W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures

W3C Recommendation 5 April 2012

This version:
http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/
Latest version:
http://www.w3.org/TR/xmlschema11-1/
Previous version:
http://www.w3.org/TR/2012/PR-xmlschema11-1-20120119/
Editors (Version 1.1):
Shudi (Sandy) Gao 高殊镝, IBM <sandygao@ca.ibm.com>
C. M. Sperberg-McQueen, Black Mesa Technologies LLC <cmsmcq@blackmesatech.com>
Henry S. Thompson, University of Edinburgh <ht@inf.ed.ac.uk>
Editors (Version 1.0):
Henry S. Thompson, University of Edinburgh <ht@inf.ed.ac.uk>
Noah Mendelsohn, IBM (retired) <nrm@arcanedomain.com>
David Beech, Oracle Corporation (retired) <davidbeech@earthlink.net>
Murray Maloney, Muzmo Communications <murray@muzmo.com>

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

See also **translations**.

This document is also available in these non-normative formats: XML, XHTML with changes since version 1.0 marked, XHTML with changes since previous Working Draft marked, Independent copy of the schema for schema documents, Independent copy of the DTD for schema documents, Independent tabulation of components and microcomponents, and List of translations.

Abstract

This document specifies the XML Schema Definition Language, which offers facilities for describing the structure and constraining the contents of XML documents, including those which exploit the XML Namespace facility. The schema language, which is itself represented in an XML vocabulary and uses namespaces, substantially reconstructs and considerably extends the capabilities found in XML document type definitions (DTDs). This specification depends on *XML Schema Definition Language 1.1 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 W3C Recommendation specifies the W3C XML Schema Definition Language (XSD) 1.1. It is here made available for use by W3C members and the public. XSD 1.1 retains all the essential features of XSD 1.0 but adds several new features to support functionality requested by users, fixes many errors in XSD 1.0, and clarifies wording.

This draft was published on 5 April 2012. The major revisions since the previous public working draft include the following:

- Some minor errors, typographic and otherwise, have been corrected.

For those primarily interested in the changes since version 1.0, the appendix Changes since version 1.0 (non-normative) (§G) is the recommended starting point. It summarizes both changes made since XSD 1.0 and some changes which were expected (and predicted in earlier drafts of this specification) but have not been made after all. Accompanying versions of this document display in color all changes to normative text since version 1.0 and since the previous Working Draft.

Comments on this document should be made in W3C's public installation of Bugzilla, specifying "XML Schema" as the product. Instructions can be found at http://www.w3.org/XML/2006/01/public-bugzilla. If access to Bugzilla is not feasible, please send your comments to the W3C XML Schema comments mailing list, www-xml-schema-comments@w3.org (archive) and note explicitly that you have not made a Bugzilla entry for the comment. Each Bugzilla entry and email message should contain only one comment.

This document has been reviewed by W3C Members, by software developers, and by other W3C groups and interested parties, and is endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited 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.

An implementation report for XSD 1.1 was prepared and used in the Director's decision to publish the previous version of this specification as a Proposed Recommendation. The Director's decision to publish this document as a W3C Recommendation is based on consideration of reviews of the Proposed Recommendation by the public and by the members of the W3C Advisory committee.

The W3C XML Schema Working Group intends to process comments made about this recommendation, with any approved changes being handled as errata to be published separately.

This document has been produced by the W3C XML Schema Working Group as part of the W3C XML Activity. The goals of XSD 1.1 are discussed in the document *Requirements for XML Schema 1.1*. 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 by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; 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) must 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/2003/03/Translations/byTechnology?technology=xmlschema.

Table of Contents

Appendices

---

1 Introduction

This document sets out the structural part of the XML Schema Definition Language.

Chapter 2 presents a Conceptual Framework (§2) for XSD, including an introduction to the nature of XSD schemas and an introduction to the XSD 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 an XSD schema for an XSD 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 Schema Documents (Structures) (normative) (§A) for the XML representation of schemas and Normative (§L.1).

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

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 Introduction to Version 1.1

The Working Group has three main goals for this version of W3C XML Schema:

- Significant improvements in simplicity of design and clarity of exposition *without* loss of backward *or* forward compatibility;
- Provision of support for versioning of XML languages defined using this specification, including the XML vocabulary specified here for use in schema documents.
- Provision of support for co-occurrence constraints, that is constraints which make the presence of an attribute or element, or the values allowable for it, depend on the value or presence of other attributes or elements.

These goals are in tension with one another. The Working Group's strategic guidelines for changes between versions 1.0 and 1.1 can be summarized as follows:

1. Support for versioning (acknowledging that this *may* be slightly disruptive to the XML transfer syntax at the margins)
2. Support for co-occurrence constraints (which will certainly involve additions to the XML transfer syntax, which will not be understood by 1.0 processors)
3. Bug fixes (unless in specific cases we decide that the fix is too disruptive for a point release)
4. Editorial changes
5. Design cleanup will possibly change behavior in edge cases

6. Non-disruptive changes to type hierarchy (to better support current and forthcoming international standards and W3C recommendations)

7. Design cleanup will possibly change component structure (changes to functionality restricted to edge cases)

8. No significant changes in existing functionality

9. No changes to XML transfer syntax except those required by version control hooks, co-occurrence constraints and bug fixes

The aim with regard to compatibility is that

- All schema documents conformant to version 1.0 of this specification [XSD 1.0 2E] should also conform to version 1.1, and should have the same ·assessment· behavior across 1.0 and 1.1 implementations (except possibly in edge cases and in the details of the resulting PSVI);

- The vast majority of schema documents conformant to version 1.1 of this specification should also conform to version 1.0, leaving aside any incompatibilities arising from support for versioning or co-occurrence constraints, and when they are conformant to version 1.0 (or are made conformant by the removal of versioning information), should have the same ·assessment· behavior across 1.0 and 1.1 implementations (again except possibly in edge cases and in the details of the resulting PSVI);

1.2 Purpose

The purpose of *XML Schema Definition Language: Structures* is to define the nature of XSD 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 XSD 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 can 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 Definition Language: 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 formalism defined here to express syntactic, structural and value constraints applicable to its document instances. The XSD 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 applications. Some applications will require constraint capabilities not expressible in this language, and so will need to perform their own additional validations.

1.3 Namespaces and Language Identifiers

1.3.1 XSD Namespaces

1.3.1.1 The Schema Namespace (xs)

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.

> **Note:** The namespace for schema documents is unchanged from version 1.0 of this specification [XSD 1.0 2E], because any schema document valid under the rules of version 1.0 has essentially the same ·assessment· semantics under this specification as it did under version 1.0 (Second Edition). There are a few exceptions to this rule, involving errors in version 1.0 of this specification which were not reparable by errata and which have therefore been fixed only in this version of this specification, not in version 1.0.
>
> **Note:** The data model used by [XPath 2.0] and other specifications, namely [XDM], makes use of type labels in the XSD namespace (`untyped`, `untypedAtomic`) which are not defined in this specification; see the [XDM] specification for details of those types.

Users of the namespaces defined here should be aware, as a matter of namespace policy, that more names in this namespace may be given definitions in future versions of this or other specifications.

1.3.1.2 The Schema Instance Namespace (xsi)

This specification defines several attributes for direct use in any XML documents, as described in Schema-Related Markup in Documents Being Validated (§2.7). These attributes are in the namespace whose name is `http://www.w3.org/2001/XMLSchema-instance`. For brevity, the text and examples in this specification use the prefix `xsi:` to stand for this namespace; in practice, any prefix can be used.

Users of the namespaces defined here should be aware, as a matter of namespace policy, that more names in this namespace may be given definitions in future versions of this or other specifications.

1.3.1.3 The Schema Versioning Namespace (vc)

The pre-processing of schema documents described in Conditional inclusion (§4.2.2) uses attributes in the namespace `http://www.w3.org/2007/XMLSchema-versioning`. For brevity, the text and examples in this specification use the prefix `vc:` to stand for this namespace; in practice, any prefix can be used.

Users of the namespaces defined here should be aware, as a matter of namespace policy, that more names in this namespace may be given definitions in future versions of this or other specifications.

1.3.2 Namespaces with Special Status

Except as otherwise specified elsewhere in this specification, if components are ·present· in a schema, or source declarations are included in an XSD schema document, for components in any of the following namespaces, then the components, or the declarations, SHOULD agree with the descriptions given in the relevant specifications and with the declarations given in any applicable XSD schema documents maintained by the

World Wide Web Consortium for these namespaces. If they do not, the effect is ·implementation-dependent· and not defined by this specification.

- http://www.w3.org/XML/1998/namespace
- http://www.w3.org/2001/XMLSchema
- http://www.w3.org/2001/XMLSchema-instance
- http://www.w3.org/2007/XMLSchema-versioning

Note: Depending on implementation details, some processors may be able to process and use (for example) variant forms of the schema for schema documents devised for specialized purposes; if so, this specification does not forbid the use of such variant components. Other processors, however, may find it impossible to validate and use alternative components for these namespaces; this specification does not require them to do so. Users who have an interest in such specialized processing should be aware of the attending interoperability problems and should exercise caution.

This flexibility does not extend to the components described in this specification or in [XML Schema: Datatypes] as being included in every schema, such as those for the primitive and other built-in datatypes. Since those components are by definition part of every schema, it is not possible to have different components with the same expanded names present in the schema without violating constraints defined elsewhere against multiple components with the same expanded names.

Components and source declarations MUST NOT specify http://www.w3.org/2000/xmlns/ as their target namespace. If they do, then the schema and/or schema document is in ·error·.

Note: Any confusion in the use, structure, or meaning of this namespace would have catastrophic effects on the interpretability of this specification.

1.3.3 Conventional Namespace Bindings

Several namespace prefixes are conventionally used in this document for notational convenience. The following bindings are assumed.

- fn bound to http://www.w3.org/2005/xpath-functions (defined in [Functions and Operators]
- html bound to http://www.w3.org/1999/xhtml
- my (in examples) bound to the target namespace of the example schema document
- rdd1 bound to http://www.rdd1.org/
- vc bound to http://www.w3.org/2007/XMLSchema-versioning (defined in this and related specifications)
- xhtml bound to http://www.w3.org/1999/xhtml
- xlink bound to http://www.w3.org/1999/xlink
- xml bound to http://www.w3.org/XML/1998/namespace (defined in [XML 1.1] and [XML Namespaces 1.1])
- xs bound to http://www.w3.org/2001/XMLSchema (defined in this and related specifications)

- `xsi` bound to `http://www.w3.org/2001/XMLSchema-instance` (defined in this and related specifications)
- `xsl` bound to `http://www.w3.org/1999/XSL/Transform`

In practice, any prefix bound to the appropriate namespace name MAY be used (unless otherwise specified by the definition of the namespace in question, as for `xml` and `xmlns`).

1.3.4 Schema Language Identifiers

Sometimes other specifications or Application Programming Interfaces (APIs) need to refer to the XML Schema Definition Language in general, sometimes they need to refer to a specific version of the language, possibly even to a version defined in a superseded draft. To make such references easy and enable consistent identifiers to be used, we provide the following URIs to identify these concepts.

`http://www.w3.org/XML/XMLSchema`

> Identifies the XML Schema Definition Language in general, without referring to a specific version of it.

`http://www.w3.org/XML/XMLSchema/v`*X*`.`*Y*

> Identifies the language described in version ***X*.*Y*** of the XSD specification. URIs of this form refer to a numbered version of the language in general. They do not distinguish among different working drafts or editions of that version. For example, `http://www.w3.org/XML/XMLSchema/v1.0` identifies XSD version 1.0 and `http://www.w3.org/XML/XMLSchema/v1.1` identifies XSD version 1.1.

`http://www.w3.org/XML/XMLSchema/v`*X*`.`*Y*`/`*N*`e`

> Identifies the language described in the ***N***-th edition of version *X*.*Y* of the XSD specification. For example, `http://www.w3.org/XML/XMLSchema/v1.0/2e` identifies the second edition of XSD version 1.0.

`http://www.w3.org/XML/XMLSchema/v`*X*`.`*Y*`/`*N*`e/yyyymmdd`

> Identifies the language described in the ***N***-th edition of version *X*.*Y* of the XSD specification published on the particular date `yyyy-mm-dd`. For example, `http://www.w3.org/XML/XMLSchema/v1.0/1e/20001024` identifies the language defined in the XSD version 1.0 Candidate Recommendation (CR) published on 24 October 2000, and `http://www.w3.org/XML/XMLSchema/v1.0/2e/20040318` identifies the language defined in the XSD version 1.0 Second Edition Proposed Edited Recommendation (PER) published on 18 March 2004.

Please see XSD Language Identifiers (non-normative) (§K) for a complete list of XML Schema Definition Language identifiers which exist to date.

1.4 Dependencies on Other Specifications

The definition of *XML Schema Definition Language: Structures* depends on the following specifications: [XML Infoset], [XML Namespaces 1.1], [XPath 2.0], and [XML Schema: 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.

[XML Schema: Datatypes] defines some datatypes which depend on definitions in [XML 1.1] and [XML Namespaces 1.1]; those definitions, and therefore the datatypes based on them, vary between version 1.0 ([XML 1.0], [Namespaces in XML 1.0]) and version 1.1 ([XML 1.1], [XML Namespaces 1.1]) of those specifications. In any given schema-validity-·assessment· episode, the choice of the 1.0 or the 1.1 definition of those datatypes is ·implementation-defined·.

Conforming implementations of this specification MAY provide either the 1.1-based datatypes or the 1.0-based datatypes, or both. If both are supported, the choice of which datatypes to use in a particular assessment episode SHOULD be under user control.

> **Note:** It is a consequence of the rule just given that implementations MAY provide the heuristic of using the 1.1 datatypes if the input is labeled as XML 1.1, and the 1.0 datatypes if the input is labeled 1.0. It should be noted however that the XML version number is not required to be present in the input to an assessment episode, and in any case the heuristic SHOULD be subject to override by users, to support cases where users wish to accept XML 1.1 input but validate it using the 1.0 datatypes, or accept XML 1.0 input and validate it using the 1.1 datatypes.
>
> **Note:** Some users will perhaps wish to accept only XML 1.1 input, or only XML 1.0 input. The rules just given ensure that conforming implementations of this specification which accept XML input MAY accept XML 1.0, XML 1.1, or both and MAY provide user control over which versions of XML to accept.

1.5 Documentation Conventions and Terminology

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

Unless otherwise noted, the entire text of this specification is normative. Exceptions include:

- notes
- sections explicitly marked non-normative
- examples and their commentary
- informal descriptions of the consequences of rules formally and normatively stated elsewhere (such informal descriptions are typically introduced by phrases like "Informally, ..." or "It is a consequence of ... that ...")

Explicit statements that some material is normative are not to be taken as implying that material not so described is non-normative (other than that mentioned in the list just given).

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}`
A Component component. Required.
An example 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}.

For a given component ***C***, an expression of the form "***C***.{example property}" denotes the (value of the) property {example property} for component ***C***. The leading "***C***." (or more) is sometimes omitted, if the identity of the component and any other omitted properties is understood from the context. This "dot operator" is left-associative, so "***C***.{p1}.{p2}" means the same as "(***C***.{p1}) . {p2}" and denotes the value of property {p2} within the component or ‘property record’ which itself is the value of ***C***'s {p1} property. White space on either side of the dot operator has no significance and is used (rarely) solely for legibility.

For components C_1 and C_2, an expression of the form "C_1 . {example property 1} = C_2 . {example property 2}" means that C_1 and C_2 have the same value for the property (or properties) in question. Similarly, "C_1 = C_2" means that C_1 and C_2 are identical, and "C_1.{example property} = C_2" that C_2 is the value of C_1.{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 determines which of several different components corresponds to the source declaration, 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 Schema: 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 Schema Documents (Structures) (normative) (§A). In the case of apparent conflict, the Schema for Schema Documents (Structures) (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>.

Unless otherwise specified, references to attribute values are references to the ·actual value· of the attribute information item in question, not to its ·normalized value· or to other forms or varieties of "value" associated with it. For a given element information item ***E***, expressions of the form "***E*** has `attl` = ***V***" are short-hand for "there is an attribute information item named `attl` among the [attributes] of ***E*** and its ·actual value· is ***V***." If the identity of ***E*** is clear from context, expressions of the form "`attl` = ***V***" are sometimes used. The form "`attl` ≠ ***V***" is also used to specify that the ·actual value· of `attl` is *not* ***V***.

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 "dot operator" described above for components and their properties is also used for information items and their properties. For a given information item ***I***, an expression of the form "***I*** . [new property]" denotes the (value of the) property [new property] for item ***I***.

Lists of normative constraints are typically introduced with phrase like "all of the following are true" (or "... apply"), "one of the following is true", "at least one of the following is true", "one or more of the following is true", "the appropriate case among the following is true", etc. The phrase "one of the following is true" is used in cases where the authors believe the items listed to be mutually exclusive (so that the distinction between "exactly one" and "one or more" does not arise). If the items in such a list are not in fact mutually exclusive, the phrase "one of the following" should be interpreted as meaning "one or more of the following". The phrase "the appropriate case among the following" is used only when the cases are thought by the authors to be mutually exclusive; if the cases in such a list are not in fact mutually exclusive, the first applicable case should be taken. Once a case has been encountered with a true condition, subsequent cases MUST not be tested.

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

Note: General comments directed to all readers.

Within normative prose in this specification, the words MAY, SHOULD, MUST and MUST NOT are defined as follows:

MAY

Schemas, schema documents, and processors are permitted to but need not behave as described.

SHOULD

It is recommended that schemas, schema documents, and processors behave as described, but there can be valid reasons for them not to; it is important that the full implications be understood and carefully weighed before adopting behavior at variance with the recommendation.

MUST

(Of schemas and schema documents:) Schemas and documents are required to behave as described; otherwise they are in ·error·.

(Of processors:) Processors are required to behave as described.

MUST NOT

Schemas, schema documents, and processors are forbidden to behave as described; schemas and documents which nevertheless do so are in ·error·.

error

A failure of a schema or schema document to conform to the rules of this specification.

Except as otherwise specified, processors MUST distinguish error-free (conforming) schemas and schema documents used in ·assessment· from those with errors; if a schema used in ·assessment· or a schema document used in constructing a schema is in error, processors MUST report the fact; if more than one is in error, it is ·implementation-dependent· whether more than one is reported as being in error. If one or more of the constraint codes given in Outcome Tabulations (normative) (§B) is applicable, it is ·implementation-dependent· how many of them, and which, are reported.

Note: Failure of an XML document to be valid against a particular schema is not (except for the special case of a schema document consulted in the course of building a schema) in itself a failure to conform to this specification and thus, for purposes of this specification, not an error.

Note: Notwithstanding the fact that (as just noted) failure to be schema-valid is not a violation of this specification and thus not strictly speaking an error as defined here, the names of the PSVI properties [schema error code] (for attributes) and [schema error code] (for elements) are retained for compatibility with other versions of this specification, and because in many applications of XSD, non-conforming documents *are* "in error" for purposes of those applications.

deprecated

A feature or construct defined in this specification described as **deprecated** is retained in this specification for compatibility with previous versions of the

specification, and but its use is not advisable and schema authors SHOULD avoid its use if possible.

Deprecation has no effect on the conformance of schemas or schema documents which use deprecated features. Since deprecated features are part of the specification, processors MUST support them, although some processors MAY choose to issue warning messages when deprecated features are encountered.

Features deprecated in this version of this specification may be removed entirely in future versions, if any.

[Definition:] user option

A choice left under the control of the user of a processor, rather than being fixed for all users or uses of the processor.

Statements in this specification that "Processors MAY at user option" behave in a certain way mean that processors MAY provide mechanisms to allow users (i.e. invokers of the processor) to enable or disable the behavior indicated. Processors which do not provide such user-operable controls MUST NOT behave in the way indicated. Processors which do provide such user-operable controls MUST make it possible for the user to disable the optional behavior.

Note: The normal expectation is that the default setting for such options will be to disable the optional behavior in question, enabling it only when the user explicitly requests it. This is not, however, a requirement of conformance: if the processor's documentation makes clear that the user can disable the optional behavior, then invoking the processor without requesting that it be disabled can be taken as equivalent to a request that it be enabled. It is required, however, that it in fact be possible for the user to disable the optional behavior.

Note: Nothing in this specification constrains the manner in which processors allow users to control user options. Command-line options, menu choices in a graphical user interface, environment variables, alternative call patterns in an application programming interface, and other mechanisms may all be taken as providing user options.

These definitions describe in terms specific to this document the meanings assigned to these terms by [IETF RFC 2119]. The specific wording follows that of [XML 1.1].

Where these terms appear without special highlighting, they are used in their ordinary senses and do not express conformance requirements. Where these terms appear highlighted within non-normative material (e.g. notes), they are recapitulating rules normatively stated elsewhere.

This specification provides a further description of error and of conformant processors' responsibilities with respect to errors in Schemas and Schema-validity Assessment (§5).

2 Conceptual Framework

This chapter gives an overview of *XML Schema Definition Language: 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 will find it most useful first to read [XML Schema: Primer] for a tutorial introduction, and only then to consult the sub-sections of Schema Component Details (§3) named *XML Representation of ...* for the details.

2.1 Overview of XSD

An XSD schema is a set 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 to specify additional information about those items and their descendants. These augmentations to the information set make explicit information that was implicitly present in the original document (or in the original document and the governing schema, taken together), such as normalized and/or default values for attributes and elements and the types of element and attribute information items. The input information set is also augmented with information about the validity of the item, or about other properties described in this specification. [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**. Conforming processors MAY provide access to some or all of the PSVI, as described in Subset of the Post-schema-validation Infoset (§C.1). The mechanisms by which processors provide such access to the PSVI are neither defined nor constrained by this specification.

[Definition:] As it is used in this specification, the term **schema-validity assessment** has three 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 XSD schema (specifically the ·governing· element or attribute declaration and/or ·governing· type definition);
2 Determining an overall validation outcome for the item by combining local schema-validity with the results of schema-validity assessments of its descendants, if any; and
3 Determining the appropriate augmentations to the infoset (and, if desired, exposing them to downstream applications in some way, to record this outcome).

Throughout this specification, [Definition:] the word **assessment** is used to refer to the overall process of local validation, recursive determination of validation outcome, and infoset augmentation, i.e. as a short form for "·schema-validity assessment·".

[Definition:] **Validation** is the process of determining whether an XML document, an element information item, or an attribute information item obeys the constraints expressed in a schema; in the context of XSD, this amounts to calculating the value of the appropriate item's [validity] property.

> **Note:** As just defined, validation produces not a binary result, but a ternary one: if the information item is ·strictly assessed·, it will be either valid or invalid, but if no applicable declaration is found, its validity will be unknown (and its [validity] property will have the value ***notKnown***). Whether in a particular application ***notKnown*** should be treated in the same way as ***invalid*** or differently is outside the scope of this specification; sometimes one choice is appropriate, sometimes the other.
>
> **Note:** In phrases such as "·validly substitutable·" and "length valid restriction", the word **valid** is used in its ordinary English sense of "conforming to some set of rules", not necessarily limited to rules expressed in an XSD schema.

In general, a **valid document** is a document whose contents obey the constraints expressed in a particular schema. Since a document may be validated against many different schemas, it is often clearer to speak of a document being valid *against a particular schema*. When this specification is used, document validity can be defined operationally in terms of the ·post-schema-validation infoset· properties on the nodes of the document (in particular [validity]). Several similar but distinct kinds of validity are usefully distinguished, for which terms are defined below in Schema-validity and documents (§2.5).

Because the [validity] property is part of the ·post-schema-validation infoset·, it should be evident that any full ·assessment· of an item by definition entails the ·validation· of that item. Conversely, since the [validity] property is recursive and depends upon many other pieces of information which are part of the ·post-schema-validation infoset·, ·validation· also typically entails at least partial ·assessment·. The processes denoted by the two terms thus overlap and are not always distinguishable; often the same process can be referred to by either term. In this specification, the term "·assessment·" is used when it is desired to stress the calculation of the complete ·post-schema-validation infoset·, including properties whose values have no effect on validity. The term "·validation·", in contrast, is used when it is desired to focus primarily on the ·validity· of the item, treating the other information generated in the process as merely incidental.

> **Note:** When there is no particular emphasis one way or the other, the choice of terms is necessarily arbitrary, or grounded in the history of this and related specifications. Historical reasons, rather than connotation, determine the use of the term "·validation·" instead of "·assessment·" in terms like "·post-schema-validation infoset·", "·validation root·", and "Validation Rules".

During ·assessment·, some or all of the element and attribute information items in the input document are associated with declarations and/or type definitions; these declarations and type definitions are then used in the ·assessment· of those items, in a recursive process. [Definition:] The declaration associated with an information item, if any, and with respect to which its validity is ·assessed· in a given assessment episode is said to **govern** the item, or to be its **governing** element or attribute declaration. Similarly the type definition with respect to which the type-validity of an item is assessed is its **governing** type definition.

> **Note:** See also the definitions of ·governing element declaration· and ·governing type definition· (for elements) and ·governing attribute declaration· and ·governing type definition· (for attributes).

2.2 XSD Abstract Data Model

This specification builds on [XML 1.1] and [XML Namespaces 1.1]. 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 1.1]) and namespace conformance (as defined in [XML Namespaces 1.1]) for all candidates for ·assessment· and for all ·schema documents·.

Just as [XML 1.1] and [XML Namespaces 1.1] can be described in terms of information items, XSD schemas can be described in terms of an abstract data model. In defining schemas in terms of an abstract data model, this specification rigorously specifies the information which MUST be available to a conforming XSD 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 make up the abstract data model of the schema. [Definition:] An **XSD schema** is a set of ·schema components·. There are several kinds of schema component, falling into three groups. The primary schema 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 schema components, are as follows:

- Attribute group definitions
- Identity-constraint definitions
- Type alternatives
- Assertions
- Model group definitions
- Notation declarations

Finally, the "helper" schema components provide small parts of other schema components; they are dependent on their context:

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

The name [Definition:] **Component** covers all the different kinds of schema component defined in this specification.

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 and in some cases MUST have and be identified by **name**s, which are NCNames as defined by [XML Namespaces 1.1].

[Definition:] Several kinds of component have a **target namespace**, which is either ·absent· or a namespace name, also as defined by [XML Namespaces 1.1]. The ·target namespace· serves to identify the namespace within which the association between the component and its name exists.

An expanded name, as defined in [XML Namespaces 1.1], is a pair consisting of a namespace name, which MAY be ·absent·, and a local name. The expanded name of any component with both a ·target namespace· property and a ·component name· property is the pair consisting of the values of those two properties. The expanded name of a declaration is used to help determine which information items will be ·governed· by the declaration.

> **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 is ·validated· with respect to an attribute declaration, a list of element information items 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 ·xs:anyType·, every ·type definition· is, by construction, either a ·restriction· or an ·extension· of some other type definition. The exception ·xs:anyType· is a ·restriction· of itself. With the exception of the loop on ·xs:anyType·, the graph of these relationships forms a tree known as the **Type Definition Hierarchy** with ·xs:anyType· as its root.

[Definition:] The type definition used as the basis for an ·extension· or ·restriction· is known as the **base type definition** of that definition. [Definition:] If a type definition ***D*** can reach a type definition ***B*** by following its base type definition chain, then ***D*** is said to

be **derived** from ***B***. In most cases, a type definition is derived from other type definitions. The only exception is ·xs:anyType·, which is derived from itself.

[Definition:] A type defined with the same constraints as its ·base type definition·, or with more, is said to be a **restriction**. The added constraints might include narrowed ranges or reduced alternatives. Given two types ***A*** and ***B***, if the definition of ***A*** is a ·restriction· of the definition of ***B***, then members of type ***A*** are always locally valid against 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**.

> **Note:** Conceptually, the definitions of ·restriction· and ·extension· overlap: given a type ***T***, a vacuous ·restriction· of ***T*** and a vacuous ·extension· of ***T*** will each accept the same inputs as valid. The syntax specified in this version of this specification, however, requires that each type be defined either as a restriction or as an extension, not both. Thus even though the vacuous extension of ***T*** accepts the same inputs as the vacuous restriction, it will not be accepted in contexts which require restrictions of ***T***.

[Definition:] A special complex type definition, (referred to in earlier versions of this specification as 'the ur-type definition') whose name is ***anyType*** in the XSD namespace, is present in each ·XSD schema·. The **definition of *anyType*** serves as default type definition for element declarations whose XML representation does not specify one.

[Definition:] A special simple type definition, whose name is ***error*** in the XSD namespace, is also present in each ·XSD schema·. The **XSD `error` type** has no valid instances. It can be used in any place where other types are normally used; in particular, it can be used in conditional type assignment to cause elements which satisfy certain conditions to be invalid.

For brevity, the text and examples in this specification often use the qualified names `xs:anyType` and `xs:error` for these type definitions. (In practice, any appropriately declared prefix can be used, as described in Schema-Related Markup in Documents Being Validated (§2.7).)

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 Schema: Datatypes]) or user-defined, is a ·restriction· of its ·base type definition·. [Definition:] A special ·restriction· of ·xs:anyType·, whose name is ***anySimpleType*** in the XSD namespace, is the root of the ·Type Definition Hierarchy· for all simple type definitions. ·xs:anySimpleType· has a lexical space containing all sequences of characters in the Universal Character Set (UCS) and a value space containing all atomic values and all finite-length lists of atomic values. As with ·xs:anyType·, this specification sometimes uses the qualified name `xs:anySimpleType` to designate this type definition. The built-in list datatypes all have ·xs:anySimpleType· as their ·base type definition·.

[Definition:] There is a further special datatype called ***anyAtomicType***, a ·restriction· of ·xs:anySimpleType·, which is the ·base type definition· of all the primitive datatypes. This type definition is often referred to simply as "`xs:anyAtomicType`". It too is considered to have an unconstrained lexical space. Its value space consists of the union of the value spaces of all the primitive datatypes.

[Definition:] Datatypes can be **constructed** from other datatypes by ***restricting*** the value space or lexical space of a {base type definition} using zero or more Constraining Facets, by specifying the new datatype as a ***list*** of items of some {item type definition}, or by defining it as a ***union*** of some specified sequence of {member type definitions}.

The mapping from lexical space to value space is unspecified for items whose type definition is ·xs:anySimpleType· or ·xs:anyAtomicType·. Accordingly this specification does not constrain processors' behavior in areas where this mapping is implicated, for example checking such items against enumerations, constructing default attributes or elements whose declared type definition is ·xs:anySimpleType· or ·xs:anyAtomicType·, checking identity constraints involving such items.

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

[XML Schema: Datatypes] provides mechanisms for defining new simple type definitions by ·restricting· some primitive or ordinary datatype. It also provides mechanisms for constructing new simple type definitions 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 ·xs:anySimpleType·.

For detailed information on simple type definitions, see Simple Type Definitions (§3.16) and [XML Schema: 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), or 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 ·xs:anyType· 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:** For the most part, this specification allows only appending, and not other kinds of extensions. This decision simplifies application processing required to cast instances from the derived type to the base type. One special case allows the extension of all-groups in ways that do not guarantee that the new material occurs only at the end of the content. Another special case is extension via Open Contents in ***interleave*** mode.

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). For an overview of identity constraints, see Identity-constraint Definition (§2.2.4.1).

2.2.2.2 Element Substitution Group

When XML vocabularies are defined using the DTD syntax defined by [XML 1.1], a reference in a content model to a particular name is satisfied only by an element in the XML document whose name and content correspond exactly to those given in the corresponding element type declaration.

> **Note:** The "element type declaration" of [XML 1.1] is not quite the same as the ·governing type definition· as defined in this specification: [XML 1.1] does not distinguish between element declarations and type definitions as distinct kinds of object in the way that this specification does. The "element type declaration" of [XML 1.1] specifies both the kinds of properties associated in this specification with element declarations and the kinds of properties associated here with (complex) type definitions.

[Definition:] Through the mechanism of **element substitution groups**, XSD provides a more powerful model than DTDs do 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 derived from 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 constrained by 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 or element 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 (§3.14).

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 or more of the particles).

Each model group denotes a set of sequences of element information items. Regarding that set of sequences as a language, the set of sequences recognized by a group ***G*** may be written ***L***(***G***). [Definition:] A model group ***G*** is said to **accept** or **recognize** the members of ***L***(***G***).

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.

The name [Definition:] **Term** is used to refer to any of the three kinds of components which can appear in particles. All ·Terms· are themselves ·Annotated Components·. [Definition:] A **basic term** is an Element Declaration or a Wildcard. [Definition:] A **basic particle** is a Particle whose {term} is a ·basic term·.

[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:** XSD ·content models· are similar to but more expressive than [XML 1.1] content models; unlike [XML 1.1], XSD does not restrict the form of ·content models· describing mixed content.

Each content model, indeed each particle and each term, denotes a set of sequences of element information items. Regarding that set of sequences as a language, the set of sequences recognized by a particle ***P*** may be written ***L***(***P***). [Definition:] A particle ***P*** is said to **accept** or **recognize** the members of ***L***(***P***). Similarly, a term ***T*** **accepts** or **recognizes** the members of ***L***(***T***).

> **Note:** The language accepted by a content model plays a role in determining whether an element information item is locally valid or not: if the appropriate content model does not accept the sequence of elements among its children, then the element information item is not locally valid. (Some additional constraints must also be met: not every sequence in ***L***(***P***) is locally valid against ***P***. See Principles of Validation against Groups (§3.8.4.2).)
>
> No assumption is made, in the definition above, that the items in the sequence are themselves valid; only the expanded names of the items in the sequence are relevant in determining whether the sequence is accepted by a particle. Their validity does affect whether their parent is (recursively) valid as well as locally valid.

If a sequence ***S*** is a member of ***L***(***P***), then it is necessarily possible to trace a path through the ·basic particles· within ***P***, with each item within ***S*** corresponding to a matching particle within ***P***. The sequence of particles within ***P*** corresponding to ***S*** is called the ·path· of ***S*** in ***P***.

> **Note:** This ·path· has nothing to do with XPath expressions. When there may otherwise be danger of confusion, the ·path· described here may be referred to as the ·match path· of ***S*** in ***P***.

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 used by 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 names and optionally on their local names.

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

2.2.4 Constraint Components

This section describes constructs which use [XPath 2.0] expressions to constrain the input document; using them, certain rules can be expressed conveniently which would be inconvenient or impossible to express otherwise. Identity-constraint definitions are

associated with element declarations; assertions are associated with type definitions; conditional type assignment using type alternatives allows the type of an element instance to be chosen based on properties of the element instance (in particular, based on the values of its attributes).

2.2.4.1 Identity-constraint Definition

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 2.0] 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).

> **Note:** In version 1.0 of this specification [XSD 1.0 2E], identity constraints used [XPath 1.0]; they now use [XPath 2.0].

2.2.4.2 Type Alternative

A Type Alternative component (type alternative for short) associates a type definition with a predicate. Type alternatives are used in conditional type assignment, in which the choice of ·governing type definition· for elements governed by a particular element declaration depends on properties of the document instance. An element declaration may have a {type table} which contains a sequence of type alternatives; the predicates on the alternatives are tested, and when a predicate is satisfied, the type definition paired with it is chosen as the element instance's ·governing type definition·.

> **Note:** The provisions for conditional type assignment are inspired by, but not identical to, those of [SchemaPath].

For detailed information on Type Alternatives, see Type Alternatives (§3.12).

2.2.4.3 Assertion

An assertion is a predicate associated with a type, which is checked for each instance of the type. If an element or attribute information item fails to satisfy an assertion associated with a given type, then that information item is not locally ·valid· with respect to that type.

For detailed information on Assertions, see Assertions (§3.13).

2.2.4.4 Overlapping Functionality of Constraint Components

Many rules that can be enforced by identity constraints and conditional type assignment can also be formulated in terms of assertions. That is, the various constructs have overlapping functionality. The three forms of constraint differ from each other in various ways which may affect the schema author's choice of formulation.

Most obviously, the ·post-schema-validation infoset· will differ somewhat, depending on which form of constraint is chosen.

Less obviously, identity constraints are associated with element declarations, while assertions are associated with type definitions. If it is desired to enforce a particular property of uniqueness or referential integrity associated with a particular element declaration ***E***, of type ***T***, the schema author may often choose either an identity constraint associated with ***E***, or an assertion associated with ***T***. One obvious difference is that elements substitutable for ***E*** are required to have types derived from ***T***, but are not required to enforce the identity constraints (or the nillability) of ***E***. If the constraint applicable to ***E*** should be enforced by elements substitutable for ***E***, it is often most convenient to formulate the constraint as an assertion on ***T***; conversely, if only some elements of type ***T*** are intended to be subject to the constraint, or if elements substitutable for ***E*** need not enforce the constraint, then it will be more convenient to formulate the rule as an identity constraint on ***E***.

Similar considerations sometimes apply to the choice between assertions and conditional type assignment.

Because identity constraints and conditional type assignment are simpler and less variable than assertions, it may be easier for software to exploit or optimize them. Assertions have greater expressive power, which means they are often convenient. The "rule of least power" applies here; it is often preferable to use a less expressive notation in preference to a more expressive one, when either will suffice. See [Rule of Least Power].

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 (§3.15).

2.3 Constraints and Validation Rules

The [XML 1.1] 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. They are located in the sixth sub-section of the per-component sections of Schema Component Details (§3) and tabulated in Schema Component Constraints (§B.4).

Schema Representation Constraint

> [Definition:] Constraints on the representation of schema components in XML beyond those which are expressed in Schema for Schema Documents (Structures) (normative) (§A). They are located in the third sub-section of the per-component sections of Schema Component Details (§3) and tabulated in Schema Representation Constraints (§B.3).

Validation Rules

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

Schema Information Set Contribution

> [Definition:] Augmentations to ·post-schema-validation infoset·s expressed by schema components, which follow as a consequence of ·assessment·. They are 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 (§B.2).

The last of these, schema information set contributions, are not as new as they might at first seem. XML validation augments the XML 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 leaves implicit.

2.4 Conformance

Note:

Within the context of this specification, conformance can be claimed for schema documents, for schemas, and for processors.

A ·schema document· conforms to this specification if and only if **all** of the following are true:

1 It is valid with respect to the top-level element declaration for <schema> in the schema specified in Schema for Schema Documents (Structures) (normative) (§A). That is, when ·assessed· using ·element-driven validation· and stipulating the declaration for <schema>, then in its ·post-schema-validation infoset·, the <schema> element has a

[validation attempted] property with value ***full*** or ***partial*** and a [validity] property with value ***valid***.

2 No element in the schema document violates any of the Schema Representation Constraints set out in Schema Representation Constraints (§B.3), unless that element has an <annotation> element as an ancestor.

Note: Because elements within <annotation> do not map to components, they are not required to obey the Schema Representation Constraints.

If the schema document is invalid only in consequence of invalid descendants of <annotation> elements, processors MAY treat the schema document as valid. It is ·implementation-defined· what effect, if any, invalid <annotation> elements have on the construction of schema components.

Note: While conformance of schema documents is (with the exception just noted) a precondition for the mapping from schema documents to schema components described in this specification, conformance of the schema documents does not guarantee that the result of that mapping will be a schema that conforms to this specification. Some constraints (e.g. the rule that there must be at most one top-level element declaration with a particular expanded name) can only be checked in the context of the schema as a whole. Because component correctness depends in part upon the other components present, the XML mapping rules defined in this specification do not always map conforming schema documents into components that satisfy all constraints. In some cases, the mapping will produce components which violate constraints imposed at the component level; in others, no component at all will be produced.

Note: In this version of this specification, Schema Representation Constraints concern only properties of the schema document which can be checked in isolation. In version 1.0 of this specification, some Schema Representation Constraints could not be checked against the schema document in isolation, and so it was not always possible to say, for a given schema document, whether it satisfied the constraints or not.

A schema conforms to this specification if and only if it consists of components which individually and collectively satisfy all the relevant constraints specified in this document, including but not limited to all the ·Schema Component Constraints·.

Note: This specification defines no API or other interface for interacting with schemas, so a conformance claim for a schema is not normally testable in any standardized way. However, if an interface is provided which enables a user to interrogate various properties of the schema and check their values, conformance can usefully be claimed for the schema.

This specification distinguishes several classes of conforming processors, which are defined in terms of the following concepts.

[Definition:] A **validator** (or **instance validator**) is a processor which ·validates· an XML instance document against a conforming schema and distinguishes between valid documents and others, for one or more of the definitions of validity (·root-validity·, ·deep validity·, or ·uniform validity·) defined below in section Schema-validity and documents (§2.5). Conforming validators MAY additionally support other definitions of validity defined in terms of the ·post-schema-validation infoset·.

[Definition:] A **schema-validity assessor** (or just **assessor**) is a processor which performs full or partial ·schema-validity assessment· of an XML instance document, element information item, or attribute information item, with reference to a conforming schema, and provides access to the entirety of the resulting ·post-schema-validation

infoset·. The means by which an ·assessor· provides access to the ·post-schema-validation infoset· is ·implementation-defined·.

[Definition:] A **general-purpose** processor is a ·validator· or ·assessor· which accepts schemas represented in the form of XML documents as described in Layer 2: Schema Documents, Namespaces and Composition (§4.2).

> **Note:** The ·Schema Representation Constraints· are to be enforced after, not before, the ·conditional-inclusion pre-processing· described in Conditional inclusion (§4.2.2) and the ·chameleon pre-processing· described in Assembling a schema for a single target namespace from multiple schema definition documents (<include>) (§4.2.3).

[Definition:] A schema processor which is not a ·general-purpose· processor is a **special-purpose** processor.

> **Note:** By separating the conformance requirements relating to the concrete syntax of ·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 conform to this specification as ·special-purpose· but not as ·general-purpose· processors.

[Definition:] **Web-aware** processors are network-enabled processors which are not only ·general-purpose· but which additionally MUST be capable of accessing schema documents from the World Wide Web as described in Representation of Schemas on the World Wide Web (§2.8) and How schema definitions are located on the Web (§4.3.2). .

> **Note:** In version 1.0 of this specification the class of ·general-purpose· processors was termed "conformant to the XML Representation of Schemas". Similarly, the class of ·Web-aware· processors was called "fully conforming".

Several important classes of processor can be defined in terms of the concepts just given:

general-purpose validators

> ·Validators· which accept arbitrary schemas expressed in the form of sets of schema documents (i.e., are ·general-purpose·); some general-purpose validators may additionally be ·Web-aware·.

general-purpose schema-validity assessors

> ·Assessors· which accept arbitrary schemas expressed in the form of sets of schema documents (i.e., are ·general-purpose·); some general-purpose assessors may additionally be ·Web-aware·.

special-purpose validators

> ·Validators· which do not accept arbitrary schemas expressed in the form of sets of schema documents
>
> > **Note:** Typically a special-purpose validator will either have a built-in (hard-coded) schema, or else will accept arbitrary schemas in some form other than schema documents.

other special-purpose tools

Processors (other than those otherwise defined) which perform some task, service, or activity which depends at least in part on the contents of some schema, and which do so in ways which are consistent with the provisions of this specification.

> **Note:** The class of ·other special-purpose tools· is not, as defined here, a particularly informative description of a piece of software. It is expected that other specifications may wish to define processes depending in part upon schemas, and to require that implementations of those processes conform to this specification; this conformance class provides a reference point for such requirements, and for such claims of conformance.

A claim that a processor conforms to this specification MUST specify to which processor classes defined here the processor belongs.

> **Note:** Although this specification provides just these standard levels of conformance, it is anticipated that other conventions can be established in the future. There is no need to modify or republish this specification to define such additional levels of conformance.

See Checklist of implementation-defined features (§E.1) and Terminology for implementation-defined features (normative) (§C) for terminology and concepts which may be helpful in defining the behavior of conforming processors and/or claiming conformance to this specification.

2.5 Schema-validity and documents

As noted above, in general a document is **valid against a particular schema** if it obeys the constraints imposed by that schema. Depending on the nature of the application and on the specific invariants to be enforced, different forms of validity may be appropriately required by an application, a specification, or other users of XSD. This section defines terminology for use in describing the requirements of applications or other technologies which use XSD schema to describe constraints on XML documents.

> **Note:** Conformance to this specification cannot be claimed for XML documents other than schema documents; this specification imposes no requirements on documents, validity-related or otherwise, and the terms defined in this section play no role in conformance to this specification. They are defined here for the convenience of users of this specification who do wish to impose specific requirements on documents.
>
> The terms defined capture some commonly used requirements, but the specification of which documents should be regarded as acceptable for a specific application, or as conforming to a given specification, is out of scope for this specification. Applications and specifications which use XSD are free to specify whatever constraints they see fit on documents; the provision of terms for the concepts identified here should not be taken to imply that other rules for document acceptability are discouraged or inappropriate.

All the terms defined below require that the document's root element be ·assessed· using either ·element-driven validation· (when the intended root element of the schema is clearly specified) or else ·strict wildcard validation· (if several different root elements are acceptable).

root-valid document

> A document is **root-valid** against a given XSD schema if and only if after ·assessment· the document's root element has [validity] = ***valid*** and [validation attempted] = ***full*** or ***partial***.

deep-valid document

A document is **deep-valid** against a given XSD schema if and only if after ·assessment· **all** of the following are true:
1 The document's root element has [validity] = ***valid***.
2 The document's root element has [validation attempted] = ***full*** or ***partial***.
3 No element in the document has [validity] = ***invalid***.
4 No attribute in the document has [validity] = ***invalid***.

Note: The second and third clauses are necessary to ensure that invalid descendants of laxly validated elements are caught; they do not cause their laxly validated ancestor to have [validity] = ***invalid***.

uniformly valid document

A document is **uniformly valid** against a given XSD schema if and only if after ·assessment· **all** of the following are true:
1 The document's root element has [validity] = ***valid***.
2 The document's root element has [validation attempted] = ***full***.

Note: See Assessment Outcome (Element) (§3.3.5.1) for the definition of the [validation attempted] property.)

It follows from the first and second clauses that every element and attribute in the document has been ·validated· and each of them is valid. (This distinguishes **uniform validity** from **deep validity**; a deep-valid document may include elements and attributes whose validity is ***notKnown***, perhaps because they are laxly ·assessed· and no declarations were found for them, or because they were ·skipped·.)

Note: The absence of error codes does not suffice to make a document valid according to any of the definitions just given; the [schema error code] property will be empty (or ·absent·) for any root element with [validity] = ***notKnown***. Validators which expose only the [schema error code] property and fail to distinguish in their behavior between [validity] = ***notKnown*** and [validity] = ***valid*** can thus easily mislead unwary users. A frequent cause of [validity] = ***notKnown*** is the failure of the element information item to ·match· any declaration in the schema.

2.6 Names and Symbol Spaces

As discussed in XSD 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. Within a given schema there are distinct symbol spaces for each kind of named definition and declaration component identified in XSD Abstract Data Model (§2.2), except that simple type definitions and complex type definitions share a symbol space. Within a given symbol space, names MUST be unique; as a consequence, each expanded name within a given symbol space uniquely identifies a single component. The same expanded name MAY however appear in more than one symbol space without conflict. For example, assuming that the namespace prefix `my` is bound to some particular namespace, both a simple type definition and a top-level element declaration can bear the name `my:abc` without conflict or necessary relation between the two. But it is not possible for both a simple type definition and a complex type definition, or two distinct top-level element declarations, to share the name `my:abc`.

Locally scoped attribute and element declarations are special with regard to symbol spaces. Their names are not included in the global symbol spaces for attribute and element names; each complex type definition defines its own attribute symbol space, and elements local to a complex type definition are constrained by Element Declarations Consistent (§3.8.6.3), but not by means of symbol spaces. Their names are not regarded as being in any particular symbol space. 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.

2.7 Schema-Related Markup in Documents Being Validated

XML Schema Definition Language: Structures defines several attributes for direct use in any XML documents. These attributes are in the schema instance namespace (`http://www.w3.org/2001/XMLSchema-instance`) described in The Schema Instance Namespace (xsi) (§1.3.1.2) above. All schema processors MUST have appropriate attribute declarations for these attributes built in, see Attribute Declaration for the 'type' attribute (§3.2.7.1), Attribute Declaration for the 'nil' attribute (§3.2.7.2), Attribute Declaration for the 'schemaLocation' attribute (§3.2.7.3) and Attribute Declaration for the 'noNamespaceSchemaLocation' attribute (§3.2.7.4).

> **Note:** As described above (Conventional Namespace Bindings (§1.3.3)), the attributes described in this section are referred to in this specification as "`xsi:type`", "`xsi:nil`", etc. This is shorthand for "an attribute information item whose [namespace name] is `http://www.w3.org/2001/XMLSchema-instance` and whose [local name] is `type`" (or `nil`, etc.).

2.7.1 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 resolution (Instance) (§3.17.6.3) for the means by which the ·QName· is associated with a type definition.

2.7.2 xsi:nil

XML Schema Definition Language: Structures introduces a mechanism for signaling that an element MUST 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 can 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.

2.7.3 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 can be used for ·assessment·. See How schema definitions are located on the Web (§4.3.2) for details on the use of these attributes.

> **Note:** The `xsi:schemaLocation` attribute typically appears in XML document instances being ·validated·; it is distinct from the `schemaLocation` attribute defined for some

elements in schema documents (which is not always a hint but sometimes a firm directive).

2.8 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 (§4.2.3)), 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, decimal) 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· MUST NOT exist, so in some cases re-entrant chains of properties will exist.

> **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".

Component properties are simply named values. Most properties have either other components or literals (that is, strings or booleans or enumerated keywords) for values, but in a few cases, where more complex values are involved, [Definition:] a property value may itself be a collection of named values, which we call a **property record**.

[Definition:] Throughout this specification, the term **absent** is used as a distinguished property value denoting absence. Again this should not be interpreted as constraining implementations, as for instance between using a ***null*** value for such properties or not representing them at all. [Definition:] A property value which is not ·absent· is **present**.

Any property not defined as optional is always 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 might 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 might 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 1.1].

> **Note:** It is ·implementation-defined· whether a schema processor uses the definition of legal character from [XML 1.1] or [XML 1.0].

3.1.2 XML Representations of Components

The principal purpose of *XML Schema Definition Language: 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 Schema Documents (Structures) (normative) (§A) and DTD for Schemas (non-normative) (§I)) and schema components. The key element information items in the XML representation of a schema are in the XSD namespace, that is their [namespace name] is `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 XSD 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.

A recurrent pattern in the XML representation of schemas may also be mentioned here. In many cases, the same element name (e.g. `element` or `attribute` or `attributeGroup`), serves both to define a particular schema component and to incorporate it by reference. In the first case the `name` attribute is required, in the second the `ref` attribute is required. These two usages are mutually exclusive, and sometimes also depend on context.

The descriptions of the XML representation of components, and the ·Schema Representation Constraints·, apply to schema documents *after*, not before, the

·conditional-inclusion pre-processing· described in Conditional inclusion (§4.2.2) and the ·chameleon pre-processing· described in Assembling a schema for a single target namespace from multiple schema definition documents (<include>) (§4.2.3).

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 expressed in the Schema for Schema Documents (Structures) (normative) (§A). Neither the correspondences described nor the XML Representation Constraints apply to elements in the Schema namespace which occur as descendants of <appinfo> or <documentation>.

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 Schema Documents (Structures) (normative) (§A), there is always a simple type definition associated with any such attribute information item. [Definition:] With reference to any string, interpreted as denoting an instance of a given datatype, the term **actual value** denotes the value to which the lexical mapping of that datatype maps the string. In the case of attributes in schema documents, the string used as the lexical representation is normally the ·normalized value· of the attribute. The associated datatype is, unless otherwise specified, the one identified in the declaration of the attribute, in the schema for schema documents; in some cases (e.g. the enumeration facet, or fixed and default values for elements and attributes) the associated datatype will be a more specific one, as specified in the appropriate XML mapping rules. The ·actual value· will often be a string, but can 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 ·assessed·.

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 will in some cases 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·, it is possible that an appropriately-named component will 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 most ·validation· purposes. For the exception to this rule, see the discussion of comments and processing instructions in Assertion Satisfied (§3.13.4.1).

[Definition:] The **normalized value** of an element or attribute information item is an ·initial value· which has been normalized according to the value of the whiteSpace facet, and the values of any other pre-lexical facets, associated with the simple type definition used in its ·validation·. The keywords for whitespace normalization have the following meanings:

preserve

> No normalization is done, the whitespace-normalized value is the ·initial 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 #x20s are collapsed to a single #x20, and initial and/or final #x20s are deleted.

Similarly, the **normalized value** of any string with respect to a given simple type definition is the string resulting from normalization using the whiteSpace facet and any other pre-lexical facets, associated with that simple type definition.

When more than one pre-lexical facet applies, the whiteSpace facet is applied first; the order in which ·implementation-defined· facets are applied is ·implementation-defined·.

If the simple type definition used in an item's ·validation· is ·xs:anySimpleType·, then the ·normalized value· MUST be determined as in the **preserve** case above.

There are three alternative validation rules which help supply the necessary background for the above: Attribute Locally Valid (§3.2.4.1) (clause 3), Element Locally Valid (Type) (§3.3.4.4) (clause 3.1.3) or Element Locally Valid (Complex Type) (§3.4.4.2) (clause 1.2).

These three levels of normalization correspond to the processing mandated in XML for element content, CDATA attribute content and tokenized attributed content, respectively. See Attribute Value Normalization in [XML 1.1] for the precedent for **replace** and **collapse** for attributes. Extending this processing to element content is necessary to ensure 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, it is possible that *further* normalization will take place, as for instance when character entity references in attribute values result in white space characters other than spaces in their ·initial value·s.

Note: The values **replace** and **collapse** may appear to provide a convenient way to "unwrap" text (i.e. undo the effects of pretty-printing and word-wrapping). In some cases, especially highly constrained data consisting of lists of artificial tokens such as part numbers or other identifiers, this appearance is correct. For natural-language data, however, the whitespace processing prescribed for these values is not only unreliable but will systematically remove the information needed to perform unwrapping correctly. For Asian scripts, for example, a correct unwrapping process will replace line boundaries not with blanks but with zero-width separators or nothing. In consequence, it is normally unwise to use these values for natural-language data, or for any data other than lists of highly constrained tokens.

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, a kind of Annotated Component**

{annotations}
: A sequence of Annotation components.

{name}
: An xs:NCName value. Required.

{target namespace}
: An xs:anyURI value. Optional.

{type definition}
: A Simple Type Definition component. Required.

{scope}
: A Scope property record. Required.

{value constraint}
: A Value Constraint property record. Optional.

{inheritable}
: An xs:boolean value. Required.

Property Record: Scope

{variety}
: One of {***global***, ***local***}. Required.

{parent}
: Either a Complex Type Definition or a Attribute Group Definition. Required if {variety} is ***local***, otherwise MUST be ·absent·

Property Record: Value Constraint

{variety}
: One of {***default***, ***fixed***}. Required.

{value}
: An ·actual value·. Required.

{lexical form}
: A character string. Required.

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

The value of each attribute validated 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.

For an attribute declaration ***A***, if ***A***.{scope}.{variety} = ***global***, then ***A*** is available for use throughout the schema. If ***A***.{scope}.{variety} = ***local***, then ***A*** is available for use only within (the Complex Type Definition or Attribute Group Definition) ***A***.{scope}.{parent}.

The {value constraint} property reproduces the functions of XML default and #FIXED attribute values. A {variety} of ***default*** specifies that the attribute is to appear unconditionally in the ·post-schema-validation infoset·, with {value} and {lexical form} used whenever the attribute is not actually present; ***fixed*** indicates that the attribute value if present MUST be equal or identical to {value}, and if absent receives {value} and {lexical form} as for ***default***. Note that it is *values* that are checked, not strings, and that the test is for either equality or identity.

See Annotations (§3.15) for information on the role of the {annotations} 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) (§3.4.4.2).)

[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 (§3.2.6.3). 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 after the appropriate ·pre-processing· and the properties of the component are given in this section.

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. When no simple type definition is referenced or provided, the default is ·`xs:anySimpleType`·, which imposes no constraints at all.

XML Representation Summary: `attribute` Element Information Item

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

An <attribute> element maps to an attribute declaration, and allows the type definition of that declaration to be specified either by reference or by explicit inclusion.

Top-level <attribute> elements (i.e. those which appear within the schema document as children of <schema> elements) produce ***global*** attribute declarations; <attribute>s within <attributeGroup> or <complexType> produce either attribute uses which contain ***global*** attribute 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 user-defined global type definition. Otherwise an anonymous <simpleType> is provided inline.

> **Note:** Children of <override> are not strictly speaking top-level declarations, but they will become top-level declarations if they override corresponding declarations in the ·target set· of their parent. See Overriding component definitions (`<override>`) (§4.2.5) for details.

Attribute information items ·validated· by a top-level declaration MUST be qualified with the {target namespace} of that declaration. If the {target namespace} 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.

The following sections specify several sets of XML mapping rules which apply in different circumstances.

- If the <attribute> element information item has <schema> as its parent, then it maps to a global Attribute Declaration as described in Mapping Rules for Global Attribute Declarations (§3.2.2.1).
- If the <attribute> element information item has <complexType> or <attributeGroup> as an ancestor, and the `ref` [attribute] is absent, and the `use` [attribute] is not `"prohibited"`, then it maps both to an Attribute Declaration and to an Attribute Use component, as described in Mapping Rules for Local Attribute Declarations (§3.2.2.2).

  On Attribute Use components, see Attribute Uses (§3.5).
- If the <attribute> element information item has <complexType> or <attributeGroup> as an ancestor, and the `ref` [attribute] is ·present·, and the `use` [attribute] is not `"prohibited"`, then it maps to an Attribute Use component, as described in Mapping Rules for References to Top-level Attribute Declarations (§3.2.2.3).
- If the <attribute> element information item has `use='prohibited'`, then it does not map to, or correspond to, any schema component at all.

  Note: The `use` attribute is not allowed on top-level <attribute> elements, so this can only happen with <attribute> elements appearing within a <complexType> or <attributeGroup> element.

3.2.2.1 Mapping Rules for Global Attribute Declarations

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

XML Mapping Summary for 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 ·`xs:anySimpleType`·.
{scope}	A Scope as follows:

Property	Value
{variety}	***global***
{parent}	·absent·

{value constraint}	If there is a `default` or a `fixed` [attribute], then a Value Constraint as follows, otherwise ·absent·.

Property	Value
{variety}	either ***default*** or ***fixed***, as appropriate
{value}	the ·actual value· (with respect to the {type definition}) of the [attribute]
{lexical form}	the ·normalized value· (with respect to the {type definition}) of the [attribute]

{inheritable}	The ·actual value· of the `inheritable` [attribute], if present, otherwise ***false***.
{annotations}	The ·annotation mapping· of the <attribute> element, as defined in XML Representation of Annotation Schema Components (§3.15.2).

3.2.2.2 Mapping Rules for Local Attribute Declarations

If the <attribute> element information item has <complexType> or <attributeGroup> as an ancestor and the `ref` [attribute] is absent, it maps both to an attribute declaration (see below) and to an attribute use with properties as follows (unless `use='prohibited'`, in which case the item corresponds to nothing at all):

XML Mapping Summary for Attribute Use Schema Component

Property	Representation
{required}	***true*** if the <attribute> element has `use` = `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 Value Constraint as follows, otherwise ·absent·.

Property	Value
{variety}	either ***default*** or ***fixed***, as appropriate
{value}	the ·actual value· of the [attribute] (with respect to {attribute declaration}.{type definition})
{lexical form}	the ·normalized value· of the [attribute] (with respect to {attribute declaration}.{type definition})

{inheritable}	The ·actual value· of the `inheritable` [attribute], if present, otherwise ***false***.
{annotations}	The same annotations as the {annotations} of the Attribute Declaration. See below.

The <attribute> element also maps to the {attribute declaration} of the attribute use, as follows:

XML Mapping Summary for Attribute Declaration Schema Component

Property	Representation
{name}	The ·actual value· of the `name` [attribute]
{target namespace}	The appropriate **case** among the following: 1 **If** `targetNamespace` is present , **then** its ·actual value·. 2 **If** `targetNamespace` is not present and **one** of the following is true 2.1 `form` = `qualified` 2.2 `form` is absent and the <schema> ancestor has `attributeFormDefault` = `qualified` **then** the ·actual value· of the `targetNamespace` [attribute] of the ancestor <schema> element information item, or ·absent· if there is none. 3 **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 ·`xs:anySimpleType`·.
{scope}	A Scope as follows:

Property	Value
{variety}	***local***
{parent}	If the <attribute> element information item has <complexType> as an ancestor, the Complex Type Definition corresponding to that item, otherwise (the <attribute> element information item is within an <attributeGroup> element information item), the Attribute Group Definition corresponding to that item.

{value constraint}	·absent·.
{inheritable}	The ·actual value· of the `inheritable` [attribute], if present, otherwise ***false***.
{annotations}	The ·annotation mapping· of the <attribute> element, as defined in XML Representation of Annotation Schema Components (§3.15.2).

3.2.2.3 Mapping Rules for References to Top-level Attribute Declarations

If the <attribute> element information item has <complexType> or <attributeGroup> as an ancestor and the `ref` [attribute] is present, it maps to an attribute use with properties as follows (unless `use='prohibited'`, in which case the item corresponds to nothing at all):

XML Mapping Summary for Attribute Use Schema Component

Property	Representation

{required}	***true*** if `use` = `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 Value Constraint as follows, otherwise ·absent·.

Property	Value
{variety}	either ***default*** or ***fixed***, as appropriate
{value}	the ·actual value· of the [attribute] (with respect to {attribute declaration}.{type definition})
{lexical form}	the ·normalized value· of the [attribute] (with respect to {attribute declaration}.{type definition})

{inheritable}	The ·actual value· of the `inheritable` [attribute], if present, otherwise {attribute declaration}.{inheritable}.
{annotations}	The ·annotation mapping· of the <attribute> element, as defined in XML Representation of Annotation Schema Components (§3.15.2).

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 schema documents, **all** of the following also apply:

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` is present, but not both.
  3.2 If `ref` is present, then all of <simpleType>, `form` and `type` are absent.
4 The `type` attribute and a <simpleType> child element MUST NOT both be present.
5 If `fixed` and `use` are both present, `use` MUST NOT have the ·actual value· `prohibited`.
6 If the `targetNamespace` attribute is present then **all** of the following MUST be true:
  6.1 The `name` attribute is present.
  6.2 The `form` attribute is absent.
  6.3 If the ancestor <schema> does not have a `targetNamespace` [attribute] or its ·actual value· is different from the ·actual value· of `targetNamespace` of <attribute>, then **all** of the following are true:
    6.3.1 <attribute> has <complexType> as an ancestor
    6.3.2 There is a <restriction> ancestor between the <attribute> and the nearest <complexType> ancestor, and the ·actual value· of the `base` [attribute] of <restriction> does not ·match· the name of ·`xs:anyType`·.

3.2.4 Attribute Declaration Validation Rules

3.2.4.1 Attribute Locally Valid

Informally, an attribute in an XML instance is locally ·valid· against an attribute declaration if and only if (a) the name of the attribute matches the name of the

declaration, (b) after whitespace normalization its ·normalized value· is locally valid against the type declared for the attribute, and (c) the attribute obeys any relevant value constraint. Additionally, for `xsi:type`, it is required that the type named by the attribute be present in the schema. A logical prerequisite for checking the local validity of an attribute against an attribute declaration is that the attribute declaration itself and the type definition it identifies both be present in the schema.

Local validity of attributes is tested as part of schema-validity ·assessment· of attributes (and of the elements on which they occur), and the result of the test is exposed in the [validity] property of the ·post-schema-validation infoset·.

A more formal statement is given in the following constraint.

Validation Rule: Attribute Locally Valid

For an attribute information item ***A*** to be locally ·valid· with respect to an attribute declaration ***D*** **all** of the following MUST be true:

1 ***D*** is not ·absent· (see Missing Sub-components (§5.3) for how this can fail to be the case) and ***D*** and ***A*** have the same expanded name.
2 ***D***.{type definition} is not ·absent·.
3 ***A***'s ·initial value· is locally ·valid· with respect to ***D***.{type definition} as per String Valid (§3.16.4).
4 If ***D***.{value constraint} is present and ***D***.{value constraint}.{variety} = ***fixed***, then ***A***'s ·actual value· is equal or identical to ***D***.{value constraint}.{value}.
5 If ***D*** is the built-in declaration for `xsi:type` (Attribute Declaration for the 'type' attribute (§3.2.7.1)), then ***A***'s ·actual value· ·resolves· to a type definition.

3.2.4.2 Governing Attribute Declaration and Governing Type Definition

[Definition:] In a given schema-validity ·assessment· episode, the ·governing· declaration of an attribute (its **governing attribute declaration**) is the first of the following which applies:

1 A declaration which was stipulated by the processor (see Assessing Schema-Validity (§5.2)).
2 Its ·context-determined declaration·;
3 A declaration ·resolved· to by its [local name] and [namespace name], provided the attribute is not ·skipped· and the processor has not stipulated a type definition at the start of ·assessment·.

If none of these applies, the attribute has no ·governing attribute declaration· (or, in equivalent words, the ·governing attribute declaration· is ·absent·).

Note: As a consequence, unless ·skipped· or stipulated otherwise, attributes named `xsi:type`, `xsi:nil`, `xsi:schemaLocation`, or `xsi:noNamespaceSchemaLocation` are always governed by their corresponding built-in declarations (see Built-in Attribute Declarations (§3.2.7)).

[Definition:] The **governing type definition** of an attribute, in a given schema-validity ·assessment· episode, is the first of the following which applies:

1 A type definition stipulated by the processor (see Assessing Schema-Validity (§5.2)).
2 The {type definition} of the ·governing attribute declaration·.

If neither of these applies, there is no ·governing type definition· (or, in equivalent words, it is ·absent·).

3.2.4.3 Schema-Validity Assessment (Attribute)

Schema-validity assessment of an attribute information item involves identifying its ·governing attribute declaration· and checking its local validity against the declaration. If the ·governing type definition· is not present in the schema, then assessment is necessarily incomplete.

Validation Rule: Schema-Validity Assessment (Attribute)

The schema-validity assessment of an attribute information item depends on its local ·validation· alone.

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 is known for it, namely its ·governing· declaration.
2 Its local ·validity· with respect to that declaration has been evaluated as per Attribute Locally Valid (§3.2.4.1).
3 Both clause 1 and clause 2 of Attribute Locally Valid (§3.2.4.1) are satisfied.

[Definition:] For attribute information items, there is no difference between assessment and strict assessment, so the attribute information item has been **strictly assessed** if and only if its schema-validity has been assessed.

3.2.5 Attribute Declaration Information Set Contributions

3.2.5.1 Assessment Outcome (Attribute)

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) (§3.2.4.3), 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 locally ·valid· as defined by Attribute Locally Valid (§3.2.4.1), **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 (§3.4.5.1) for the other possible value.

3.2.5.2 Validation Failure (Attribute)

Schema Information Set Contribution: Validation Failure (Attribute)

If and only if the local ·validity·, as defined by Attribute Locally Valid (§3.2.4.1) above, of an attribute information item has been assessed, then 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 ·invalid·, **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) (§B)) herein.
2 **otherwise** ·absent·.

3.2.5.3 Attribute Declaration

Schema Information Set Contribution: Attribute Declaration

If and only if a ·governing· declaration is known for an attribute information item then in the ·post-schema-validation infoset· the attribute information item has a property:

PSVI Contributions for attribute information items

`[attribute declaration]`
An ·item isomorphic· to the ·governing· declaration component itself.
`[schema default]`
If the attribute information item is ·attributed· to an Attribute Use then the {lexical form} of the ·effective value constraint·, otherwise the {lexical form} of the declaration's {value constraint}.

3.2.5.4 Attribute Validated by Type

Schema Information Set Contribution: Attribute Validated by Type

If and only if a ·governing type definition· is known for an attribute information item, then in the ·post-schema-validation infoset· the attribute information item has the properties:

PSVI Contributions for attribute information items

`[schema normalized value]`
If the attribute's ·normalized value· is ·valid· with respect to the ·governing type definition·, then the ·normalized value· as ·validated·, otherwise ·absent·.
`[schema actual value]`
If the [schema normalized value] is not ·absent·, then the corresponding ·actual value·; otherwise ·absent·.
`[type definition]`
An ·item isomorphic· to the ·governing type definition· component.
`[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 the {name} is not ·absent·. If the ·type definition·'s {name} property is ·absent·, then schema processors MAY, but need not, provide a value which uniquely identifies this type definition among those with the same target namespace. It is ·implementation-defined· whether a processor provides a name for such a type definition. If a processor does provide a value in this situation, the choice of what value to use is ·implementation-dependent·.

Note: The [type definition type], [type definition namespace], [type definition name], and [type definition anonymous] properties are redundant with the [type definition] property; they are defined for the convenience of implementations which wish to expose those specific properties but not the entire type definition.

If the attribute's ·initial value· is ·valid· with respect to the ·governing type definition· as defined by String Valid (§3.16.4) and the ·governing type definition· has {variety} ***union***, then there are four additional properties:

PSVI Contributions for attribute information items

`[member type definition]`
an ·item isomorphic· to the ·validating type· of the [schema actual value].

`[member type definition namespace]`
The {target namespace} of the ·validating type·.

`[member type definition anonymous]`
true if the {name} of the ·validating type· is ·absent·, otherwise ***false***.

`[member type definition name]`
The {name} of the ·validating type·, if it is not ·absent·. If it is ·absent·, schema processors MAY, but need not, provide a value unique to the definition. It is ·implementation-defined· whether a processor provides a name for such a type definition. If a processor does provide a value in this situation, the choice of what value to use is ·implementation-dependent·.

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.

If **all** of the following are true:
1 the attribute's ·normalized value· is ·valid· with respect to the ·governing type definition·;
2 **One** of the following is true:
  2.1 the ·governing type definition· has {variety} = ***list***;
  2.2 the ·governing type definition· has {variety} = ***union*** and the ·validating type· of the [schema actual value] has {variety} = ***list***;
3 the {item type definition} of the list type (from the previous clause) has {variety} = ***union***;
then there is an additional property:

PSVI Contributions for attribute information items

`[member type definitions]`
a sequence of Simple Type Definition components, with the same length as the [schema actual value], each one an ·item isomorphic· to the ·validating type· of the corresponding item in the [schema actual value].

See also Attribute Default Value (§3.4.5.1), Match Information (§3.4.5.2) and Schema Information (§3.17.5.1), which describe other information set contributions related to attribute information items.

3.2.6 Constraints on Attribute Declaration Schema Components

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

3.2.6.1 Attribute Declaration Properties Correct

Schema Component Constraint: Attribute Declaration Properties Correct

All of the following MUST be true:
1 The values of the properties of an attribute declaration are 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}, then it is a valid default with respect to the {type definition} as defined in Simple Default Valid (§3.2.6.2).

Note: The use of ID and related types together with value constraints goes beyond what is possible with XML DTDs, and SHOULD be avoided if compatibility with DTDs is desired.

3.2.6.2 Simple Default Valid

Schema Component Constraint: Simple Default Valid

For a Value Constraint ***V*** to be a valid default with respect to a Simple Type Definition ***T*** **all** of the following MUST be true:
1 ***V***.{lexical form} is ·valid· with respect to ***T*** as defined by Datatype Valid in [XML Schema: Datatypes].
2 ***V***.{lexical form} maps to ***V***.{value} in the value space of ***T***.

3.2.6.3 `xmlns` Not Allowed

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:*`.

3.2.6.4 `xsi:` Not Allowed

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 in consequence of the rule just given MUST NOT be declared.

Note: It is legal for Attribute Uses that refer to `xsi:` attributes to specify default or fixed value constraints (e.g. in a component corresponding to a schema document construct of the form `<xs:attribute ref="xsi:type" default="xs:integer"/>`), but the practice is not recommended; including such attribute uses will tend to mislead readers of the schema document, because the attribute uses would have no effect; see Element Locally Valid (Complex Type) (§3.4.4.2) and Attribute Default Value (§3.4.5.1) for details.

3.2.7 Built-in Attribute Declarations

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

3.2.7.1 `xsi:type`

The `xsi:type` attribute is used to signal use of a type other than the declared type of an element. See xsi:type (§2.7.1).

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}	A Scope as follows:

Property	Value
{variety}	***global***
{parent}	·absent·

{value constraint}	·absent·
{annotations}	· the empty sequence ·

3.2.7.2 `xsi:nil`

The `xsi:nil` attribute is used to signal that an element's content is "nil" (or "null"). See xsi:nil (§2.7.2).

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}	A Scope as follows:

Property	Value
{variety}	***global***
{parent}	·absent·

{value constraint}	·absent·
{annotations}	· the empty sequence ·

3.2.7.3 *xsi:schemaLocation*

The xsi:schemaLocation attribute is used to signal possible locations of relevant schema documents. See xsi:schemaLocation, xsi:noNamespaceSchemaLocation (§2.7.3).

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 ·xs:anySimpleType·
{facets}	·absent·
{variety}	***list***
{item type definition}	The built-in anyURI simple type definition
{annotations}	· the empty sequence ·

{scope}	A Scope as follows:

Property	Value
{variety}	***global***
{parent}	·absent·

{value constraint}	·absent·
{annotations}	· the empty sequence ·

3.2.7.4 *xsi:noNamespaceSchemaLocation*

The xsi:noNamespaceSchemaLocation attribute is used to signal possible locations of relevant schema documents. See xsi:schemaLocation, xsi:noNamespaceSchemaLocation (§2.7.3).

Attribute Declaration for the 'noNamespaceSchemaLocation' attribute

Property	Value
{name}	noNamespaceSchemaLocation
{target namespace}	http://www.w3.org/2001/XMLSchema-instance
{type definition}	The built-in anyURI simple type definition
{scope}	A Scope as follows:

Property	Value
{variety}	***global***
{parent}	·absent·

{value constraint}	·absent·
{annotations}	· the empty sequence ·

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 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: , a kind of Term

{annotations}
A sequence of Annotation components.
{name}
An xs:NCName value. Required.
{target namespace}
An xs:anyURI value. Optional.
{type definition}
A Type Definition component. Required.
{type table}
A Type Table property record. Optional.
{scope}
A Scope property record. Required.
{value constraint}
A Value Constraint property record. Optional.
{nillable}
An xs:boolean value. Required.
{identity-constraint definitions}

A set of Identity-Constraint Definition components.

{substitution group affiliations}
A set of Element Declaration components.

{substitution group exclusions}
A subset of {***extension***, ***restriction***}.

{disallowed substitutions}
A subset of {*substitution*, *extension*, *restriction*}.

{abstract}
An xs:boolean value. Required.

Property Record: Type Table

{alternatives}
A sequence of Type Alternative components.

{default type definition}
A Type Alternative component. Required.

Property Record: Scope

{variety}
One of {***global***, ***local***}. Required.

{parent}
Either a Complex Type Definition or a Model Group Definition. Required if {variety} is ***local***, otherwise MUST be ·absent·

Property Record: Value Constraint

{variety}
One of {***default***, ***fixed***}. Required.

{value}
An ·actual value·. Required.

{lexical form}
A character string. Required.

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

For an element declaration ***E***, if ***E***.{scope}.{variety} = ***global***, then ***E*** is available for use throughout the schema. If ***E***.{scope}.{variety} = ***local***, then ***E*** is available for use only within (the Complex Type Definition or Model Group Definition) ***E***.{scope}.{parent}.

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 normally required to satisfy 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·. The type definition against which an element information item is validated (its ·governing type definition·) can be different from the declared {type definition}. The {type table} property of an Element Declaration, which governs conditional type assignment, and the `xsi:type` attribute of an element information item (see xsi:type (§2.7.1)) can cause the ·governing type definition· and the declared {type definition} to be different.

If {nillable} is ***true***, then an element with no text or element content can be ·valid· despite a {type definition} which would otherwise require content, if it carries the attribute `xsi:nil` with the value `true` (see xsi:nil (§2.7.2)). Formal details of element ·validation· are described in Element Locally Valid (Element) (§3.3.4.3).

[Definition:] An element information item ***E*** is **nilled** with respect to some element declaration ***D*** if and only if **all** of the following are true:
1 ***E*** has `xsi:nil` = ***true***.
2 ***D***.{nillable} = ***true***.
If ***E*** is said to be ·nilled· without the identity of ***D*** being clear from the context, then ***D*** is assumed to be ***E***'s ·governing element declaration·.

{value constraint} establishes a default or fixed value for an element. If a {value constraint} with {variety} = ***default*** is present, and if the element being ·validated· is empty, then for purposes of calculating the [schema normalized value] and other contributions to the ·post-schema-validation infoset· the element is treated as if {value constraint}.{lexical form} was used as the content of the element. If ***fixed*** is specified, then the element's content MUST either be empty, in which case ***fixed*** behaves as ***default***, or its value MUST be equal or identical to {value constraint}.{value}.

> **Note:** When a default value is supplied and used, as described in the second sentence of the preceding paragraph, the default value is used to calculate the [schema normalized value], etc., but the actual content of the element is not changed: the element contained no character information items in the input information set, and it contains none in the PSVI.

> **Note:** The provision of defaults for elements goes beyond what is possible in XML 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).

The {substitution group affiliations} property of an element declaration indicates which ·substitution groups·, if any, it can potentially be a member of. Potential membership is transitive but not symmetric; an element declaration is a potential member of any group named in its {substitution group affiliations}, and also of any group of which any entry in its {substitution group affiliations} 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 named in the {substitution group affiliations} of other element declarations having the same declared {type definition} or some type ·derived· therefrom. The explicit values of {substitution group exclusions}, ***extension*** or ***restriction***, rule out element declarations having types whose derivation from {type definition} involves any ***extension*** steps, or ***restriction*** steps, respectively.

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 (§2.7.1) 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 MUST NOT themselves ever be used to ·validate· element content.

See Annotations (§3.15) for information on the role of the {annotations} 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 after the appropriate ·pre-processing· and the properties of the component(s) it corresponds to are given in this section.

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 = List of QName
  targetNamespace = anyURI
  type = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, ((simpleType | complexType)?, alternative*, (unique |
key | keyref)*))
</element>
```

An <element> element information item in a schema document maps to an element declaration and allows the type definition of that declaration to be specified either by reference or by explicit inclusion.

Top-level <element> elements (i.e. those which appear within the schema document as children of <schema> elements) 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 user-defined global type definition. Otherwise an anonymous <simpleType> or <complexType> is provided inline.

> **Note:** Children of <override> are not strictly speaking top-level declarations, but they will become top-level declarations if they override corresponding declarations in the ·target set· of their parent. See Overriding component definitions (<override>) (§4.2.5) for details.

Element information items ·validated· by a top-level declaration MUST be qualified with the {target namespace} of that declaration. If the {target namespace} 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}.

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. The names of locally-scoped element declarations need not be unique and thus reside in no symbol

space at all (but the element declarations are constrained by Element Declarations Consistent (§3.8.6.3)).

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 first substitution-group head named in the `substitutionGroup` [attribute], if present, otherwise ·`xs:anyType`·. 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 XML Representation of Identity-constraint Definition Schema Components (§3.11.2) for <key>, <unique> and <keyref>.

The following sections specify several sets of XML mapping rules which apply in different circumstances.

- If the <element> element information item has <schema> as its parent, it maps to an Element Declaration using the mappings described in Common Mapping Rules for Element Declarations (§3.3.2.1) and Mapping Rules for Top-Level Element Declarations (§3.3.2.2).
- If the <element> element information item has <complexType> or <group> as an ancestor, and the `ref` [attribute] is absent, and it does not have `minOccurs=maxOccurs=0`, then it maps both to a Particle, as described in Mapping Rules for Local Element Declarations (§3.3.2.3), and also to an Element Declaration, using the mappings described in Common Mapping Rules for Element Declarations (§3.3.2.1) and Mapping Rules for Local Element Declarations (§3.3.2.3).
- If the <element> element information item has <complexType> or <group> as an ancestor, and the `ref` [attribute] is present, and it does not have `minOccurs=maxOccurs=0`, then it maps to a Particle as described in References to Top-Level Element Declarations (§3.3.2.4).
- If the <element> element information item has `minOccurs=maxOccurs=0`, then it maps to no component at all.

  Note: The `minOccurs` and `maxOccurs` attributes are not allowed on top-level <element> elements, so in valid schema documents this will happen only when the <element> element information item has <complexType> or <group> as an ancestor.

3.3.2.1 Common Mapping Rules for Element Declarations

The following mapping rules apply in all cases where an <element> element maps to an Element Declaration component.

XML Mapping Summary for Element Declaration Schema Component

Property	Representation
{name}	The ·actual value· of the `name` [attribute].
{type definition}	The first of the following that applies: 1 The type definition corresponding to the <simpleType> or <complexType> element information item in the [children], if either is present.

2 The type definition ·resolved· to by the ·actual value· of the `type` [attribute], if it is present.
3 The declared {type definition} of the Element Declaration ·resolved· to by the first QName in the ·actual value· of the `substitutionGroup` [attribute], if present.
4 ·`xs:anyType`·.

{type table} — A Type Table corresponding to the <alternative> element information items among the [children], if any, as follows, otherwise ·absent·.

Property	Value
{alternatives}	A sequence of Type Alternatives, each corresponding, in order, to one of the <alternative> elements which have a `test` [attribute].
{default type definition}	Depends upon the final <alternative> element among the [children]. If it has no `test` [attribute], the final <alternative> maps to the {default type definition}; if it does have a `test` attribute, it is covered by the rule for {alternatives} and the {default type definition} is taken from the declared type of the Element Declaration. So the value of the {default type definition} is given by the appropriate **case** among the following: 1 **If** the <alternative> has no `test` [attribute], **then** a Type Alternative corresponding to the <alternative>. 2 **otherwise** (the <alternative> has a `test`) a Type Alternative with the following properties:

Property	Value
{test}	·absent·.
{type definition}	the {type definition} property of the parent Element Declaration.
{annotations}	the empty sequence.

{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 Value Constraint as follows, otherwise ·absent·. [Definition:] Use the name **effective simple type definition** for the declared {type definition}, if it is a simple type definition, or, if {type definition}.{content type}.{variety}

= ***simple***, for {type definition}.{content type}.{simple type definition}, or else for the built-in string simple type definition).

Property	Value
{variety}	either ***default*** or ***fixed***, as appropriate
{value}	the ·actual value· (with respect to the ·effective simple type definition·) of the [attribute]
{lexical form}	the ·normalized value· (with respect to the ·effective simple type definition·) of the [attribute]

{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 affiliations}	A set of the element declarations ·resolved· to by the items in the ·actual value· of the `substitutionGroup` [attribute], if present, otherwise the empty set.
{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***.
{annotations}	The ·annotation mapping· of the <element> element and any of its <unique>, <key> and <keyref> [children] with a `ref` [attribute], as defined in XML Representation of Annotation Schema Components (§3.15.2).

3.3.2.2 Mapping Rules for Top-Level Element Declarations

If the <element> element information item has <schema> as its parent, it maps to a global Element Declaration, using the mapping given in Common Mapping Rules for

Element Declarations (§3.3.2.1), supplemented by the following.

XML Mapping Summary for Element Declaration Schema Component

Property	Representation
{target namespace}	The ·actual value· of the `targetNamespace` [attribute] of the parent <schema> element information item, or ·absent· if there is none.
{scope}	A Scope as follows

Property	Value
{variety}	***global***
{parent}	·absent·

3.3.2.3 Mapping Rules for Local Element Declarations

If the <element> element information item has <complexType> or <group> as an ancestor, and the `ref` [attribute] is absent, and it does not have `minOccurs=maxOccurs=0`, then it maps both to a Particle and to a local Element Declaration which is the {term} of that Particle. The Particle is as follows:

XML Mapping Summary for 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.
{annotations}	The same annotations as the {annotations} of the {term}.

The <element> element also maps to an element declaration using the mapping rules given in Common Mapping Rules for Element Declarations (§3.3.2.1), supplemented by those below:

XML Mapping Summary for Element Declaration Schema Component

Property	Representation
{target namespace}	The appropriate **case** among the following: 1 **If** `targetNamespace` is present , **then** its ·actual value·. 2 **If** `targetNamespace` is not present and **one** of the following is true 2.1 `form` = `qualified` 2.2 `form` is absent and the <schema> ancestor has `elementFormDefault` = `qualified` **then** the ·actual value· of the `targetNamespace` [attribute] of the ancestor <schema> element information item, or ·absent· if there is none. 3 **otherwise** ·absent·.

{scope}	A Scope as follows:

Property	Value
{variety}	***local***
{parent}	If the <element> element information item has <complexType> as an ancestor, the Complex Type Definition corresponding to that item, otherwise (the <element> element information item is within a named <group> element information item), the Model Group Definition corresponding to that item.

3.3.2.4 References to Top-Level Element Declarations

If the <element> element information item has <complexType> or <group> as an ancestor, and the `ref` [attribute] is present, and it does not have `minOccurs=maxOccurs=0`, then it maps to a Particle as follows.

XML Mapping Summary for 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].
{annotations}	The ·annotation mapping· of the <element> element, as defined in XML Representation of Annotation Schema Components (§3.15.2).

3.3.2.5 Examples of Element Declarations

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 `xs:anyType`. 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.

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).

Example

The following example illustrates conditional type assignment to an element, based on the value of one of the element's attributes. Each instance of the `message` element will be assigned either to type `messageType` or to a more specific type derived from it.

The type `messageType` accepts any well-formed XML or character sequence as content, and carries a `kind` attribute which can be used to describe the kind or format of the message. The value of `kind` is either one of a few well known keywords or, failing that, any string.

```
<xs:complexType name="messageType" mixed="true">
 <xs:sequence>
  <xs:any processContents="skip" minOccurs="0" maxOccurs="unbounded"/>
 </xs:sequence>
 <xs:attribute name="kind">
  <xs:simpleType>
   <xs:union>
    <xs:simpleType>
     <xs:restriction base="xs:string">
      <xs:enumeration value="string"/>
      <xs:enumeration value="base64"/>
      <xs:enumeration value="binary"/>
      <xs:enumeration value="xml"/>
      <xs:enumeration value="XML"/>
     </xs:restriction>
    </xs:simpleType>
    <xs:simpleType>
     <xs:restriction base="xs:string"/>
    </xs:simpleType>
   </xs:union>
  </xs:simpleType>
 </xs:attribute>
 <xs:anyAttribute processContents="skip"/>
</xs:complexType>
```

Three restrictions of `messageType` are defined, each corresponding to one of the three well-known formats: `messageTypeString` for `kind="string"`, `messageTypeBase64` for `kind="base64"` and `kind="binary"`, and `messageTypeXML` for `kind="xml"` or `kind="XML"`.

```
<xs:complexType name="messageTypeString">
 <xs:simpleContent>
  <xs:restriction base="messageType">
   <xs:simpleType>
    <xs:restriction base="xs:string"/>
   </xs:simpleType>
  </xs:restriction>
 </xs:simpleContent>
</xs:complexType>

<xs:complexType name="messageTypeBase64">
 <xs:simpleContent>
```

```
  <xs:restriction base="messageType">
   <xs:simpleType>
    <xs:restriction base="xs:base64Binary"/>
   </xs:simpleType>
  </xs:restriction>
 </xs:simpleContent>
</xs:complexType>

<xs:complexType name="messageTypeXML">
 <xs:complexContent>
  <xs:restriction base="messageType">
   <xs:sequence>
    <xs:any processContents="strict"/>
   </xs:sequence>
  </xs:restriction>
 </xs:complexContent>
</xs:complexType>
```

The `message` element itself uses `messageType` both as its declared type and as its default type, and uses `test` attributes on its <alternative> [children] to assign the appropriate specialized message type to messages with the well known values for the `kind` attribute. Because the declared type and the default type are the same, the last <alternative> (without the `test` attribute) can be omitted.

```
<xs:element name="message" type="messageType">
  <xs:alternative test="@kind='string'" type="messageTypeString"/>
  <xs:alternative test="@kind='base64'" type="messageTypeBase64"/>
  <xs:alternative test="@kind='binary'" type="messageTypeBase64"/>
  <xs:alternative test="@kind='xml'"    type="messageTypeXML"/>
  <xs:alternative test="@kind='XML'"    type="messageTypeXML"/>
  <xs:alternative                       type="messageType"/>
</xs:element>
```

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 schema documents: **all** of the following MUST be true:

1 `default` and `fixed` are not both present.

2 If the item's parent is not <schema>, then **all** of the following are true:
- 2.1 One of `ref` or `name` is present, but not both.
- 2.2 If `ref` is present, then no unqualified attributes are present other than `minOccurs`, `maxOccurs`, and `id`, and no children in the Schema namespace (`xs`) other than <annotation>.

3 The <element> element does not have both a <simpleType> or <complexType> child and a `type` attribute.

4 If `targetNamespace` is present then **all** of the following are true:
- 4.1 `name` is present.
- 4.2 `form` is not present.
- 4.3 If the ancestor <schema> does not have a `targetNamespace` [attribute] or its ·actual value· is different from the ·actual value· of `targetNamespace` of <element>, then **all** of the following are true:
  - 4.3.1 <element> has <complexType> as an ancestor
  - 4.3.2 There is a <restriction> ancestor between the <element> and the nearest <complexType> ancestor, and the ·actual value· of the `base` [attribute] of <restriction> does not ·match· the name of ·`xs:anyType`·.

5 Every <alternative> element but the last has a `test` [attribute]; the last <alternative> element MAY have such an [attribute].

3.3.4 Element Declaration Validation Rules

When an element is ·assessed·, it is first checked against its ·governing element declaration·, if any; this in turn entails checking it against its ·governing type definition·. The second step is recursive: the element's [attributes] and [children] are ·assessed· in turn with respect to the declarations assigned to them by their parent's ·governing type definition·.

3.3.4.1 Selected and Instance-specified Type Definitions

The ·governing type definition· of an element is normally the declared {type definition} associated with the ·governing element declaration·, but this may be ·overridden· using conditional type assignment in the Element Declaration or using an ·instance-specified type definition·, or both. When the element is declared with conditional type assignment, the ·selected type definition· is used as the ·governing type definition· unless ·overridden· by an ·instance-specified type definition·.

[Definition:] The **selected type definition** ***S*** of an element information item ***E*** is a type definition associated with ***E*** in the following way. Let ***D*** be the ·governing element declaration· of ***E***. Then:
1 If ***D*** has a {type table}, then ***S*** is the type ·conditionally selected· for ***E*** by ***D***.{type table}.
2 If ***D*** has no {type table}, then ***S*** is ***D***.{type definition}.
If ***E*** has no ·governing element declaration·, then ***E*** has no selected type definition.

> **Note:** It is a consequence of Element Declaration Properties Correct (§3.3.6.1) that if ***D*** is valid, then ***S*** will be ·validly substitutable· for ***D***'s declared {type definition}, or else that ***S*** will be ·`xs:error`·.

[Definition:] Given a Type Table ***T*** and an element information item ***E***, ***T*** **conditionally selects** a type ***S*** for ***E*** in the following way. The {test} expressions in ***T***'s {alternatives} are evaluated, in order, until one of the Type Alternatives ·successfully selects· a type definition for ***E***, or until all have been tried without success. If any Type Alternative ·successfully selects· a type definition, none of the following Type Alternatives are tried. Then the type ***S*** **conditionally selected** for ***E*** by ***T*** is as described in the appropriate **case** among the following:
1 If at least one Type Alternative in ***T***.{alternatives} ·successfully selects· a type definition for ***E***, then ***S*** is the type definition selected by the first such Type Alternative.
2 If no Type Alternative in ***T***.{alternatives} ·successfully selects· a type definition, then ***S*** is ***T***.{default type definition}.{type definition}.

[Definition:] An **instance-specified type definition** is a type definition associated with an element information item in the following way:
1 Among the element's attribute information items is one named `xsi:type`.
2 The ·normalized value· of that attribute information item is a qualified name ·valid· with respect to the built-in QName simple type, as defined by String Valid (§3.16.4).
3 The ·actual value· (a ·QName·) ·resolves· to a type definition. It is this type definition which is the **instance-specified type definition**.

3.3.4.2 Type Override and Valid Substitutability

[Definition:] An ·instance-specified type definition· ***S*** is said to **override** another type definition ***T*** if and only if all of the following are true:
1 ***S*** is the ·instance-specified type definition· on some element information item ***E***. A ·governing element declaration· may or may not be known for ***E***.
2 ***S*** is ·validly substitutable· for ***T***, subject to the blocking keywords of the {disallowed substitutions} of ***E***'s ·governing element declaration·, if any, or ·validly substitutable

without limitation· for ***T*** (if no ·governing element declaration· is known).

Note: Typically, ***T*** would be the ·governing type definition· for ***E*** if it were not overridden. (This will be the case if ***T*** was stipulated by the processor, as described in Assessing Schema-Validity (§5.2), or ***E*** has a ·governing element declaration· and ***T*** is its declared type, or ***T*** is the ·locally declared type· of ***E***.)

Note: The use of the term "·override·" to denote the relation between an ·instance-specified type definition· ***S*** and another type ***T*** has nothing to do with the <override> element; the two mechanisms are distinct and unrelated.

[Definition:] A type definition ***S*** is **validly substitutable** for another type ***T***, subject to a set of blocking keywords ***K*** (typically drawn from the set {***substitution***, ***extension***, ***restriction***, ***list***, ***union***} used in the {disallowed substitutions} and {prohibited substitutions} of element declarations and type definitions), if and only if either

- ***S*** and ***T*** are both complex type definitions and ***S*** is validly derived from ***T*** subject to the blocking keywords in the union of ***K*** and ***T***. {prohibited substitutions}, as defined in Type Derivation OK (Complex) (§3.4.6.5)

or

- ***S*** is a complex type definition, ***T*** is a simple type definition, and ***S*** is validly ·derived· from ***T*** subject to the blocking keywords in ***K***, as defined in Type Derivation OK (Complex) (§3.4.6.5)

or

- ***S*** is a simple type definition and ***S*** is validly ·derived· from ***T*** subject to the blocking keywords in ***K***, as defined in Type Derivation OK (Simple) (§3.16.6.3).

[Definition:] If the set of keywords controlling whether a type ***S*** is ·validly substitutable· for another type ***T*** is the empty set, then ***S*** is said to be **validly substitutable** for ***T*** **without limitation** or **absolutely**. The phrase **validly substitutable**, without mention of any set of blocking keywords, means "validly substitutable without limitation".

Sometimes one type ***S*** is ·validly substitutable· for another type ***T*** only if ***S*** is derived from ***T*** by a chain of restrictions, or if ***T*** is a union type and ***S*** a member type of the union. The concept of ·valid substitutability· is appealed to often enough in such contexts that it is convenient to define a term to cover this specific case. [Definition:] A type definition ***S*** is **validly substitutable as a restriction** for another type ***T*** if and only if ***S*** is ·validly substitutable· for ***T***, subject to the blocking keywords {***extension***, ***list***, ***union***}.

3.3.4.3 Element Locally Valid (Element)

The concept of local validity of an element information item against an element declaration is an important part of the schema-validity ·assessment· of elements. (The other important part is the recursive ·assessment· of attributes and descendant elements.) Local validity partially determines the element information item's [validity] property, and fully determines the [local element validity] property, in the ·post-schema-validation infoset·.

Informally, an element is locally valid against an element declaration when:

1. The declaration is present in the schema and the name of the element matches the name of the declaration.

2. The element is declared concrete (i.e. not abstract).

3. Any `xsi:nil` attribute on the element obeys the rules. The element is allowed to have an `xsi:nil` attribute only if the element is declared nillable, and `xsi:nil = 'true'` is allowed only if the element itself is empty. If the element declaration specifies a fixed value for the element, `xsi:nil='true'` will make the element invalid.

4. Any `xsi:type` attribute present names a type which is ·validly substitutable· for the element's declared {type definition}.

5. The element's content satisfies the appropriate constraints: If the element is empty and the declaration specifies a default value, the default is checked against the appropriate type definitions. Otherwise, the content of the element is checked against the ·governing type definition·; additionally, if the element declaration specifies a fixed value, the content is checked against that value.

6. The element satisfies all the identity constraints specified on the element declaration.

7. Additionally, on the ·validation root·, document-level ID and IDREF constraints are checked.

The following validation rule gives the normative formal definition of local validity of an element against an element declaration.

Validation Rule: Element Locally Valid (Element)

For an element information item ***E*** to be locally ·valid· with respect to an element declaration ***D*** **all** of the following MUST be true:
1 ***D*** is not ·absent· and ***E*** and ***D*** have the same expanded name.
2 ***D***.{abstract} = ***false***.
3 **One** of the following is true:
  3.1 ***D***.{nillable} = ***false***, and ***E*** has no `xsi:nil` attribute.
  3.2 ***D***.{nillable} = ***true*** and **one** of the following is true
    3.2.1 ***E*** has no `xsi:nil` attribute information item.
    3.2.2 ***E*** has `xsi:nil` = `false`.
    3.2.3 ***E*** has `xsi:nil` = `true` (that is, ***E*** is ·nilled·), and **all** of the following are true:
      3.2.3.1 ***E*** has no character or element information item [children].
      3.2.3.2 ***D*** has no {value constraint} with {variety} = ***fixed***.
4 If ***E*** has an ·instance-specified type definition· ***T***, then ***T*** ·overrides· the ·selected type definition· of ***E***.
5 The appropriate **case** among the following is true:
  5.1 **If** ***D*** has a {value constraint}, and ***E*** has neither element nor character [children], and ***E*** is not ·nilled· with respect to ***D*** , **then all** of the following are true:
    5.1.1 If ***E***'s ·governing type definition· is an ·instance-specified type definition·, then ***D***.{value constraint} is a valid default for the ·governing type definition· as defined in Element Default Valid (Immediate) (§3.3.6.2).
    5.1.2 The element information item with ***D***.{value constraint}.{lexical form} used as its ·normalized value· is locally ·valid· with respect to the ·governing type definition· as defined by Element Locally Valid (Type) (§3.3.4.4).
  5.2 **If** ***D*** has no {value constraint}, or ***E*** has either element or character [children], or ***E*** is ·nilled· with respect to ***D***, **then all** of the following are true:
    5.2.1 ***E*** is locally ·valid· with respect to the ·governing type definition· as defined by Element Locally Valid (Type) (§3.3.4.4).
    5.2.2 If ***D***.{value constraint}.{variety} = ***fixed*** and ***E*** is not ·nilled· with respect to ***D***, then **all** of the following are true:
      5.2.2.1 ***E*** has no element information item [children].
      5.2.2.2 The appropriate **case** among the following is true:
        5.2.2.2.1 **If** ***E***'s ·governing type definition· is a Complex Type Definition with {content type}.{variety} = ***mixed*** , **then** the ·initial value· of ***E*** matches ***D***.

{value constraint}.{lexical form}.

5.2.2.2.2 **If** ***E***'s ·governing type definition· is a Simple Type Definition or a Complex Type Definition with {content type}.{variety} = ***simple***, **then** the ·actual value· of ***E*** is equal or identical to ***D***.{value constraint}.{value}.

6 ***E*** is ·valid· with respect to each of the {identity-constraint definitions} as per Identity-constraint Satisfied (§3.11.4).

7 If ***E*** is the ·validation root·, then it is ·valid· per Validation Root Valid (ID/IDREF) (§3.3.4.5).

Note: If an element has an `xsi:type` attribute, the type definition indicated by that attribute normally takes precedence over the ·selected type definition· which would otherwise govern the element. If an `xsi:type` attribute is present and ·resolves· to a known type definition, but fails to ·override· the ·selected type definition·, then ***E*** is not locally valid against ***D***, since ***E*** has failed to satisfy clause 4. In this case (or if `xsi:type` fails to ·resolve·), the ·governing type definition· of the element is the ·selected type definition· of its ·governing element declaration·, and the element is validated against that type as described in clause 5. The local validity of the element with respect to the ·governing type definition· is recorded in the [local type validity] property. The use of the ·selected type definition· when the ·instance-specified type definition· cannot be used allows useful validation to proceed in some cases (not all) even when the schema is incomplete. It also helps ensure consistent typing for sub-elements with the same name.

3.3.4.4 Element Locally Valid (Type)

The following validation rule specifies formally what it means for an element to be locally valid against a type definition. This concept is appealed to in the course of checking an element's local validity against its ·governing type definition·. It is also part of schema-validity ·assessment· of an element when the element is ·laxly assessed·, by checking its local validity against `xs:anyType`.

Informally, local validity against a type requires first that the type definition be present in the schema and not declared abstract. For a simple type definition, the element must lack attributes (except for namespace declarations and the special attributes in the `xsi` namespace) and child elements, and must be type-valid against that simple type definition. For a complex type definition, the element must be locally valid against that complex type definition.

Validation Rule: Element Locally Valid (Type)

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

1 ***T*** is not ·absent·;

2 If ***T*** is a complex type definition, then ***T***.{abstract} = ***false***.

3 The appropriate **case** among the following is true:

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

3.1.1 ***E***.[attributes] is empty, except for attributes named `xsi:type`, `xsi:nil`, `xsi:schemaLocation`, or `xsi:noNamespaceSchemaLocation`.

3.1.2 ***E*** has no element information item [children].

3.1.3 If ***E*** is not ·nilled·, then the ·initial value· is ·valid· with respect to ***T*** as defined by String Valid (§3.16.4).

3.2 **If** ***T*** is a complex type definition, **then** ***E*** is locally ·valid· with respect to ***T*** as per Element Locally Valid (Complex Type) (§3.4.4.2);

3.3.4.5 Validation Root Valid (ID/IDREF)

The following validation rule specifies document-level ID/IDREF constraints checked on the ·validation root· if it is an element; this rule is not checked on other elements. Informally, the requirement is that each ID identifies a single element within the ·validation root·, and that each IDREF value matches one ID.

Validation Rule: Validation Root Valid (ID/IDREF)

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

1 There is no **ID/IDREF binding** in ***E***.[ID/IDREF table] whose [binding] is the empty set.

2 There is no **ID/IDREF binding** in ***E***.[ID/IDREF table] whose [binding] has more than one member.

See ID/IDREF Table (§3.17.5.2) for the definition of **ID/IDREF binding**.

Note: The first clause above is violated when there is a reference to an undefined ID. The second is violated when there is a multiply-defined ID. The cases are separated out to ensure that distinct error codes (see Outcome Tabulations (normative) (§B)) are associated with these two cases.

Note: Since an element governed by type `xs:ID` provides a unique identifier for the element's parent element, it is not useful to have an element governed by `xs:ID` when the element has no parent element or when the parent element lies outside the scope of validation.

In the following examples, `DOC` and `Y` are governed by type ·`xs:anyType`·, the element `X` and the attribute `xml:id` are governed by `xs:ID`, and the element `Z` is governed by a complex type with simple content derived from `xs:ID`.

- In the document `<DOC><X>abcd</X></DOC>`, the ID value '`abcd`' will normally be bound to the `DOC` element. But if the `X` element is the validation root, then '`abcd`' will have no element binding, because `DOC` is outside the scope of the validation episode. So the first clause is violated and the document is invalid.

- The superficially similar case `<DOC><Y xml:id="abcd"/></DOC>` will, in contrast, be valid whether the `DOC` element or the `Y` element is the validation root. The ID/IDREF table will have one entry in either case, binding '`abcd`' to the `Y` element.

- For the document `<DOC><Z xml:id="abcd">abcd</Z></DOC>`, if `Z` is the validation root, then the ID/IDREF table for the document will have a single entry for '`abcd`' and will be valid. The single binding comes from the `xml:id` attribute; the content of `Z` produces no binding, just as the content of `X` above produces no binding.

  But if `DOC` is the validation root, then the ID/IDREF table for the document will have two entries for '`abcd`' (one, from the `xml:id` attribute, binding '`abcd`' to the `Z` element, one from the content of `Z` binding '`abcd`' to the `DOC` element) and will be invalid.

Note: Although this rule applies at the ·validation root·, in practice processors, particularly streaming processors, will perhaps wish to detect and signal the clause 2 case as it arises.

Note: This reconstruction of [XML 1.1]'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.

3.3.4.6 Schema-Validity Assessment (Element)

This section gives the top-level rule for ·assessment· of an element information item. Informally:

1. Assessment begins with the identification of a ·governing element declaration· for the element and then checks that the element is locally valid against the declaration; if no ·governing element declaration· is available, a ·governing type definition· can be used instead.

2. The element's attributes are to be ·assessed· recursively, unless they match a ***skip*** wildcard and are thus ·skipped·.

3. The element's children are to be ·assessed· recursively, unless they match a ***skip*** wildcard and are thus ·skipped·. For each child element, the ·governing element declaration· is the one identified in the course of checking the local validity of the parent, unless that declaration is not available. If the ·governing element declaration· is not available, the element may still be ·strictly assessed· if a ·governing type definition· can be identified (e.g. via the `xsi:type` attribute), otherwise the element will be ·laxly assessed·.

[Definition:] The **governing element declaration** of an element information item ***E***, in a given schema-validity ·assessment· episode, is the first of the following which applies:
1 A declaration stipulated by the processor (see Assessing Schema-Validity (§5.2)).
2 ***E***'s ·context-determined declaration·.
3 A declaration ·resolved· to by ***E***'s [local name] and [namespace name], provided that ***E*** is ·attributed· either to a ***strict*** ·wildcard particle· or to a ***lax*** ·wildcard particle·.
4 A declaration ·resolved· to by ***E***'s [local name] and [namespace name], provided that **none** of the following is true:
  4.1 ***E*** is ·skipped·
  4.2 the processor has stipulated a type definition for ***E***
  4.3 a ·non-absent· ·locally declared type· exists for ***E***
If none of these applies, ***E*** has no ·governing element declaration· (or, in equivalent words, ***E***'s ·governing element declaration· is ·absent·).

[Definition:] The **governing type definition** of an element information item ***E***, in a given schema-validity ·assessment· episode, is the first of the following which applies:
1 An ·instance-specified type definition· which ·overrides· a type definition stipulated by the processor (see Assessing Schema-Validity (§5.2)).
2 A type definition stipulated by the processor (see Assessing Schema-Validity (§5.2)).
3 An ·instance-specified type definition· which ·overrides· the ·selected type definition· of ***E***.
4 The ·selected type definition· of ***E***.
5 The value ·absent· if ***E*** is ·skipped·.
6 An ·instance-specified type definition· which ·overrides· the ·locally declared type·.
7 The ·locally declared type·.
8 An ·instance-specified type definition·.
If none of these applies, there is no ·governing type definition· (or, in equivalent words, it is ·absent·).

Validation Rule: Schema-Validity Assessment (Element)

The schema-validity assessment of an element information item ***E*** is performed as follows:
1 If ***E*** has a ·governing element declaration· or a ·governing type definition·, then ***E*** MUST be ·strictly assessed·.
2 If ***E*** is ·skipped·, then ***E*** MUST NOT be ·assessed·.
3 Otherwise, ***E*** MUST be ·laxly assessed·.

[Definition:] An element information item ***E*** is said to be **strictly assessed** if and only if **all** of the following are true:

1 **One** of the following is true:
  1.1 **All** of the following are true:
    1.1.1 A ·non-absent· element declaration is known for ***E***, namely its ·governing· declaration.
    1.1.2 ***E***'s local ·validity· with respect to that declaration has been evaluated as per Element Locally Valid (Element) (§3.3.4.3).
    1.1.3 If that evaluation involved the evaluation of Element Locally Valid (Type) (§3.3.4.4), clause 1 thereof is satisfied.
  1.2 **All** of the following are true:
    1.2.1 ***E*** does not have a ·governing element declaration·.
    1.2.2 A ·non-absent· type definition is known for ***E***, namely its ·governing type definition·
    1.2.3 The local ·validity· of ***E*** with respect to its ·governing type definition· has been evaluated as per Element Locally Valid (Type) (§3.3.4.4).
2 For each of the attribute information items among ***E***.[attributes], the appropriate **case** among the following is true:
  2.1 **If** the attribute has a ·governing attribute declaration· , **then** its schema-validity is assessed with respect to that declaration, as defined in Schema-Validity Assessment (Attribute) (§3.2.4.3).
  2.2 **otherwise** its schema-validity is not assessed.
3 For each of the element information items among its [children], the appropriate **case** among the following is true:
  3.1 **If** the child has a ·governing element declaration· or a ·governing type definition· , **then** its schema-validity is assessed with respect to that declaration or type definition, as defined in Schema-Validity Assessment (Element) (§3.3.4.6).
  3.2 **If** the child is ·attributed to· a ***skip*** Wildcard , **then** its schema-validity is not assessed.
  3.3 **otherwise** its schema-validity is ·laxly assessed· with respect to ·`xs:anyType`·.

[Definition:] The schema validity of an element information item ***E*** is said to be **laxly assessed** if and only if **both** of the following are true:

1 ***E*** has neither a ·governing element declaration· nor a ·governing type definition·.
2 ***E*** is locally ·validated· with respect to ·`xs:anyType`· as defined in Element Locally Valid (Type) (§3.3.4.4), and the schema-validity of ***E***'s [attributes] and [children] is assessed as described in clause 2 and clause 3 of Schema-Validity Assessment (Element) (§3.3.4.6).

> **Note:** It follows from the definitions given that no element information item can be both ·strictly assessed· and ·laxly assessed· in the same schema-validity ·assessment· episode.

3.3.5 Element Declaration Information Set Contributions

3.3.5.1 Assessment Outcome (Element)

Schema Information Set Contribution: Assessment Outcome (Element)

If and only if the schema-validity of an element information item has been assessed as per Schema-Validity Assessment (Element) (§3.3.4.6), 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 **One** of the following is true:

1.1.1.1 clause 1.1 of Schema-Validity Assessment (Element) (§3.3.4.6) applied and the item was locally ·valid· as defined by Element Locally Valid (Element) (§3.3.4.3);

1.1.1.2 clause 1.2 of Schema-Validity Assessment (Element) (§3.3.4.6) applied and the item was locally ·valid· as defined by Element Locally Valid (Type) (§3.3.4.4).

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) which is ·attributed· to a ***strict*** ·wildcard particle· and 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***.

3.3.5.2 Validation Failure (Element)

Schema Information Set Contribution: Validation Failure (Element)

If and only if the local ·validity·, as defined by Element Locally Valid (Element) (§3.3.4.3) above and/or Element Locally Valid (Type) (§3.3.4.4) below, of an element information item has been assessed, then 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 ·invalid·, **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) (§B)) herein.

2 **otherwise** ·absent·.

`[subsequence-valid]`

The appropriate **case** among the following:

1 **If** the element information item is locally ·invalid·, because unexpected attributes or elements were found among its [attributes] and [children] **and** clause 1 of Element Locally Valid (Complex Type) (§3.4.4.2) would be satisfied, if those unexpected attributes and children (those with [match information] = ***none***) were removed, **then *true***

2 **otherwise** ***false***

[failed identity constraints]
A list of Identity-Constraint Definitions that are not satisfied by the element information item, as defined by Identity-constraint Satisfied (§3.11.4).

Note: In principle, the value of this property includes all of the Identity-Constraint Definitions which are not satisfied for this element item; in practice, some processors will expose a subset of the items in this value, rather than the full value. For example, a processor could choose not to check further identity constraints after detecting the first failure.

[failed assertions]
A list of Assertions that are not satisfied by the element information item, as defined by Assertion Satisfied (§3.13.4.1).

Note: In principle, the value of this property includes all of the Assertions which are not satisfied by this element item; in practice, some processors will expose a subset of the items in this value, rather than the full value. For example, a processor could choose not to check further assertions after detecting the first failure.

3.3.5.3 Element Declaration

Schema Information Set Contribution: Element Declaration

If and only if a ·governing element declaration· is known for an element information item, then in the ·post-schema-validation infoset· the element information item has the properties:

PSVI Contributions for element information items

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

[nil]
true if clause 3.2.3 of Element Locally Valid (Element) (§3.3.4.3) above is satisfied, otherwise ***false***

[expected element declaration]
if the element information item is ·attributed· to an ·element particle·, then the {term} of that Particle, otherwise ·absent·

Note: The [element declaration] either is the same as or is in the ·substitution group· of the [expected element declaration].

[declared type]
an ·item isomorphic· to the declared {type definition} of the ·governing element declaration·

[local element validity]
The appropriate **case** among the following:
1 **If** the item was locally ·valid· as defined by Element Locally Valid (Element) (§3.3.4.3), **then** ***valid***
2 **otherwise** (the item was locally ·invalid· as defined by Element Locally Valid (Element) (§3.3.4.3)) ***invalid***.

3.3.5.4 Element Validated by Type

Schema Information Set Contribution: Element Validated by Type

If and only if a ·governing type definition· is known for an element information item, then in the ·post-schema-validation infoset· the item has the properties:

PSVI Contributions for element information items

[schema normalized value]
The appropriate **case** among the following:
1 **If** the element information item is not ·nilled· and either the ·governing type definition· is a simple type definition or its {content type} has {variety} ***simple***, **then** the appropriate **case** among the following:
1.1 **If** clause 5.1 of Element Locally Valid (Element) (§3.3.4.3) above has applied, **then** the {lexical form} of the {value constraint}
1.2 **If** clause 5.1 of Element Locally Valid (Element) (§3.3.4.3) above has *not* applied and the element's ·initial value· is ·valid· with respect to the simple type definition as defined by String Valid (§3.16.4), **then** the ·normalized value· of the item as ·validated·
1.3 **otherwise** ·absent·.
2 **otherwise** ·absent·.

[schema actual value]
If the [schema normalized value] is not ·absent·, then the corresponding ·actual value· ; otherwise ·absent·.

[type definition]
An ·item isomorphic· to the ·governing type definition· component itself.

[type definition type]
simple or ***complex***, depending on the [type definition].

[type definition namespace]
[type definition].{target namespace}.

[type definition anonymous]
true if [type definition].{name} is ·absent·, otherwise ***false***.

[type definition name]
If [type definition].{name} is not ·absent·, then [type definition].{name}, otherwise schema processors MAY, but need not, provide a value unique to the definition. It is ·implementation-defined· whether a processor provides a name for such a type definition. If a processor does provide a value in this situation, the choice of what value to use is ·implementation-dependent·.

[type fallback]
A keyword indicating whether the expected type definition was unavailable and the element had a fallback type as its ·governing type definition·

- ***declared*** if the element information item has a ·governing element declaration· which has no {type table}, and also an `xsi:type` attribute which fails to ·resolve· to a type definition that ·overrides· the declared {type definition}
- ***selected*** if the element information item has a ·governing element declaration· with a {type table} and also has an `xsi:type` attribute which fails to ·resolve· to a type definition that ·overrides· the ·selected type definition·
- ***lax*** if the element was ·laxly assessed· using ·`xs:anyType`·
- ***none*** otherwise

[type alternative]
If the element's ·governing element declaration· does not have a {type table}, then ·absent·; otherwise the first Type Alternative that ·successfully

selected· the element's ·selected type definition·, if any; otherwise the {default type definition}.

[local type validity]
The appropriate **case** among the following:
1 **If** the element information item was locally ·valid· as defined by Element Locally Valid (Type) (§3.3.4.4), **then** ***valid***
2 **otherwise** (the item was locally ·invalid· as defined by Element Locally Valid (Type) (§3.3.4.4)) ***invalid***.

[descendant validity]
The appropriate **case** among the following:
1 **If** neither its [children] nor its [attributes] contains an information item **I** (element or attribute respectively) where either **I**'s [validity] is ***invalid*** or **I** is ·attributed· to a strict ·wildcard particle· and **I**'s [validity] is ***notKnown***, **then** ***valid***;
2 **otherwise** ***invalid***.

Note: The [type definition type], [type definition namespace], [type definition name], and [type definition anonymous] properties are redundant with the [type definition] property; they are defined for the convenience of implementations which wish to expose those specific properties but not the entire type definition.

Note: When clause 5.1 of Element Locally Valid (Element) (§3.3.4.3) above applies and the default or fixed value constraint {value} is of type QName or NOTATION, it is ·implementation-dependent· whether ·namespace fixup· occurs; if it does not, the prefix used in the lexical representation (in [schema normalized value]) will not necessarily map to the namespace name of the value (in [schema actual value]). To reduce problems and confusion, users may prefer to ensure that the required namespace information item is present in the input infoset.

If the [schema normalized value] is not ·absent· and the ·governing type definition· is a simple type definition with {variety} ***union***, or its {content type} has {variety} ***simple*** and {simple type definition} a simple type definition with {variety} ***union***, then there are four additional properties:

PSVI Contributions for element information items

[member type definition]
An ·item isomorphic· to the ·validating type· of the [schema actual value].

[member type definition namespace]
The {target namespace} of the ·validating type·.

[member type definition anonymous]
true if the {name} of the ·validating type· is ·absent·, otherwise ***false***.

[member type definition name]
The {name} of the ·validating type·, if it is not ·absent·. If it is ·absent·, schema processors MAY, but need not, provide a value unique to the definition. It is ·implementation-defined· whether a processor provides a name for such a type definition. If a processor does provide a value in this situation, the choice of what value to use is ·implementation-dependent·.

The [type definition] property 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 [type definition type], [type definition namespace], [type definition name], and [type definition anonymous] properties are defined for the convenience of those specifying lighter-weight interfaces, in which exposing the entire type hierarchy and full component details might be a significant burden.

If **all** of the following are true:

1 the [schema normalized value] is not ·absent·;
2 **One** of the following is true:
  2.1 the simple type definition used to validate the ·normalized value· (either the ·governing type definition· or its {simple type definition}) has {variety} = ***list***;
  2.2 the simple type definition has {variety} = ***union*** and the ·validating type· of the [schema actual value] has {variety} = ***list***;
3 the {item type definition} of the list type (from the previous clause) has {variety} = ***union***;
then there is an additional property:

PSVI Contributions for element information items
`[member type definitions]` a sequence of Simple Type Definition components, with the same length as the [schema actual value], each one an ·item isomorphic· to the ·validating type· of the corresponding item in the [schema actual value].

Also, if the declaration has a {value constraint}, the item has a property:

PSVI Contributions for element information items
`[schema default]` The {lexical form} of the declaration's {value constraint}.

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

3.3.5.5 Element Default Value

Schema Information Set Contribution: Element Default Value

If and only if the local ·validity·, as defined by Element Locally Valid (Element) (§3.3.4.3) 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** clause 5.1 of Element Locally Valid (Element) (§3.3.4.3) above applies, **then** ***schema***. 2 **otherwise** ***infoset***.

See also Match Information (§3.4.5.2), Identity-constraint Table (§3.11.5), Validated with Notation (§3.14.5), and Schema Information (§3.17.5.1), which describe other information set contributions related to element information items.

3.3.5.6 Inherited Attributes

Schema Information Set Contribution: Inherited Attributes

[Definition:] An attribute information item ***A***, whether explicitly specified in the input information set or defaulted as described in Attribute Default Value (§3.4.5.1), is

potentially inherited by an element information item ***E*** if and only if **all** of the following are true:
1 ***A*** is among the [attributes] of one of ***E***'s ancestors.
2 ***A*** and ***E*** have the same [validation context].
3 **One** of the following is true:
  3.1 ***A*** is ·attributed to· an Attribute Use whose {inheritable} = ***true***.
  3.2 ***A*** is *not* ·attributed to· any Attribute Use but ***A*** has a ·governing attribute declaration· whose {inheritable} = ***true***.

If and only if an element information item ***P*** is not ·skipped· (that is, it is either ·strictly· or ·laxly· assessed), in the ·post-schema-validation infoset· each of ***P***'s element information item [children] ***E*** which is not ·attributed to· a ***skip*** Wildcard, has a property:

PSVI Contributions for element information items

`[inherited attributes]`
A list of attribute information items. An attribute information item ***A*** is included if and only if **all** of the following are true:
1 ***A*** is ·potentially inherited· by ***E***.
2 Let ***O*** be ***A***'s [owner element]. ***A*** does not have the same expanded name as another attribute which is also ·potentially inherited· by ***E*** and whose [owner element] is a descendant of ***O***.

3.3.6 Constraints on Element Declaration Schema Components

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

3.3.6.1 Element Declaration Properties Correct

Schema Component Constraint: Element Declaration Properties Correct

For any element declaration ***E***, **all** of the following MUST be true:
1 The values of ***E***'s properties . are 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 ***E*** has a ·non-absent· {value constraint}, then ***E***.{value constraint} is a valid default with respect to ***E***.{type definition} as defined in Element Default Valid (Immediate) (§3.3.6.2).
3 If ***E***.{substitution group affiliations} is non-empty, then ***E***.{scope}.{variety} = ***global***.
4 For each member ***M*** of ***E***.{substitution group affiliations}, ***E***.{type definition} is ·validly substitutable· for ***M***.{type definition}, subject to the blocking keywords in ***M***.{substitution group exclusions}.
5 There are no circular substitution groups. That is, it is not possible to return to ***E*** by repeatedly following any member of the {substitution group affiliations} property.
6 If ***E***.{type table} exists, then for each Type Alternative in ***E***.{type table}.{alternatives}, the {test} property is not ·absent·.
7 If ***E***.{type table} exists, then for each {type definition} ***T*** in ***E***.{type table}.{alternatives}, and also for ***E***.{type table}.{default type definition}.{type definition}, **one** of the following is true
  7.1 ***T*** is ·validly substitutable· for ***E***.{type definition}, subject to the blocking keywords of ***E***.{disallowed substitutions}.
  7.2 ***T*** is the type ·`xs:error`·.

3.3.6.2 Element Default Valid (Immediate)

This and the following sections define relations appealed to elsewhere in this specification.

Schema Component Constraint: Element Default Valid (Immediate)

For a Value Constraint ***V*** to be a valid default with respect to a type definition ***T*** the appropriate **case** among the following MUST be true:

1 **If** ***T*** is a simple type definition or a complex type definition with {content type}.{variety} = ***simple*** , **then** ***V*** is a valid default with respect either to ***T*** (if ***T*** is simple) or (if ***T*** is complex) to ***T***.{content type}.{simple type definition} as defined by Simple Default Valid (§3.2.6.2).

2 **If** ***T*** is a complex type definition with {content type}.{variety} ≠ ***simple*** , **then all** of the following are true:

2.1 ***T***.{content type}.{variety} = ***mixed***.

2.2 The particle ***T***.{content type}.{particle} is ·emptiable· as defined by Particle Emptiable (§3.9.6.3).

3.3.6.3 Substitution Group OK (Transitive)

Schema Component Constraint: Substitution Group OK (Transitive)

For an element declaration (call it **M**, for member) to be substitutable for another element declaration (call it **H**, for head) at least **one** of the following MUST be true:

1 **M** and **H** are the same element declaration.

2 **All** of the following are true:

2.1 **H**. {disallowed substitutions} does not contain ***substitution***.

2.2 There is a chain of {substitution group affiliations} properties from **M** to **H**, that is, either **M**.{substitution group affiliations} contains **H**, or **M**.{substitution group affiliations} contains a declaration whose {substitution group affiliations} contains **H**, or . . .

2.3 The set of all {derivation method}s involved in the ·derivation· of ***M***.{type definition} from ***H***.{type definition} does not intersect with the union of (1) ***H***.{disallowed substitutions}, (2) ***H***.{type definition}.{prohibited substitutions} (if **H**.{type definition} is complex, otherwise the empty set), and (3) the {prohibited substitutions} (respectively the empty set) of any intermediate declared {type definition}s in the ·derivation· of ***M***.{type definition} from ***H***.{type definition}.

3.3.6.4 Substitution Group

[Definition:] One element declaration is **substitutable** for another if together they satisfy constraint Substitution Group OK (Transitive) (§3.3.6.3).

[Definition:] Every element declaration (call this **HEAD**) in the {element declarations} of a schema defines a **substitution group**, a subset of those {element declarations}. An element declaration is in the **substitution group** of **HEAD** if and only if it is ·substitutable· for **HEAD**.

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.
- Constraining elements and attributes to exist, not to exist, or to have specified values, with Assertion (§2.2.4.3)s.
- 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, a kind of Type Definition

{annotations}
: A sequence of Annotation components.

{name}
: An xs:NCName value. Optional.

{target namespace}
: An xs:anyURI value. Optional.

{base type definition}
: A type definition component. Required.

{final}
: A subset of {*extension*, *restriction*}.

{context}
: Required if {name} is ·absent·, otherwise MUST be ·absent·.
Either an Element Declaration or a Complex Type Definition.

{derivation method}
: One of {***extension***, ***restriction***}. Required.

{abstract}
: An xs:boolean value. Required.

{attribute uses}
: A set of Attribute Use components.

{attribute wildcard}
: A Wildcard component. Optional.

{content type}
: A Content Type property record. Required.

{prohibited substitutions}
: A subset of {***extension***, ***restriction***}.

{assertions}
: A sequence of Assertion components.

Property Record: Content Type

{variety}
: One of {***empty***, ***simple***, ***element-only***, ***mixed***}. Required.

{particle}
: A Particle component. Required if {variety} is ***element-only*** or ***mixed***, otherwise MUST be ·absent·.

{open content}
: An Open Content property record. Optional if {variety} is ***element-only*** or ***mixed***, otherwise MUST be ·absent·.

{simple type definition}
: A Simple Type Definition component. Required if {variety} is ***simple***, otherwise MUST be ·absent·.

Property Record: Open Content

{mode}
: One of {***interleave***, ***suffix***}. Required.

{wildcard}
: A Wildcard component. Required.

Complex type 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 ·XSD 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 (§2.7.1)), and for use in the XML

representation of schema components (specifically in <element>). See References to schema components across namespaces (<import>) (§4.2.6) 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.

The {context} property is only relevant for anonymous type definitions, for which its value is the component in which this type definition appears as the value of a property, e.g. {type definition}.

Complex types for which {abstract} is ***true*** have no valid instances and thus cannot be used in the normal way as the {type definition} for the ·validation· of element information items (if for some reason an abstract type is identified as the ·governing type definition· of an element information item, the item will invariably be invalid). It follows that such abstract types MUST NOT be referenced from an xsi:type (§2.7.1) attribute in an instance document. Abstract complex types can be used as {base type definition}s, or even as the declared {type definition}s of element declarations, provided in every case a concrete ·derived· type definition is used for ·validation·, either via xsi:type (§2.7.1) or the operation of a ·substitution group·.

{attribute uses} are a set of attribute uses. See Element Locally Valid (Complex Type) (§3.4.4.2) and Attribute Locally Valid (§3.2.4.1) for details of attribute ·validation·.

{attribute wildcard}s provide a more flexible specification for ·validation· of attributes not explicitly included in {attribute uses}. See Element Locally Valid (Complex Type) (§3.4.4.2), The Wildcard Schema Component (§3.10.1) and Wildcard allows Expanded Name (§3.10.4.2) for formal details of attribute wildcard ·validation·.

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

- A {content type} with {variety} ***empty*** ·validates· elements with no character or element information item [children].
- A {content type} with {variety} ***simple*** ·validates· elements with character-only [children]using its {simple type definition}.
- A {content type} with {variety} ***element-only*** ·validates· elements with [children] that conform to the ·content model· supplied by its {particle}.
- A {content type} with {variety} ***mixed*** ·validates· elements whose element [children] (i.e. specifically ignoring other [children] such as character information items) conform to the ·content model· supplied by its {particle}.

- A {content type} with ·non-absent· {open content} ·validates· elements with some [children] conforming to the ·content model· and others conforming to the {open content}.

  Note: Not all combinations of {derivation method} and {content type} are compatible with all properties of the {base type definition}. For example, it is not allowed to derive a complex type with complex content from a simple type. The XML mapping rules specified in the following section (in particular clause 5 of the rule for the {simple type definition} in the rule for {content type} of complex types with simple content, and clause 4.1 and clause 4.2.1 of the rule for {content type} for complex types with complex content) do not detect such incompatible combinations of properties; in such cases the mapping rules will build a complex type regardless of the fact that the properties specified are incompatible. But the resulting complex type does not satisfy component rules outlined in Derivation Valid (Extension) (§3.4.6.2) or Derivation Valid (Restriction, Complex) (§3.4.6.3).

The {prohibited substitutions} property of a complex type definition ***T*** determines whether type definitions derived from ***T*** are or are not ·validly substitutable· for ***T***. Examples include (but are not limited to) the substitution of another type definition:

- as the ·governing type definition· of an element instance ***E***, when ***T*** is the ·selected type definition· of ***E*** (often, the declared {type definition} of ***E***'s ·governing element declaration·); this can occur when ***E*** specifies a type definition using the `xsi:type` attribute; see xsi:type (§2.7.1);

- as the ·selected type definition· of an element instance ***E***, when ***T*** is the declared {type definition} of ***E***'s ·governing element declaration·; this can occur when conditional type assignment is used; see Type Alternatives (§3.12);

- as the ·governing type definition· of element instances whose ·governing element declaration· is included in a model group only ·implicitly·, by virtue of being included in the ·substitution group· of some element declaration present ·directly· ·indirectly· in the model group, whose declared {type definition} is ***T***.

- as the {type definition} of an Element Declaration ***E1*** where

  - ***E1*** is contained in a Complex Type Definition ***D***

  - ***D*** is derived from another Complex Type Definition ***B***

  - ***B*** contains an Element Declaration ***E2*** that has the same expanded name as ***E1***

  - ***E2*** has ***T*** as its {type definition}.

If {prohibited substitutions} is empty, then all such substitutions are allowed; if it contains the keyword ***restriction***, then no type definition is ·validly substitutable· for ***T*** if its derivation from ***T*** involves any restriction steps; if {prohibited substitutions} contains the keyword ***extension***, then no type definition is ·validly substitutable· for ***T*** if its derivation from ***T*** involves any extension steps.

{assertions} constrain elements and attributes to exist, not to exist, or to have specified values. Though specified as a sequence, the order among the assertions is not significant during assessment. See Assertions (§3.13).

See Annotations (§3.15) for information on the role of the {annotations} property.

3.4.2 XML Representation of Complex Type Definition Schema Components

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 {content type} with {variety} ***simple*** is significantly different from that of those with other {content type}s, and this is reflected in the presentation below, which describes the mappings for the two cases in separate subsections. Common mapping rules are factored out and given in separate sections. As always, the mapping rules given here apply after, not before, the appropriate ·pre-processing·.

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
  name = NCName
  defaultAttributesApply = boolean : true
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (simpleContent | complexContent | (openContent?,
(group | all | choice | sequence)?, ((attribute | attributeGroup)*,
anyAttribute?), assert*)))
</complexType>
```

Note: It is a consequence of the concrete syntax given above that a top-level type definition need consist of no more than a name, i.e. that `<complexType name="anyThing"/>` is allowed.

Note: Aside from the simple coherence requirements outlined below, the requirement that type definitions identified as restrictions actually *be* restrictions — that is, the requirement that they accept as valid only a subset of the items which are accepted as valid by their base type definition — is enforced in Constraints on Complex Type Definition Schema Components (§3.4.6).

The following sections describe different sets of mapping rules for complex types; some are common to all or many source declarations, others only in specific circumstances.

- If the <complexType> source declaration has a <simpleContent> element as a child, then it maps to a Complex Type Definition using the mapping rules in
  - Mapping Rules for Complex Types with Simple Content (§3.4.2.2),
  - Common Mapping Rules for Complex Type Definitions (§3.4.2.1),
  - Mapping Rule for Attribute Uses Property (§3.4.2.4), and
  - Mapping Rule for Attribute Wildcard Property (§3.4.2.5).
- If the <complexType> source declaration has a <complexContent> element as a child, then it maps to a Complex Type Definition using the mapping rules in
  - Mapping Rules for Complex Types with Explicit Complex Content (§3.4.2.3.1),
  - Mapping Rules for Content Type Property of Complex Content (§3.4.2.3.3),
  - Common Mapping Rules for Complex Type Definitions (§3.4.2.1),

- Mapping Rule for Attribute Uses Property (§3.4.2.4), and
- Mapping Rule for Attribute Wildcard Property (§3.4.2.5).

- If the <complexType> source declaration has neither a <simpleContent> nor a <complexContent> element as a child, then it maps to a Complex Type Definition using the mapping rules in
  - Mapping Rules for Complex Types with Implicit Complex Content (§3.4.2.3.2),
  - Mapping Rules for Content Type Property of Complex Content (§3.4.2.3.3),
  - Common Mapping Rules for Complex Type Definitions (§3.4.2.1),
  - Mapping Rule for Attribute Uses Property (§3.4.2.4), and
  - Mapping Rule for Attribute Wildcard Property (§3.4.2.5).

Where convenient, the mapping rules are described exclusively in terms of the schema document's information set. The mappings, however, depend not only upon the source declaration but also upon the schema context. Some mappings, that is, depend on the properties of other components in the schema. In particular, several of the mapping rules given in the following sections depend upon the {base type definition} having been identified before they apply.

3.4.2.1 Common Mapping Rules for Complex Type Definitions

Whichever alternative for the content of <complexType> is chosen, the following property mappings apply. Except where otherwise specified, attributes and child elements are to be sought among the [attributes] and [children] of the <complexType> element.

XML Mapping Summary for 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 to 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].
{context}	If the `name` [attribute] is present, then ·absent·, otherwise (among the ancestor element information items there will be a nearest <element>), the Element Declaration corresponding to the nearest <element> information item among the the ancestor element information items.
{assertions}	A sequence whose members are Assertions drawn from the following sources, in order: 1 The {assertions} of the {base type definition}. 2 Assertions corresponding to all the <assert> element information items among the [children] of <complexType>, <restriction> and <extension>, if any, in document order.
{annotations}	The ·annotation mapping· of the set of elements containing the <complexType>, the <openContent> [child], if present, the <attributeGroup> [children], if present, the <simpleContent> and <complexContent> [children], if present, and their <restriction> and <extension> [children], if present, and their <openContent> and <attributeGroup> [children], if present, as defined in XML Representation of Annotation Schema Components (§3.15.2).

Note: If the {base type definition} is a complex type definition, then the {assertions} always contain members of the {assertions} of the {base type definition}, no matter which alternatives are chosen in the XML representation, <simpleContent> or <complexContent>, <restriction> or <extension>.

3.4.2.2 Mapping Rules for Complex Types with Simple Content

When the <complexType> source declaration has a <simpleContent> child, the following elements are relevant (as are <attribute>, <attributeGroup>, and <anyAttribute>), and the property mappings are as below, supplemented by the mappings in Common Mapping Rules for Complex Type Definitions (§3.4.2.1), Mapping Rule for Attribute Uses Property (§3.4.2.4), and Mapping Rule for Attribute Wildcard Property (§3.4.2.5). Note that either <restriction> or <extension> MUST appear in the content of <simpleContent>.

XML Representation Summary: `simpleContent` Element Information Item et al.

```
<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 | assertion | {any with
namespace: ##other})*)?, ((attribute | attributeGroup)*, anyAttribute?), assert*)
</restriction>
```

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

When the <complexType> element has a <simpleContent> child, then the <complexType> element maps to a complex type with simple content, as follows.

XML Mapping Summary for 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] on the <restriction> or <extension> element appearing as a child of <simpleContent>
{derivation method}	If the <restriction> alternative is chosen, then ***restriction***, otherwise (the <extension> alternative is chosen) ***extension***.
{content type}	A Content Type as follows:

Property	Value
{variety}	***simple***
{particle}	·absent·
{open content}	·absent·
{simple type definition}	the appropriate **case** among the following: 1 **If** the {base type definition} is a complex type definition whose own {content type} has {variety} ***simple*** and the <restriction> alternative is chosen, **then** let ***B*** be 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 {simple type definition} of the {content type} of the {base type definition} a simple type definition as follows:

Property	Value
{name}	·absent·
{target namespace}	The ·actual value· of the `targetNamespace` [attribute] of the ancestor <schema>

	element information item if present, otherwise ·absent·
{final}	The empty set
{context}	The Complex Type Definition whose {content type}.{simple type definition} is being defined
{base type definition}	***B***
{facets}	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) (§3.16.6.4);
{fundamental facets}	Based on {variety}, {facets}, {base type definition} and {member type definitions}, a set of Fundamental Facet components, one each as specified in The ordered Schema Component , The bounded Schema

	Component , The cardinality Schema Component and The numeric Schema Component
{variety}	***B***.{variety}
{primitive type definition}	***B***.{primitive type definition}
{item type definition}	***B***.{item type definition}
{member type definitions}	***B***.{member type definitions}
{annotations}	The empty sequence

2 **If** the {base type definition} is a complex type definition whose own {content type} has {variety} ***mixed*** and {particle} a Particle which is ·emptiable·, as defined in Particle Emptiable (§3.9.6.3) and the <restriction> alternative is chosen, **then** (let S_B be the simple type definition corresponding to the <simpleType> among the [children] of <restriction> if any, otherwise ·xs:anySimpleType·) a simple type definition which restricts S_B 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) (§3.16.6.4);

Note: If there is no <simpleType> among the [children] of <restriction> (and if therefore S_B is ·xs:anySimpleType·), the result will be a simple type definition component which fails to obey the constraints on simple type definitions, including for example clause 1.1 of Derivation Valid (Restriction, Simple) (§3.16.6.2).

3 **If** the {base type definition} is a complex type definition whose own {content type} has {variety} ***simple*** and the

<extension> alternative is chosen, **then** the {simple type definition} of the {content type} of that complex type definition;

4 **If** the {base type definition} is a simple type definition and the <extension> alternative is chosen, **then** that simple type definition;

5 **otherwise** ·xs:anySimpleType·.

3.4.2.3 Mapping Rules for Complex Types with Complex Content

When the <complexType> element does not have a <simpleContent> child element, then it maps to a complex type with complex content. The following elements are relevant (as are the <attribute>, <attributeGroup>, and <anyAttribute> elements, which are described more fully in XML Representation of Attribute Declaration Schema Components (§3.2.2), Mapping Rule for Attribute Uses Property (§3.4.2.4), and XML Representation of Wildcard Schema Components (§3.10.2), respectively, and which are not repeated here), and the additional property mappings are as below, supplemented by the mappings in Common Mapping Rules for Complex Type Definitions (§3.4.2.1), Mapping Rule for Attribute Uses Property (§3.4.2.4), Mapping Rule for Attribute Wildcard Property (§3.4.2.5), Mapping Rules for Local Attribute Declarations (§3.2.2.2), and Mapping Rules for References to Top-level Attribute Declarations (§3.2.2.3). Note that either <restriction> or <extension> MUST appear in the content of <complexContent>, but their content models are different in this case from the case above when they occur as children of <simpleContent>.

XML Representation Summary: `complexContent` Element Information Item et al.

```
<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?, openContent?, (group | all | choice | sequence)?,
((attribute | attributeGroup)*, anyAttribute?), assert*)
</restriction>
```

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

```
<openContent
  id = ID
  mode = (none | interleave | suffix) : interleave
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, any?)
</openContent>
```

Complex types with complex content can be the image of two different forms of <complexType> element: one with a <complexContent> child (discussed in Mapping Rules for Complex Types with Explicit Complex Content (§3.4.2.3.1)), and one with neither <simpleContent> nor <complexContent> as a child (discussed in Mapping Rules for Complex Types with Implicit Complex Content (§3.4.2.3.2)). The mapping of the {content type} is the same in both cases; it is described in Mapping Rules for Content Type Property of Complex Content (§3.4.2.3.3).

3.4.2.3.1 Mapping Rules for Complex Types with Explicit Complex Content

When the <complexType> source declaration has a <complexContent> child, the following mappings apply, supplemented by those specified in Mapping Rules for Content Type Property of Complex Content (§3.4.2.3.3), Common Mapping Rules for Complex Type Definitions (§3.4.2.1), Mapping Rule for Attribute Uses Property (§3.4.2.4), and Mapping Rule for Attribute Wildcard Property (§3.4.2.5).

XML Mapping Summary for 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***.

3.4.2.3.2 Mapping Rules for Complex Types with Implicit Complex Content

When the <complexType> source declaration has neither <simpleContent> nor <complexContent> as a child, it is taken as shorthand for complex content restricting ·`xs:anyType`·. The mapping rules specific to this situation are as follows; the mapping rules for properties not described here are as given in Mapping Rules for Content Type Property of Complex Content (§3.4.2.3.3), Common Mapping Rules for Complex Type Definitions (§3.4.2.1), Mapping Rule for Attribute Uses Property (§3.4.2.4), and Mapping Rule for Attribute Wildcard Property (§3.4.2.5).

XML Mapping Summary for Complex Type Definition with complex content Schema Component

Property	Representation
{base type definition}	·`xs:anyType`·
{derivation method}	***restriction***

3.4.2.3.3 Mapping Rules for Content Type Property of Complex Content

For complex types with complex content, the {content type} property is calculated as follows. (For the {content type} on complex types with simple content, see Mapping Rules for Complex Types with Simple Content (§3.4.2.2).)

> **Note:** The mapping rule below refers here and there to elements not necessarily present within a <complexType> source declaration. For purposes of evaluating tests like "If the `abc` attribute is present on the `xyz` element", if no `xyz` element information item is present, then no `abc` attribute is present on the (non-existent) `xyz` element.
>
> When the mapping rule below refers to "the [children]", then for a <complexType> source declaration with a <complexContent> child, then the [children] of <extension> or <restriction> (whichever appears as a child of <complexContent>) are meant. If no <complexContent> is present, then the [children] of the <complexType> source declaration itself are meant.
>
> The mapping rule also refers to the value of the {derivation method} property, whose value is determined as specified in the preceding sections.

XML Mapping Summary for Complex Type Definition with complex content Schema Component

Property	Representation
{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`. **Note:** It is a consequence of clause 5 of Complex Type Definition Representation OK (§3.4.3) that clause 1.1 and clause 1.2 above will never contradict each other in a conforming schema document. 2 [Definition:] Let the **explicit content** be the appropriate **case** among the following: 2.1 **If** at least **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 among the [children] a <choice> element whose `minOccurs` [attribute] has the ·actual value· `0` and which has no [children] of its own except for <annotation>; 2.1.4 The <group>, <all>, <choice> or <sequence> element among the [children] has a `maxOccurs` [attribute] with an ·actual value· of 0; **then *empty*** 2.2 **otherwise** the particle corresponding to the <all>, <choice>, <group> or <sequence> among the [children]. 3 [Definition:] Let the **effective content** be the appropriate **case** among the following: 3.1 **If** the ·explicit content· is ***empty*** , **then** the appropriate **case** among the following:

3.1.1 **If** the ·effective mixed· is `true`, **then** A particle whose properties are as follows:

Property	Value
{min occurs}	1
{max occurs}	1
{term}	a model group whose {compositor} is ***sequence*** and whose {particles} is empty.

3.1.2 **otherwise** ***empty***

3.2 **otherwise** the ·explicit content·.

4 [Definition:] Let the **explicit content type** be the appropriate **case** among the following:

4.1 **If** {derivation method} = ***restriction***, **then** the appropriate **case** among the following:

4.1.1 **If** the ·effective content· is ***empty*** , **then** a Content Type as follows:

Property	Value
{variety}	***empty***
{particle}	·absent·
{open content}	·absent·
{simple type definition}	·absent·

4.1.2 **otherwise** a Content Type as follows:

Property	Value
{variety}	***mixed*** if the ·effective mixed· is `true`, otherwise ***element-only***
{particle}	The ·effective content·
{open content}	·absent·
{simple type definition}	·absent·

4.2 **If** {derivation method} = ***extension***, **then** the appropriate **case** among the following:

4.2.1 **If** the {base type definition} is a simple type definition or is a complex type definition whose {content type}.{variety} = ***empty*** or ***simple***, **then** a Content Type as per clause 4.1.1 and clause 4.1.2 above;

4.2.2 **If** the {base type definition} is a complex type definition whose {content type}.{variety} = ***element-only*** or ***mixed*** and the ·effective content· is ***empty***, **then** {base type definition}. {content type};

4.2.3 **otherwise** a Content Type as follows:

Property	Value

{variety}	***mixed*** if the ·effective mixed· is `true`, otherwise ***element-only***
{particle}	[Definition:] Let the **base particle** be the particle of the {content type} of the {base type definition}. Then the appropriate **case** among the following: 4.2.3.1 **If** the {term} of the ·base particle· has {compositor} ***all*** and the ·explicit content· is empty, **then** the ·base particle·. 4.2.3.2 **If** the {term} of the ·base particle· has {compositor} ***all*** and the {term} of the ·effective content· also has {compositor} ***all***, **then** a Particle whose properties are as follows: **{min occurs}** the {min occurs} of the ·effective content·. **{max occurs}** 1 **{term}** a model group whose {compositor} is ***all*** and whose {particles} are the {particles} of the {term} of the ·base particle· followed by the {particles} of the {term} of the ·effective content·. 4.2.3.3 **otherwise** **{min occurs}** 1 **{max occurs}** 1 **{term}** a model group whose {compositor} is ***sequence*** and whose {particles} are the ·base particle· followed by the ·effective content·.
{open content}	the {open content} of the {content type} of the {base type definition}.
{simple type definition}	·absent·

5 [Definition:] Let the **wildcard element** be the appropriate **case** among the following:
  5.1 **If** the <openContent> [child] is present , **then** the <openContent> [child].
  5.2 **If** the <openContent> [child] is not present, the <schema> ancestor has a <defaultOpenContent> [child], and **one** of the following is true
    5.2.1 the ·explicit content type· has {variety} ≠ ***empty***
    5.2.2 the ·explicit content type· has {variety} = ***empty*** and the <defaultOpenContent> element has `appliesToEmpty` = ***true***
  then the <defaultOpenContent> [child] of the <schema>.
  5.3 **otherwise** ·absent·.
6 Then the value of the property is the appropriate **case** among the following:
  6.1 **If** the ·wildcard element· is ·absent· or is present and has `mode` = `'none'` , **then** the ·explicit content type·.
  6.2 **otherwise**

Property	Value
{variety}	The {variety} of the ·explicit content type· if it's not ***empty***; otherwise ***element-only***.
{particle}	The {particle} of the ·explicit content type· if the {variety} of the ·explicit content type· is not ***empty***; otherwise a Particle as follows:

Property	Value
{min occurs}	1
{max occurs}	1
{term}	a model group whose {compositor} is ***sequence*** and whose {particles} is empty.

{open content}	An Open Content as follows:

Property	Value
{mode}	The ·actual value· of the `mode` [attribute] of the ·wildcard element·, if present,

	otherwise ***interleave***.
{wildcard}	Let ***W*** be the wildcard corresponding to the <any> [child] of the ·wildcard element·. If the {open content} of the ·explicit content type· is ·absent·, then ***W***; otherwise a wildcard whose {process contents} and {annotations} are those of ***W***, and whose {namespace constraint} is the wildcard union of the {namespace constraint} of ***W*** and of {open content}. {wildcard} of the ·explicit content type·, as defined in Attribute Wildcard Union (§3.10.6.3).

{simple type definition}	·absent·

Note: It is a consequence of clause 4.2 above that when a type definition is extended, the same particles appear in both the base type definition and the extension; the particles are reused without being copied.

3.4.2.4 Mapping Rule for Attribute Uses Property

Any <complexType> source declaration can have <attribute> and <attributeGroup> elements as children, or descendants. The <attribute> element is described in XML Representation of Attribute Declaration Schema Components (§3.2.2) and will not be repeated here.

XML Representation Summary: `attributeGroup` Element Information Item

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

The <attribute> and <attributeGroup> elements map to the {attribute uses} property of the Complex Type Definition component as described below. This mapping rule is the same for all complex type definitions.

> **Note:** In the following rule, references to "the [children]" refer to the [children] of the <extension> or <restriction> element (whichever appears as a child of <simpleContent> or <complexContent> in the <complexType> source declaration), if present, otherwise to the [children] of the <complexType> source declaration itself.
>
> The rule also refers to the value of the {derivation method} property, which is described elsewhere.

XML Mapping Summary for Complex Type Definition (Attribute Uses) Schema Component

Property	Representation
{attribute uses}	If the <schema> ancestor has a `defaultAttributes` attribute, and the <complexType> element does not have `defaultAttributesApply` = `false`, then the {attribute uses} property is computed as if there were an <attributeGroup> [child] with empty content and a `ref` [attribute] whose ·actual value· is the same as that of the `defaultAttributes` [attribute] appearing after any other <attributeGroup> [children]. Otherwise proceed as if there were no such <attributeGroup> [child]. Then the value is 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 "inherited" from the {base type definition} ***T***, as described by the appropriate **case** among the following: 3.1 **If** ***T*** is a complex type definition and {derivation method} = ***extension***, **then** the attribute uses in ***T***.{attribute uses} are inherited. 3.2 **If** ***T*** is a complex type definition and {derivation method} = ***restriction***, **then** the attribute uses in ***T***.{attribute uses} are inherited, with the exception of those with an {attribute declaration} whose expanded name is **one** of the following:

3.2.1 the expanded name of the {attribute declaration} of an attribute use which has already been included in the set, following the rules in clause 1 or clause 2 above;
3.2.2 the expanded name of the {attribute declaration} of what would have been an attribute use corresponding to an <attribute> [child], if the <attribute> had not had `use` = ***prohibited***.

Note: This sub-clause handles the case where the base type definition ***T*** allows the attribute in question, but the restriction prohibits it.

3.3 **otherwise** no attribute use is inherited.

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). It is pointless, though not an error, for the `use` attribute to have the value ***prohibited*** in other contexts (e.g. in complex type extensions or named model group definitions), in which cases the <attribute> element is simply ignored, provided that it does not violate other constraints in this specification.

3.4.2.5 Mapping Rule for Attribute Wildcard Property

The {attribute wildcard} property of a Complex Type Definition depends on the <anyAttribute> element which may be present within the <complexType> element or within the attribute groups referred to within <complexType>. The <attributeGroup> element is described in the preceding section Mapping Rule for Attribute Uses Property (§3.4.2.4) and will not be repeated here.

Note: The following mapping rule is the same for all complex type definitions.

References to "the [children]" refer to the [children] of the <extension> or <restriction> element (whichever appears as a child of <simpleContent> or <complexContent> in the <complexType> source declaration), if present, otherwise to the [children] of the <complexType> source declaration itself.

The rule also refers to the value of the {derivation method} property, which is described elsewhere.

XML Mapping Summary for Complex Type Definition (Attribute Wildcard) Schema Component

Property	Representation
{attribute wildcard}	If the <schema> ancestor has a `defaultAttributes` attribute, and the <complexType> element does not have `defaultAttributesApply` = `false`, then the {attribute wildcard} property is computed as if there were an <attributeGroup> [child] with empty content and a `ref` [attribute] whose ·actual value· is the same as that of the `defaultAttributes` [attribute] appearing after any other

<attributeGroup> [children]. Otherwise proceed as if there were no such <attributeGroup> [child].

1 [Definition:] Let the **complete wildcard** be the Wildcard computed as described in Common Rules for Attribute Wildcards (§3.6.2.2).
2 The value is then determined by the appropriate **case** among the following:
2.1 **If** {derivation method} = ***restriction***, **then** the ·complete wildcard·;
2.2 **If** {derivation method} = ***extension***, **then**
2.2.1 [Definition:] let the **base wildcard** be defined as the appropriate **case** among the following:
2.2.1.1 **If** the {base type definition} is a complex type definition with an {attribute wildcard}, **then** that {attribute wildcard}.
2.2.1.2 **otherwise** ·absent·.
2.2.2 The value is then determined by the first **case** among the following which applies:
2.2.2.1 **If** the ·base wildcard· is ·absent·, **then** the ·complete wildcard·;
2.2.2.2 **If** the ·complete wildcard· is ·absent·, **then** the ·base wildcard·;
2.2.2.3 **otherwise** a wildcard whose {process contents} and {annotations} are those of the ·complete wildcard·, and whose {namespace constraint} is the wildcard union of the {namespace constraint} of the ·complete wildcard· and of the ·base wildcard·, as defined in Attribute Wildcard Union (§3.10.6.3).

3.4.2.6 Examples of Complex Type Definitions

Example: Three ways to define a type for length

The following declaration defines a type for specifications of length by creating a complex type with simple content, with `xs:nonNegativeInteger` as the type of the content, and a `unit` attribute to give the unit of measurement.

```
<xs:complexType name="lengthl">
 <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="lengthl"/>
```

An instance using this type might look like this:

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

A second approach to defining length uses two elements, one for size and one for the unit of measure. The definition of the type and the declaration of the element might look like this:

```
<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"/>
```

An instance using this method might look like this:

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

A third definition of type leaves the base type implicit; at the component level, the following declaration is equivalent to the preceding one.

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

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 child and fixing another to occur exactly once; an element declared by reference to it, and a ·valid· instance thereof.

Example

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

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

Example

```
<xs:complexType name="name">
  <xs:openContent>
    <xs:any namespace="##other" processContents="skip"/>
  </xs:openContent>
  <xs:sequence>
    <xs:element name="given" type="xs:string"/>
    <xs:element name="middle" type="xs:string" minOccurs="0"/>
    <xs:element name="family" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

A complex type definition that allows three explicitly declared child elements, in the specified order (but not necessarily adjacent), and furthermore allows additional elements of any name from any namespace other than the target namespace to appear anywhere in the children.

Example

To restrict away a local element declaration that ·competes· with a wildcard, use a wildcard in the derived type that explicitly disallows the element's expanded name. For example:

```
<xs:complexType name="computer">
 <xs:all>
  <xs:element name="CPU" type="CPUType"/>
  <xs:element name="memory" type="memoryType"/>
  <xs:element name="monitor" type="monitorType"/>
  <xs:element name="speaker" type="speakerType"
              minOccurs="0"/>
  <!-- Any additional information about the computer -->
  <xs:any processContents="lax"
          minOccurs="0" maxOccurs="unbounded"/>
 </xs:all>
</xs:complexType>

<xs:complexType name="quietComputer">
 <xs:complexContent>
  <xs:restriction base="computer">
   <xs:all>
    <xs:element name="CPU" type="CPUType"/>
    <xs:element name="memory" type="memoryType"/>
    <xs:element name="monitor" type="monitorType"/>
    <!-- Any additional information about the computer -->
    <xs:any processContents="lax" notQName="speaker"
            minOccurs="0" maxOccurs="unbounded"/>
   </xs:all>
  </xs:restriction>
 </xs:complexContent>
</xs:complexType>
```

The restriction type `quietComputer` has a ***lax*** wildcard, which ·matches· any element but one with the name `speaker`.

Without the specification of the `notQName` attribute, the wildcard would ·match· elements named `speaker`, as well. In that case, the restriction would be valid only if there is a top-level declaration for `speaker` that also has type `speakerType` or a type derived from it. Otherwise, there would be instances locally valid against the restriction `quietComputer` that are not locally valid against the base type `computer`.

For example, if there is no `notQName` attribute on the wildcard and no top-level declaration for `speaker`, then the following is allowed by `quietComputer`, but not by `computer`:

```
    <speaker xsi:type="xs:string"/>
```

The specific rule violated in this case is clause 2 of constraint Content type restricts (Complex Content) (§3.4.6.4)

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 schema documents, **all** of the following also apply:

1 If the <simpleContent> alternative is chosen, the <complexType> element MUST NOT have `mixed` = ***true***.

2 If <restriction> is present under <simpleContent>, then no facet-specifying element other than `xs:enumeration`, `xs:pattern`, or `xs:assertion` MAY appear more than once among the [children] of <restriction>.

3 If <openContent> is present and has `mode` ≠ `'none'`, then there MUST be an <any> among the [children] of <openContent>.

4 If <openContent> is present and has `mode` = `'none'`, then there MUST NOT be an <any> among the [children] of <openContent>.

5 If the <complexContent> alternative is chosen and the `mixed` [attribute] is present on both <complexType> and <complexContent>, then ·actual values· of those [attributes] MUST be the same.

3.4.4 Complex Type Definition Validation Rules

3.4.4.1 Locally Declared Type

This section defines the concept of ·locally declared type·; this concept plays a role in determining whether restrictions and extensions of complex type definitions are legitimate. The ·locally declared type· is also used to help determine the ·governing element declaration· and ·governing type definition· of an element information item.

[Definition:] Every Complex Type Definition determines a partial functional mapping from element or attribute information items (and their expanded names) to type definitions. This mapping serves as a **locally declared type** for elements and attributes which are allowed by the Complex Type Definition.

The attribute case is simpler and will be taken first.

[Definition:] For a given Complex Type Definition ***CTD*** and a given attribute information item ***A***, the ·locally declared type· of ***A*** within ***CTD*** is the appropriate **case** among the following:
1 **If** ***CTD*** is ·`xs:anyType`· , **then** ·absent·.
2 **If** ***A*** has the same expanded name as some attribute declaration ***D*** which is the {attribute declaration} of some Attribute Use contained by ***CTD***'s {attribute uses} , **then** the {type definition} of ***D***.
3 **otherwise** the ·locally declared type· of ***A*** within ***CTD***.{base type definition}.

The definition for elements is slightly more complex.

[Definition:] For a given Complex Type Definition ***CTD*** and a given element information item ***E***, the ·locally declared type· of ***E*** within ***CTD*** is the appropriate **case** among the following:
1 **If** ***CTD*** is ·`xs:anyType`· , **then** ·absent·.
2 **If** ***E*** has the same expanded name as some element declaration ***D*** which is ·contained· by ***CTD***'s content model, whether ·directly·, ·indirectly·, or ·implicitly· , **then** the declared {type definition} of ***D***.
3 **otherwise** the ·locally declared type· of ***E*** within ***CTD***.{base type definition}.

> **Note:** The constraint Element Declarations Consistent (§3.8.6.3) ensures that even if there is more than one such declaration ***D***, there will be just one type definition.
>
> **Note:** The reference to ·implicit· containment ensures that if ***E*** has a ·governing element declaration· ·substitutable· for a declaration ·contained· by ***CTD***'s content model, a ·locally declared type· is defined for ***E***.

3.4.4.2 Element Locally Valid (Complex Type)

Validation Rule: Element Locally Valid (Complex Type)

For an element information item ***E*** to be locally ·valid· with respect to a complex type definition ***T*** **all** of the following MUST be true:
1 If ***E*** is not ·nilled·, then **all** of the following are true:
1.1 **If** ***T***.{content type}.{variety} = ***empty***, **then** ***E*** has no character or element information item [children].

1.2 **If** ***T***.{content type}.{variety} = ***simple***, **then** ***E*** has no element information item [children], and the ·initial value· of ***E*** is ·valid· with respect to ***T***.{content type}.{simple type definition} as defined by String Valid (§3.16.4).

1.3 **If** ***T***.{content type}.{variety} = ***element-only***, **then** ***E*** has no character information item [children] other than those whose [character code] is defined as a white space in [XML 1.1].

1.4 **If** ***T***.{content type}.{variety} = ***element-only*** or ***T***.{content type}.{variety} = ***mixed***, **then** the sequence of element information items in ***E***.[children], if any, taken in order, is ·valid· with respect to ***T***.{content type}, as defined in Element Sequence Locally Valid (Complex Content) (§3.4.4.3).

2 For each attribute information item ***A*** in ***E***.[attributes] excepting those named `xsi:type`, `xsi:nil`, `xsi:schemaLocation`, or `xsi:noNamespaceSchemaLocation` (see Built-in Attribute Declarations (§3.2.7)), the appropriate **case** among the following is true:

2.1 **If** there is among the {attribute uses} an attribute use ***U*** whose {attribute declaration} has the same expanded name as ***A*** , **then** ***A*** is locally ·valid· with respect to ***U*** as per Attribute Locally Valid (Use) (§3.5.4). In this case ***U***.{attribute declaration} is the ·context-determined declaration· for ***A*** with respect to Schema-Validity Assessment (Attribute) (§3.2.4.3) and Assessment Outcome (Attribute) (§3.2.5.1). Also ***A*** is ·attributed to· ***U***.

2.2 **otherwise all** of the following are true:

2.2.1 There is an {attribute wildcard}.

2.2.2 ***A*** is ·valid· with respect to it as defined in Item Valid (Wildcard) (§3.10.4.1). In this case ***A*** is ·attributed to· the {attribute wildcard}.

3 For each attribute use ***U*** in ***T***.{attribute uses}, if ***U***.{required} = ***true***, then ***U***.{attribute declaration} has the same expanded name as one of the attribute information items in ***E***.[attributes].

Note: It is a consequence that (with few exceptions) each such ***U*** will have the matching attribute information item ·attributed to· it by clause 2.1 above. The exceptions are uses of `xsi:type` and the other attributes named in clause 2.1; no ·attribution· is performed for them.

4 For each ·defaulted attribute· ***A*** belonging to ***E***, the {lexical form} of ***A***'s ·effective value constraint· is ·valid· with respect to ***A***.{attribute declaration}.{type definition} as defined by String Valid (§3.16.4).

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 2.

5 For each element information item in ***E***.[children] and each attribute information item in ***E***.[attributes], if neither the ·governing type definition· nor the ·locally declared type· is ·absent·, then the ·governing type definition· is the same as, or is ·validly substitutable· for, the ·locally declared type·, ·without limitation·.

6 ***E*** is ·valid· with respect to each of the assertions in ***T***.{assertions} as per Assertion Satisfied (§3.13.4.1).

[Definition:] A **defaulted attribute** belonging to an element information item ***E*** ·governed by· a complex type ***T*** is any Attribute Use ***U*** for which **all** of the following are true:

1 ***U*** is a member of ***T***.{attribute uses}.

2 ***U***.{required} = ***false***.

3 ***U***'s ·effective value constraint· is not ·absent·.

4 ***U***.{attribute declaration} is not one of the Attribute Declarations from Built-in Attribute Declarations (§3.2.7).

5 ***U***.{attribute declaration} does not match any of the attribute information items in ***E***. [attributes] as per clause 2.1 of Element Locally Valid (Complex Type) (§3.4.4.2) above.

3.4.4.3 Element Sequence Locally Valid (Complex Content)

Validation Rule: Element Sequence Locally Valid (Complex Content)

For a sequence ***S*** (possibly empty) of element information items to be locally ·valid· with respect to a Content Type ***CT***, the appropriate **case** among the following MUST be true:

1 **If** ***CT***.{open content} is ·absent· , **then** ***S*** is ·valid· with respect to ***CT***.{particle}, as defined in Element Sequence Locally Valid (Particle) (§3.9.4.2).

2 **If** ***CT***.{open content}.{mode} = ***suffix*** , **then** ***S*** can be represented as two subsequences ***S1*** and ***S2*** (either can be empty) such that **all** of the following are true:

2.1 ***S*** = ***S1*** + ***S2***

2.2 ***S1*** is ·valid· with respect to ***CT***.{particle}, as defined in Element Sequence Locally Valid (Particle) (§3.9.4.2).

2.3 If ***S2*** is not empty, let ***E*** be the first element in ***S2***, then ***S1*** + ***E*** does *not* have a ·path· in ***CT***.{particle}

2.4 Every element in ***S2*** is ·valid· with respect to the wildcard ***CT***.{open content}.{wildcard}, as defined in Item Valid (Wildcard) (§3.10.4.1).

3 **otherwise** (***CT***.{open content}.{mode} = ***interleave***) ***S*** can be represented as two subsequences ***S1*** and ***S2*** (either can be empty) such that **all** of the following are true:

3.1 ***S*** is a member of ***S1*** × ***S2*** (where × is the interleave operator, see All-groups (§3.8.4.1.3))

3.2 ***S1*** is ·valid· with respect to ***CT***.{particle}, as defined in Element Sequence Locally Valid (Particle) (§3.9.4.2).

3.3 For every element ***E*** in ***S2***, let ***S3*** be the longest prefix of ***S1*** where members of ***S3*** are before ***E*** in ***S***, then ***S3*** + ***E*** does *not* have a ·path· in ***CT***.{particle}

3.4 Every element in ***S2*** is ·valid· with respect to the wildcard ***CT***.{open content}.{wildcard}, as defined in Item Valid (Wildcard) (§3.10.4.1).

[Definition:] A sequence of element information items is **locally valid** with respect to a Content Type if and only if it satisfies Element Sequence Locally Valid (Complex Content) (§3.4.4.3) with respect to that Content Type.

3.4.4.4 Attribution

[Definition:] During ·validation· of an element information item against its (complex) ·governing type definition·, associations between element and attribute information items among the [children] and [attributes] on the one hand, and the attribute uses, attribute wildcard, particles and open content property record on the other, are established. The element or attribute information item is **attributed to** the corresponding component.

When an attribute information item has the same expanded name as the {attribute declaration} of an Attribute Use, then the item is **attributed to** that attribute use. Otherwise, if the item matches an attribute wildcard, as described in Item Valid (Wildcard) (§3.10.4.1), then the item is **attributed to** that wildcard. Otherwise the item is *not* **attributed to** any component.

When a sequence ***S*** of [child] element information items are checked against the ·governing type definition·'s {content type} ***CT***, let ***S1*** and ***S2*** be subsequences of ***S*** such that

1. ***S*** is a member of ***S1*** × ***S2***

2. ***S1*** is a prefix of some element sequence that is ·locally valid· with respect to ***CT***, as defined in Element Sequence Locally Valid (Complex Content) (§3.4.4.3).

3. for every element ***E*** in ***S2***, let ***S3*** be the longest prefix of ***S1*** where members of ***S3*** are before ***E*** in ***S***, then ***S3*** + ***E*** is *not* a prefix of any element sequence that is ·locally valid· with respect to ***CT***.

Then members of ***S1*** that form a ·validation-path· in ***CT***.{particle} (see Element Sequence Locally Valid (Complex Content) (§3.4.4.3)) are **attributed to** the particles they match up with in the ·validation-path·. Other members of ***S1*** are **attributed to** the {open content} of ***CT***. Members of ***S2*** are *not* **attributed to** any component.

> **Note:** The above definition makes sure that ·attribution· happens even when the sequence of element information items is not ·locally valid· with respect to a Content Type. For example, if a complex type definition has the following content model:
>
> ```
> <xs:sequence>
>   <xs:element name="a"/>
>   <xs:element name="b"/>
>   <xs:element name="c"/>
> </xs:sequence>
> ```
>
> and an input sequence
>
> ```
> <a/><b/><d/>
> ```
>
> Then the element <a> is ·attributed· to the particle whose term is the "a" element declaration. Similarly, <b> is ·attributed· to the "b" particle.

[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 also established. When an item is ·attributed· to an ·element particle·, then it is associated with the declaration which is the {term} of the particle. Similarly, when an attribute information item is ·attributed to· an Attribute Use, then the item is associated with the {attribute declaration} of that Attribute Use. Such declarations are called the **context-determined declarations**. See clause 2.1 (in Element Locally Valid (Complex Type) (§3.4.4.2)) for attribute declarations, clause 2 (in Element Sequence Locally Valid (Particle) (§3.9.4.2)) for element declarations.

3.4.5 Complex Type Definition Information Set Contributions

3.4.5.1 Attribute Default Value

Schema Information Set Contribution: Attribute Default Value

For each ·defaulted attribute·, the ·post-schema-validation infoset· has an attribute information item whose properties are as below added to the [attributes] of the element information item.

In addition, if necessary ·namespace fixup· is performed on the element information item for the {attribute declaration}'s {target namespace}.

[local name]

The {attribute declaration}'s {name}.

[namespace name]

The {attribute declaration}'s {target namespace}.

[prefix]

If the {attribute declaration} has a ·non-absent· {target namespace} ***N*** and the [in-scope namespaces] property binds ***N*** to one or more prefixes, then a namespace prefix bound to ***N*** in the [in-scope namespaces] property of the element information item in the ·post-schema-validation infoset·.

If more than one prefix is bound to ***N*** in the [in-scope namespaces], it is ·implementation-dependent· which of those prefixes is used.

If the {attribute declaration} has a ·non-absent· {target namespace} ***N*** and no prefix is bound to ***N*** in the [in-scope namespaces] property, then (as described in the discussion of ·namespace fixup·) an ·implementation-dependent· prefix.

> **Note:** In this case (i.e. when ·namespace fixup· is performed), the [in-scope namespaces] property is also augmented and an appropriate namespace attribute is added to the parent element's [namespace attributes].

If the {attribute declaration}'s {target namespace} is ·absent·, then ·absent·.

[normalized value]

The ·effective value constraint·'s {lexical form}.

[owner element]

The element information item being assessed.

[schema normalized value]

The ·effective value constraint·'s {lexical form}.

[schema actual value]

The ·effective value constraint·'s {value}.

[schema default]

The ·effective value constraint·'s {lexical form}.

[validation context]

The nearest ancestor element information item with a [schema information] property.

[validity]

valid.

[validation attempted]

full.

[schema specified]

schema.

The added items also have [type definition] (and [member type definition] and [member type definitions] if appropriate) properties, and their lighter-weight alternatives, as specified in Attribute Validated by Type (§3.2.5.4).

[Definition:] When default values are supplied for attributes, **namespace fixup** may be required, to ensure that the ·post-schema-validation infoset· includes the namespace bindings needed and maintains the consistency of the namespace information in the infoset. To perform namespace fixup on an element information item ***E*** for a namespace ***N***:

1 If the [in-scope namespaces] of ***E*** binds a prefix to ***N***, no namespace fixup is needed; the properties of ***E*** are not changed.
2 Otherwise, first select some prefix ***P*** which is not bound by the [in-scope namespaces] of ***E*** (the choice of prefix is ·implementation-dependent·).
3 Add an entry to the [in-scope namespaces] of ***E*** binding ***P*** to ***N***.
4 Add a namespace attribute to the [namespace attributes] of ***E***.
5 Maintain the consistency of the information set by adjusting the namespace bindings on the descendants of ***E***. This may be done in either of two ways:
  5.1 Add the binding of ***P*** to ***N*** to the [in-scope namespaces] of all descendants of ***E***, except where that binding is overridden by another binding for ***P***.
  5.2 Add to the [namespace attributes] of each child of ***E*** a namespace attribute which undeclares the binding for ***P*** (i.e. a namespace attribute for prefix ***P*** whose ·normalized value· is the empty string), unless that child already has a namespace declaration for prefix ***P***. Note that this approach is possible only if [XML Namespaces 1.1] is in use, rather than [Namespaces in XML 1.0].

The choice between the two methods of maintaining consistency in the information set is ·implementation-dependent·.

If the ·namespace fixup· is occasioned by a defaulted attribute with a non-absent target namespace, then (as noted above), the [prefix] of the attribute information item supplied in the ·post-schema-validation infoset· is set to ***P***.

Note: When a default value of type QName or NOTATION is applied, it is ·implementation-dependent· whether ·namespace fixup· occurs; if it does not, the prefix used in the lexical representation (in [normalized value] or [schema normalized value]) will not necessarily map to the namespace name of the value (in [schema actual value]). To reduce problems and confusion, users may prefer to ensure that the required namespace information item is present in the input infoset.

3.4.5.2 Match Information

Schema Information Set Contribution: Match Information

To allow users of the ·post-schema-validation infoset· to distinguish element information items which are ·attributed· to ·element particles· from those ·attributed· to ·wildcard particles·, if and only if the local ·validity· of an element information item has been assessed as defined by Element Locally Valid (Complex Type) (§3.4.4.2), then each attribute information item in its [attributes] has the following properties in the ·post-schema-validation infoset·:

PSVI Contributions for attribute information items

`[attribute attribution]`
The appropriate **case** among the following:
1 **If** the attribute information item is ·attributed· to an Attribute Use, **then** an ·item isomorphic· to the Attribute Use.

2 **If** the attribute information item is ·attributed· to an {attribute wildcard}, **then** an ·item isomorphic· to the attribute wildcard.
3 **otherwise** (the item is not ·attributed· to an Attribute Use or an {attribute wildcard}) ·absent·.

`[match information]`
A keyword indicating what kind of component the attribute information item is ·attributed· to. The appropriate **case** among the following:
1 **If** the item is ·attributed· to an Attribute Use, **then** ***attribute***
2 **If** the item is ·attributed· to a ***strict*** {attribute wildcard}, **then** ***strict***
3 **If** the item is ·attributed· to a ***lax*** {attribute wildcard}, **then** ***lax***
4 **If** the item is ·attributed· to a ***skip*** {attribute wildcard}, **then** ***skip***
5 **otherwise** (the item is not ·attributed· to an Attribute Use or an {attribute wildcard}) ***none***

And each element information item in its [children] has the following properties in the ·post-schema-validation infoset·:

PSVI Contributions for element information items

`[element attribution]`
The appropriate **case** among the following:
1 **If** the element information item is ·attributed· to an ·element particle· or a ·wildcard particle·, **then** an ·item isomorphic· to the Particle.
2 **If** the item is ·attributed· to an Open Content, **then** an ·item isomorphic· to the Open Content.
3 **otherwise** (the item is not ·attributed· to a Particle or an Open Content) ·absent·.

`[match information]`
A keyword indicating what kind of Particle the item is ·attributed· to. The appropriate **case** among the following:
1 **If** the item is ·attributed· to an ·element particle·, **then** ***element***
2 **If** the item is ·attributed· to a ***strict*** ·wildcard particle·, **then** ***strict***
3 **If** the item is ·attributed· to a ***lax*** ·wildcard particle·, **then** ***lax***
4 **If** the item is ·attributed· to a ***skip*** ·wildcard particle·, **then** ***skip***
5 **If** the item is ·attributed· to an Open Content, **then** ***open***
6 **otherwise** (the item is not ·attributed· to a Particle or an Open Content) ***none***

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.

3.4.6.1 Complex Type Definition Properties Correct

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 are 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} is ***extension***.
3 There are no circular definitions, except for that of ·`xs:anyType`·. That is, it is possible to reach the definition of ·`xs:anyType`· by repeatedly following the {base type definition}.

4 No two distinct members of the {attribute uses} have {attribute declaration}s with the same expanded name.
5 If {content type}.{open content} is ·non-absent·, then {content type}.{variety} is either ***element-only*** or ***mixed***.

3.4.6.2 Derivation Valid (Extension)

Schema Component Constraint: Derivation Valid (Extension)

For every complex type ***T*** with {base type definition} ***B*** where ***T***.{derivation method} = ***extension***, the appropriate **case** among the following MUST be true:
1 **If** ***B*** is a complex type definition, **then all** of the following are true:
  1.1 ***B***.{final} does not contain ***extension***.
  1.2 ***B***.{attribute uses} is a subset of ***T***.{attribute uses}. That is, for every attribute use ***U*** in ***B***.{attribute uses}, there is an attribute use in ***T***.{attribute uses} whose properties, recursively, are identical to those of ***U***.
  1.3 If ***B*** has an {attribute wildcard}, then ***T*** also has one, and ***B***.{attribute wildcard}.{namespace constraint} is a subset of ***T***.{attribute wildcard}.{namespace constraint}, as defined by Wildcard Subset (§3.10.6.2).
  1.4 **One** of the following is true:
    1.4.1 ***B*** and ***T*** both have {content type}.{variety} = ***simple*** and both have the same {content type}.{simple type definition}.
    1.4.2 ***B*** and ***T*** both have {content type}.{variety} = ***empty***.
    1.4.3 **All** of the following are true:
      1.4.3.1 ***T***.{content type}.{variety} = ***element-only*** or ***mixed***.
      1.4.3.2 **One** of the following is true:
        1.4.3.2.1 ***B***.{content type}.{variety} = ***empty***.
        1.4.3.2.2 **All** of the following are true:
          1.4.3.2.2.1 Both ***B*** and ***T*** have {content type}.{variety} = ***mixed*** or both have {content type}.{variety} = ***element-only***.
          1.4.3.2.2.2 ***T***.{content type}.{particle} is a ·valid extension· of ***B***.{content type}.{particle}, as defined in Particle Valid (Extension) (§3.9.6.2).
          1.4.3.2.2.3 **One or more** of the following is true:
            1.4.3.2.2.3.1 ***B***.{content type}.{open content} (call it **BOT**) is ·absent·.
            1.4.3.2.2.3.2 ***T***.{content type}.{open content} (call it **EOT**) has {mode} ***interleave***.
            1.4.3.2.2.3.3 Both **BOT** and **EOT** have {mode} ***suffix***.
          1.4.3.2.2.4 If neither **BOT** nor **EOT** is ·absent·, then **BOT**.{wildcard}.{namespace constraint} is a subset of **EOT**.{wildcard}.{namespace constraint}, as defined by Wildcard Subset (§3.10.6.2).
  1.5 It is in principle possible to ·derive· ***T*** 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 ·`xs:anyType`·.

    Note: This requirement ensures that nothing removed by a restriction is subsequently added back by an extension in an incompatible way (for example, with a conflicting type assignment or value constraint).

    Constructing the intermediate type definition to check this constraint is straightforward: simply re-order the ·derivation· to put all the extension steps first, then collapse them into a single extension. If the resulting definition can be the basis for a valid restriction to the desired definition, the constraint is satisfied.

  1.6 For any element or attribute information item, its ·locally declared type· within ***T*** is ·validly substitutable· for the ·locally declared type· within ***B***, ·without limitation·, if neither is ·absent·.

1.7 ***B***.{assertions} is a prefix of ***T***.{assertions}.

2 **If** ***B*** is a simple type definition, **then all** of the following are true:

2.1 ***T***.{content type}.{variety} = ***simple*** and ***T***.{content type}.{simple type definition} = ***B***.

2.2 ***B***.{final} does not contain ***extension***.

.

[Definition:] A complex type **T** is a **valid extension** of its {base type definition} if and only if **T**.{derivation method} = ***extension*** and ***T*** satisfies the constraint Derivation Valid (Extension) (§3.4.6.2).

3.4.6.3 Derivation Valid (Restriction, Complex)

Schema Component Constraint: Derivation Valid (Restriction, Complex)

For every complex type ***T*** with {base type definition} ***B*** where ***T***.{derivation method} = ***restriction***, **all** of the following MUST be true:

1 ***B*** is a complex type definition whose {final} does not contain ***restriction***.

2 **One or more** of the following is true:

2.1 ***B*** is ·xs:anyType·.

2.2 **All** of the following are true:

2.2.1 ***T***.{content type}.{variety} = ***simple***

2.2.2 **One** of the following is true:

2.2.2.1 Let S_B be ***B***.{content type}.{simple type definition}, and S_T be ***T***.{content type}.{simple type definition}. Then S_T is validly derived from S_B as defined in Type Derivation OK (Simple) (§3.16.6.3).

2.2.2.2 ***B***.{content type}.{variety} = ***mixed*** and ***B***.{content type}.{particle} is a Particle which is ·emptiable· as defined in Particle Emptiable (§3.9.6.3).

2.3 **All** of the following are true:

2.3.1 ***T***.{content type}.{variety} = ***empty***.

2.3.2 **One** of the following is true:

2.3.2.1 ***B***.{content type}.{variety} = ***empty***.

2.3.2.2 ***B***.{content type}.{variety} = ***element-only*** or ***mixed***, and ***B***.{content type}.{particle} is a Particle which is ·emptiable· as defined in Particle Emptiable (§3.9.6.3).

2.4 **All** of the following are true:

2.4.1 **One** of the following is true:

2.4.1.1 ***T***.{content type}.{variety} = ***element-only*** and ***B***.{content type}.{variety} = ***element-only*** or ***mixed***.

2.4.1.2 ***T*** and ***B*** both have {content type}.{variety} = ***mixed***.

2.4.2 The {content type} of ***T*** ·restricts· that of ***B*** as defined in Content type restricts (Complex Content) (§3.4.6.4).

3 For every element information item ***E***, if the [attributes] of ***E*** satisfy clause 2 and clause 3 of Element Locally Valid (Complex Type) (§3.4.4.2) with respect to ***T***, then they also satisfy the same clauses with respect to ***B***, and for every attribute information item ***A*** in ***E***.[attributes], ***B***'s ·default binding· for ***A*** ·subsumes· that defined by ***T***.

4 For any element or attribute information item, its ·locally declared type· within ***T*** is ·validly substitutable· for its ·locally declared type· within ***B***, subject to the blocking keywords {***extension***, ***list***, ***union***}, if the item has a ·locally declared type· both in ***T*** and in ***B***.

5 ***B***.{assertions} is a prefix of ***T***.{assertions}.

[Definition:] A complex type definition with {derivation method} = ***restriction*** is a **valid restriction** of its {base type definition} if and only if the constraint Derivation Valid (Restriction, Complex) (§3.4.6.3) is satisfied.

Note: Valid restriction involves both a subset relation on the set of elements valid against ***T*** and those valid against ***B***, and a derivation relation, explicit in the type hierarchy, between the types assigned to attributes and child elements by ***T*** and those assigned to the same attributes and children by ***B***.

The constraint just given, like other constraints on schemas, MUST be satisfied by every complex type ***T*** to which it applies.

However, under certain conditions conforming processors need not (although they MAY) detect some violations of this constraint. If (1) the type definition being checked has ***T*** . {content type} . {particle} . {term} . {compositor} = ***all*** and (2) an implementation is unable to determine by examination of the schema in isolation whether or not clause 2.4.2 is satisfied, then the implementation MAY provisionally accept the derivation. If any instance encountered in the ·assessment· episode is valid against ***T*** but not against ***T***.{base type definition}, then the derivation of ***T*** does not satisfy this constraint, the schema does not conform to this specification, and no ·assessment· can be performed using that schema.

It is ·implementation-defined· whether a processor (a) always detects violations of clause 2.4.2 by examination of the schema in isolation, (b) detects them only when some element information item in the input document is valid against ***T*** but not against ***T***.{base type definition}, or (c) sometimes detects such violations by examination of the schema in isolation and sometimes not. In the latter case, the circumstances in which the processor does one or the other are ·implementation-dependent·.

3.4.6.4 Content Type Restricts (Complex Content)

Schema Component Constraint: Content type restricts (Complex Content)

[Definition:] A Content Type **R** (for "restriction") with complex content (i.e. one with a ·non-absent· {particle}) **restricts** another Content Type **B** (for "base") with complex content if and only if **all** of the following are true:

1 Every sequence of element information items which is ·locally valid· with respect to **R** is also ·locally valid· with respect to **B**.

2 For all sequences of element information items **ES** which are ·locally valid· with respect to **R**, for all elements **E** in **ES**, **B**'s ·default binding· for **E** ·subsumes· that defined by **R**.

[Definition:] When a sequence of element information items **ES** is ·locally valid· with respect to a Content Type **CT** or when a set of attribute information items **AS** satisfies clause 2 and clause 3 of Element Locally Valid (Complex Type) (§3.4.4.2) with respect to a Complex Type Definition, there is a (partial) functional mapping from the element information items ***E*** in the sequence ***ES*** or the attribute information items in ***AS*** to a **default binding** for the item, where the default binding is an Element Declaration, an Attribute Use, or one of the keywords ***strict***, ***lax***, or ***skip***, as follows:

1 When the item has a ·governing element declaration·, the default binding is that Element Declaration.

2 When the item has a ·governing attribute declaration· and it is ·attributed· to an Attribute Use, the default binding is that Attribute Use.

3 When the item has a ·governing attribute declaration· and it is ·attributed· to an attribute wildcard, the default binding is an Attribute Use whose {attribute declaration} is the ·governing attribute declaration·, whose {value constraint} is ·absent·, and whose {inheritable} is the ·governing attribute declaration·'s {inheritable} (the other properties in the Attribute Use are not relevant).

4 When the item is ·attributed· to a ***strict*** ·wildcard particle· or attribute wildcard or an Open Content with a ***strict*** Wildcard and it does not have a ·governing element declaration· or a ·governing attribute declaration·, then the default binding is the keyword ***strict***.

5 When the item is ·attributed· to a ***lax*** ·wildcard particle· or attribute wildcard or an Open Content with a ***lax*** Wildcard and it does not have a ·governing element declaration· or a ·governing attribute declaration·, then the default binding is the keyword ***lax***.
6 When the item is ·attributed· to a ***skip*** ·wildcard particle· or attribute wildcard or an Open Content with a ***skip*** Wildcard then the default binding is the keyword ***skip***.

[Definition:] A ·default binding· **G** (for general) **subsumes** another ·default binding· **S** (for specific) if and only if **one** of the following is true
1 **G** is ***skip***.
2 **G** is ***lax*** and **S** is not ***skip***.
3 Both **G** and **S** are ***strict***.
4 **G** and **S** are both Element Declarations and **all** of the following are true:
  4.1 Either **G**.{nillable} = ***true*** or **S**.{nillable} = ***false***.
  4.2 Either **G** has no {value constraint}, or it is not ***fixed***, or **S** has a ***fixed*** {value constraint} with an equal or identical value.
  4.3 **S**.{identity-constraint definitions} is a superset of **G**.{identity-constraint definitions}.
  4.4 **S** disallows a superset of the substitutions that **G** does.
  4.5 **S**'s declared {type definition} is ·validly substitutable as a restriction· for **G**'s declared {type definition}.
  4.6 **S**.{type table} and **G**.{type table} either are both ·absent· or are both ·present· and ·equivalent·.
5 **G** and **S** are both Attribute Uses and **all** of the following are true:
  5.1 ***S***.{attribute declaration}.{type definition} is validly ·derived· from ***G***.{attribute declaration}.{type definition}, as defined in Type Derivation OK (Simple) (§3.16.6.3).
  5.2 Let **GVC** be ***G***'s ·effective value constraint· and **SVC** be ***S***'s ·effective value constraint·, then **one or more** of the following is true:
    5.2.1 **GVC** is ·absent· or has {variety} ***default***.
    5.2.2 **SVC**.{variety} = ***fixed*** and **SVC**.{value} is equal or identical to **GVC**.{value}.
  5.3 ***G***.{inheritable} = ***S***.{inheritable}.

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.

Note: To restrict away a local element declaration that ·competes· with a wildcard, use a wildcard in the derived type that explicitly disallows the element's expanded name. See the example given in XML Representation of Complex Type Definition Schema Components (§3.4.2).

3.4.6.5 Type Derivation OK (Complex)

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) subject to the blocking keywords in 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** is not in the subset.
2 **One or more** of the following is true:
  2.1 **B** = **D**.
  2.2 **B** = **D**.{base type definition}.
  2.3 **All** of the following are true:

2.3.1 **D**.{base type definition} ≠ ·xs:anyType·.
2.3.2 The appropriate **case** among the following is true:
2.3.2.1 **If D**.{base type definition} is complex, **then** it is validly ·derived· from **B** subject to the subset as defined by this constraint.
2.3.2.2 **If D**.{base type definition} is simple, **then** it is validly ·derived· from **B** subject to the subset as defined in Type Derivation OK (Simple) (§3.16.6.3).

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 type 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 it is possible that conforming implementations will disagree as to whether components are identical.

Note: When a complex type definition ***S*** is said to be "validly ·derived·" from a type definition ***T***, without mention of any specific set of blocking keywords, or with the explicit phrase "without limitation", then what is meant is that ***S*** is validly derived from ***T***, subject to the empty set of blocking keywords, i.e. without any particular limitations.

3.4.7 Built-in Complex Type Definition

There is a complex type definition for ·xs:anyType· present in every schema by definition. It has the following properties:

Complex Type Definition of anyType	
Property	**Value**
{name}	anyType
{target namespace}	http://www.w3.org/2001/XMLSchema
{base type definition}	itself
{derivation method}	***restriction***
{content type}	A Content Type as follows:

Property	Value
{variety}	***mixed***
{particle}	a Particle with the properties shown below in Outer Particle for Content Type of anyType (§3.4.7).
{simple type definition}	·absent·

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

Property	Value
{namespace constraint}	A Namespace Constraint with the following properties:

Property	Value
{variety}	***any***
{namespaces}	The empty set
{disallowed names}	The empty set

Property	Value
{process contents}	***lax***

{final}	The empty set
{context}	·absent·
{prohibited substitutions}	The empty set
{assertions}	The empty sequence
{abstract}	***false***

The outer particle of ·xs:anyType· contains a sequence with a single term:

Outer Particle for Content Type of anyType

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 properties shown below in Inner Particle for Content Type of anyType (§3.4.7).

The inner particle of ·xs:anyType· contains a wildcard which matches any element:

Inner Particle for Content Type of anyType

Property	Value
{min occurs}	0
{max occurs}	***unbounded***
{term}	a wildcard with the following properties:

Property	Value
{namespace constraint}	A Namespace Constraint with the following properties:

Property	Value
{variety}	***any***
{namespaces}	The empty set
{disallowed names}	The empty set

{process contents}	***lax***

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 Attribute Uses

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:decimal" 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, a kind of Annotated Component

{annotations}
: A sequence of Annotation components.

{required}
: An xs:boolean value. Required.

{attribute declaration}
: An Attribute Declaration component. Required.

{value constraint}
: A Value Constraint property record. Optional.

{inheritable}
: An xs:boolean value. Required.

Property Record: Value Constraint

{variety}
: One of {***default***, ***fixed***}. Required.

{value}
: An ·actual value·. Required.

{lexical form}
: A character string. Required.

{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, or a value equal or identical to it.

See Annotations (§3.15) for information on the role of the {annotations} property.

3.5.2 XML Representation of Attribute Use Schema 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

[Definition:] The **effective value constraint** of an attribute use ***U*** is ***U***.{value constraint}, if present, otherwise ***U***.{attribute declaration}.{value constraint}, if present, otherwise the **effective value constraint** is ·absent·.

Validation Rule: Attribute Locally Valid (Use)

For an attribute information item to be ·valid· with respect to an attribute use its ·actual value· MUST be equal or identical to the {value} of the attribute use's {value constraint}, if it is present and has {variety} ***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 Attribute Uses (§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 ***U*** are 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 ***U***.{value constraint} is not ·absent·, then it is a valid default with respect to ***U***.{attribute declaration}.{type definition} as defined in Simple Default Valid (§3.2.6.2).

3 If ***U***.{attribute declaration} has {value constraint}.{variety} = ***fixed*** and ***U*** itself has a {value constraint}, then ***U***.{value constraint}.{variety} = ***fixed*** and ***U***.{value constraint}.{value} is identical to ***U***.{attribute declaration}.{value constraint}.{value}.

3.6 Attribute Group Definitions

A schema can name a group of attribute declarations so that they can 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.

The example above illustrates the pattern mentioned in XML Representations of Components (§3.1.2): The same element, in this case `attributeGroup`, serves both to define and to incorporate by reference. In the first `attributeGroup` element in the example, the `name` attribute is required and the `ref` attribute is forbidden; in the second the `ref` attribute is required, the `name` attribute is forbidden.

3.6.1 The Attribute Group Definition Schema Component

The attribute group definition schema component has the following properties:

Schema Component: Attribute Group Definition, a kind of Annotated Component

{annotations}
A sequence of Annotation components.

{name}
An xs:NCName value. Required.

{target namespace}
An xs:anyURI value. Optional.

{attribute uses}
A set of Attribute Use components.

{attribute wildcard}
A Wildcard component. Optional.

Attribute groups are identified by their {name} and {target namespace}; attribute group identities MUST be unique within an ·XSD schema·. See References to schema components across namespaces (<import>) (§4.2.6) for the use of component identifiers when importing one schema into another.

{attribute uses} is a set of 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 (§3.15) for information on the role of the {annotations} property.

3.6.2 XML Representation of Attribute Group Definition Schema Components

3.6.2.1 XML Mapping Rule for Named Attribute Groups

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 after the appropriate ·pre-processing· and the properties of the component it corresponds to are given in this section.

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

> **Note:** If the <attributeGroup> is a child of <override>, and it overrides a corresponding declaration in the ·target set· of its parent, it will also correspond to an attribute group definition as shown below. See Overriding component definitions (<override>) (§4.2.5) for details.

XML Mapping Summary for 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 <schema> ancestor element information item if present, otherwise ·absent·.
{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. **Note:** As described below, circular references from <attributeGroup> to <attributeGroup> are not errors.
{attribute wildcard}	The Wildcard determined by applying the attribute-wildcard mapping described in Common Rules for Attribute Wildcards (§3.6.2.2) to the <attributeGroup> element information item.
{annotations}	The ·annotation mapping· of the <attributeGroup> element and its <attributeGroup> [children], if present, as defined in XML Representation of Annotation Schema Components (§3.15.2).

> **Note:** It is a consequence of this rule and the rule in XML Representation of Complex Type Definition Schema Components (§3.4.2) that any annotations

specified in attribute group references are included in the sequence of Annotations of the enclosing Complex Type Definition or Attribute Group Definition components.

The rules given above for {attribute uses} and {attribute wildcard} specify that if an <attributeGroup> element ***A*** contains a reference to another attribute group ***B*** (i.e. ***A***'s [children] include an <attributeGroup> with a `ref` attribute pointing at ***B***), then ***A*** maps to an Attribute Group Definition component whose {attribute uses} reflect not only the <attribute> [children] of ***A*** but also those of ***B*** and of any <attributeGroup> elements referred to in ***B***. The same is true for attribute groups referred to from complex types.

Circular reference is *not* disallowed. That is, it is not an error if ***B***, or some <attributeGroup> element referred to by ***B*** (directly, or indirectly at some remove) contains a reference to ***A***. An <attributeGroup> element involved in such a reference cycle maps to a component whose {attribute uses} and {attribute wildcard} properties reflect all the <attribute> and <any> elements contained in, or referred to (directly or indirectly) by elements in the cycle.

> **Note:** In version 1.0 of this specification, circular group reference was not allowed except in the [children] of <redefine>. As described above, this version allows it. The effect is to take the transitive closure of the reference relation between <attributeGroup> elements and take into account all their {attribute uses} and {attribute wildcard} properties.

3.6.2.2 Common Rules for Attribute Wildcards

The following mapping for attribute-wildcards forms part of the XML mapping rules for different kinds of source declaration (most prominently <attributeGroup>). It can be applied to any element which can have an <anyAttribute> element as a child, and produces as a result either a Wildcard or the special value ·absent·. The mapping depends on the concept of the ·local wildcard·:

[Definition:] The **local wildcard** of an element information item ***E*** is the appropriate **case** among the following:

1 **If** ***E*** has an <anyAttribute> child, **then** the Wildcard mapped to by the <anyAttribute> element using the wildcard mapping set out in XML Representation of Wildcard Schema Components (§3.10.2);

2 **otherwise** ·absent·.

The mapping is defined as follows:

1 Let ***L*** be the ·local wildcard·

2 Let ***W*** be a sequence containing all the ·non-absent· {attribute wildcard}s of the attribute groups referenced by ***E***, in document order.

3 The value is then determined by the appropriate **case** among the following:

  3.1 **If** ***W*** is empty, **then** the ·local wildcard· ***L***.

  3.2 **otherwise** the appropriate **case** among the following:

    3.2.1 **If** ***L*** is ·non-absent·, **then** a wildcard whose properties are as follows:

Property	Value
{process contents}	***L***.{process contents}
{annotations}	***L***.{annotations}
{namespace constraint}	the wildcard intersection of ***L***.{namespace constraint} and of the {namespace constraint}s of all the the wildcards in ***W***, as defined in Attribute Wildcard Intersection (§3.10.6.4)

3.2.2 **otherwise** (no <anyAttribute> is present) a wildcard whose properties are as follows:

Property	Value
{process contents}	The {process contents} of the first wildcard in ***W***
{namespace constraint}	The wildcard intersection of the {namespace constraint}s of all the wildcards in ***W***, as defined in Attribute Wildcard Intersection (§3.10.6.4)
{annotations}	·The empty sequence·

3.6.3 Constraints on XML Representations of Attribute Group Definitions

Schema Representation Constraint: Attribute Group Definition Representation OK

None as such.

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 are 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 No two distinct members of the {attribute uses} have {attribute declaration}s with the same expanded name.

3.7 Model Group Definitions

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

Model group definitions are provided primarily for reference from the XML Representation of Complex Type Definition Schema Components (§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, a kind of Annotated Component

{annotations}
: A sequence of Annotation components.

{name}
: An xs:NCName value. Required.

{target namespace}
: An xs:anyURI value. Optional.

{model group}
: A Model Group component. Required.

Model group definitions are identified by their {name} and {target namespace}; model group identities MUST be unique within an ·XSD schema·. See References to schema components across namespaces (<import>) (§4.2.6) 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 for which the model group definition provides a name.

See Annotations (§3.15) for information on the role of the {annotations} 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 after the appropriate ·pre-processing· and the properties of the component it corresponds to are given in this section.

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 the item has <schema> or <redefine> as its parent (in which case there will be a `name` [attribute], then the item maps to a model group definition component with properties as follows:

> **Note:** If the item is a child of <override>, it will also have a `name` [attribute]; if it overrides a corresponding declaration in the ·target set· of its parent, it will also (in the overridden schema document) map to a component as described below. See Overriding component definitions (<override>) (§4.2.5) for details.

XML Mapping Summary for 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 <schema> ancestor element information item if present, otherwise ·absent·.
{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 exactly one).
{annotations}	The ·annotation mapping· of the <group> element, as defined in XML Representation of Annotation Schema Components (§3.15.2).

Otherwise, if the item has a `ref` [attribute] and does *not* have `minOccurs=maxOccurs=0` , then the <group> element maps to a particle component with properties as follows:

XML Mapping Summary for 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]
{annotations}	The ·annotation mapping· of the <group> element, as defined in XML Representation of Annotation Schema Components (§3.15.2).

Otherwise, the <group> has `minOccurs=maxOccurs=0`, in which case it maps to no component at all.

> **Note:** 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 schema documents 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.

3.7.3 Constraints on XML Representations of Model Group Definitions

None as such.

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 (§3.16)), 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 model groups is therefore recursive. [Definition:] A model group **directly contains** the particles in the value of its {particles} property. [Definition:] A model group **indirectly contains** the particles, groups, wildcards, and element declarations which are ·contained· by the particles it ·directly contains·. [Definition:] A model group **contains** the components which it either ·directly contains· or ·indirectly contains·.

Example

```
<xs:group name="otherPets">
  <xs:all>
   <xs:element name="birds"/>
   <xs:element name="fish"/>
  </xs:all>
 </xs:group>

<xs:all>
 <xs:element ref="cats"/>
 <xs:element ref="dogs"/>
 <xs:group ref="otherPets"/>
</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, a kind of Term

{annotations}
: A sequence of Annotation components.

{compositor}
: One of {***all***, ***choice***, ***sequence***}. Required.

{particles}
: A sequence of Particle components.

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***) correspond to exactly one of the specified {particles};
- (***all***) correspond to the specified {particles}. The elements can occur in any order.

When two or more element declarations contained ·directly·, ·indirectly·, or ·implicitly· in the {particles} of a model group have identical names, the type definitions of those declarations MUST be the same.

See Annotations (§3.15) for information on the role of the {annotations} 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 after the appropriate ·pre-processing· and the properties of the component they correspond to are given in this section.

XML Representation Summary: `all` Element Information Item et al.

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

XML Mapping Summary for 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.
{annotations}	The same annotations as the {annotations} of the model group. See below.

The particle just described has a Model Group as the value of its {term} property, as follows.

XML Mapping Summary for 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.
{annotations}	The ·annotation mapping· of the <all>, <choice>, or <sequence> element, whichever is present, as defined in XML Representation of Annotation Schema Components (§3.15.2).

3.8.3 Constraints on XML Representations of Model Groups

None as such.

3.8.4 Model Group Validation Rules

In order to define the validation rules for model groups clearly, it will be useful to define some basic terminology; this is done in the next two sections, before the validation rules themselves are formulated.

3.8.4.1 Language Recognition by Groups

Each model group ***M*** denotes a language ***L***(***M***), whose members are the sequences of element information items ·accepted· by ***M***.

Within ***L***(***M***) a smaller language ***V***(***M***) can be identified, which is of particular importance for schema-validity assessment. The difference between the two languages is that ***V***(***M***) enforces some constraints which are ignored in the definition of ***L***(***M***). Informally ***L***(***M***) is the set of sequences which are accepted by a model group if no account is taken of the schema component constraint Unique Particle Attribution (§3.8.6.4) or the related provisions in the validation rules which specify how to choose a unique ·path· in a non-deterministic model group. By contrast, ***V***(***M***) takes account of those constraints and includes only the sequences which are ·locally valid· against ***M***. For all model groups ***M***, ***V***(***M***) is a subset of ***L***(***M***). ***L***(***M***) and related concepts are described in this section; ***V***(***M***) is described in the next section, Principles of Validation against Groups (§3.8.4.2).

[Definition:] When a sequence ***S*** of element information items is checked against a model group ***M***, the sequence of ·basic particles· which the items of ***S*** match, in order, is a **path** of ***S*** in ***M***. For a given ***S*** and ***M***, the path of ***S*** in ***M*** is not necessarily unique. Detailed rules for the matching, and thus for the construction of paths, are given in Language Recognition by Groups (§3.8.4.1) and Principles of Validation against Particles (§3.9.4.1). Not every sequence has a path in every model group, but every sequence accepted by the model group does have a path. [Definition:] For a model group ***M*** and a sequence ***S*** in ***L***(***M***), the path of ***S*** in ***M*** is a **complete path**; prefixes of complete paths which are themselves not complete paths are **incomplete paths**. For example, in the model group

```
<xs:sequence>
 <xs:element name="a"/>
 <xs:element name="b"/>
 <xs:element name="c"/>
</xs:sequence>
```

the sequences (<a/><b/><c/>) and (<a/><b/>) have ·paths· (the first a ·complete path· and the second an ·incomplete path·), but the sequences (<a/><b/><c/><d/>) and (<a/><x/>) do not have paths.

Note: It is possible, but unusual, for a model group to have some paths which are neither complete paths, nor prefixes of complete paths. For example, the model group

```
<xs:sequence>
 <xs:element name="a"/>
 <xs:element name="b"/>
 <xs:choice/>
</xs:sequence>
```

accepts no sequences because the empty `choice` recognizes no input sequences. But the sequences (<a/>) and (<a/><b/>) have paths in the model group.

The definitions of ***L***(***M***) and ·paths· in ***M***, when ***M*** is a ·basic term· or a ·basic particle·, are given in Principles of Validation against Particles (§3.9.4.1). The definitions for groups are given below.

3.8.4.1.1 Sequences

This section defines ***L***(***M***), the set of ·paths· in ***M***, and ***V***(***M***), if ***M*** is a sequence group.

If ***M*** is a Model Group, and the {compositor} of ***M*** is ***sequence***, and the {particles} of ***M*** is the sequence P_1, P_2, ..., P_n, then ***L***(***M***) is the set of sequences $S = S_1 + S_2 + ... + Sn$ (taking "+" as the concatenation operator), where S_i is in $L(P_i)$ for $0 < i \leq n$. The sequence of sequences S_1, S_2, ..., Sn is a ·partition· of ***S***. Less formally, when ***M*** is a sequence of P_1, P_2, ... P_n, then ***L***(***M***) is the set of sequences formed by taking one sequence which is accepted by P_1, then one accepted by P_2, and so on, up through P_n, and then concatenating them together in order.

[Definition:] A **partition** of a sequence is a sequence of sub-sequences, some or all of which MAY be empty, such that concatenating all the sub-sequences yields the original sequence.

When ***M*** is a sequence group and ***S*** is a sequence of input items, the set of ·paths· of ***S*** in ***M*** is the set of all paths $Q = Q_1 + Q_2 + ... + Q_j$, where

- $j \leq n$, and
- $S = S_1 + S_2 + ... + S_j$ (i.e. S_1, S_2, ..., S_j is a ·partition· of ***S***), and
- S_i is in $L(P_i)$ for $0 < i < j$, and
- Q_i is a ·path· of S_i in P_i for $0 < i \leq j$.

Example

By this definition, some sequences which do not satisfy the entire model group nevertheless have ·paths· in a model group. For example, given the model group ***M***

```
<xs:sequence>
 <xs:element name="a"/>
 <xs:element name="b"/>
 <xs:element name="c"/>
</xs:sequence>
```

and an input sequence ***S***

<a/><b/>
where ***n*** = 3, ***j*** = 2, then S_1 is (<a/>), S_2 is (<b/>), and ***S*** has a ·path· in ***M***, even though ***S*** is not in ***L***(***M***). The ·path· has two items, first the Particle for the a element, then the Particle for the b element.

When ***M*** is a sequence group, the set ***V***(***M***) (the set of sequences ·locally valid· against ***M***) is the set of sequences ***S*** which are in ***L***(***M***) and which have a ·validation-path· in ***M***. Informally, ***V***(***M***) contains those sequences which are accepted by ***M*** and for which no element information item is ever ·attributed to· a ·wildcard particle· if it can, in context, instead be ·attributed to· an ·element particle·. There will invariably be a ·partition· of ***S*** whose members are ·locally valid· against {particles} of ***M***.

Note: For sequences with more than one ·path· in ***M***, the ·attributions· of the ·validation-path· are used in validation and for determining the contents of the ·post-schema-validation infoset·. For example, if ***M*** is

```
<xs:sequence>
 <xs:any minOccurs="0"/>
 <xs:element name="a" minOccurs="0"/>
</xs:sequence>
```

then the sequence (<a/>) has two ·paths· in ***M***, one containing just the ·wildcard particle· and the other containing just the ·element particle·. It is the latter which is a ·validation-path· and which determines which Particle the item in the input is ·attributed to·.

Note: There are model groups for which some members of ***L***(***M***) are not in ***V***(***M***). For example, if ***M*** is

```
<xs:sequence>
 <xs:any minOccurs="0"/>
 <xs:element name="a"/>
</xs:sequence>
```

then the sequence (<a/><a/>) is in ***L***(***M***), but not in ***V***(***M***), because the validation rules require that the first a be ·attributed to· the ·wildcard particle·. In a ·validation-path· the initial a will invariably be ·attributed to· the ·element particle·, and so no sequence with an initial a can be ·locally valid· against this model group.

3.8.4.1.2 Choices

This section defines ***L***(***M***), the set of ·paths· in ***M***, and ***V***(***M***), if ***M*** is a choice group.

When the {compositor} of ***M*** is ***choice***, and the {particles} of ***M*** is the sequence P_1, P_2, ..., P_n, then ***L***(***M***) is ***L***(P_1) ∪ ***L***(P_2) ∪ ... ∪ ***L***(P_n), and the set of ·paths· of ***S*** in ***P*** is the set ***Q*** = $Q_1 \cup Q_2 \cup ... \cup Q_n$, where Q_i is the set of ·paths· of ***S*** in P_i, for $0 < i \leq n$. Less formally, when ***M*** is a choice of P_1, P_2, ... P_n, then ***L***(***M***) contains any sequence accepted by any of the particles P_1, P_2, ... P_n, and any ·path· of ***S*** in any of the particles P_1, P_2, ... P_n is a ·path· of ***S*** in ***P***.

The set ***V***(***M***) (the set of sequences ·locally valid· against ***M***) is the set of sequences ***S*** which are in ***L***(***M***) and which have a ·validation-path· in ***M***. In effect, this means that if one of the choices in ***M*** ·attributes· an initial element information item to a ·wildcard

particle·, and another ·attributes· the same item to an ·element particle·, then the latter choice is used for validation.

Note: For example, if ***M*** is

```
<xs:choice>
 <xs:any/>
 <xs:element name="a"/>
</xs:choice>
```

then the ·validation-path· for the sequence (<a/>) contains just the ·element particle· and it is to the ·element particle· that the input element will be ·attributed·; the alternate ·path· containing just the ·wildcard particle· is not relevant for validation as defined in this specification.

3.8.4.1.3 All-groups

This section defines ***L***(***M***), the set of ·paths· in ***M***, and ***V***(***M***), if ***M*** is an all-group.

When the {compositor} of ***M*** is ***all***, and the {particles} of ***M*** is the sequence P_1, P_2, ..., P_n, then ***L***(***M***) is the set of sequences $S = S_1 \times S_2 \times ... \times Sn$ (taking "×" as the interleave operator), where for $0 < i \leq n$, S_i is in $L(P_i)$. The set of sequences $\{S_1, S_2, ..., Sn\}$ is a ·grouping· of ***S***. The set of ·paths· of ***S*** in ***P*** is the set of all ·paths· $Q = Q_1 \times Q_2 \times ... \times Q_n$, where Q_i is a ·path· of S_i in P_i, for $0 < i \leq n$.

Less formally, when ***M*** is an ***all***-group of P_1, P_2, ... P_n, then ***L***(***M***) is the set of sequences formed by taking one sequence which is accepted by P_1, then one accepted by P_2, and so on, up through P_n, and then interleaving them together. Equivalently, ***L***(***M***) is the set of sequences ***S*** such that the set $\{S_1, S_2, ..., Sn\}$ is a ·grouping· of ***S***, and for $0 < i \leq n$, S_i is in $L(P_i)$.

[Definition:] A **grouping** of a sequence is a set of sub-sequences, some or all of which may be empty, such that each member of the original sequence appears once and only once in one of the sub-sequences and all members of all sub-sequences are in the original sequence.

For example, given the model group ***M***

```
<xs:all>
 <xs:element name="a" minOccurs="0" maxOccurs="5"/>
 <xs:element name="b" minOccurs="1" maxOccurs="1"/>
 <xs:element name="c" minOccurs="0" maxOccurs="5"/>
</xs:all>
```

and an input sequence ***S***

```
<a/><b/><a/>
```

where $n = 3$, then S_1 is (<a/><a/>), S_2 is (<b/>), and the ·path· of ***S*** in ***M*** is the sequence containing first the Particle for the `a` element, then the Particle for the `b` element, then once more the Particle for the `a` element.

The set ***V***(***M***) (the set of sequences ·locally valid· against ***M***) is the set of sequences ***S*** which are in ***L***(***M***) and which have a ·validation-path· in ***M***. In effect, this means that if one of the Particles in ***M*** ·attributes· an element information item to a ·wildcard particle·, and a ·competing· Particle ·attributes· the same item to an ·element particle·, then the ·element particle· is used for validation.

Note: For example, if ***M*** is

```
<xs:all>
 <xs:any/>
 <xs:element name="a"/>
</xs:all>
```

then ***M*** accepts sequences of length two, containing one `a` element and one other element.

The other element can be anything at all, including a second `a` element. After the first `a` the ·element particle· accepts no more elements and so no longer ·competes· with the ·wildcard particle·. So if the sequence (`<a/><a/>`) is checked against ***M***, in the ·validation-path· the first `a` element will be ·attributed to· the ·element particle· and the second to the ·wildcard particle·.

If the intention is not to allow the second `a`, use a wildcard that explicitly disallows it. That is,

```
<xs:all>
 <xs:any notQName="a"/>
 <xs:element name="a"/>
</xs:all>
```

Now the sequence (`<a/><a/>`) is not accepted by the particle.

3.8.4.1.4 Multiple Paths in Groups

It is possible for a given sequence of element information items to have multiple ·paths· in a given model group ***M***; this is the case, for example, when ***M*** is ambiguous, as for example

```
<xs:choice>
 <xs:sequence>
  <xs:element ref="my:a" maxOccurs="unbounded"/>
  <xs:element ref="my:b"/>
 </xs:sequence>
 <xs:sequence>
  <xs:element ref="my:a"/>
  <xs:element ref="my:b" maxOccurs="unbounded"/>
 </xs:sequence>
</xs:choice>
```

which can match the sequence (`<a/><b/>`) in more than one way. It may also be the case with unambiguous model groups, if they do not correspond to a deterministic expression (as it is termed in [XML 1.1]) or a "1-unambiguous" expression, as it is defined by [Brüggemann-Klein / Wood 1998]. For example,

```
<xs:sequence>
 <xs:element name="a" minOccurs="0"/>
 <xs:element name="a"/>
</xs:sequence>
```

Note: Because these model groups do not obey the constraint Unique Particle Attribution (§3.8.6.4), they cannot appear in a conforming schema.

3.8.4.2 Principles of Validation against Groups

As noted above, each model group ***M*** denotes a language ***L***(***M***), whose members are sequences of element information items. Each member of ***L***(***M***) has one or more ·paths·

in ***M***, as do other sequences of element information items.

By imposing conditions on ·paths· in a model group ***M*** it is possible to identify a set of ·validation-paths· in ***M***, such that if ***M*** is a model group which obeys the Unique Particle Attribution (§3.8.6.4) constraint, then any sequence ***S*** has at most one ·validation-path· in ***M***. The language ***V***(***M***) can then be defined as the set of sequences which have ·validation-paths· in ***M***.

[Definition:] Two Particles P_1 and P_2 contained in some Particle ***P*** **compete** with each other if and only if some sequence ***S*** of element information items has two ·paths· in ***P*** which are identical except that one path has P_1 as its last item and the other has P_2.

For example, in the content model

```
<xs:sequence>
 <xs:element name="a"/>
 <xs:choice>
  <xs:element name="b"/>
  <xs:any/>
 </xs:choice>
</xs:sequence>
```

the sequence (<a/><b/>) has two paths, one (Q_1) consisting of the Particle whose {term} is the declaration for a followed by the Particle whose {term} is the declaration for b, and a second (Q_2) consisting of the Particle whose {term} is the declaration for a followed by the Particle whose {term} is the wildcard. The sequences Q_1 and Q_2 are identical except for their last items, and so the two Particles which are the last items of Q_1 and Q_2 are said to ·compete· with each other.

By contrast, in the content model

```
<xs:choice>
 <xs:sequence>
  <xs:element name="a"/>
  <xs:element name="b"/>
 </xs:sequence>
 <xs:sequence>
  <xs:element name="c"/>
  <xs:any/>
 </xs:sequence>
</xs:choice>
```

the Particles for b and the wildcard do not ·compete·, because there is no pair of ·paths· in ***P*** which differ only in one having the ·element particle· for b and the other having the ·wildcard particle·.

[Definition:] Two (or more) ·paths· of a sequence ***S*** in a Particle ***P*** are **competing paths** if and only if they are identical except for their final items, which differ.

[Definition:] For any sequence ***S*** of element information items and any particle ***P***, a ·path· of ***S*** in ***P*** is a **validation-path** if and only if for each prefix of the ·path· which ends with a ·wildcard particle·, the corresponding prefix of ***S*** has no ·competing path· which ends with an ·element particle·.

> **Note:** It is a consequence of the definition of ·validation-path· that for any content model ***M*** which obeys constraint Unique Particle Attribution (§3.8.6.4) and for any sequence ***S*** of element information items, ***S*** has at most one ·validation-path· in ***M***.

[Definition:] A sequence ***S*** of element information items is **locally valid** against a particle ***P*** if and only if ***S*** has a ·validation-path· in ***P***. The set of all such sequences is written ***V***(***P***).

3.8.4.3 Element Sequence Valid

Validation Rule: Element Sequence Valid

For a sequence ***S*** (possibly empty) of element information items to be locally ·valid· with respect to a model group ***M***, ***S*** MUST be in ***V***(***M***).

> **Note:** It is possible to define groups whose {particles} is empty. When a ***choice***-group ***M*** has an empty {particles} property, then ***L***(***M***) is the empty set. When ***M*** is a ***sequence***- or ***all***-group with an empty {particles} property, then ***L***(***M***) is the set containing the empty (zero-length) sequence.

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.

3.8.6.1 Model Group Correct

Schema Component Constraint: Model Group Correct

All of the following MUST be true:

1 The values of the properties of a model group are 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 There are no circular groups. That is, within the {particles} of a group there is no particle at any depth whose {term} is the group itself.

3.8.6.2 All Group Limited

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 more 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} = 1 which is the {particle} of the {content type} of a complex type definition.

1.3 the {term} property of a Particle ***P*** with {min occurs} = {max occurs} = 1, where ***P*** is among the {particles} of a Model Group whose {compositor} is ***all***.

2 For every particle ***P*** in its {particles}, if ***P***.{term} is a model group, then ***P***.{term}.{compositor} = ***all***.

3.8.6.3 Element Declarations Consistent

Schema Component Constraint: Element Declarations Consistent

If the {particles} property contains, either directly, indirectly (that is, within the {particles} property of a contained model group, recursively), or ·implicitly·, two or more element declarations with the same expanded name, then all their type definitions MUST be the same top-level definition, that is, **all** of the following MUST be true:

1 All their declared {type definition}s have a ·non-absent· {name}.

2 All their declared {type definition}s have the same {name}.

3 All their declared {type definition}s have the same {target namespace}.
4 All their {type table}s are either all ·absent· or else all are present and ·equivalent·.

If **all** of the following are true:
1 The {particles} property contains (either directly, indirectly, or ·implicitly·) one or more element declarations with the same expanded name ***Q***; call these element declarations ***EDS***.
2 At least **one** of the following is true
  2.1 The {particles} property contains one or more ***strict*** or ***lax*** ·wildcard particles· which ·match· ***Q***.
  2.2 The Model Group is the {term} of the ·content model· of some Complex Type Definition ***CTD*** and ***CTD***.{content type} has an {open content} with a ***strict*** or ***lax*** Wildcard which ·matches· ***Q***.
3 There exists a top-level element declaration ***G*** with the expanded name ***Q***.
then the {type table}s of ***EDS*** and the {type table} of ***G*** MUST either all be ·absent· or else all be present and ·equivalent·.

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

[Definition:] A Type Table ***T1*** is **equivalent** to a Type Table ***T2*** if and only if **all** of the following are true:
1 ***T1***.{alternatives} has the same length as ***T2***.{alternatives} and their corresponding entries are ·equivalent·.
2 ***T1***.{default type definition} and ***T2***.{default type definition} are ·equivalent·.

[Definition:] Any Type Alternative is **equivalent** to itself. Otherwise, any Type Alternative ***T1*** is **equivalent** to a different Type Alternative ***T2*** if and only if ***T1***.{test} and ***T2***.{test} are true for the same set of input element information items and ***T1***.{type definition} and ***T2***.{type definition} accept the same set of input information items as valid. In the general case, equivalence and non-equivalence can be difficult to establish. It is ·implementation-defined· under just what conditions a processor detects that two type alternatives are equivalent, but all processors MUST detect ***T1*** and ***T2*** as equivalent if **all** of the following are true:
1 ***T1***.{test}.{namespace bindings} and ***T2***.{test}.{namespace bindings} have the same number of Namespace Bindings, and for each entry in ***T1***.{test}.{namespace bindings} there is a corresponding entry in ***T2***.{test}.{namespace bindings} with the same {prefix} and {namespace}.
2 ***T1***.{test}.{default namespace} and ***T2***.{test}.{default namespace} either are both ·absent· or have the same value.
3 ***T1***.{test}.{base URI} and ***T2***.{test}.{base URI} either are both ·absent· or have the same value.
4 ***T1***.{test}.{expression} and ***T2***.{test}.{expression} have the same value.
5 ***T1***.{type definition} and ***T2***.{type definition} are the same type definition.
A processor MAY treat two type alternatives as non-equivalent if they do not satisfy the conditions just given and the processor does not detect that they are nonetheless equivalent.

> **Note:** In the general case, equivalence can be difficult to prove, so the minimum required of implementations is kept relatively simple. Schema authors can avoid interoperability issues by ensuring that any type alternatives for which equivalence must be established do satisfy the tests above. Implementations MAY recognize cases when differences of namespace bindings, base URIs, and white space in the XPath expression do not affect the meaning of the expression.

3.8.6.4 Unique Particle Attribution

[Definition:] An **element particle** is a Particle whose {term} is an Element Declaration.
[Definition:] A **wildcard particle** is a Particle whose {term} is a Wildcard. Wildcard particles may be referred to as "strict", "lax", or "skip" particles, depending on the {process contents} property of their {term}.

Schema Component Constraint: Unique Particle Attribution

A content model MUST NOT contain two ·element particles· which ·compete· with each other, nor two ·wildcard particles· which ·compete· with each other.

Note: Content models in which an ·element particle· and a ·wildcard particle· ·compete· with each other are *not* prohibited. In such cases, the Element Declaration is chosen; see the definitions of ·attribution· and ·validation-path·.

Note: This constraint reconstructs for XSD the equivalent constraints of [XML 1.1] and SGML. See Analysis of the Unique Particle Attribution Constraint (non-normative) (§J) 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: It is a consequence of Unique Particle Attribution (§3.8.6.4), together with the definition of ·validation-path·, that any sequence ***S*** of element information items has at most one ·validation-path· in any particle ***P***. This means in turn that each item in ***S*** is attributed to at most one particle in ***P***. No item can match more than one Wildcard or more than one Element Declaration (because no two ·wildcard particles· and no two ·element particles· MAY ·compete·), and if an item matches both a ·wildcard particle· and an ·element particle·, it is ·attributed· by the rules for ·validation-paths· to the ·element particle·.

Note: Because locally-scoped element declarations sometimes have and sometimes do not have a {target namespace}, the scope of declarations is *not* relevant to enforcing either the Unique Particle Attribution (§3.8.6.4) constraint or the Element Declarations Consistent (§3.8.6.3) constraint.

*3.8.6.5 Effective Total Range (**all** and **sequence**)*

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 ***P*** whose {term} is a group ***G*** whose {compositor} is ***all*** or ***sequence*** is a pair of minimum and maximum, as follows:

minimum

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

maximum

unbounded if the {max occurs} of any wildcard or element declaration particle in ***G***.{particles} or the maximum part of the effective total range of any of the group

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

*3.8.6.6 Effective Total Range (**choice**)*

Schema Component Constraint: Effective Total Range (*choice*)

The effective total range of a particle ***P*** whose {term} is a group ***G*** whose {compositor} is ***choice*** is a pair of minimum and maximum, as follows:

minimum

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

maximum

unbounded if the {max occurs} of any wildcard or element declaration particle in ***G***.{particles} or the maximum part of the effective total range of any of the group particles in ***G***.{particles} is ***unbounded***, or if any of those is non-zero and ***P***.{max occurs} = ***unbounded***, otherwise the product of ***P***.{max occurs} and the maximum of the {max occurs} of every wildcard or element declaration particle in ***G***.{particles} and the maximum part of the effective total range of each of the group particles in ***G***.{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.

When an element is validated against a complex type, its sequence of child elements is checked against the content model of the complex type and the children are ·attributed to· to Particles of the content model. The attribution of items to Particles determines the calculation of the items' ·context-determined declarations· and thus partially determines the ·governing element declarations· for the children: when an element information item is ·attributed to· an ·element particle·, that Particle's Element Declaration, or an Element Declaration ·substitutable· for it, becomes the item's ·context-determined declaration· and thus normally its ·governing element declaration·; when the item is ·attributed to· a ·wildcard particle·, the ·governing element declaration· depends on the {process contents} property of the wildcard and on QName resolution (Instance) (§3.17.6.3).

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, a kind of Component

{min occurs}
An xs:nonNegativeInteger value. Required.
{max occurs}
Either a positive integer or ***unbounded***. Required.
{term}
A Term component. Required.
{annotations}
A sequence of Annotation components.

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}.

[Definition:] A particle **directly contains** the component which is the value of its {term} property. [Definition:] A particle **indirectly contains** the particles, groups, wildcards, and element declarations which are contained by the value of its {term} property. [Definition:] A particle **contains** the components which it either ·directly contains· or ·indirectly contains·.

See Annotations (§3.15) for information on the role of the {annotations} property.

3.9.2 XML Representation of Particle Schema Components

Particles in the schema typically correspond to element information items that can bear `minOccurs` and `maxOccurs` attributes in the schema document:

- Local <element>, see XML Representation of Element Declaration Schema Components (§3.3.2)
- Model groups <all>, <sequence>, and <choice>, see XML Representation of Model Group Schema Components (§3.8.2)
- Group references <group>, see XML Representation of Model Group Definition Schema Components (§3.7.2)
- Wildcard <any>, see XML Representation of Wildcard Schema Components (§3.10.2)

Sometimes particles do not correspond to any of these elements. For example, particles may be synthesized in complex type extension.

3.9.3 Constraints on XML Representations of Particles

None as such.

3.9.4 Particle Validation Rules

3.9.4.1 Principles of Validation against Particles

Every particle ***P*** ·recognizes· some language ***L***(***P***). When {min occurs} and {max occurs} of ***P*** are both 1, ***L***(***P***) is the language of ***P***'s {term}, as described in Validation of Basic Terms (§3.9.4.1.2). The following section (Language Recognition for Repetitions (§3.9.4.1.1)) describes how more complicated counts are handled.

3.9.4.1.1 Language Recognition for Repetitions

When ***P***.{min occurs} = ***P***.{max occurs} = ***n***, and ***P***.{term} = ***T***, then ***L***(***P***) is the set of sequences ***S*** = S_1 + S_2 + ... + ***Sn*** such that S_i is in ***L***(***T***) for 0 < ***i*** ≤ ***n***. Less formally: ***L***(***P***) is the set of sequences which have ·partitions· into ***n*** sub-sequences for which each of the ***n*** subsequences is in the language accepted by the {term} of ***P***.

When ***P***.{min occurs} = ***j*** and ***P***.{max occurs} = ***k***, and ***P***.{term} = ***T***, then ***L***(***P***) is the set of sequences ***S*** = S_1, + S_2 + ... + ***Sn***, i.e. the set of sequences which have ·partitions· into ***n*** sub-sequences such that ***n*** ≥ ***j*** and ***n*** ≤ ***k*** (or ***k*** is ***unbounded***) and S_i is in ***L***(***T***) for 0 < ***i*** ≤ ***n***.

When ***P***.{min occurs} = 0, then ***L***(***P***) also includes the empty sequence.

If (1) Particle ***P*** has {min occurs} = ***j***, {max occurs} = ***k***, and {term} = ***T***, and (2) ***S*** is a sequence of element information items such that ***S*** = S_1 + S_2 + ... + ***Sn*** (i.e. S_1, S_2, ..., ***Sn*** is a ·partition· of ***S***), and (3) ***n*** ≤ ***k*** (or ***k*** is ***unbounded***), and (4) S_i is in ***L***(***T***) for 0 < ***i*** < ***n***, then:

- If ***T*** is a model group, then the set of ·paths· of ***S*** in ***P*** is the set of all ·paths· ***Q*** such that ***Q*** = Q_1 + Q_2 + ... + Q_n, where Q_i is a ·path· of S_i in ***T*** for 0 < ***i*** ≤ ***n***. (For the definition of ·paths· in model groups, see Language Recognition by Groups (§3.8.4.1).)
- If ***T*** is a ·basic term·, then the (sole) ·path· of ***S*** in ***P*** is a sequence of ***n*** occurrences of ***P***.

Note: Informally: the path of an input sequence ***S*** in a particle ***P*** may go through the ·basic particles· in ***P*** as many times as is allowed by ***P***.{max occurs}. If the path goes through ***P*** more than once, each time before the last one must correspond to a sequence accepted by ***P***.{term}; because the last iteration in the path may not be complete, it need not be accepted by the {term}.

3.9.4.1.2 Validation of Basic Terms

In the preceding section (Language Recognition for Repetitions (§3.9.4.1.1)), the language ***L***(***P***) ·accepted· by a Particle ***P*** is defined in terms of the language ·accepted· by ***P***'s {term}. This section defines ***L***(***T***) for ·basic terms·; for the definition of ***L***(***T***) when ***T*** is a group, see Language Recognition by Groups (§3.8.4.1).

[Definition:] For any Element Declaration ***D***, the language ***L*(*D*)** ·accepted· by ***D*** is the set of all sequences of length 1 whose sole member is an element information item which ·matches· ***D***.

[Definition:] An element information item ***E*** **matches** an Element Declaration ***D*** if and only if **one** of the following is true:
1 ***E*** and ***D*** have the same expanded name,
2 The expanded name of ***E*** ·resolves· to an element declaration D_2 which is ·substitutable· for ***D***.

[Definition:] An expanded name ***E*** **matches** an ·NCName· ***N*** and a namespace name ***NS*** (or, equivalently, ***N*** and ***NS*** **match** ***E***) if and only if all of the following are true:

- The local name of ***E*** is identical to ***N***.
- Either the namespace name of ***E*** is identical to ***NS***, or else ***E*** has no namespace name (***E*** is an unqualified name) and ***NS*** is ·absent·.

Note: For convenience, expanded names are sometimes spoken of as ·matching· a Type Definition, an Element Declaration, an Attribute Declaration, or other schema component which has both a {name} and a {target namespace} property (or vice versa, the component is spoken of as ·matching· the expanded name), when what is meant is, strictly speaking, that the expanded name ·matches· the {name} and {target namespace} properties of the component.

[Definition:] For any Wildcard ***W***, the language ***L*(*W*)** ·accepted· by ***W*** is the set of all sequences of length 1 whose sole member is an element information item which ·matches· ***W***.

[Definition:] An element information item ***E*** **matches** a Wildcard ***W*** (or a ·wildcard particle· whose {term} is ***W***) if and only if ***E*** is locally ·valid· with respect to ***W***, as defined in the validation rule Item Valid (Wildcard) (§3.10.4.1).

[Definition:] Two namespace names N_1 and N_2 are said to **match** if and only if they are identical or both are ·absent·.

For principles of validation when the {term} is a model group instead of a ·basic particle·, see Language Recognition by Groups (§3.8.4.1) and Principles of Validation against Groups (§3.8.4.2).

3.9.4.2 Element Sequence Locally Valid (Particle)

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 **all** of the following MUST be true:
1 The sequence must be accepted by the Particle, as defined in Element Sequence Accepted (Particle) (§3.9.4.3).

3.9.4.3 Element Sequence Accepted (Particle)

Validation Rule: Element Sequence Accepted (Particle)

For a sequence (possibly empty) of element information items to be accepted by a Particle ***P***,

the appropriate **case** among the following MUST be true:
1 **If** ***P***.{term} is a wildcard, **then all** of the following are true:
  1.1 The length of the sequence is greater than or equal to ***P***.{min occurs}.
  1.2 If ***P***.{max occurs} is a number, then the length of the sequence is less than or equal to the {max occurs}.
  1.3 Each element information item in the sequence is ·valid· with respect to the wildcard as defined by Item Valid (Wildcard) (§3.10.4.1).
  In this case, each element information item in the sequence is ·attributed to· ***P*** and has no ·context-determined declaration·.
2 **If** ***P***.{term} is an element declaration ***D***, **then all** of the following are true:
  2.1 The length of the sequence is greater than or equal to ***P***.{min occurs}.
  2.2 If ***P***.{max occurs} is a number, then the length of the sequence is less than or equal to the {max occurs}.
  2.3 For each element information item ***E*** in the sequence **one or more** of the following is true:
    2.3.1 ***D*** has the same expanded name as ***E***.

    In this case ***D*** is the ·context-determined declaration· for ***E*** with respect to Schema-Validity Assessment (Element) (§3.3.4.6) and Assessment Outcome (Element) (§3.3.5.1).

    2.3.2 ***D*** is top-level (i.e. ***D***.{scope}.{variety} = ***global***), its {disallowed substitutions} does not contain ***substitution***, ***E***'s expanded name ·resolves· to an element declaration ***S*** — [Definition:] call this declaration the **substituting declaration** — and ·***S***· is ·substitutable· for ***D*** as defined in Substitution Group OK (Transitive) (§3.3.6.3).

    In this case ·***S***· is the ·context-determined declaration· for ***E*** with respect to Schema-Validity Assessment (Element) (§3.3.4.6) and Assessment Outcome (Element) (§3.3.5.1).

  In this case ***E*** is ·attributed to· ***P***.

    Note: This clause is equivalent to requiring that the sequence of length 1 containing ***E*** is in ·***L***(***D***)·.

3 **If** ***P***.{term} is a model group, **then all** of the following are true:
  3.1 There is a ·partition· of the sequence into ***n*** sub-sequences such that ***n*** is greater than or equal to ***P***.{min occurs}.
  3.2 If ***P***.{max occurs} is a number, ***n*** is less than or equal to ***P***.{max occurs}.
  3.3 Each sub-sequence in the ·partition· is ·valid· with respect to that model group as defined in Element Sequence Valid (§3.8.4.3).
  In this case, the element information items in each sub-sequence are ·attributed to· Particles within the model group which is the {term}, as described in Language

Recognition by Groups (§3.8.4.1).

Note: The rule just given does not require that the content model be deterministic. In practice, however, most non-determinism in content models is ruled out by the schema component constraint Unique Particle Attribution (§3.8.6.4). Non-determinism can occur despite that constraint for several reasons. In some such cases, some particular element information item may be accepted by either a Wildcard or an Element Declaration. In such situations, the validation process defined in this specification matches the element information item against the Element Declaration, both in identifying the Element Declaration as the item's ·context-determined declaration·, and in choosing alternative paths through a content model. Other cases of non-determinism involve nested particles each of which has {max occurs} greater than 1, where the input sequence can be partitioned in multiple ways. In those cases, there is no fixed rule for eliminating the non-determinism.

Note: clause 1 and clause 2.3.2 do not interact: an element information item validatable by a declaration with a substitution group head is *not* validatable by a wildcard which accepts the head's (namespace, name) pair but not its own.

3.9.5 Particle Information Set Contributions

None as such.

3.9.6 Constraints on Particle Schema Components

3.9.6.1 Particle Correct

All particles (see Particles (§3.9)) MUST satisfy the following constraint.

Schema Component Constraint: Particle Correct

All of the following MUST be true:

1 The values of the properties of a particle are 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 are true:

2.1 {min occurs} is not greater than {max occurs}.

3.9.6.2 Particle Valid (Extension)

The following constraint defines a relation 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 or more** of the following is true:

1 They are the same particle.

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

3 **All** of the following are true:

3.1 **E**.{min occurs} = **B**.{min occurs}.

3.2 Both **E** and **B** have ***all*** groups as their {term}s.

3.3 The {particles} of **B**'s ***all*** group is a prefix of the {particles} of **E**'s ***all*** group.

3.9.6.3 Particle Emptiable

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

Schema Component Constraint: Particle Emptiable

[Definition:] For a particle to be **emptiable** **one or more** of the following is 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***) (§3.8.6.5) (if the group is ***all*** or ***sequence***) or Effective Total Range (***choice***) (§3.8.6.6) (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 names and optionally on their local names.

Example

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

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

<xs:any namespace="http://www.w3.org/1999/XSL/Transform"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        notQName="xsl:comment xsl:fallback"/>

<xs:any notNamespace="##targetNamespace ##local"/>

<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, a kind of Term

{annotations}
A sequence of Annotation components.
{namespace constraint}
A Namespace Constraint property record. Required.
{process contents}
One of {***skip***, ***strict***, ***lax***}. Required.

Property Record: **Namespace Constraint**

{variety}
One of {***any***, ***enumeration***, ***not***}. Required.
{namespaces}
A set each of whose members is either an xs:anyURI value or the distinguished value ·absent·. Required.
{disallowed names}
A set each of whose members is either an xs:QName value or the keyword ***defined*** or the keyword ***sibling***. Required.

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

1. ({variety} ***any***) have any namespace or are not namespace-qualified;
2. ({variety} ***not*** and {namespaces} a set whose members are either namespace names or ·absent·) have any namespace other than the specified namespaces and/or, if ·absent· is included in the set, are namespace-qualified; (see this example, which accepts only namespace-qualified names distinct from the target namespace; the "##local" in the schema document maps to the value ·absent· in the {namespace constraint} property)
3. ({variety} ***enumeration*** and {namespaces} 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.
4. ({disallowed names} contains QName members) have any expanded name other than the specified names.
5. ({disallowed names} contains the keyword ***defined***) have any expanded name other than those matching the names of global element or attribute declarations.
6. ({disallowed names} contains the keyword ***sibling***) have any expanded name other than those matching the names of element or attribute declarations in the containing complex type definition.

{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 declaration, that is, ·validate· if you can, don't worry if you can't.

See Annotations (§3.15) for information on the role of the {annotations} 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.

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)) )
  notNamespace = List of (anyURI | (##targetNamespace | ##local))
  notQName = List of (QName | (##defined | ##definedSibling))
  processContents = (lax | skip | strict) : strict
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</any>
```

XML Representation Summary: `anyAttribute` Element Information Item

```
<anyAttribute
  id = ID
  namespace = ((##any | ##other) | List of (anyURI | (##targetNamespace | ##local)) )
  notNamespace = List of (anyURI | (##targetNamespace | ##local))
  notQName = List of (QName | ##defined)
  processContents = (lax | skip | strict) : strict
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</anyAttribute>
```

An <any> information item corresponds both to a wildcard component and to a particle containing that wildcard (unless `minOccurs=maxOccurs=0`, in which case the item corresponds to no component at all). The mapping rules are given in the following two subsections. As always, the mapping rules given here apply after, not before, the appropriate ·pre-processing·.

Processors MAY issue a warning if a list of `xs:anyURI` values in the `namespace` or `noNamespace` attributes includes any values beginning with the string '##'.

> **Note:** Strings beginning with '##' are not IRIs or URIS according to the definitions given in the relevant RFCs (for references see [XML Schema: Datatypes]) and users of this specification are discouraged from using them as namespace names, to avoid confusion with the keyword values defined by this specification.

3.10.2.1 Mapping from <any> to a Particle

The mapping from an <any> information item to a particle is as follows.

XML Mapping Summary for Particle Schema Component

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

{annotations}	The same annotations as the {annotations} of the wildcard. See below.

3.10.2.2 Mapping from <any> and <anyAttribute> to a Wildcard Component

The mapping from an <any> information item to a wildcard component is as follows. This mapping is also used for mapping <anyAttribute> information items to wildcards, although in some cases the result of the mapping is further modified, as specified in the rules for <attributeGroup> and <complexType>.

XML Mapping Summary for Wildcard Schema Component

Property	Representation
{namespace constraint}	A Namespace Constraint with the following properties:

Property	Value
{variety}	the appropriate **case** among the following: 1 **If** the `namespace` [attribute] is present, **then** the appropriate **case** among the following: 1.1 **If** `namespace` = `"##any"`, **then** ***any***; 1.2 **If** `namespace` = `"##other"`, **then** ***not***; 1.3 **otherwise** ***enumeration***; 2 **If** the `notNamespace` [attribute] is present, **then** ***not***; 3 **otherwise** (neither `namespace` nor `notNamespace` is present) ***any***.
{namespaces}	the appropriate **case** among the following: 1 **If** neither `namespace` nor `notNamespace` is present, **then** the empty set; 2 **If** `namespace` = `"##any"`, **then** the empty set; 3 **If** `namespace` = `"##other"`, **then** a set consisting of ·absent· and, if the `targetNamespace` [attribute] of the <schema> ancestor element information item is present, its ·actual value·; 4 **otherwise** a set whose members are namespace names corresponding to the space-delimited substrings of the ·actual value· of the `namespace` or `notNamespace` [attribute] (whichever is present), except 4.1 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·; 4.2 if one such substring is `##local`, the corresponding member is ·absent·.

{disallowed names}	If the `notQName` [attribute] is present, then a set whose members correspond to the items in the ·actual value· of the `notQName` [attribute], as follows. • If the item is a QName value (i.e. an expanded name), then that QName value is a member of the set. • If the item is the token "`##defined`", then the keyword ***defined*** is a member of the set. • If the item is the token "`##definedSibling`", then the keyword ***sibling*** is a member of the set. If the `notQName` [attribute] is not present, then the empty set.
{process contents}	The ·actual value· of the `processContents` [attribute], if present, otherwise ***strict***.
{annotations}	The ·annotation mapping· of the <any> element, as defined in XML Representation of Annotation Schema Components (§3.15.2). **Note:** When this rule is used for an attribute wildcard (see XML Representation of Complex Type Definition Schema Components (§3.4.2)), the {annotations} is the ·annotation mapping· of the <anyAttribute> element.

Wildcards are subject to the same ambiguity constraints (Unique Particle Attribution (§3.8.6.4)) as other content model particles: If an instance element could match 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> and <anyAttribute> element information items by the schema for schema documents, `namespace` and `notNamespace` attributes MUST NOT both be present.

3.10.4 Wildcard Validation Rules

3.10.4.1 Item Valid (Wildcard)

Validation Rule: Item Valid (Wildcard)

For an element or attribute information item ***I*** to be locally ·valid· with respect to a wildcard ***W*** **all** of the following MUST be true:
1 The expanded name of ***I*** is ·valid· with respect to ***W***.{namespace constraint}, as defined in Wildcard allows Expanded Name (§3.10.4.2).
2 **If** ***W***.{namespace constraint}.{disallowed names} contains the keyword ***defined***, **then both** of the following are true:

2.1 **If** ***W*** is an element wildcard (i.e., ***W*** appears in a content model), **then** the expanded name of ***I*** does not ·resolve· to an element declaration. (Informally, no such top-level element is declared in the schema.)
2.2 **If** ***W*** is an attribute wildcard, **then** the expanded name of ***I*** does not ·resolve· to an attribute declaration.

3 **If all** of the following are true:
3.1 ***W***.{namespace constraint}.{disallowed names} contains the keyword ***sibling***;
3.2 ***W*** is an element wildcard
3.3 ***I*** is an element information item
3.4 ***I*** has a [parent] ***P*** that is also an element information item
3.5 ***I*** and ***P*** have the same [validation context]
3.6 ***P*** has an ·governing type definition· ***T*** (which is always a complex type and contains ***W*** in its ·content model·)
then the expanded name of ***I*** does not ·match· any element declaration ·contained· in the content model of ***T***, whether ·directly·, ·indirectly·, or ·implicitly·.

Informally, the keyword ***sibling*** disallows any element declared as a possible sibling of the wildcard ***W***.

When an element or attribute information item is ·attributed· to a wildcard and the preceding constraint (Item Valid (Wildcard) (§3.10.4.1)) is satisfied, then the item has no ·context-determined declaration·. Its ·governing· declaration, if any, is found by matching its expanded name as described in QName resolution (Instance) (§3.17.6.3). Note that QName resolution is performed only if the item is ·attributed· to a ***strict*** or ***lax*** wildcard; if the wildcard has a {process contents} property of ***skip***, then the item has no ·governing· declaration.

[Definition:] An element or attribute information item is **skipped** if it is ·attributed· to a ***skip*** wildcard or if one of its ancestor elements is.

3.10.4.2 Wildcard allows Expanded Name

Validation Rule: Wildcard allows Expanded Name

For an expanded name ***E***, i.e. a (namespace name, local name) pair, to be ·valid· with respect to a namespace constraint ***C*** **all** of the following MUST be true:
1 The namespace name is ·valid· with respect to ***C***, as defined in Wildcard allows Namespace Name (§3.10.4.3);
2 ***C***.{disallowed names} does not contain ***E***.

3.10.4.3 Wildcard allows Namespace Name

Validation Rule: Wildcard allows Namespace Name

For a value ***V*** which is either a namespace name or ·absent· to be ·valid· with respect to a namespace constraint ***C*** (the value of a {namespace constraint}) **one** of the following MUST be true:
1 ***C***.{variety} = ***any***.
2 ***C***.{variety} = ***not***, and ***V*** is not identical to any of the members of ***C***.{namespaces}.
3 ***C***.{variety} = ***enumeration***, and ***V*** is identical to one of the members of ***C***.{namespaces}.

3.10.5 Wildcard Information Set Contributions

None as such.

3.10.6 Constraints on Wildcard Schema Components

3.10.6.1 Wildcard Properties Correct

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

Schema Component Constraint: Wildcard Properties Correct

All of the following MUST be true:

1 The values of the properties of a wildcard are as described in the property tableau in The Wildcard Schema Component (§3.10.1), modulo the impact of Missing Sub-components (§5.3).
2 If {variety} is ***not***, {namespaces} has at least one member.
3 If {variety} is ***any***, {namespaces} is empty.
4 The namespace name of each QName member in {disallowed names} is allowed by the {namespace constraint}, as defined in Wildcard allows Namespace Name (§3.10.4.3).
5 Attribute wildcards do not contain ***sibling*** in their {namespace constraint}.{disallowed names}.

3.10.6.2 Wildcard Subset

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

Schema Component Constraint: Wildcard Subset

[Definition:] Given two Namespace Constraints **sub** and **super**, **sub** is a **wildcard subset** of **super** if and only if **one** of the following is true

1 **super**.{variety} = ***any***.
2 Both **sub** and **super** have {variety} = ***enumeration***, and **super**.{namespaces} is a superset of **sub**.{namespaces}.
3 **sub**.{variety} = ***enumeration***, **super**.{variety} = ***not***, and the {namespaces} of the two are disjoint.
4 Both **sub** and **super** have {variety} = ***not***, and **super**.{namespaces} is a subset of **sub**.{namespaces}.

And **all** of the following are true:

1 Each QName member of **super**.{disallowed names} is not allowed by **sub**, as defined in Wildcard allows Expanded Name (§3.10.4.2).
2 If **super**.{disallowed names} contains ***defined***, then **sub**.{disallowed names} also contains ***defined***.
3 If **super**.{disallowed names} contains ***sibling***, then **sub**'s {disallowed names} also contains ***sibling***.

3.10.6.3 Attribute Wildcard Union

Schema Component Constraint: Attribute Wildcard Union

[Definition:] Given three Namespace Constraints ***O***, ***O1***, and ***O2***, ***O*** is the **wildcard union** of ***O1*** and ***O2*** if and only if, first, the {variety} and {namespaces}, and, second, the {disallowed names} of ***O*** are consistent with ***O*** being the union of ***O1*** and ***O2***, as that is defined below.

The {variety} and {namespaces} of ***O*** are consistent with ***O*** being the wildcard union of ***O1*** and ***O2*** if and only if **one or more** of the following is true:

1 ***O***, ***O1***, and ***O2*** all have the same {variety} and {namespaces}.

2 Either ***O1***.{variety} = ***any*** or ***O2***.{variety} = ***any***, and ***O***.{variety} = ***any***.
3 ***O***, ***O1***, and ***O2*** all have {variety} ***enumeration***, and ***O***.{namespaces} is the union of ***O1***.{namespaces} and ***O2***.{namespaces}.
4 ***O1***.{variety} = ***O2***.{variety} = ***not***, and **one** of the following is true
  4.1 The intersection of the {namespaces} of ***O1*** and ***O2*** is the empty set, and ***O***.{variety} = ***any***.
  4.2 ***O***.{variety} = ***not***, and ***O***.{namespaces} is the non-empty intersection of ***O1***.{namespaces} and ***O2***.{namespaces}.
5 Either ***O1*** or ***O2*** has {variety} = ***not*** and {namespaces} = ***S1***, and the other has {variety} = ***enumeration*** and {namespaces} = ***S2***, and **one** of the following is true
  5.1 The set difference ***S1*** minus ***S2*** is the empty set, and ***O***.{variety} = ***any***.
  5.2 ***O***.{variety} = ***not*** and ***O***.{namespaces} is the non-empty set difference ***S1*** minus ***S2***.

The {disallowed names} property of ***O*** is consistent with ***O*** being the wildcard union of ***O1*** and ***O2*** if and only if ***O***.{disallowed names} includes all and only the following:
1 QName members of ***O1***.{disallowed names} that are not allowed by ***O2***, as defined in Wildcard allows Expanded Name (§3.10.4.2).
2 QName members of ***O2***.{disallowed names} that are not allowed by ***O1***.
3 The keyword ***defined*** if it is contained in both ***O1***.{disallowed names} and ***O2***.{disallowed names}.

> **Note:** When one of the wildcards has ***defined*** in {disallowed names} and the other does not, then ***defined*** is *not* included in the union. This may allow QNames that are not allowed by either wildcard. This is to ensure that all unions are expressible. If ***defined*** is intended to be included, then it is necessary to have it in both wildcards.

In the case where there are more than two Namespace Constraints to be combined, the wildcard union is determined by identifying the wildcard union of two of them as above, then the wildcard union of the result with the third, and so on as required.

3.10.6.4 Attribute Wildcard Intersection

Schema Component Constraint: Attribute Wildcard Intersection

[Definition:] Given three Namespace Constraints ***O***, ***O1***, and ***O2***, ***O*** is the **wildcard intersection** of ***O1*** and ***O2*** if and only if both its {variety} and {namespaces} properties, on the one hand, and its {disallowed names} property, on the other, are consistent with ***O*** being the intersection of ***O1*** and ***O2***, as that is defined below.

The {variety} and {namespaces} of ***O*** are consistent with ***O*** being the wildcard intersection of ***O1*** and ***O2*** if and only if **one or more** of the following is true:
1 ***O***, ***O1***, and ***O2*** have the same {variety} and {namespaces}.
2 Either ***O1*** or ***O2*** has {variety} = ***any*** and ***O*** has {variety} and {namespaces} identical to those of the other.
3 ***O***, ***O1***, and ***O2*** all have {variety} = ***enumeration***, and ***O***.{namespaces} is the intersection of the {namespaces} of ***O1*** and ***O2***.
4 ***O***, ***O1***, and ***O2*** all have {variety} = ***not***, and ***O***.{namespaces} is the union of the {namespaces} of ***O1*** and ***O2***.
5 Either ***O1*** or ***O2*** has {variety} = ***not*** and {namespaces} = ***S1*** and the other has {variety} = ***enumeration*** and {namespaces} = ***S2***, and ***O***.{variety} = ***enumeration*** and ***O***.{namespaces} = the set difference ***S2*** minus ***S1***.

The {disallowed names} property of ***O*** is consistent with ***O*** being the wildcard intersection of ***O1*** and ***O2*** if and only if ***O***.{disallowed names} includes all and only the following:

1 QName members of ***O1***.{disallowed names} whose namespace names are allowed by ***O2***, as defined in Wildcard allows Namespace Name (§3.10.4.3).
2 QName members of ***O2***.{disallowed names} whose namespace names are allowed by ***O1***.
3 The keyword ***defined*** if it is a member of either {disallowed names}.

In the case where there are more than two Namespace Constraints to be combined, the wildcard intersection is determined by identifying the wildcard intersection of two of them as above, then the wildcard intersection of the result with the third, 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, a kind of Annotated Component

{annotations}
A sequence of Annotation components.
{name}
An xs:NCName value. Required.
{target namespace}

An xs:anyURI value. Optional.

{identity-constraint category}
One of {***key***, ***keyref***, ***unique***}. Required.

{selector}
An XPath Expression property record. Required.

{fields}
A sequence of XPath Expression property records.

{referenced key}
An Identity-Constraint Definition component. Required if {identity-constraint category} is ***keyref***, otherwise ({identity-constraint category} is ***key*** or ***unique***) MUST be ·absent·.
If a value is present, its {identity-constraint category} must be ***key*** or ***unique***.

Identity-constraint definitions are identified by their {name} and {target namespace}; identity-constraint definition identities MUST be unique within an ·XSD schema·. See References to schema components across namespaces (<import>) (§4.2.6) 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 alongside the specification of types for the attributes and elements involved, i.e. something declared as of type integer can 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 *values* of the fields selected, not their lexical representation, so that for example `3.0` and `3` would be conflicting keys if they were both decimal, but non-conflicting if they were both strings, or one was a string and one a decimal. When equality and identity differ for the simple types involved, all three forms of identity-constraint test for *either* equality *or* identity of values.

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 performed on typed values, not on the lexical representations of those values.

{selector} specifies a restricted XPath ([XPath 2.0]) expression relative to instances of the element being declared. This MUST identify a sequence of element nodes that are contained within the declared element to which the constraint applies.

{fields} specifies XPath expressions relative to each element selected by a {selector}. Each XPath expression in the {fields} property 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 `xs1:key`.

> **Note:** In version 1.0 of this specification [XSD 1.0 2E], identity constraints used [XPath 1.0]. They now use [XPath 2.0].

See Annotations (§3.15) for information on the role of the {annotations} 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 after the appropriate ·pre-processing· and the properties of the component they correspond to are as follows:

XML Representation Summary: `unique` Element Information Item et al.

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

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

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

```
<selector
  id = ID
  xpath = a subset of XPath expression, see below
  xpathDefaultNamespace = (anyURI | (##defaultNamespace | ##targetNamespace | ##local))
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</selector>
```

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

If the ref [attribute] is absent, the corresponding schema component is as follows:

XML Mapping Summary for 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 <schema> ancestor element information item if present, otherwise ·absent·.
{identity-constraint category}	One of ***key***, ***keyref*** or ***unique***, depending on the item.
{selector}	An XPath Expression property record, as described in section XML Representation of Assertion Schema Components (§3.13.2), with <selector> as the "host element" and xpath as the designated expression [attribute].
{fields}	A sequence of XPath Expression property records, corresponding to the <field> element information item [children], in order, following the rules given in XML Representation of Assertion Schema Components (§3.13.2), with <field> as the "host element" and xpath as the designated expression [attribute].
{referenced key}	If the item is a <keyref>, the identity-constraint definition ·resolved· to by the ·actual value· of the refer [attribute], otherwise ·absent·.
{annotations}	The ·annotation mapping· of the set of elements containing the <key>, <keyref>, or <unique> element, whichever is present, and the <selector> and <field> [children], if present, as defined in XML Representation of Annotation Schema Components (§3.15.2).

Otherwise (the ref [attribute] is present), the corresponding schema component is the identity-constraint definition ·resolved· to by the ·actual value· of the ref [attribute].

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>

 <!-- vehicles are keyed by their plate within states -->
 <xs:key name="reg">
  <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>

 <!-- states are keyed by their code -->
 <xs:key name="state">
  <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>

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

 <!-- people's cars are a reference -->
 <xs:keyref name="carRef" refer="regKey">
  <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 vehicles via the carRef constraint. The requirement that a vehicle's state match its containing state's code is not expressed here.

Example

```
<xs:element name="stateList">
 <xs:complexType>
  <xs:sequence>
   . . .
   <xs:element name="state" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      . . .
      <xs:element name="code" type="twoLetterCode"/>
      . . .
     </xs:sequence>
    </xs:complexType>
   </xs:element>
   . . .
  </xs:sequence>
 </xs:complexType>

 <xs:key ref="state"/> <!-- reuse the key constraint from the above example -->
</xs:element>
```

A list of state elements can appear as child elements under stateList. A ***key*** constraint can be used to ensure that there is no duplicate state code. We already defined a ***key*** in the above example for the exact same purpose (the key constraint is named "state"). We can reuse it directly via the ref attribute on the key element.

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 schema documents, **all** of the following MUST be true:

1 One of ref or name is present, but not both.
2 If name is present, then <selector> appears in [children].
3 If name is present on <keyref>, then refer is also present.
4 If ref is present, then only id and <annotation> are allowed to appear together with ref.
5 If ref is present, then the {identity-constraint category} of the identity-constraint definition ·resolved· to by the ·actual value· of the ref [attribute] matches the name of the element information item.

3.11.4 Identity-constraint Definition Validation Rules

Validation Rule: Identity-constraint Satisfied

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

1 A data model instance is constructed from the input information set, as described in [XDM]. The {selector}, with the element node corresponding to ***E*** as the context node, evaluates to a sequence of element nodes, as defined in XPath Evaluation (§3.13.4.2). [Definition:] The **target node set** is the set of nodes in that sequence,

after omitting all element nodes corresponding to element information items that are ·skipped·.

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

3 For each node ***N*** in the ·target node set· and each field ***F*** in {fields}, when ***N*** is taken as the context node ***F*** evaluates (as defined in XPath Evaluation (§3.13.4.2)) to a sequence of nodes ***S***. ***S*** contains zero or more ·skipped· nodes, at most one node whose ·governing· type definition is either a simple type definition or a complex type definition with {variety} ***simple***, and no other nodes. As a consequence, for each field ***F*** the sequence ***S*** contains at most one node with a ·non-absent· [schema actual value]. Call that one node's [schema actual value] ***V***(***N***,***F***). [Definition:] The **key-sequence** of ***N*** is the sequence consisting of the values ***V***(***N***,***F***), in the order of the {fields} property.

Note: The use of [schema actual value] in the definition of ·key sequence· above means that ***default*** or ***fixed*** value constraints MAY play a part in ·key sequence·s.

Note: In some cases, the ·key-sequence· of node ***N*** will be shorter than {fields} in length, for example whenever any field evaluates to the empty sequence or to a sequence consisting only of ·skipped· or ·nilled· nodes. In such a case, the ·key-sequence· of ***N*** does not participate in value comparison, because ***N*** is not a member of the ·qualified node set·. See below.

4 [Definition:] The **qualified node set** is the subset of the ·target node set· consisting of those nodes whose ·key-sequence· has the same length as the {fields}. The appropriate **case** among the following is 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 or identical, as defined by Equality and Identity in [XML Schema: Datatypes].

4.2 **If** the {identity-constraint category} is ***key***, **then all** of the following are 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 or identical, as defined by Equality and Identity in [XML Schema: 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 is a ·node table· associated with the {referenced key} in the [identity-constraint table] of ***E*** (see Identity-constraint Table (§3.11.5), which is understood as logically prior to this clause of this constraint, below) and there is an entry in that table whose ·key-sequence· is equal or identical to the **keyref member's** ·key-sequence· member for member, as defined by Equality and Identity in [XML Schema: Datatypes].

Note: For ***unique*** identity constraints, the ·qualified node set· is allowed to be different from the ·target node set·. That is, a selected unique node may have fields that do not have corresponding [schema actual value]s.

Note: Because the validation of ***keyref*** (see clause 4.3) depends on finding appropriate entries in an 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 (§3.3.5.3)), 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.

For purposes of checking identity-constraints, single atomic values are not distinguished from lists with single items. An atomic value ***V*** and a list ***L*** with a single item are treated as equal, for purposes of this specification, if ***V*** is equal to the atomic value which is the single item of ***L***. And similarly for identity.

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 is 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 can be defined on domains distinct from the embedded domain of the identity-constraint they reference, or on domains which are 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.

3.11.6 Constraints on Identity-constraint Definition Schema Components

3.11.6.1 Identity-constraint Definition Properties Correct

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 are 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} is equal to that of the {fields} of the {referenced key}.

3.11.6.2 Selector Value OK

Schema Component Constraint: Selector Value OK

All of the following MUST be true:

1 The {selector} satisfies the constraint XPath Valid (§3.13.6.2).

2 **One or more** of the following is true:

2.1 Its {expression} conforms 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 Its {expression} is 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.

[Definition:] The subset of XPath defined in Selector Value OK (§3.11.6.2) is called the **selector subset** of XPath.

3.11.6.3 Fields Value OK

Schema Component Constraint: Fields Value OK

All of the following MUST be true:

1 Each member of the {fields} satisfies the constraint XPath Valid (§3.13.6.2).

2 For each member of the {fields} **one or more** of the following is true:

2.1 Its {expression} conforms 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 Its {expression} is 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.

[Definition:] The subset of XPath defined in Fields Value OK (§3.11.6.3) is called the **field subset** of XPath.

3.12 Type Alternatives

Type Alternative components provide associations between boolean conditions (as XPath expressions) and Type Definitions. They are used in conditional type assignment.

3.12.1 The Type Alternative Schema Component

The type alternative schema component has the following properties:

Schema Component: Type Alternative, a kind of Annotated Component

{annotations}
A sequence of Annotation components.

{test}
An XPath Expression property record. Optional.

{type definition}
A Type Definition component. Required.

Type alternatives can be used by an Element Declaration to specify a condition ({test}) under which a particular type ({type definition}) is used as the ·governing type definition·

for element information items governed by that Element Declaration. Each Element Declaration MAY have multiple Type Alternatives in its {type table}.

3.12.2 XML Representation of Type Alternative Schema Components

The XML representation for a type alternative schema component is an <alternative> element information item. The correspondences between the properties of that information item after the appropriate ·pre-processing· and the properties of the component it corresponds to are as follows:

XML Representation Summary: `alternative` Element Information Item

```
<alternative
  id = ID
  test = an XPath expression
  type = QName
  xpathDefaultNamespace = (anyURI | (##defaultNamespace | ##targetNamespace | ##local))
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (simpleType | complexType)?)
</alternative>
```

Each <alternative> element maps to a Type Alternative component as follows.

XML Mapping Summary for Type Alternative Schema Component

Property	Representation
{test}	If the `test` [attribute] is not present, then ·absent·; otherwise an XPath Expression property record, as described in section XML Representation of Assertion Schema Components (§3.13.2), with <alternative> as the "host element" and `test` as the designated expression [attribute].
{type definition}	The type definition ·resolved· to by the ·actual value· of the `type` [attribute], if one is present, otherwise the type definition corresponding to the `complexType` or `simpleType` among the [children] of the <alternative> element.
{annotations}	The ·annotation mapping· of the <alternative> element, as defined in XML Representation of Annotation Schema Components (§3.15.2).

3.12.3 Constraints on XML Representations of Type Alternatives

Schema Representation Constraint: Type Alternative Representation OK

In addition to the conditions imposed on <alternative> element information items by the schema for schema documents, each <alternative> element MUST have one (and only one) of the following: a `type` attribute, or a `complexType` child element, or a `simpleType` child element.

3.12.4 Type Alternative Validation Rules

[Definition:] A Type Alternative ***A*** **successfully selects** a Type Definition ***T*** for an element information item ***E*** if and only if ***A***.{test} evaluates to `true` and ***A***.{type definition} = ***T***. The {test} is evaluated in the following way:
1 An instance of the [XDM] data model is constructed as follows:

1.1 An information set is constructed by copying the base information set properties (and not any of the properties specific to ·post-schema-validation infoset·) of the following information items:
1.1.1 ***E*** itself.
1.1.2 ***E***'s [attributes] (but not its [children]).
1.1.3 ***E***'s [inherited attributes] which do not have the same expanded names as any of ***E***'s [attributes]. They are copied as if they were among ***E***'s [attributes] and had ***E*** as their [owner element]. When an attribute with a non-empty [namespace name] is copied, ·namespace fixup· may need to be performed on the resulting information set to ensure that a prefix ***P*** is bound to the [namespace name] and the [prefix] of the copied attribute is set to ***P***.

[Definition:] By **base information set properties** are meant the properties listed in Required Information Set Items and Properties (normative) (§D).

1.2 In the copy of ***E***, the [parent] property is modified to have no value. The [children] property is modified to have the empty list as its value.
1.3 An [XDM] data model instance is constructed from that information set, following the rules given in [XDM].

2 The XPath expression which is the value of the {test}, is evaluated as described in XPath Evaluation (§3.13.4.2). If a dynamic error or a type error is raised during evaluation, then the {test} is treated as if it had evaluated (without error) to `false`.

Note: Dynamic errors and type errors in the evaluation of {test} expressions cause neither the schema nor the document instance to be invalid. But conforming processors MAY issue a warning if they occur. However, if type errors are detected statically by the XPath processor, the effect is the same as for other static errors in the XPath expression.

Note: As a consequence of the rules just given, the root node of the [XDM] instance is necessarily constructed from ***E***; the ancestors, siblings, children, and descendants of ***E*** are not represented in the data model instance, and they are thus not accessible to the tests expressed in the {test}s in the {type table}. The element ***E*** and its [attributes] will be represented in the data model instance by nodes labeled as untyped. If the {test} expressions being evaluated include comparisons which require type information, then explicit casts will sometimes be necessary.

3.12.5 Type Alternative Information Set Contributions

None.

3.12.6 Constraints on Type Alternative Schema Components

All type alternatives (see Type Alternatives (§3.12)) MUST satisfy the following constraints.

Schema Component Constraint: Type Alternative Properties Correct

All of the following MUST be true:
1 The values of the properties of a type alternatives are as described in the property tableau in The Type Alternative Schema Component (§3.12.1), modulo the impact of Missing Sub-components (§5.3).
2 If the {test} is not ·absent·, then it satisfies the constraint XPath Valid (§3.13.6.2). The function signatures in the static context MUST include signatures for **all** of the following:
2.1 The fn:not function defined in the [Functions and Operators] specification.
2.2 Constructor functions for the built-in datatypes.
The further contents of function signatures are ·implementation-defined·.

A conforming processor MUST accept and process any XPath expression conforming to the "required subset" of [XPath 2.0] defined by the following grammar.

Note: Any XPath expression containing no static errors as defined in [XPath 2.0] may appear in a conforming schema. Conforming processors MAY but are not required to support XPath expressions not belonging to the required subset of XPath.

An XPath expression belongs to the required subset of XPath if and only if **all** of the following are true:

1 The {expression} property of the XPath Expression is an XPath expression containing no static errors, as defined in [XPath 2.0].

2 **One or more** of the following is true:

2.1 It conforms to the following extended BNF:

Test XPath expressions

```
[8]   Test                ::=  OrExpr
[9]   OrExpr              ::=  AndExpr ( 'or' AndExpr )*
[10]  AndExpr             ::=  BooleanExpr ( 'and' BooleanExpr )*
[11]  BooleanExpr         ::=  '(' OrExpr ')' |
                               BooleanFunction |
                               ValueExpr ( Comparator ValueExpr )?
[12]  BooleanFunction     ::=  QName '(' OrExpr ')'
[13]  Comparator          ::=  '=' | '!=' | '<' | '<=' | '>' | '>='
[14]  ValueExpr           ::=  CastExpr | ConstructorFunction
[15]  CastExpr            ::=  SimpleValue ( 'cast' 'as' QName '?'? )?
[16]  SimpleValue         ::=  AttrName | Literal
[17]  AttrName            ::=  '@' NameTest
[18]  ConstructorFunction ::=  QName '(' SimpleValue ')'
```

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

Note: For readability, [XPath 2.0] allows whitespace to be used between tokens in XPath expressions, even though this is not explicitly shown in the grammar. For details of whitespace handling, consult [XPath 2.0].

3 Any strings matching the *BooleanFunction* production are function calls to fn:not defined in the [Functions and Operators] specification. Any strings matching the *ConstructorFunction* production are function calls to constructor functions for the built-in datatypes.

Note: The minimal content of the function signatures in the static context is given in clause 2 of Type Alternative Properties Correct (§3.12.6): `fn:not` and constructors for the built-in datatypes.

Note: The above extended BNF is ambiguous. For example, the string "`a:b('123')`" has 2 paths in the grammar, by matching either *BooleanFunction* or *ConstructorFunction*. The rules given here require different function names for the productions. As a result, the ambiguity can be resolved based on the function name.

4 Any explicit casts (i.e. any strings which match the optional "`cast as`" *QName* in the *CastExpr* production) are casts to built-in datatypes.

Note: Implementations MAY choose to support a bigger subset of [XPath 2.0].

Note: The rule given above for the construction of the data model instance has as a consequence that even when implementations support full [XPath 2.0] expressions, it is not possible to refer successfully to the children, siblings, ancestors, etc. of the element whose type is being selected.

3.13 Assertions

Assertion components constrain the existence and values of related elements and attributes.

Example

```
<xs:assert test="@min le @max"/>
```

The XML representation for assertions.

The <assert> element requires that the value of the `min` attribute be less than or equal to that of the `max` attribute, and fails if that is not the case.

3.13.1 The Assertion Schema Component

The assertion schema component has the following properties:

Schema Component: Assertion, a kind of Annotated Component

{annotations}
A sequence of Annotation components.
{test}
An XPath Expression property record. Required.

Property Record: XPath Expression

{namespace bindings}
A set of Namespace Binding property records.
{default namespace}
An xs:anyURI value. Optional.
{base URI}
An xs:anyURI value. Optional.
{expression}
An [XPath 2.0] expression. Required.

Property Record: Namespace Binding

{prefix}
An xs:NCName value. Required.
{namespace}

An xs:anyURI value. Required.

To check an assertion, an instance of the XPath 2.0 data model ([XDM]) is constructed, in which the element information item being ·assessed· is the (parentless) root node, and elements and attributes are assigned types and values according to XPath 2.0 data model construction rules, with some exceptions. See Assertion Satisfied (§3.13.4.1) for details about how the data model is constructed. When evaluated against this data model instance, {test} evaluates to either `true` or `false` (if any other value is returned, it's converted to either `true` or `false` as if by a call to the XPath fn:boolean function).

See Annotations (§3.15) for information on the role of the {annotations} property.

3.13.2 XML Representation of Assertion Schema Components

The XML representation for an assertion schema component is an <assert> element information item. The correspondences between the properties of that information item after the appropriate ·pre-processing· and the properties of the component it corresponds to are as follows:

XML Representation Summary: `assert` Element Information Item

```
<assert
  id = ID
  test = an XPath expression
  xpathDefaultNamespace = (anyURI | (##defaultNamespace | ##targetNamespace | ##local))
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</assert>
```

The <assert> element maps to an Assertion component as follows.

XML Mapping Summary for Assertion Schema Component

Property	Representation
{test}	An XPath Expression property record, as described below, with <assert> as the "host element" and `test` as the designated expression [attribute].
{annotations}	The ·annotation mapping· of the <assert> element, as defined in XML Representation of Annotation Schema Components (§3.15.2).

Assertions, like identity constraints and conditional type assignment, use [XPath 2.0] expressions. The expression itself is recorded, together with relevant parts of the context, in an XPath Expression property record. The mapping is as described below; in each case, the XPath expression itself is given in a **designated attribute** of the appropriate "**host element**".

XML Mapping Summary for XPath Expression Property Record

Property	Representation
{namespace bindings}	A set of Namespace Binding property records. Each member corresponds to an entry in the [in-scope namespaces] of the host element, with {prefix} being the [prefix] and {namespace} the [namespace name].

{default namespace}	Let **D** be the ·actual value· of the xpathDefaultNamespace [attribute], if present on the host element, otherwise that of the xpathDefaultNamespace [attribute] of the <schema> ancestor. Then the value is the appropriate **case** among the following: 1 **If D** is ##defaultNamespace, **then** the appropriate **case** among the following: 1.1 **If** there is an entry in the [in-scope namespaces] of the host element whose [prefix] is ·absent·, **then** the corresponding [namespace name]; 1.2 **otherwise** ·absent·; 2 **If D** is ##targetNamespace, **then** the appropriate **case** among the following: 2.1 **If** the targetNamespace [attribute] is present on the <schema> ancestor, **then** its ·actual value·; 2.2 **otherwise** ·absent·; 3 **If D** is ##local, **then** ·absent·; 4 **otherwise** (**D** is an xs:anyURI value) **D**.
{base URI}	The [base URI] of the host element.
{expression}	An XPath expression corresponding to the ·actual value· of the designated [attribute] of the host element.

Example

```
<xs:complexType name="intRange">
 <xs:attribute name="min" type="xs:int"/>
 <xs:attribute name="max" type="xs:int"/>
 <xs:assert test="@min le @max"/>
</xs:complexType>
```

The value of the min attribute must be less than or equal to that of the max attribute. Note that the attributes are validated before the assertion on the parent element is checked, so the typed values of the attributes are available for comparison; it is not necessary to cast the values to int or some other numeric type before comparing them.

Example

```
<xs:complexType name="arrayType">
 <xs:sequence>
  <xs:element name="entry" minOccurs="0" maxOccurs="unbounded"/>
 </xs:sequence>
 <xs:attribute name="length" type="xs:int"/>
 <xs:assert test="@length eq fn:count(./entry)"/>
</xs:complexType>
```

The value of the length attribute must be the same as the number of occurrences of entry sub-elements.

3.13.3 Constraints on XML Representations of Assertions

None as such.

3.13.4 Assertion Validation Rules

3.13.4.1 Assertion Satisfied

Validation Rule: Assertion Satisfied

An element information item ***E*** is locally ·valid· with respect to an assertion if and only if the {test} evaluates to `true` (see below) without raising any dynamic error or type error.

Evaluation of {test} is performed as defined in [XPath 2.0], with the following conditions:

1 A data model instance (see [XDM]) is constructed in the following way:
  1.1 ***E*** is validated with respect to its ·governing element declaration·, as defined in Element Locally Valid (Element) (§3.3.4.3), if the ·governing element declaration· exists, otherwise against its ·governing type definition·, as defined in Element Locally Valid (Complex Type) (§3.4.4.2), except that for ***E*** itself (though not for its element information item descendants), clause 6 of Element Locally Valid (Complex Type) (§3.4.4.2) is skipped. (Informally, the element is validated normally, except that assertions are not checked.)

    Note: It is a consequence of this rule that the [attributes] and [children] of ***E*** will be validated in the usual way.

  1.2 A "partial" ·post-schema-validation infoset· describing the results of this partial validation of ***E*** is constructed. The ·post-schema-validation infoset· properties of ***E***'s [children] and [attributes] are defined in the usual way. On ***E*** itself, all ·post-schema-validation infoset· properties are supplied as described elsewhere in this specification if their values are known. The element's [validity] property is given the value ***invalid*** if and only if the element is known to be invalid; otherwise it is given the value ***notKnown***. The element's [validation attempted] property is given the value ***partial***.

    Note: Since the assertions of its ·governing type definition· have not been checked, ***E*** has been only partially validated, and can be known to be invalid, but not known to be valid. The values of the [validity] and [validation attempted] properties are set accordingly.

  By default, comments and processing instructions are excluded from the partial ·post-schema-validation infoset·, but ·at user option· processors MAY retain comments and processing instructions instead of excluding them.

    Note: If comments and processing instructions are retained, the consequence will be that assertions are able to test for their presence or absence and to examine their contents.

  1.3 From the "partial" ·post-schema-validation infoset·, a data model instance is constructed as described in [XDM]. The root node of the [XDM] instance is constructed from ***E***; the data model instance contains only that node and nodes constructed from the [attributes], [children], and descendants of ***E***.

    Note: It is a consequence of this construction that attempts to refer, in an assertion, to the siblings or ancestors of ***E***, or to any part of the input document outside of ***E*** itself, will be unsuccessful. Such attempted references are not in themselves errors, but the data model instance used to evaluate them does not include any representation of any parts of the document outside of ***E***, so they cannot be referred to.

2 The XPath expression {test} is evaluated, following the rules given in XPath Evaluation (§3.13.4.2), with the following conditions and modifications:
  2.1 The root node of the [XDM] instance described in clause 1 serves as the context node for evaluation of the XPath expression.

2.2 The static context is augmented with the variable "$value", as described in Assertion Properties Correct (§3.13.6.1).
2.3 The variable "$value" appears as a member of the variable values in the dynamic context. The expanded QName of that member has no namespace URI and has "value" as the local name. The value of the member is determined by the appropriate **case** among the following:
2.3.1 **If all** of the following are true:
2.3.1.1 ***E***'s [validity] in the "partial" ·post-schema-validation infoset· is not ***invalid***;
2.3.1.2 ***E***'s [nil] in the "partial" ·post-schema-validation infoset· does not exist or has value ***false***;
2.3.1.3 the {content type} of ***E***'s ·governing type definition· has {variety} ***simple***, **then** the value is the XDM representation of ***E***.[schema actual value] under the {content type} . {simple type definition} of ***E***'s ·governing type definition·.

Note: This clause provides type information to simple contents of elements, to make type-aware comparisons and operations possible without explicit casting in the XPath expressions.

Note: For complex types with simple content, the element node may be referred to as ".", while its content may be referred to as "$value". Since the element node, as a consequence of clause 1.2, will normally have the type annotation anyType, its atomized value will be a single atomic value of type untypedAtomic. By contrast, $value will be a sequence of one or more atomic values, whose types are the most specific (narrowest) built-in types available.

2.3.2 **otherwise** (in the "partial" ·post-schema-validation infoset·, ***E***.[validity] = ***invalid*** or ***E***.[nil] = ***true*** or ***E***'s ·governing type definition· does not have {content type} . {variety} = ***simple***) the value is the empty sequence.
3 The evaluation result is converted to either true or false as if by a call to the XPath fn:boolean function.

Note: Although the rules just given describe how an ·post-schema-validation infoset· and a [XDM] instance are constructed, processors are not required to construct actual data structures representing them. However, the result of XPath evaluation MUST be the same as if such ·post-schema-validation infoset· and [XDM] instance data structures were constructed.

3.13.4.2 XPath Evaluation

Validation Rule: XPath Evaluation

An XPath Expression property record ***X***, with a context node ***E***, is evaluated as defined in [XPath 2.0], with a static context as described in XPath Valid (§3.13.6.2) (unless otherwise specified elsewhere) and with the following dynamic context (again, unless otherwise specified elsewhere):
1 The context item is ***E***.
2 The context position is 1.
3 The context size is 1.
4 The variable values is the empty set.
5 The function implementations include an implementation of every function in the function signatures of the static context. See XPath Valid (§3.13.6.2).
6 The current dateTime is ·implementation-defined·, but is constant during an ·assessment· episode.
7 The implicit timezone is ·implementation-defined·, but is constant during an ·assessment· episode.

8 The available documents is the empty set.
9 The available collections is the empty set.
10 The default collection is the empty sequence.

It is ·implementation-defined· (both in this specification and in [XPath 2.0]) when type errors are detected and whether, when detected, they are treated as static or dynamic errors.

> **Note:** It is a consequence of this rule that a conforming processor which treats a type error in an XPath expression as a dynamic error will treat the expression as having evaluated to false, while a conforming processor which treats type errors as static errors will report an error in the schema.

3.13.5 Assertion Information Set Contributions

None as such.

3.13.6 Constraints on Assertion Schema Components

All assertions (see Assertions (§3.13)) MUST satisfy the following constraints.

3.13.6.1 Assertion Properties Correct

Schema Component Constraint: Assertion Properties Correct

All of the following MUST be true:
1 The values of the properties of an assertion are as described in the property tableau in The Assertion Schema Component (§3.13.1), modulo the impact of Missing Sub-components (§5.3).
2 The {test} satisfies the constraint XPath Valid (§3.13.6.2), with the following modifications to the static context:
  2.1 The in-scope variables is a set with a single member. The `expanded QName` of that member has no namespace URI and has `value` as the local name. The (static) `type` of the member is `anyAtomicType*`.
  2.2 The function signatures includes signatures for **all** of the following:
    2.2.1 Functions in the `http://www.w3.org/2005/xpath-functions` namespace as defined in the [Functions and Operators] specification.
    2.2.2 Constructor functions for the built-in datatypes.

> **Note:** The XDM type label `anyAtomicType*` simply says that for static typing purposes the variable `$value` will have a value consisting of a sequence of zero or more atomic values.

3.13.6.2 XPath Valid

Schema Component Constraint: XPath Valid

For an XPath Expression property record ***X*** to be valid, **all** of the following MUST be true:
1 The {expression} of ***X*** is a valid XPath expression, as defined in [XPath 2.0].
2 ***X*** does not produce any static error, under the following conditions (except as specified elsewhere):
  2.1 The Static Typing Feature is disabled.
  2.2 The static context is given as follows:
    2.2.1 XPath 1.0 compatibility mode is ***false***.
    2.2.2 The statically known namespaces is the {namespace bindings} of ***X***.

2.2.3 The default element/type namespace is the {default namespace} of ***X***.
2.2.4 The default function namespace is `http://www.w3.org/2005/xpath-functions`.
2.2.5 The in-scope schema definitions are those components that are present in every schema by definition, as defined in Built-in Attribute Declarations (§3.2.7), Built-in Complex Type Definition (§3.4.7) and Built-in Simple Type Definitions (§3.16.7).
2.2.6 The in-scope variables is the empty set.
2.2.7 The context item static type is not applicable, because the Static Typing Feature is disabled.
2.2.8 The function signatures are ·implementation-defined·.

Note: If ***X*** belongs to an Assertion or a Type Alternative, Assertion Properties Correct (§3.13.6.1) and Type Alternative Properties Correct (§3.12.6) impose additional constraints on the set of required functions.

2.2.9 The statically known collations are ·implementation-defined·, but always include the Unicode codepoint collation (`http://www.w3.org/2005/xpath-functions/collation/codepoint`) defined by [Functions and Operators].
2.2.10 The default collation is the Unicode codepoint collation.
2.2.11 The base URI is the {base URI} of ***X***.
2.2.12 The statically known documents is the empty set.
2.2.13 The statically known collections is ·implementation-defined·.
2.2.14 The statically known default collection type is ·implementation-defined·.

It is ·implementation-defined· (both in this specification and in [XPath 2.0]) when type errors are detected and whether, when detected, they are treated as static or dynamic errors.

Note: It is a consequence of this rule that a conforming processor which treats a type error in an XPath expression as a dynamic error will treat the expression as having evaluated to false, while a conforming processor which treats type errors as static errors will report an error in the schema.

3.14 Notation Declarations

3.14.1 The Notation Declaration Schema Component
3.14.2 XML Representation of Notation Declaration Schema Components
3.14.3 Constraints on XML Representations of Notation Declarations
3.14.4 Notation Declaration Validation Rules
3.14.5 Notation Declaration Information Set Contributions
3.14.6 Constraints on Notation Declaration Schema Components

Notation declarations reconstruct XML NOTATION declarations.

Example

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

The XML representation of a notation declaration.

3.14.1 The Notation Declaration Schema Component

The notation declaration schema component has the following properties:

Schema Component: Notation Declaration, a kind of Annotated Component

{annotations}

A sequence of Annotation components.

{name}
An xs:NCName value. Required.

{target namespace}
An xs:anyURI value. Optional.

{system identifier}
An xs:anyURI value. Required if {public identifier} is ·absent·, otherwise ({public identifier} is present) optional.

{public identifier}
A publicID value. Required if {system identifier} is ·absent·, otherwise ({system identifier} is present) optional.
As defined in [XML 1.0] or [XML 1.1].

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. An element or attribute information item with its ·governing· type definition or its ·validating type· derived from the NOTATION simple type is ·valid· only if its value was among the enumerations of such simple type. As a consequence such a value is required to be the {name} of a notation declaration.

See Annotations (§3.15) for information on the role of the {annotations} property.

3.14.2 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 after the appropriate ·pre-processing· and the 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>
```

The <notation> element maps to a Notation Declaration component as follows.

XML Mapping Summary for 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 <schema> ancestor element information item if present, otherwise ·absent·.
{system identifier}	The ·actual value· of the `system` [attribute], if present, otherwise ·absent·.
{public identifier}	The ·actual value· of the `public` [attribute], if present, otherwise ·absent·.
{annotations}	The ·annotation mapping· of the <notation> element, as defined in XML Representation of Annotation Schema Components (§3.15.2).

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>
```

3.14.3 Constraints on XML Representations of Notation Declarations

None as such.

3.14.4 Notation Declaration Validation Rules

None as such.

3.14.5 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 has the following properties:

PSVI Contributions for element information items

[notation]
: An ·item isomorphic· to the notation declaration ·resolved· to by the attribute item's ·actual value·

[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.

Note: Element as well as attribute information items may be ·valid· with respect to a NOTATION, but only attribute information items cause a notation declaration to appear in the ·post-schema-validation infoset· as a property of their parent.

3.14.6 Constraints on Notation Declaration Schema Components

All notation declarations (see Notation Declarations (§3.14)) 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 (§3.14.1), modulo the impact of Missing Sub-components (§5.3).

3.15 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.

3.15.1 The Annotation Schema Component

The annotation schema component has the following properties:

Schema Component: Annotation, a kind of Component

{application information}
: A sequence of Element information items.

{user information}
: A sequence of Element information items.

{attributes}
: A set 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 XSD 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.

The mapping defined in this specification from XML representations to components does not apply to XML elements contained within an <annotation> element; such elements do not correspond to components, when the mapping defined here is used.

It is ·implementation-defined· what effect, if any, the invalidity of a descendant of <annotation> has on the construction of schema components from the enclosing schema document.

The name [Definition:] **Annotated Component** covers all the different kinds of component which may have annotations.

3.15.2 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 after the appropriate ·pre-processing· and the properties of the component it corresponds to are as follows:

XML Representation Summary: `annotation` Element Information Item et al.

```
<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>
```

The <annotation> element and its descendants map to an Annotation component as follows.

XML Mapping Summary for 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 set 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}.

Virtually every kind of schema component defined in this specification has an {annotations} property. When the component is described in a schema document, the mapping from the XML representation of the component to the Annotation components in the appropriate {annotations} property follows the rules described in the next paragraph.

[Definition:] The **annotation mapping** of a set of element information items ***ES*** is a sequence of annotations ***AS***, with the following properties:

1 For every <annotation> element information item among the [children] of any element information item in ***ES***, there is a corresponding Annotation component in ***AS***.

> **Note:** As noted above, the {attributes} property of each Annotation component includes all the attribute information items on the <annotation> element itself, on the XML element which represents (and maps to) the component being annotated, and on any intervening XML elements, if those attribute information items have [namespace name]s different from the XSD namespace.

2 If any element information item ***E*** in ***ES*** has any attribute information items ***A*** such that **all** of the following are true:
  2.1 ***A*** is in ***E***.[attributes].
  2.2 ***A***.[namespace name] is present and not the XSD namespace.
  2.3 ***A*** is not included in the [attributes] property of any annotation component described in clause 1.
  then for each such ***E***, an Annotation component ***C*** will appear in ***AS***, with ***C***.{application information} and ***C***.{user information} each being the empty sequence and ***C***.{attributes} containing all and only those attribute information items ***A*** among ***E***.[attributes].

3 ***AS*** contains no other Annotation components.

[Definition:] The **annotation mapping** of a single element information item is the ·annotation mapping· of the singleton set containing that element.

> **Note:** The order of Annotation components within the sequence is ·implementation-dependent·.

> **Note:** When the input set has more than one member, the Annotation components in the resulting sequence do not record which element in the set they correspond to. The attribute information items in the {attributes} of any Annotation similarly do not indicate which element information item in the schema document was their parent.

3.15.3 Constraints on XML Representations of Annotations

None as such.

3.15.4 Annotation Validation Rules

None as such.

3.15.5 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.

3.15.6 Constraints on Annotation Schema Components

All annotations (see Annotations (§3.15)) 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 (§3.15.1), modulo the impact of Missing Sub-components (§5.3).

3.16 Simple Type Definitions

Note: This section consists of a combination of copies of normative material from [XML Schema: Datatypes], for local cross-reference purposes, and material unique to this specification, 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="celsiusWaterTemp">
 <xs:restriction base="xs:decimal">
  <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.

3.16.1 The Simple Type Definition Schema Component

The simple type definition schema component has the following properties:

Schema Component: Simple Type Definition, a kind of Type Definition

{annotations}
A sequence of Annotation components.

{name}
An xs:NCName value. Optional.

{target namespace}
An xs:anyURI value. Optional.

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

{context}
Required if {name} is ·absent·, otherwise MUST be ·absent·.
Either an Attribute Declaration, an Element Declaration, a Complex Type Definition, or a Simple Type Definition.

{base type definition}
A Type Definition component. Required.
With one exception, the {base type definition} of any Simple Type Definition is a Simple Type Definition. The exception is ·xs:anySimpleType·, which has ·xs:anyType·, a Complex Type Definition, as its {base type definition}.

{facets}
A set of Constraining Facet components.

{fundamental facets}
A set of Fundamental Facet components.

{variety}
One of {***atomic***, ***list***, ***union***}. Required for all Simple Type Definitions except ·xs:anySimpleType·, in which it is ·absent·.

{primitive type definition}
A Simple Type Definition component. With one exception, required if {variety} is ***atomic***, otherwise MUST be ·absent·. The exception is ·xs:anyAtomicType·, whose {primitive type definition} is ·absent·.
If non-·absent·, MUST be a primitive definition.

{item type definition}
A Simple Type Definition component. Required if {variety} is ***list***, otherwise MUST be ·absent·.
The value of this property MUST be a primitive or ordinary simple type definition with {variety} = ***atomic***, or an ordinary simple type definition with {variety} = ***union*** whose basic members are all atomic; the value MUST NOT itself be a list type (have {variety} = ***list***) or have any basic members which are list types.

{member type definitions}
A sequence of primitive or ordinary Simple Type Definition components.
MUST be present (but MAY be empty) if {variety} is ***union***, otherwise MUST be ·absent·.

The sequence may contain any primitive or ordinary simple type definition, but MUST NOT contain any special type definitions.

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 ·XSD 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 (§2.7.1)), and for use in the XML representation of schema components (specifically in <element> and <attribute>). See References to

schema components across namespaces (<import>) (§4.2.6) 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 Schema: 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 ·xs:anySimpleType· if and only if the type definition in question is ·xs:anyAtomicType· or a list or union type definition which is not itself ·derived· by restriction from a list or union respectively. A type definition has ·xs:anyAtomicType· as its {base type definition} if and only if it is one of the primitive datatypes. Each *atomic* type is ultimately a restriction of exactly one such primitive datatype, which is its {primitive type definition}.

The {facets} property contains a set of constraining facets which are used to specify constraints on the datatype described by the simple type definition. 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}).

Constraining facets are defined in [XML Schema: Datatypes]. All conforming implementations of this specification MUST support all of the facets defined in [XML Schema: Datatypes]. It is ·implementation-defined· whether additional facets are supported; if they are, the implementation MUST satisfy the rules for ·implementation-defined· facets described in [XML Schema: Datatypes].

As specified in [XML Schema: 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 Schema: 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.

·xs:anySimpleType· or ·xs:anyAtomicType· MUST *not* be named as the {base type definition} of any user-defined atomic simple type definitions: as they allow no constraining facets, this would be incoherent.

See Annotations (§3.15) for information on the role of the {annotations} property.

3.16.2 XML Representation of Simple Type Definition Schema Components

As always, the mapping rules given in this section apply after, not before, the appropriate ·pre-processing·.

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

XML Representation Summary: `simpleType` Element Information Item et al.

```
<simpleType
  final = (#all | List of (list | union | restriction | extension))
  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 | assertion |
explicitTimezone | {any with namespace: ##other})*))
</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>
```

The <simpleType> element and its descendants normally, when there are no errors, map to a Simple Type Definition component. The case in which an unknown facet is used in the definition of a simple type definition is handled specially: the <simpleType> in question is not in error, but it does not map to any component at all.

> **Note:** The effect of the special handling of unknown facets is to ensure (1) that ·implementation-defined· facets which are not supported by a particular implementation result in the types which depend upon them not being present in the schema, and (2) that the presence of references to unknown facets in a schema document does not prevent the rest of the schema document being processed and used.

The following subsections define one set of common mapping rules for simple type definitions, and three specialized sets of mapping rules for atomic, list, and union datatypes, respectively.

- If the <simpleType> element has a <restriction> element among its children, and the base type definition has {variety} = ***atomic***, then the mapping rules in Common mapping rules for Simple Type Definitions (§3.16.2.1) and Mapping Rules for Atomic Simple Type Definitions (§3.16.2.2) apply.

- If the <simpleType> element has a <list> element among its children, or if it has a <restriction> child and the base type definition has {variety} = ***list***, then the mapping rules in Common mapping rules for Simple Type Definitions (§3.16.2.1) and Mapping Rules for Lists (§3.16.2.3) apply.

- If the <simpleType> element has a <union> element among its children, or if it has a <restriction> child and the base type definition has {variety} = ***union***, then the mapping rules in Common mapping rules for Simple Type Definitions (§3.16.2.1) and Mapping Rules for Unions (§3.16.2.4) apply.

3.16.2.1 Common mapping rules for Simple Type Definitions

The following rules apply to all simple type definitions.

XML Mapping Summary for Simple Type Definition Schema Component

Property	Representation
{name}	The ·actual value· of the `name` [attribute] if present on the <simpleType> element, otherwise ·absent·.
{target namespace}	The ·actual value· of the `targetNamespace` [attribute] of the ancestor <schema> 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** ·`xs:anySimpleType`·.
{final}	A subset of {***restriction***, ***extension***, ***list***, ***union***}, determined as follows. [Definition:] Let **FS** be the ·actual value· of the `final` [attribute], if present, otherwise the ·actual value· of the `finalDefault` [attribute] of the ancestor `schema` element, if present, otherwise the empty string. Then the property value is the appropriate **case** among the following: 1 **If** ·FS· is the empty string, **then** the empty set; 2 **If** ·FS· is "`#all`", **then** {***restriction***, ***extension***, ***list***, ***union***}; 3 **otherwise** Consider ·FS· as a space-separated list, and include ***restriction*** if "`restriction`" is in that list, and similarly for ***extension***, ***list*** and ***union***.
{context}	The appropriate **case** among the following: 1 **If** the `name` [attribute] is present, **then** ·absent· 2 **otherwise** the appropriate **case** among the following: 2.1 **If** the parent element information item is <attribute>, **then** the corresponding Attribute Declaration 2.2 **If** the parent element information item is <element>, **then** the corresponding Element Declaration 2.3 **If** the parent element information item is <list> or <union>, **then** the Simple Type Definition corresponding to the grandparent <simpleType> element information item 2.4 **If** the parent element information item is <alternative>, **then** the Element Declaration corresponding to the nearest enclosing <element> element information item

	2.5 **otherwise** (the parent element information item is <restriction>), the appropriate **case** among the following: 2.5.1 **If** the grandparent element information item is <simpleType>, **then** the Simple Type Definition corresponding to the grandparent 2.5.2 **otherwise** (the grandparent element information item is <simpleContent>), the Simple Type Definition which is the {content type}.{simple type definition} of the Complex Type Definition corresponding to the great-grandparent <complexType> element information item.
{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}.
{facets}	The appropriate **case** among the following: 1 **If** the <restriction> alternative is chosen **and** the children of the <restriction> element are all either <simpleType> elements, <annotation> elements, or elements which specify constraining facets supported by the processor, **then** the set of Constraining Facet components obtained by ·overlaying· the {facets} of the {base type definition} with the set of Constraining Facet components corresponding to those [children] of <restriction> which specify facets, as defined in Simple Type Restriction (Facets) (§3.16.6.4). 2 **If** the <restriction> alternative is chosen **and** the children of the <restriction> element include at least one element of which the processor has no prior knowledge (i.e. not a <simpleType> element, an <annotation> element, or an element denoting a constraining facet known to and supported by the processor), **then** the <simpleType> element maps to no component at all (but is not in error solely on account of the presence of the unknown element). 3 **If** the <list> alternative is chosen, **then** a set with one member, a whiteSpace facet with {value} = ***collapse*** and {fixed} = ***true***. 4 **otherwise** the empty set
{fundamental facets}	Based on {variety}, {facets}, {base type definition} and {member type definitions}, a set of Fundamental Facet components, one each as specified in The ordered Schema Component , The bounded Schema Component , The cardinality Schema Component and The numeric Schema Component .
{annotations}	The ·annotation mapping· of the set of elements containing the <simpleType>, and one of the <restriction>, <list> or <union> [children], whichever is present, as defined in XML Representation of Annotation Schema Components (§3.15.2).

3.16.2.2 Mapping Rules for Atomic Simple Type Definitions

The following rule applies if the {variety} is ***atomic***

[Definition:] The **ancestors** of a ·type definition· are its {base type definition} and the ·ancestors· of its {base type definition}. (The ancestors of a Simple Type Definition ***T*** in the type hierarchy are themselves ·type definitions·; they are distinct from the XML elements which may be ancestors, in the XML document hierarchy, of the <simpleType> element which declares ***T***.)

XML Mapping Summary for Atomic Simple Type Definition Schema Component

Property	Representation
{primitive type definition}	From among the ·ancestors· of this Simple Type Definition, that Simple Type Definition which corresponds to a primitive datatype.

3.16.2.3 Mapping Rules for Lists

If the {variety} is ***list***, the following additional property mapping applies:

XML Mapping Summary for List Simple Type Definition Schema Component

Property	Representation
{item type definition}	The appropriate **case** among the following: 1 **If** the {base type definition} is ·xs:anySimpleType·, **then** the Simple Type Definition (a) ·resolved· to by the ·actual value· of the itemType [attribute] of <list>, or (b), corresponding to the <simpleType> among the [children] of <list>, whichever is present. **Note:** In this case, a <list> element will invariably be present; it will invariably have either an itemType [attribute] or a <simpleType> [child], but not both. 2 **otherwise** (that is, the {base type definition} is not ·xs:anySimpleType·), the {item type definition} of the {base type definition}. **Note:** In this case, a <restriction> element will invariably be present.

3.16.2.4 Mapping Rules for Unions

If the {variety} is ***union***, the following additional property mapping applies:

XML Mapping Summary for Union Simple Type Definition Schema Component

Property	Representation
{member type definitions}	The appropriate **case** among the following: 1 **If** the {base type definition} is ·xs:anySimpleType·, **then** the sequence of Simple Type Definitions (a) ·resolved· to by the items in the ·actual value· of the memberTypes [attribute] of <union>, if any, and (b) corresponding to the <simpleType>s among the [children] of <union>, if any, in order. **Note:** In this case, a <union> element will invariably be present; it will invariably have either a memberTypes [attribute] or one or more <simpleType> [children], or both.

2 **otherwise** (that is, the {base type definition} is not ·xs:anySimpleType·), the {member type definitions} of the {base type definition}.

Note: In this case, a <restriction> element will invariably be present.

3.16.3 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 schema documents, **all** of the following MUST be true:

1 No two elements among the [children] of <restriction> have the same expanded name in the Schema (xs) namespace, unless that expanded name is one of xs:enumeration, xs:pattern, or xs:assertion.

Note: That is, most of the facets for simple types defined by this specification are forbidden to occur more than once. But <enumeration>, <pattern>, and <assertion> may occur multiple times, and facets defined in other namespaces and made available as extensions to this specification may occur multiple times.

2 If the <restriction> alternative is chosen, it has either a base [attribute] or a <simpleType> among its [children], but not both.

3 If the <list> alternative is chosen, it has either an itemType [attribute] or a <simpleType> among its [children], but not both.

4 If the <union> alternative is chosen, either it has a non-empty memberTypes [attribute] or it has at least one simpleType [child].

3.16.4 Simple Type Definition Validation Rules

Validation Rule: String Valid

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

1 The ·normalized value· of ***S***, ***N***, is calculated using the whiteSpace facet, and any other pre-lexical facets associated with ***T***, as described in the definition of the term "·normalized value·".

2 ***N*** is schema-valid with respect to ***T*** as defined by Datatype Valid in [XML Schema: Datatypes].

3 Let ***V*** be the ·actual value· of ***N*** with respect to ***T***. Then:

Every ·ENTITY value· in ***V*** is a ·declared entity name·.

[Definition:] When a string ***N*** is schema-valid with respect to a simple type definition ***T*** as defined by Datatype Valid with the corresponding ·actual value· ***V***,

1 The **validating type** of ***V*** is ***T*** if ***T*** is not a union type, otherwise the validating type is the basic member of ***T***'s transitive membership which actually ·validated· ***N***.

2 If the ·validating type· of ***V*** is a list type ***L*** and the {item type definition} of ***L*** is ***I***, then the **validating type** of an (atomic) item value ***A*** occurring in ***V*** is ***I*** if ***I*** is not a union type, otherwise the validating type is the basic member of ***I***'s transitive membership which actually ·validated· the substring in ***N*** that corresponds to ***A***.

[Definition:] When the ·validating type· of an ·actual value· is or is ·derived· from a simple type definition ***T***, the value is also referred to as a ***T*** **value**. For example, an **ENTITY value** is an ·actual value· whose ·validating type· is or is derived from the

built-in simple type definition ENTITY, and an **ID value** is one whose ·validating type· is or is derived from ID.

[Definition:] A string is a **declared entity name** if and only 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.

3.16.5 Simple Type Definition Information Set Contributions

None as such.

3.16.6 Constraints on Simple Type Definition Schema Components

3.16.6.1 Simple Type Definition Properties Correct

All simple type definitions 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 are as described in the property tableau in The Simple Type Definition Schema Component, modulo the impact of Missing Sub-components (§5.3).

2 All simple type definitions are, or are ·derived· ultimately from, ·xs:anySimpleType· (so circular definitions are disallowed). That is, it is possible to reach a primitive datatype or ·xs:anySimpleType· by following the {base type definition} zero or more times.

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

4 There is not more than one member of {facets} of the same kind.

5 Each member of {facets} is supported by the processor.

> **Note:** As specified normatively elsewhere, all conforming processors MUST support the facets defined by [XML Schema: Datatypes]; support for additional facets is ·implementation-defined·. If a schema document applies an unknown facet, the immediate result will be a violation of this constraint, so that the simple type defined by means of that facet will be excluded from the schema, and any references to it will be treated as undischarged references.

3.16.6.2 Derivation Valid (Restriction, Simple)

Schema Component Constraint: Derivation Valid (Restriction, Simple)

For any Simple Type Definition ***D*** whose {base type definition} is some Simple Type Definition ***B***, the appropriate **case** among the following MUST be true:

1 **If** ***D***.{variety} = ***atomic***, **then all** of the following are true:

1.1 Either ***D*** is ·xs:anyAtomicType·, or else ***B*** is an atomic simple type definition.

> **Note:** The type ·xs:anyAtomicType· is an exception because its {base type definition} is ·xs:anySimpleType·, whose {variety} is ·absent·.

1.2 ***B***.{final} does not contain ***restriction***.

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

1.3.1 **DF** is applicable to ***D***, as specified in Applicable Facets of [XML Schema: Datatypes].

1.3.2 **DF** satisfies the constraints on facet components given in the appropriate subsection of Constraining Facets in [XML Schema: Datatypes].

2 **If** ***D***.{variety} = ***list***, **then all** of the following are true:

2.1 ***D***.{item type definition} is not a special type definition and either ***D***.{item type definition}.{variety} = ***atomic*** or ***D***.{item type definition}.{variety} = ***union*** and there are no types whose {variety} is ***list*** among the union's transitive membership.
2.2 The appropriate **case** among the following is true:
2.2.1 **If** ***B*** is ·xs:anySimpleType· , **then all** of the following are true:
2.2.1.1 ***D***.{item type definition}.{final} does not contain ***list***.
2.2.1.2 ***D***.{facets} contains only the ***whiteSpace*** facet component with {value} = ***collapse*** and {fixed} = ***true***.
2.2.2 **otherwise all** of the following are true:
2.2.2.1 ***B***.{variety} = ***list***.
2.2.2.2 ***B***.{final} does not contain ***restriction***.
2.2.2.3 ***D***.{item type definition} is validly ·derived· from ***B***.{item type definition}, as defined in Type Derivation OK (Simple) (§3.16.6.3).
2.2.2.4 All facets in {facets} are applicable to ***D***, as specified in Applicable Facets.
2.2.2.5 All facets in {facets} satisfy the constraints on facet components given in the appropriate subsection of Constraining Facets.

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

3 **If** ***D***.{variety} is ***union***, **then all** of the following are true:
3.1 ***D***.{member type definitions} does not contain a special type definition.
3.2 The appropriate **case** among the following is true:
3.2.1 **If** ***B*** is ·xs:anySimpleType· , **then all** of the following are true:
3.2.1.1 All of the {member type definitions} have a {final} which does not contain ***union***.
3.2.1.2 ***D***.{facets} is empty.
3.2.2 **otherwise all** of the following are true:
3.2.2.1 ***B***.{variety} = ***union***.
3.2.2.2 ***B***.{final} does not contain ***restriction***.
3.2.2.3 Each type definition in ***D***.{member type definitions} is validly ·derived· from the corresponding type definition in ***B***.{member type definitions}, as defined in Type Derivation OK (Simple) (§3.16.6.3).
3.2.2.4 All facets in {facets} are applicable to ***D***, as specified in Applicable Facets.
3.2.2.5 All facets in {facets} satisfy the constraints on facet components given in the appropriate subsection of Constraining Facets.

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

3.3 Neither ***D*** nor any type ·derived· from it is a member of its own transitive membership.

[Definition:] A simple type definition **T** is a **valid restriction** of its {base type definition} if and only if **T** satisfies constraint Derivation Valid (Restriction, Simple) (§3.16.6.2).

3.16.6.3 Type Derivation OK (Simple)

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) subject to a set of blocking keywords drawn from the

set {***extension***, ***restriction***, ***list***, ***union***} (of which only ***restriction*** is actually relevant; call this set ***S***) **one** of the following MUST be true:
1 They are the same type definition.
2 **All** of the following are true:
  2.1 ***restriction*** is not in ***S***, or in **D**.{base type definition}.{final};
  2.2 **One or more** of the following is true:
    2.2.1 **D**.{base type definition} = **B**.
    2.2.2 **D**.{base type definition} is not ·xs:anyType· and is validly ·derived· from **B** given ***S***, as defined by this constraint.
    2.2.3 **D**.{variety} = ***list*** or ***union*** and **B** is ·xs:anySimpleType·.
    2.2.4 **All** of the following are true:
      2.2.4.1 **B**.{variety} = ***union***.
      2.2.4.2 **D** is validly ·derived· from a type definition **M** in **B**'s transitive membership given ***S***, as defined by this constraint.
      2.2.4.3 The {facets} property of **B** and of any intervening union datatypes is empty.

Note: It is a consequence of this requirement that the value space, lexical space, and lexical mapping of **D** will be subsets of those of **B**.

Note: With respect to clause 1, see the Note on identity at the end of (§3.4.6.5) above.

Note: When a simple type definition ***S*** is said to be "validly ·derived·" from a type definition ***T***, without mention of any specific set of blocking keywords, then what is meant is that ***S*** is validly derived from ***T***, subject to the empty set of blocking keywords, i.e. without any particular limitations.

Note: It is a consequence of clause 2.2.4 that the constraint (§3.16.6.2) can hold between a Simple Type Definition in the transitive membership of a union type, and the union type, even though neither is actually ·derived· from the other. The slightly misleading terminology is retained for historical reasons and for compatibility with version 1.0 of this specification.

3.16.6.4 Simple Type Restriction (Facets)

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 {facets} of ***B*** ·overlaid· with ***S***.

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

[Definition:] Given two sets of facets ***B*** and ***S***, the result of **overlaying *B*** with ***S*** is the set of facets ***R*** for which **all** of the following are true:
1 Every facet in **S** is in **R**.
2 Every facet in **B** is in **R**, unless it is of the same kind as some facet in **S**, in which case it is not included in **R**.
3 Every facet in **R** is required by clause 1 or clause 2 above.

3.16.7 Built-in Simple Type Definitions

3.16.7.1 xs:anySimpleType

The Simple Type Definition of anySimpleType is present in every schema. It has the following properties:

Simple Type Definition of anySimpleType

Property	Value
{name}	'anySimpleType'
{target namespace}	'http://www.w3.org/2001/XMLSchema'
{final}	The empty set
{context}	·absent·
{base type definition}	·xs:anyType·
{facets}	The empty set
{fundamental facets}	The empty set
{variety}	·absent·
{primitive type definition}	·absent·
{item type definition}	·absent·
{member type definitions}	·absent·
{annotations}	The empty sequence

The definition of ·xs:anySimpleType· 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 ·xs:anyType·, which is *its* {base type definition}.

3.16.7.2 xs:anyAtomicType

The Simple Type Definition of anyAtomicType is present in every schema. It has the following properties:

Simple Type Definition of anyAtomicType

Property	Value
{name}	'anyAtomicType'
{target namespace}	'http://www.w3.org/2001/XMLSchema'
{final}	The empty set
{context}	·absent·
{base type definition}	·xs:anySimpleType·
{facets}	The empty set

{fundamental facets}	The empty set
{variety}	***atomic***
{primitive type definition}	·absent·
{item type definition}	·absent·
{member type definitions}	·absent·
{annotations}	The empty sequence

3.16.7.3 xs:error

A Simple Type Definition for ·xs:error· is present in every schema by definition. It has the following properties:

Simple Type Definition of xs:error

Property	**Value**
{name}	'error'
{target namespace}	'http://www.w3.org/2001/XMLSchema'
{final}	{***extension***, ***restriction***, ***list***, ***union***}
{context}	·absent·
{base type definition}	·xs:anySimpleType·
{facets}	The empty set
{fundamental facets}	The empty set
{variety}	***union***
{primitive type definition}	·absent·
{item type definition}	·absent·
{member type definitions}	The empty sequence
{annotations}	The empty sequence

Note: The datatype xs:error has no valid instances (i.e. it has an empty value space and an empty lexical space). This is a natural consequence of its construction: a value is a value of a union type if and only if it is a value of at least one member of the {member type definitions} of the union. Since xs:error has no member type definitions, there can be no values which are values of at least one of its member types. And since the value space is empty, the lexical space is also empty.

The type xs:error is expected to be used mostly in conditional type assignment. Whenever it serves as the ·governing· type definition for an attribute or element

information item, that item will be invalid.

3.16.7.4 Built-in primitive datatypes

Simple type definitions corresponding to all the built-in primitive datatypes, namely ***string***, ***boolean***, ***float***, ***double***, ***decimal***, ***dateTime***, ***duration***, ***time***, ***date***, ***gMonth***, ***gMonthDay***, ***gDay***, ***gYear***, ***gYearMonth***, ***hexBinary***, ***base64Binary***, ***anyURI***, ***QName*** and ***NOTATION*** (see the Primitive Datatypes section of [XML Schema: Datatypes]) are present by definition in every schema as follows:

Simple Type Definition corresponding to the built-in primitive datatypes

Property	**Value**
{name}	[as appropriate]
{target namespace}	'http://www.w3.org/2001/XMLSchema'
{base type definition}	·xs:anyAtomicType·
{final}	The empty set
{variety}	***atomic***
{primitive type definition}	[this simple type definition itself]
{facets}	{a whitespace facet with [value] = ***collapse*** and [fixed] = ***true*** in all cases except string, which has [value] = ***preserve*** and [fixed] = ***false***}
{fundamental facets}	[as appropriate]
{context}	·absent·
{item type definition}	·absent·
{member type definitions}	·absent·
{annotations}	The empty sequence

All conforming implementations of this specification MUST support all the primitive datatypes defined in [XML Schema: Datatypes]. It is ·implementation-defined· whether additional primitive datatypes are supported, and whether, if so, they are automatically incorporated in every schema or not. If ·implementation-defined· primitives are supported, the implementation MUST satisfy the rules for ·implementation-defined· primitive datatypes described in [XML Schema: Datatypes].

[Definition:] A type about which a processor possesses prior knowledge, and which the processor can support without any declaration of the type being supplied by the user, is said to be **automatically known** to the processor.

Note: By their nature, primitive types can be supported by a processor only if ·automatically known· to that processor.

Types ·automatically known· to a processor, whether primitive or derived, can be included automatically by that processor in all schemas, but need not be. It is possible, for example, for a processor to have built-in prior knowledge of a set of primitive and derived types, but to include them in the schema only when the relevant namespace is explicitly imported, or a given run-time option is selected, or on some other conditions; such conditions are not defined by this specification.

Note: The definition of "·automatically known·" is not intended to prevent implementations from allowing users to specify new primitive types. If an implementation defines a mechanism by which users can define or supply an implementation of a primitive type, then when those mechanisms are successfully used, such user-supplied types are ·automatically known· to the implementation, as that term is used in this specification.

3.16.7.5 Other built-in datatypes

Similarly, simple type definitions corresponding to all the other built-in datatypes (see the Other Built-in Datatypes section of [XML Schema: Datatypes]) are present by definition in every schema, with properties as specified in [XML Schema: Datatypes] and as represented in XML in Illustrative XML representations for the built-in ordinary type definitions.

Simple Type Definition corresponding to the built-in ordinary datatypes

Property	Value
{name}	[as appropriate]
{target namespace}	'http://www.w3.org/2001/XMLSchema'
{base type definition}	[as specified in the appropriate sub-section of Other Built-in Datatypes]
{final}	The empty set
{variety}	[***atomic*** or ***list***, as specified in the appropriate sub-section of Other Built-in Datatypes]
{primitive type definition}	[if {variety} is ***atomic***, then the {primitive type definition} of the {base type definition}, otherwise ·absent·]
{facets}	[as specified in the appropriate sub-section of Other Built-in Datatypes]
{fundamental facets}	[as specified in the appropriate sub-section of Other Built-in Datatypes]
{context}	·absent·
{item type definition}	if {variety} is ***atomic***, then ·absent·, otherwise as specified in the appropriate sub-section of Other Built-in Datatypes]
{member type definitions}	·absent·
{annotations}	As shown in the XML representations of the ordinary built-in datatypes in Illustrative XML representations for the built-in ordinary type definitions

All conforming implementations of this specification MUST support all the built-in datatypes defined in [XML Schema: Datatypes]. It is ·implementation-defined· whether additional derived types are ·automatically known· to the implementation without declaration and whether, if so, they are automatically incorporated in every schema or not.

3.17 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.

3.17.1 The Schema Itself

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

Schema Component: Schema, a kind of Annotated Component

{annotations}
: A sequence of Annotation components.

{type definitions}
: A set of Type Definition components.

{attribute declarations}
: A set of Attribute Declaration components.

{element declarations}
: A set of Element Declaration components.

{attribute group definitions}
: A set of Attribute Group Definition components.

{model group definitions}
: A set of Model Group Definition components.

{notation declarations}
: A set of Notation Declaration components.

{identity-constraint definitions}
: A set of Identity-Constraint Definition components.

3.17.2 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 (§4.2.6.2).

As always, the mapping rules given in this section apply after, not before, the appropriate ·pre-processing·.

XML Representation Summary: schema Element Information Item et al.

```
<schema
  attributeFormDefault = (qualified | unqualified) : unqualified
  blockDefault = (#all | List of (extension | restriction | substitution))  : ''
  defaultAttributes = QName
  xpathDefaultNamespace = (anyURI | (##defaultNamespace | ##targetNamespace | ##local))
 : ##local
  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 | override | annotation)*,
(defaultOpenContent, annotation*)?, ((simpleType | complexType | group |
attributeGroup | element | attribute | notation), annotation*)*)
</schema>
```

```
<defaultOpenContent
  appliesToEmpty = boolean : false
  id = ID
  mode = (interleave | suffix) : interleave
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, any)
</defaultOpenContent>
```

The <schema> element information item maps to a Schema component as follows.

XML Mapping Summary for 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 definitions brought in via <include> (see Assembling a schema for a single target namespace from multiple schema definition documents (<include>) (§4.2.3)), <override> (see Overriding component definitions (<override>) (§4.2.5)), <redefine> (see Including modified component definitions (<redefine>) (§4.2.4)), and <import> (see References to schema components across namespaces (<import>) (§4.2.6)).
{attribute declarations}	The (top-level) attribute declarations corresponding to all the <attribute> element information items in the [children], if any, plus any declarations brought in via <include>, <override>, <redefine>, and <import>.
{element declarations}	The (top-level) element declarations corresponding to all the <element> element information items in the [children], if any, plus

	any declarations brought in via <include>, <override>, <redefine>, and <import>.
{attribute group definitions}	The attribute group definitions corresponding to all the <attributeGroup> element information items in the [children], if any, plus any definitions brought in via <include>, <override>, <redefine>, and <import>.
{model group definitions}	The model group definitions corresponding to all the <group> element information items in the [children], if any, plus any definitions brought in via <include>, <redefine> and <import>.
{notation declarations}	The notation declarations corresponding to all the <notation> element information items in the [children], if any, plus any declarations brought in via <include>, <override>, <redefine>, and <import>.
{identity-constraint definitions}	The identity-constraint definitions corresponding to all the <key>, <keyref>, and <unique> element information items anywhere within the [children], if any, plus any definitions brought in via <include>, <override>, <redefine>, and <import>.
{annotations}	The ·annotation mapping· of the set of elements containing the <schema> and all the <include>, <redefine>, <override>, <import>, and <defaultOpenContent> [children], if any, as defined in XML Representation of Annotation Schema Components (§3.15.2).

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 XSD 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:** [XML Namespaces 1.1] 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 corresponds.

3.17.2.1 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 (<import>) (§4.2.6)) 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 1.1]. 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 Schema: Datatypes]. For brevity, the term ·QName· is also used to refer to ·actual values· in the value space of the QName simple type, which are expanded names with a [Definition:] **local name** and a [Definition:] **namespace name**.

[Definition:] An **NCName** is a name with no colon, as defined in [XML Namespaces 1.1]. 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 Schema: Datatypes].

> **Note:** It is ·implementation-defined· whether a schema processor supports the definitions of QName and NCName found in [XML Namespaces 1.1] or those found in [Namespaces in XML 1.0] or both.

[Definition:] A ·QName· in a schema document **resolves to** a component in a schema if and only if in the context of that schema the QName and the component together satisfy the rule QName resolution (Schema Document) (§3.17.6.2). A ·QName· in an input document, or a pair consisting of a local name and a namespace name, **resolves to** a component in a schema if and only if in the context of that schema the QName (or the name + namespace pair) and the component together satisfy the rule QName resolution (Instance) (§3.17.6.3).

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 (<import>) (§4.2.6) for a discussion of importing.

3.17.2.2 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 using these names will not work to reference schema components from outside the context of schema documents.

The preferred way to refer to schema components is to use [XML Schema: Component Designators], which defines a mechanism to designate schema components using the `xscd()` [XPointer] scheme.

Alternatively, references to specific element information items in the schema document can be interpreted as to their corresponding schema documents. This can be done using mechanisms supported by [XPointer], for example, by using shorthand pointers. It is a matter for applications to specify whether they interpret these references 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.

3.17.3 Constraints on XML Representations of Schemas

None as such.

3.17.4 Validation Rules for Schemas as a Whole

None as such.

3.17.5 Schema Information Set Contributions

3.17.5.1 Schema Information

Schema Information Set Contribution: Schema Information

Schema components provide a wealth of information about the basis of ·assessment·, which can often be useful for 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.

The ·validation root· has the following properties:

PSVI Contributions for element or attribute information items

[schema information]
A set of **namespace schema information** information