schema Requirements
XML Schema Requirements , Ashok Malhotra and Murray Maloney, eds., W3C, 15 February 1999. See http://www.w3.org/TR/1999/NOTE-xml-schema-req-19990215

XML Schemas: Datatypes
XML Schema Part 2: Datatypes, Paul V. Biron and Ashok Malhotra, eds., W3C, 2 May 2001. See http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html

XML-Infoset
XML Information Set, John Cowan and Richard Tobin, eds., W3C, 16 March 2001. See http://www.w3.org/TR/2001/WD-xml-infoset-20010316/

XML-Namespaces
Namespaces in XML, Tim Bray et al., eds., W3C, 14 January 1999. See http://www.w3.org/TR/1999/REC-xml-names-19990114/

XPath
XML Path Language, James Clark and Steve DeRose, eds., W3C, 16 November 1999. See http://www.w3.org/TR/1999/REC-xpath-19991116

XPointer

XML Pointer Language (XPointer), Eve Maler and Steve DeRose, eds., W3C, 8 January 2001. See http://www.w3.org/TR/2001/WD-xptr-20010108/

C Outcome Tabulations (normative)

To facilitate consistent reporting of schema errors and ·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 clause 1.2 of Derivation Valid (Extension) (§3.4.6).

C.1 Validation Rules

cvc-assess-attr
Schema-Validity Assessment (Attribute)
cvc-assess-elt
Schema-Validity Assessment (Element)
cvc-attribute
Attribute Locally Valid
cvc-au
Attribute Locally Valid (Use)
cvc-complex-type
Element Locally Valid (Complex Type)
cvc-datatype-valid
Datatype Valid
cvc-elt
Element Locally Valid (Element)
cvc-enumeration-valid
enumeration valid
cvc-facet-valid
Facet Valid
cvc-fractionDigits-valid
fractionDigits Valid
cvc-id
Validation Root Valid (ID/IDREF)
cvc-identity-constraint
Identity-constraint Satisfied
cvc-length-valid
Length Valid
cvc-maxExclusive-valid
maxExclusive Valid
cvc-maxInclusive-valid
maxInclusive Valid
cvc-maxLength-valid
maxLength Valid
cvc-minExclusive-valid
minExclusive Valid
cvc-minInclusive-valid
minInclusive Valid
cvc-minLength-valid
minLength Valid
cvc-model-group
Element Sequence Valid
cvc-particle
Element Sequence Locally Valid (Particle)
cvc-pattern-valid
pattern valid
cvc-resolve-instance
QName resolution (Instance)
cvc-simple-type
String Valid
cvc-totalDigits-valid
totalDigits Valid
cvc-type
Element Locally Valid (Type)
cvc-wildcard
Item Valid (Wildcard)
cvc-wildcard-namespace
Wildcard allows Namespace Name

C.2 Contributions to the post-schema-validation infoset

attribute information item properties
[attribute declaration] (Attribute Declaration)
[member type definition] (Attribute Validated by Type)
[member type definition anonymous] (Attribute Validated by Type)
[member type definition name] (Attribute Validated by Type)
[member type definition namespace] (Attribute Validated by Type)
[schema default] (Attribute Validated by Type)
[schema error code] (Validation Failure (Attribute))
[schema normalized value] (Attribute Validated by Type)
[schema specified] (Assessment Outcome (Attribute))

[type definition] (Attribute Validated by Type)
[type definition anonymous] (Attribute Validated by Type)
[type definition name] (Attribute Validated by Type)
[type definition namespace] (Attribute Validated by Type)
[type definition type] (Attribute Validated by Type)
[validation attempted] (Assessment Outcome (Attribute))
[validation context] (Assessment Outcome (Attribute))
[validity] (Assessment Outcome (Attribute))

element information item properties

[element declaration] (Element Declaration)
[ID/IDREF table] (ID/IDREF Table)
[identity-constraint table] (Identity-constraint Table)
[member type definition] (Element Validated by Type)
[member type definition anonymous] (Element Validated by Type)
[member type definition name] (Element Validated by Type)
[member type definition namespace] (Element Validated by Type)
[nil] (Element Declaration)
[notation] (Validated with Notation)
[notation public] (Validated with Notation)
[notation system] (Validated with Notation)
[schema default] (Element Validated by Type)
[schema error code] (Validation Failure (Element))
[schema information] (Schema Information)
[schema normalized value] (Element Validated by Type)
[schema specified] (Element Default Value)
[type definition] (Element Validated by Type)
[type definition anonymous] (Element Validated by Type)
[type definition name] (Element Validated by Type)
[type definition namespace] (Element Validated by Type)
[type definition type] (Element Validated by Type)
[validation attempted] (Assessment Outcome (Element))
[validation context] (Assessment Outcome (Element))
[validity] (Assessment Outcome (Element))

ID/IDREF binding information item properties

[binding] (ID/IDREF Table)
[id] (ID/IDREF Table)

Identity-constraint Binding information item properties

[definition] (Identity-constraint Table)
[node table] (Identity-constraint Table)

namespace schema information information item properties

[schema components] (Schema Information)
[schema documents] (Schema Information)
[schema namespace] (Schema Information)

schema document information item properties

[document] (Schema Information)
[document location] (Schema Information)

C.3 Schema Representation Constraints

schema_reference

Schema Document Location Strategy

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-list-itemType-or-simpleType

itemType attribute or simpleType child

src-model_group

Model Group Representation OK

src-model_group_defn

Model Group Definition Representation OK

src-multiple-enumerations

Multiple enumerations

src-multiple-patterns

Multiple patterns

src-notation

Notation Definition Representation OK

src-qname

QName Interpretation

src-redefine
Redefinition Constraints and Semantics

src-resolve
QName resolution (Schema Document)

src-restriction-base-or-simpleType
base attribute or simpleType child

src-simple-type
Simple Type Definition Representation OK

src-single-facet-value
Single Facet Value

src-union-memberTypes-or-simpleTypes
memberTypes attribute or simpleType children

src-wildcard
Wildcard Representation OK

C.4 Schema Component Constraints

a-props-correct
Attribute Declaration Properties Correct

ag-props-correct
Attribute Group Definition Properties Correct

an-props-correct
Annotation Correct

au-props-correct
Attribute Use Correct

c-fields-xpaths
Fields Value OK

c-props-correct
Identity-constraint Definition Properties Correct

c-selector-xpath
Selector Value OK

cos-all-limited
All Group Limited

cos-applicable-facets
applicable facets

cos-aw-intersect
Attribute Wildcard Intersection

cos-aw-union
Attribute Wildcard Union

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-list-of-atomic
list of atomic

cos-no-circular-unions
no circular unions

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)

ct-props-correct
Complex Type Definition Properties Correct

derivation-ok-restriction
Derivation Valid (Restriction, Complex)

e-props-correct
Element Declaration Properties Correct

enumeration-required-notation
enumeration facet value required for NOTATION

enumeration-valid-restriction
enumeration valid restriction
fractionDigits-totalDigits
fractionDigits less than or equal to totalDigits
fractionDigits-valid-restriction
fractionDigits valid restriction
length-minLength-maxLength
length and minLength or maxLength
length-valid-restriction
length valid restriction
maxExclusive-valid-restriction
maxExclusive valid restriction
maxInclusive-maxExclusive
maxInclusive and maxExclusive
maxInclusive-valid-restriction
maxInclusive valid restriction
maxLength-valid-restriction
maxLength valid restriction
mg-props-correct
Model Group Correct
mgd-props-correct
Model Group Definition Properties Correct
minExclusive-less-than-equal-to-maxExclusive
minExclusive <= maxExclusive
minExclusive-less-than-maxInclusive
minExclusive < maxInclusive
minExclusive-valid-restriction
minExclusive valid restriction
minInclusive-less-than-equal-to-maxInclusive
minInclusive <= maxInclusive
minInclusive-less-than-maxExclusive
minInclusive < maxExclusive
minInclusive-minExclusive
minInclusive and minExclusive
minInclusive-valid-restriction
minInclusive valid restriction
minLength-less-than-equal-to-maxLength
minLength <= maxLength
minLength-valid-restriction
minLength valid restriction
n-props-correct
Notation Declaration Correct
no-xmlns
xmlns Not Allowed
no-xsi
xsi: Not Allowed
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)
sch-props-correct
Schema Properties Correct
st-props-correct
Simple Type Definition Properties Correct
st-restrict-facets
Simple Type Restriction (Facets)
totalDigits-valid-restriction
totalDigits valid restriction
w-props-correct
Wildcard Properties Correct
whiteSpace-valid-restriction
whiteSpace valid restriction

D Required Information Set Items and Properties (normative)

This specification requires as a precondition for ·assessment· an information set as defined in [XML-Infoset] which supports at least the following information items and properties:

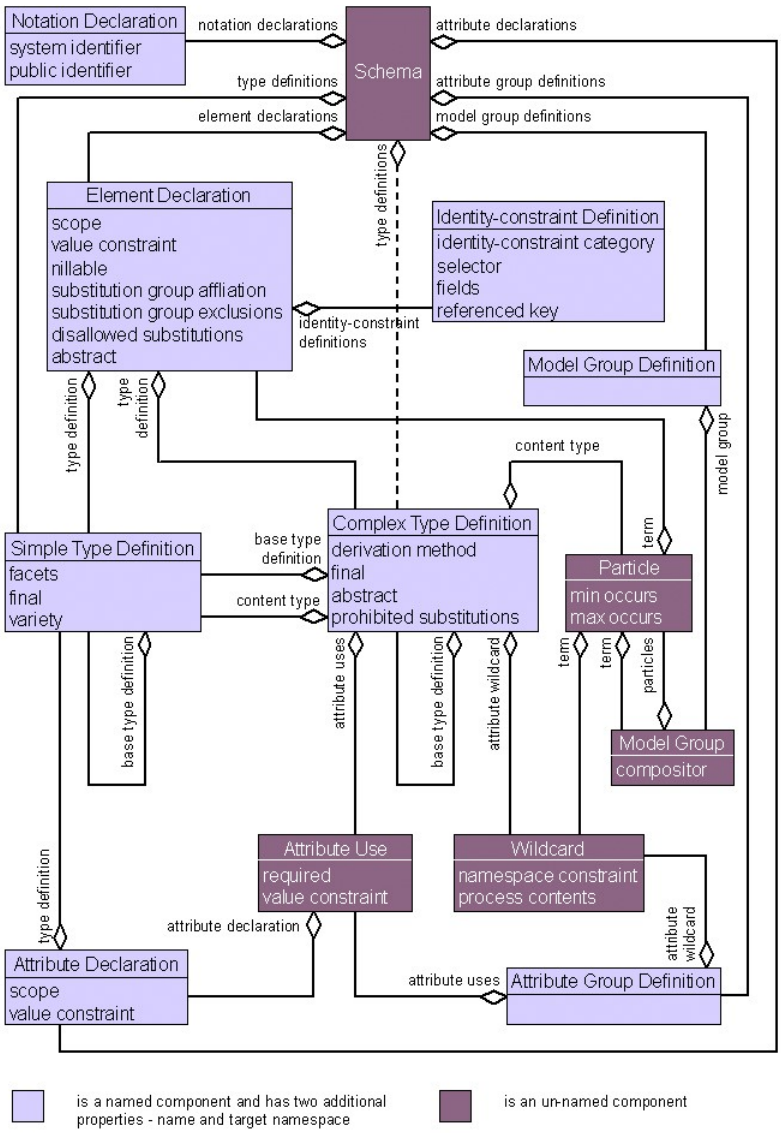
Attribute Information Item
[local name], [namespace name], [normalized value]
Character Information Item
[character code]
Element Information Item
[local name], [namespace name], [children], [attributes], [in-scope namespaces] or [namespace attributes]
Namespace Information Item
[prefix], [namespace name]

In addition, infosets should support the [unparsedEntities] property of the Document Information Item. Failure to do so will mean all items of type ENTITY or ENTITIES will fail to ·validate·.

This specification does not require any destructive alterations to the input information set: all the information set contributions specified herein are additive.

This appendix is intended to satisfy the requirements for Conformance to the [XML-Infoset] specification.

E Schema Components Diagram (non-normative)



XML Schema Component Data Model

F Glossary (non-normative)

The listing below is for the benefit of readers of a printed version of this document: it collects together all the definitions which appear in the document above.

absent
Throughout this specification, the term **absent** is used as a distinguished property value denoting absence

actual value
The phrase **actual value** is used to refer to the member of the value space of the simple type definition associated with an attribute information item which corresponds to its ·normalized value·

assessment
the word **assessment** is used to refer to the overall process of local validation, schema-validity assessment and infoset augmentation

base type definition
A type definition used as the basis for an ·extension· or ·restriction· is known as the **base type definition** of that definition

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

conformance to the XML Representation of Schemas
·Minimally conforming· processors which accept schemas represented in the form of XML documents as described in Layer 2: Schema Documents, Namespaces and Composition (§4.2) are additionally said to provide **conformance to the XML Representation of Schemas**.

content model
A particle can be used in a complex type definition to constrain the ·validation· of the [children] of an element information item; such a particle is called a **content model**

context-determined declaration
During ·validation·, associations between element and attribute information items among the [children] and [attributes] on the one hand, and element and attribute declarations on the other, are established as a side-effect. Such declarations are called the **context-determined declarations**

declaration
declaration components are associated by (qualified) name to information items being ·validated·

declared entity name
A string is a **declared entity name** if it is equal to the [name] of some unparsed entity information item in the value of the [unparsedEntities] property of the document information item at the root of the infoset containing the element or attribute information item whose ·normalized value· the string is.

definition
definition components define internal schema components that can be used in other schema components

element substitution group
Through the new mechanism of **element substitution groups**, XML Schemas provides a more powerful model supporting substitution of one named element for another

extension
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**

final
the complex type is said to be **final**, because no further derivations are possible

fully conforming
Fully conforming processors are network-enabled processors which are not only both ·minimally conforming· and ·in conformance to the XML Representation of Schemas·, but which additionally must 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 (§4.3.2).

implicitly contains
A list of particles **implicitly contains** an element declaration if a member of the list contains that element declaration in its ·substitution group·

initial value
the **initial value** of some attribute information item is the value of the [normalized value] property of that item. Similarly, the **initial value** of an element information item is the string composed of, in order, the [character code] of each character information item in the [children] of that element information item

item isomorphic to a component
by an **item isomorphic** to a component is meant an information item whose type is equivalent to the component's, 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

laxly assessed
an element information item's schema validity may be **laxly assessed** if its ·context-determined declaration· is not *skip* by ·validating· with respect to the ·ur-type definition· as per Element Locally Valid (Type) (§3.3.4)

minimally conforming
Minimally conforming processors must completely and correctly implement the ·Schema Component Constraints·, ·Validation Rules·, and ·Schema Information Set Contributions· contained in this specification

NCName
An **NCName** is a name with no colon, 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 NCName as defined in [XML Schemas: Datatypes]

normalized value
The **normalized value** of an element or attribute information item is an ·initial value· whose white space, if any, has been normalized according to the value of the whiteSpace facet of the simple type definition used in its ·validation·:

partition
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

post-schema-validation infoset
We refer to the augmented infoset which results from conformant processing as defined in this specification as the **post-schema-validation infoset**, or PSVI

QName
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 or references to them, this refers to the simple type QName as defined in [XML Schemas: Datatypes]

resolve
Whenever the word **resolve** in any form is used in this chapter in connection with a ·QName· in a schema document, the following definition QName resolution (Schema Document) (§3.15.3) should be understood

restriction

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**

schema component
Schema component is the generic term for the building blocks that comprise the abstract data model of the schema.

Schema Component Constraint
Constraints on the schema components themselves, i.e. conditions components must satisfy to be components at all. Located in the sixth sub-section of the per-component sections of Schema Component Details (§3) and tabulated in Schema Component Constraints (§C.4)

schema document
A document in this form (i.e. a <schema> element information item) is a **schema document**

Schema Information Set Contribution
Augmentations to ·post-schema-validation infoset·s expressed by schema components, which follow as a consequence of ·validation· and/or ·assessment·. Located in the fifth sub-section of the per-component sections of Schema Component Details (§3) and tabulated in Contributions to the post-schema-validation infoset (§C.2)

Schema Representation Constraint
Constraints on the representation of schema components in XML beyond those which are expressed in Schema for Schemas (normative) (§A). Located in the third sub-section of the per-component sections of Schema Component Details (§3) and tabulated in Schema Representation Constraints (§C.3)

simple ur-type definition
the **simple ur-type definition**, a special restriction of the ·ur-type definition·, whose name is **anySimpleType** in the XML Schema namespace

substitution group
Every element declaration (call this **HEAD**) in the {element declarations} of a schema defines a **substitution group**, a subset of those {element declarations}, as follows:

symbol space
this specification introduces the term **symbol space** to denote a collection of names, each of which is unique with respect to the others

target namespace
Several kinds of component have a **target namespace**, which is either ·absent· or a namespace name, also as defined by [XML-Namespaces]

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

Type Definition Hierarchy
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**

ur-type definition
A distinguished complex type definition, the **ur-type definition**, whose name is *anyType* in the XML Schema namespace, is present in each ·XML Schema·, serving as the root of the type definition hierarchy for that schema

valid
the word **valid** and its derivatives are used to refer to clause 1 above, the determination of local schema-validity

valid extension
If this constraint Derivation Valid (Extension) (§3.4.6) holds of a complex type definition, it is a **valid extension** of its {base type definition}

valid restriction
If this constraint Derivation Valid (Restriction, Complex) (§3.4.6) holds of a complex type definition, it is a **valid restriction** of its {base type definition}

valid restriction
If this constraint Derivation Valid (Restriction, Simple) (§3.14.6) holds of a simple type definition, it is a **valid restriction** of its ·base type definition·

validation root
This item, that is the element information item at which ·assessment· began, is called the **validation root**

Validation Rules
Contributions to ·validation· associated with schema components. Located in the fourth sub-section of the per-component sections of Schema Component Details (§3) and tabulated in Validation Rules (§C.1)

XML Schema
An **XML Schema** is a set of ·schema components·

G DTD for Schemas (non-normative)

The DTD for schema documents is given below. Note there is *no* implication here that `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 200102//EN"
     Official Location: http://www.w3.org/2001/XMLSchema.dtd -->
<!-- Id:  structures.dtd,v 1.1 2003/08/28 13:30:52 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 simpleType element and its constituent parts
     are defined in XML Schema: Part 2: Datatypes -->
<!ENTITY % xs-datatypes PUBLIC 'datatypes' 'datatypes.dtd' >

<!ENTITY % p 'xs:'> <!-- can be overriden in the internal subset of a
                         schema document to establish a different
                         namespace prefix -->
<!ENTITY % s ':xs'> <!-- 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">

<!-- annotation elements -->
<!ENTITY % annotation "%p;annotation">
<!ENTITY % appinfo "%p;appinfo">
<!ENTITY % documentation "%p;documentation">

<!-- 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 % annotationAttrs ''>
<!ENTITY % appinfoAttrs ''>
<!ENTITY % documentationAttrs ''>

<!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/2001/XMLSchema'
   xmlns                CDATA                  #IMPLIED
```

```
   finalDefault         %complexDerivationSet; ''
   blockDefault         %blockSet;             ''
   id                   ID                     #IMPLIED
   elementFormDefault   %formValues;           'unqualified'
   attributeFormDefault %formValues;           'unqualified'
   xml:lang             CDATA                  #IMPLIED
   %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 declaration of xmlns is a convenience for schema authors -->

<!-- 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;                      #IMPLIED
          final     %complexDerivationSet;         #IMPLIED
          block     %complexDerivationSet;         #IMPLIED
          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;  ((%annotation;)?, (%restriction;|%extension;))>
<!ATTLIST %complexContent;
          mixed (true|false) #IMPLIED
          id    ID           #IMPLIED
          %complexContentAttrs;>

<!-- restriction should use the branch defined above, not the simple
     one from part2; extension should use the full model  -->

<!ELEMENT %simpleContent;  ((%annotation;)?, (%restriction;|%extension;))>
<!ATTLIST %simpleContent;
          id    ID           #IMPLIED
          %simpleContentAttrs;>

<!-- restriction should use the simple branch from part2, not the
     one defined above; extension should have no particle  -->

<!ELEMENT %extension;  ((%annotation;)?, (%particleAndAttrs;))>
<!ATTLIST %extension;
          base  %QName;      #REQUIRED
          id    ID           #IMPLIED
          %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
            nillable           %boolean;              #IMPLIED
            substitutionGroup  %QName;                #IMPLIED
            abstract           %boolean;              #IMPLIED
            final              %complexDerivationSet; #IMPLIED
            block              %blockSet;             #IMPLIED
            default            CDATA                  #IMPLIED
            fixed              CDATA                  #IMPLIED
            form               %formValues;           #IMPLIED
            %elementAttrs;>
<!-- type and ref are mutually exclusive.
     name and ref are mutually exclusive, 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   (1)                    #IMPLIED
          maxOccurs   (1)                    #IMPLIED
          id          ID                     #IMPLIED
          %allAttrs;>

<!ELEMENT %choice; ((%annotation;)?, (%element;| %group;| %cs; | %any;)*)>
<!ATTLIST %choice;
          minOccurs   %nonNegativeInteger;   #IMPLIED
          maxOccurs   CDATA                  #IMPLIED
          id          ID                     #IMPLIED
          %choiceAttrs;>

<!ELEMENT %sequence; ((%annotation;)?, (%element;| %group;| %cs; | %any;)*)>
<!ATTLIST %sequence;
          minOccurs   %nonNegativeInteger;   #IMPLIED
          maxOccurs   CDATA                  #IMPLIED
          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'
            id              ID                     #IMPLIED
            %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'
            id              ID                 #IMPLIED
            %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) #IMPLIED
          default   CDATA         #IMPLIED
          fixed     CDATA         #IMPLIED
          form      %formValues;  #IMPLIED
          %attributeAttrs;>
<!-- type and ref are mutually exclusive.
     name and ref are mutually exclusive, one is required -->
<!-- default for use is optional when nested, none otherwise -->
<!-- default and fixed are mutually exclusive -->
<!-- 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
          refer    %QName;        #REQUIRED
          id       ID             #IMPLIED
          %keyrefAttrs;>

<!ELEMENT %selector; ((%annotation;)?)>
<!ATTLIST %selector;
          xpath %XPathExpr; #REQUIRED
          id    ID          #IMPLIED
          %selectorAttrs;>
<!ELEMENT %field; ((%annotation;)?)>
<!ATTLIST %field;
          xpath %XPathExpr; #REQUIRED
          id    ID          #IMPLIED
          %fieldAttrs;>

<!-- Schema combination mechanisms -->
<!ELEMENT %include; (%annotation;)?>
<!ATTLIST %include;
          schemaLocation %URIref; #REQUIRED
          id             ID       #IMPLIED
          %includeAttrs;>

<!ELEMENT %import; (%annotation;)?>
<!ATTLIST %import;
          namespace      %URIref; #IMPLIED
          schemaLocation %URIref; #IMPLIED
          id             ID       #IMPLIED
          %importAttrs;>

<!ELEMENT %redefine; (%annotation; | %simpleType; | %complexType; |
                      %attributeGroup; | %group;)*>
<!ATTLIST %redefine;
          schemaLocation %URIref; #REQUIRED
          id             ID       #IMPLIED
          %redefineAttrs;>

<!ELEMENT %notation; (%annotation;)?>
<!ATTLIST %notation;
          name        %NCName;    #REQUIRED
          id          ID          #IMPLIED
          public      CDATA       #REQUIRED
          system      %URIref;    #IMPLIED
          %notationAttrs;>

<!-- Annotation is either application information or documentation -->
<!-- By having these here they are available for datatypes as well
     as all the structures elements -->

<!ELEMENT %annotation; (%appinfo; | %documentation;)*>
<!ATTLIST %annotation; %annotationAttrs;>

<!-- User must define annotation elements in internal subset for this
     to work -->
<!ELEMENT %appinfo; ANY>   <!-- too restrictive -->
<!ATTLIST %appinfo;
          source     %URIref;      #IMPLIED
          id         ID         #IMPLIED
          %appinfoAttrs;>
<!ELEMENT %documentation; ANY>   <!-- too restrictive -->
<!ATTLIST %documentation;
          source     %URIref;   #IMPLIED
          id         ID         #IMPLIED
          xml:lang   CDATA      #IMPLIED
          %documentationAttrs;>

<!NOTATION XMLSchemaStructures PUBLIC
           'structures' 'http://www.w3.org/2001/XMLSchema.xsd' >
<!NOTATION XML PUBLIC
           'REC-xml-1998-0210' 'http://www.w3.org/TR/1998/REC-xml-19980210' >
```

H Analysis of the Unique Particle Attribution Constraint (non-normative)

A specification of the import of Unique Particle Attribution (§3.8.6) which does not appeal to a processing model is difficult. What follows is intended as guidance, without claiming to be complete.

[Definition:] 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 whose {name} and {target namespace} are the same as those of an element declaration 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 (§3.10.6) is not the empty set.

or

- One is a wildcard and the other an element declaration, and the {target namespace} of any member of its ·substitution group· is ·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 ·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. Determinize 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-labeled transitions from it, or a QName-labeled 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.

I References (non-normative)

DCD
Document Content Description for XML (DCD), Tim Bray et al., eds., W3C, 10 August 1998. See http://www.w3.org/TR/1998/NOTE-dcd-19980731

DDML
Document Definition Markup Language, Ronald Bourret, John Cowan, Ingo Macherius, Simon St. Laurent, eds., W3C, 19 January 1999. See http://www.w3.org/TR/1999/NOTE-ddml-19990119

SOX
Schema for Object-oriented XML, Andrew Davidson et al., eds., W3C, 1998. See http://www.w3.org/1999/07/NOTE-SOX-19990730/

SOX-2
Schema for Object-oriented XML, Version 2.0, Andrew Davidson, et al., W3C, 30 July 1999. See http://www.w3.org/TR/NOTE-SOX/

XDR
XML-Data Reduced, Charles Frankston and Henry S. Thompson, 3 July 1998. See http://www.ltg.ed.ac.uk/~ht/XMLData-Reduced.htm

XML Schema: Primer
XML Schema Part 0: Primer, David C. Fallside, ed., W3C, 2 May 2001. See http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/primer.html

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

J Acknowledgements (non-normative)

The following contributed material to the first edition of this specification:

David Fallside, IBM
Scott Lawrence, Agranat Systems
Andrew Layman, Microsoft
Eve L. Maler, Sun Microsystems
Asir S. Vedamuthu, webMethods, Inc

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.

At the time the first edition of this specification was published, the members of the XML Schema Working Group were:

- Jim Barnette, Defense Information Systems Agency (DISA)
- Paul V. Biron, Health Level Seven
- Don Box, DevelopMentor
- Allen Brown, Microsoft
- Lee Buck, TIBCO Extensibility
- Charles E. Campbell, Informix
- Wayne Carr, Intel
- Peter Chen, Bootstrap Alliance and LSU
- David Cleary, Progress Software
- Dan Connolly, W3C (*staff contact*)
- Ugo Corda, Xerox

- Roger L. Costello, MITRE
- Haavard Danielson, Progress Software
- Josef Dietl, Mozquito Technologies
- David Ezell, Hewlett-Packard Company
- Alexander Falk, Altova GmbH
- David Fallside, IBM
- Dan Fox, Defense Logistics Information Service (DLIS)
- Matthew Fuchs, Commerce One
- Andrew Goodchild, Distributed Systems Technology Centre (DSTC Pty Ltd)
- Paul Grosso, Arbortext, Inc
- Martin Gudgin, DevelopMentor
- Dave Hollander, Contivo, Inc (*co-chair*)
- Mary Holstege, Invited Expert
- Jane Hunter, Distributed Systems Technology Centre (DSTC Pty Ltd)
- Rick Jelliffe, Academia Sinica
- Simon Johnston, Rational Software
- Bob Lojek, Mozquito Technologies
- Ashok Malhotra, Microsoft
- Lisa Martin, IBM
- Noah Mendelsohn, Lotus Development Corporation
- Adrian Michel, Commerce One
- Alex Milowski, Invited Expert
- Don Mullen, TIBCO Extensibility
- Dave Peterson, Graphic Communications Association
- Jonathan Robie, Software AG
- Eric Sedlar, Oracle Corp.
- C. M. Sperberg-McQueen, W3C (*co-chair*)
- Bob Streich, Calico Commerce
- William K. Stumbo, Xerox
- Henry S. Thompson, University of Edinburgh
- Mark Tucker, Health Level Seven
- Asir S. Vedamuthu, webMethods, Inc
- Priscilla Walmsley, XMLSolutions
- Norm Walsh, Sun Microsystems
- Aki Yoshida, SAP AG
- Kongyi Zhou, Oracle Corp.

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
- David Beech, Oracle Corp.
- Gabe Beged-Dov, Rogue Wave Software
- Greg Bumgardner, Rogue Wave Software
- Dean Burson, Lotus Development Corporation
- Mike Cokus, MITRE
- Andrew Eisenberg, Progress Software
- 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)
- Andrew Layman, Microsoft
- Dmitry Lenkov, Hewlett-Packard Company
- John McCarthy, Lawrence Berkeley National Laboratory
- Murata Makoto, Xerox
- Eve Maler, Sun Microsystems
- Murray Maloney, Muzmo Communication, acting for Commerce One
- Chris Olds, Wall Data
- Frank Olken, Lawrence Berkeley National Laboratory
- Shriram Revankar, Xerox
- Mark Reinhold, Sun Microsystems
- John C. Schneider, MITRE
- Lew Shannon, NCR
- William Shea, Merrill Lynch
- Ralph Swick, W3C
- Tony Stewart, Rivcom
- Matt Timmermans, Microstar
- Jim Trezzo, Oracle Corp.
- Steph Tryphonas, Microstar

The lists given above pertain to the first edition. At the time work on this second edition was completed, the membership of the Working Group was:

- Leonid Arbouzov, Sun Microsystems
- Jim Barnette, Defense Information Systems Agency (DISA)
- Paul V. Biron, Health Level Seven
- Allen Brown, Microsoft

- Charles E. Campbell, Invited expert
- Peter Chen, Invited expert
- Tony Cincotta, NIST
- David Ezell, National Association of Convenience Stores
- Matthew Fuchs, Invited expert
- Sandy Gao, IBM
- Andrew Goodchild, Distributed Systems Technology Centre (DSTC Pty Ltd)
- Xan Gregg, Invited expert
- Mary Holstege, Mark Logic
- Mario Jeckle, DaimlerChrysler
- Marcel Jemio, Data Interchange Standards Association
- Kohsuke Kawaguchi, Sun Microsystems
- Ashok Malhotra, Invited expert
- Lisa Martin, IBM
- Jim Melton, Oracle Corp
- Noah Mendelsohn, IBM
- Dave Peterson, Invited expert
- Anli Shundi, TIBCO Extensibility
- C. M. Sperberg-McQueen, W3C (*co-chair*)
- Hoylen Sue, Distributed Systems Technology Centre (DSTC Pty Ltd)
- Henry S. Thompson, University of Edinburgh
- Asir S. Vedamuthu, webMethods, Inc
- Priscilla Walmsley, Invited expert
- Kongyi Zhou, Oracle Corp.

We note with sadness the accidental death of Mario Jeckle shortly after the completion of work on this document. In addition to those named above, several people served on the Working Group during the development of this second edition:

- Oriol Carbo, University of Edinburgh
- Tyng-Ruey Chuang, Academia Sinica
- Joey Coyle, Health Level 7
- Tim Ewald, DevelopMentor
- Nelson Hung, Corel
- Melanie Kudela, Uniform Code Council
- Matthew MacKenzie, XML Global
- Cliff Schmidt, Microsoft
- John Stanton, Defense Information Systems Agency
- John Tebbutt, NIST
- Ross Thompson, Contivo
- Scott Vorthmann, TIBCO Extensibility