Schema for Object-Oriented XML 2.0

Authors:

Andrew Davidson
Matthew Fuchs
Mette Hedin
Mudita Jain
Jari Koistinen
Chris Lloyd
Murray Maloney
Kelly Schwartzhof

Status of this document

This document represents version 2.0 of the Schema for Object-Oriented XML (SOX). It replaces the previous version of the SOX language specification and represents the current, implemented version of the language.

Comments on this document should be sent to brad.snyder@commerceone.com.

---

1 Abstract

This document describes SOX 2.0, the second version of the Schema for Object-Oriented XML. SOX is a schema language (or metagrammar) for defining the syntactic structure and partial semantics of XML document types. As such, SOX is an alternative to XML DTDs and can be used to define the same class of document types (with the exception of external parsed entities). However, SOX extends the language of DTDs by supporting:

1. An extensive (and extensible) set of datatypes
2. Inheritance among element types
3. Namespaces
4. Polymorphic content
5. Embedded documentation
6. Features to enable robust distributed schema management.

All of these features are supported with strong type-checking and validation. A SOX schema is also a valid XML instance according to the SOX DTD, enabling the application of XML content management tools to schema management.

SOX was initially developed to support the development of large-scale, distributed electronic commerce applications, but it is a general-purpose schema language for the whole range of applications of markup. As compared to XML DTDs, SOX dramatically decreases the complexity of supporting interoperation among heterogenous applications by facilitating software mapping of XML data structures, expressing domain abstractions and common relationships directly and explicitly, enabling reuse at the document design and the application programming levels, and supporting the generation of common application components.

Although SOX 2.0 retains many of the features of SOX 1.0, it represents an additional year of actual implementation experience. Commerce One has a working implementation of this language and will be releasing products based on it. Our goals in releasing this second version are two-fold:

1. Ensure that the public record accurately reflects the evolution of SOX. Members of the community wishing to understand SOX, build tools using SOX, or interoperate with SOX applications, need access to the current version of the language.
2. Expose some of the fruits of our implementation experience to the general community, particularly to assist in the ongoing work at the W3C to develop an official XML Schema language.

From the markup world, the SOX proposal is informed by the XML 1.0 specification as well as the XML-Data submission and Document Content Description submission. However, many of SOX's requirements come from the distributed computing world and the development of SOX has been heavily influenced by the Java programming language.

2 Soxtype processing instruction

As a SOX document is not defined by a DTD, the native XML 1.0 Doctype declaration is not appropriate for a document instance to declare SOX Schema information (note that this may not continue to be true once the W3C Schema WG defines an official mechanism). Therefore we have created the *soxtype declaration*, which mimics the XML doctype declaration, but using a processing instruction. The soxtype declaration must be the first statement in a document following the optional XML declaration and declares that the default namespace of this document is that of the given SOX Schema, just as a doctype declaration declares that an XML 1.0 document conforms to the given Document Type Declaration (DTD). The soxtype declaration is not currently intended to coexist in the same document with a doctype declaration, but there is no particular restriction prohibiting this.

The soxtype declaration includes three parts:

3. The PI target, which is *soxtype*.
4. The uri specifying a schema to be the default namespace of the instance.

An example declaration would look like:

```
<?soxtype urn:x-commerceone:document:com:commerceone:schema1.sox$1.0?>
```

where the schema definition is located through resolving the urn **urn:x-commerceone:document:com:commerceone:schema1:schema1.sox$1.0.** The part following the "$" gives the version number.

A small sample document would look like:

```
<?soxtype urn:x-commerceone:document:com:commerceone:schema1.sox$1.0?>
<Root>
  <Body/>
</Root>
```

The root element of the instance is not required to belong to the default namespace. This will be further elaborated in the discussion of the **import** PI below.

---

3 Import processing instruction

With the arrival of namespaces and polymorphism, it is no longer necessarily possible to indicate a single schema to which the entire instance conforms. Nevertheless, we require a mechanism to indicate a set of schemata containing definitions for all the element and datatypes appearing in an instance. This function is handled by the **import** processing instruction. The **import** PI contains one argument, an absolute URI indicating a schema.

An example import PI would look like:

```
<?import urn:x-commerceone:document:com:commerceone:schema1.sox$1.0?>
```

where the schema definition is located through resolving the urn **urn:x-commerceone:document:com:commerceone:schema1.sox$1.0.** The part following the "$" gives the version number.

A small sample document would look like:

```
<?soxtype urn:x-commerceone:document:com:commerceone:schema1.sox$1.0?>
<?import urn:x-commerceone:document:com:commerceone:schema2.sox$1.0?>
<Root
  <s2:Body
     xmlns:s2="urn:x-commerceone:document:com:commerceone:schema2.sox$1.0"/
>
</Root>
```

In the following, the namespace attributes on **Root** overrides the default namespace declared in the soxtype PI, and **Body** is assumed to also be in **schema2.sox**.

```
<?soxtype urn:x-commerceone:document:com:commerceone:schema1.sox$1.0?>
<?import urn:x-commerceone:document:com:commerceone:schema2.sox$1.0?>
<Root xmlns="urn:x-commerceone:document:com:commerceone:schema2.sox$.0">
  <Body/>
</Root>
```

3.1 Validity constraint

It is not necessary that for every element type appearing in the instance there be a corresponding **`import`** or **`soxtype`** processing instruction. Nevertheless, all schema information must be available before processing of the root element begins. We can define a transitive closure property over schemata to accomplish this.

We require each schema declared in a **`soxtype`** or **`import`** PI to be processed. Furthermore, if any definition in a schema being processed refers to a definition (**`elementtype`** or **`datatype`**) in another schema, that other schema must be processed. The set of imports must be sufficient so that starting with the **`soxtype`** and the **`imports`**, and processing all schemata transitively referenced by them, all **`elementtypes`** and **`datatypes`** found in the instance will have been processed.

---

4 Schema definitions

The definition of an XML schema is performed with the **`schema`** element. The corresponding DTD fragment is:

```
<!ELEMENT schema (intro?,
                 (datatype | elementtype
                  | join
                  | comment | namespace
                  )*) >
<!ATTLIST schema
                  prefix     NMTOKEN   #IMPLIED
                  uri        CDATA     #REQUIRED
                  soxlang-version  NMTOKEN  #FIXED "V2.0">
```

The following is a minimal valid instance of a schema:

```
<schema uri = "urn:x-commerceone:sampleSchema"/>
```

However it is unlikely there will be many schemata containing no definitions at all.

A **`schema`** consists of any number of definitions of datatypes or element types (both of which will be explained subsequently). In other words, a schema is a set of definitions, not the set of files, database entries, etc., we use to store a representation of these definitions: the physical storage mechanism we use may change frequently, without the set of definitions being affected at all. A schema may start out as a single file, then be split among

several files (which would currently be linked using the **`join`** mechanism described below), before being stored in a database. But despite being stored in three different ways, the set of definitions remains the same, and it is that set which comprises the schema.

Each schema has a unique name to identify it. This name is a URI, given in the **`uri`** attribute of the **`schema`** element in a fragment. In essence, it proclaims the set of definitions in this schema element to be a subset of the definitions forming the schema identified by the uri attribute.

The **`join`** construct allows definitions from externally defined fragments belonging to the same schema to be pulled in.

There are three main classes of symbols created during the construction of a Schema: **`elementtype`** names, **`datatype`** names, and **`namespace`** prefixes. In the current version of the language, both **`elementtype`** names and **`datatype`** names are maintained in a single set; it is illegal to use the same name to represent both an element type and a datatype. Prefixes for namespaces, however, are kept separately. It is entirely legal, although potentially confusing, to use the same name for a **`namespace`** prefix as for an **`elementtype`** or **`datatype`**.

NOTE It is our intention to separate the **`datatype`** and **`elementtype`** namespaces in a future version of SOX. When we do so, this will be entirely backwards compatible. The expressive power of SOX, however, is unchanged whether the namespaces are separated or coallesced.

The **`prefix`** attribute specifies a prefix available for referencing names created in this schema. As the current schema is the default namespace for dereferencing names, this is not strictly necessary, however it can be useful when definitions from several namespaces are mixed in close proximity.

The **`uri`** attribute provides a URI establishing the namespace of this schema fragment. This attribute has become required to prevent accidental name capture through the join mechanism described below. This must be an absolute URI.

The **`namespace`** element is considered a *declaration*, not a definition. The namespace being declared is defined elsewhere (probably in a Schema file)

N.B. SOX does not deal with a number of linking issues related to the organization of schemata into multiple files.

The **`intro`** element is available to provide an introduction to the schema as a whole. It consists of a number of HTML elements. The exact allowable contents of **`intro`** is available in the **`htmltext.ent`** file reproduced at the end of this document.

4.1 Validity issues

A Schema is about types. Types are both defined and referenced. Referencing is done using names. Once a type (or namespace) is given a name (or prefix) in a fragment, that name that name is *bound* in that fragment. Names that are only referenced are *free* in that fragment.

A Schema fragment is processed in an *environment.* That environment extends beyond the fragment to include other schemata and the SOX definition itself. This document defines how the environment of a fragment is defined, but does not discuss how it is physically constructed.

A Schema fragment cannot be successfully processed unless all the names that are free in it are bound somewhere in its environment. It is an error for a name to be bound twice in a fragment's environment.

A fragment specifies its environment by providing bindings for all the free names it contains. There are four pieces to its environment that a fragment must specify:

1. The fragment itself. This includes those names defined in the fragment. The fragment specifies this by its own existence.
2. The rest of the Schema this fragment is defined in. This is specified by the **`uri`** attribute on the **`schema`** element.
3. Those schemata referenced directly or indirectly from this fragment, as explained below. These are specified in the **`namespace`** declaration element.
4. The SOX definition. This includes all names defined in this specification or its successors. The version is specified by the **`version`** attribute of the **`schema`** element.

Names are either qualified or unqualified. Unqualified names must be defined either in the current fragment, in the rest of the schema, or in the SOX definition. It can only be defined in one of the three. Qualified names must be defined in the schema declared for that name with a namespace declaration. For each qualified name in a fragment there must be a corresponding namespace declaration in the same fragment. The **`schema`** element itself is considered a namespace declaration for the current namespace, so qualified names using the value in the **`prefix`** attribute of the **`schema`** element must resolve to the current schema (although not necessarily in the same fragment). In this version of the language it is an error to create a definition whose local name is ~~either~~ one of the intrinsic datatypes.

---

5 Namespace declarations

```
<!ELEMENT namespace (explain?)>
<!ATTLIST namespace
                prefix        NMTOKEN   #REQUIRED
                namespace     CDATA     #REQUIRED >
```

All of the names defined in a single Schema belong to the same namespace, and can be used without qualifier. Schemata, however, frequently need to refer to definitions in other namespaces. A namespace declaration allows access to definitions in the referenced schema when appearing with an appropriate prefix attribute. A namespace declaration is scoped to the current schema fragment only. Namespace declarations made in one schema fragment are not visible in other fragments belonging to the same schema, even when referenced through a **`join`**.

NOTE We will use a *prefix* attribute for the prefix, instead of using colonized names, in accordance with the XSDL spec. The prefix attribute shows up on various elements, including **`attdef`**, **`scalar`**, and **`extends`**, all of which include a reference to a definition.

XML requires the use of qualified names to make such references in document instances. A qualified name in an instance consists of two parts separated by a colon:

- A prefix, which is the value of a prefix attribute specified in a namespace declaration.
- A local name, which corresponds to some name defined in the target namespace.

SOX implements qualified names through the use of two attributes, one for the name, and one for the prefix. Qualified names can be used wherever a reference to a definition is allowed - a schema cannot define a name in another schema, but it can extend an elementtype from another schema, or use a datatype from another Schema.

For example, the following fragment declares the urn:foo namespace and associates it with prefix bar:

```
<namespace prefix = “bar” namespace = “urn:foo”/>
```

If we later need to include a foobar element from the urn:foo namespace in an element type that would be done using the following fragment:

```
<elementtype name = “et”>
  <model>
    <element prefix = “bar” type = “foobar” name = “whatever”/>
    <element prefix = “bar” type = “foobar”/>
  </model>
</elementtype>
```

A valid instance of this would be:

```
<et><whatever><bar:foobar xmlns:bar=”urn:foo”/></whatever>
    <bar:foobar xmlns:bar=”urn:foo”/></et>
```

5.1 Validity issues

Each prefix must be unique within a schema fragment. While it is not a fatal error to declare a non-existent namespace, it is a fatal error to reference an element in a non-existent namespace, or to reference a non-existent element in an existing namespace. Both of these are semantic errors. It is a fatal runtime error if the processor is

unable to retrieve a definition during processing. A processor should distinguish among these cases. This specification does not address the issue of how to retrieve definitions.

The value of the namespace attribute of the namespace declaration must be the URI of a schema, as described above. In other words, it must be the same as the value appearing in the uri attribute of the schema element of files which define that schema.

---

6 Explain element

The **explain** element exists inside several different SOX constructs. It provides a hook for including documentation within a schema and exploits commonly known HTML constructs.

```
<!ELEMENT explain  (title?, synopsis?, (%html.block;)+) >
```

The **title** and **html.block** elements are common HTML constructs whose exact definitions are specified in the **htmltext.ent** file included at the end of this document. The **synopsis** is used to give a purpose or synopsis to the thing being **explain**ed. It is a single paragraph of text.

6.1 Validity constraint

Because it is HTML embedded in XML, all the HTML constructs must be used in a well-formed manner. It is common to take advantage of SGML tag minimization in writing HTML documents, but that would result in well-formedness errors in a SOX schema.

---

7 Element type definitions

In XML Schema documents, element type definitions reproduce the expressiveness of XML element type declarations using explicit element and attribute markup. An element type may be defined by using the **element-type** element with the required **name** attribute, and a subordinate **model**or **extends** element (both of which will be described subsequently).

The corresponding DTD fragment is:

```
<!ELEMENT elementtype
               (explain?, (extends | ((empty|model), (attdef)*)))>
<!ATTLIST elementtype
                name       NMTOKEN   #REQUIRED >
```

The following example defines an element type of name *inline*:

```
<elementtype name="inline">
  <explain>
    <synopsis>This defines the <em>inline</em> element</synopsis>
  </explain>
  <model>
      <string/>
  </model>
</elementtype>
```

A mechanism for attaching attributes to an element type is described later.

7.0.1 Valid instance

A valid instance for this fragment would be:

```
<inline>This is a string</inline>
```

7.1 Element type name

The name of an element type may be any valid unqualified XML element type name corresponding to the **`Name`** production in the XML 1.0 language definition. The name must be unique among the names of element types and datatypes defined in the current Schema, which includes the current document or other documents belonging to the same Schema processed through the **`join`** mechanism or other resolution mechanism.

An element type may be referenced by the **`element`** and **`extends`** elements.

It is a fatal error to re-assign an element name, or to reference an element type which is not defined.

The value of the **`name`** attribute must be unique across all **`elementtype`** names defined in this schema.

7.2 Content model

The content model of an element type defines the structure and composition of an element of that type in an XML instance. The definition of a content model in XML Schema documents extends the expressiveness of XML DTDs by providing greater specificity of the minimum and maximum number of times some content model atom may be repeated. This allows an XML Schema designer with more precise control than is offered by XML's *, ? and + occurrence indicators.

The DTD fragment corresponding to an content model definition is:

```
<!ELEMENT model
               (string|element|choice|sequence)>
```

7.2.1 Empty content specification

An **empty** atom is used to indicate that an element may not contain any content, as in the case of the **BR** element below:

```
<elementtype name="BR">
    <empty/>
</elementtype>
```

In order to properly support extensibility (explained below) an **empty** content model is considered to be an empty **sequence**.

Valid instances are:

```
<BR/>
```

or:

```
<BR></BR>
```

7.2.2 String content model atom

The **string** atom indicates that a content model is simply string content and is an evolution of **#PCDATA**. It can be used as in the example above. In addition, the string value may be constrained to be of a particular datatype defined by the optional **datatype** attribute (and **prefix**, if the datatype is in another schema) as can be seen in the DTD fragment:

```
<!ELEMENT string
                EMPTY>
<!ATTLIST string
                   prefix     NMTOKEN   #IMPLIED
                 datatype   NMTOKEN "string" >
```

Any element type with **string** in its content model is considered to be a choice group with an **occurs** value of "*". This is consistent with the XML 1.0 spec which requires that any content model containing **#PCDATA** be in a choice with a Kleene star. It is unclear if this restriction will be maintained in the Schema world.

In the example below, the *size* element type's content model is string content constrained to be an **int**:

```
<elementtype name="size">
    <model>
        <string datatype="int" />
    </model>
</elementtype>
```

A valid example of this would be:

```
<size>12345</size>
```

However the following would not be valid:

```
<size>12r34</size>
```

7.2.3 Element content model atom

A content model may also comprise zero or more repetitions of another element. The DTD fragment for this definition is:

```
<!ELEMENT element
                EMPTY>
<!ATTLIST element
                  prefix     NMTOKEN   #IMPLIED
                type       NMTOKEN   #REQUIRED
                name       NMTOKEN   #IMPLIED
                occurs     CDATA     #IMPLIED >
```

The defined **element** is an instance of either a previously defined datatype or element type, which is refered to by the required**type** attribute. As before, it is a fatal error to reference a datatype or element type that is not defined.

For purposes of extensibility, a content model with just one **element** is considered a **sequence** of length one.

The **name** attribute may be used to assign a name to the defined **element** when it appears in an instance. As datatypes are not also element types, the **name** attribute must have a value when **type** references a datatype. When **name** is specified, this shows up as an additional element wrapping an element of the referenced type.

The following fragment demonstrates the use of **name**. The type **int** refers to the built-in integer datatype:

```
<elementtype name = “paragraph”>
  <model>
    <string/>
  </model>
</elementtype>

<elementtype name = “block”>
  <model>
    <sequence>
       <element name = “p” type = “paragraph”/>
       <element name = “position” type = “int”/>
    </sequence>
  </model>
</elementtype>
```

A valid instance would look like this:

```
<block><p><paragraph>this is the paragraph</paragraph></p><position>12345</
position></block>
```

The following would not be valid:

```
<block><paragraph>you must use the name</paragraph><int>1</int></block>
```

The **occurs** attribute indicates the number of repetitions of the instanced **element**. It can take on the values of:

- one of the Kleene operators, "*****" (0 or more repetitions), "**?**" (0 or 1 repetitions) or "**+**" (1 or more repetitions)
- a value of the form "**N1,N2**" (a value between a minimum of **N1** and a maximum of **N2** occurrences) where **N1** and **N2** are non-negative integers (N1 <= N2)
- a value of the form "N1,*" where N1 is a non-negative integer. This indicates at least N1 occurrences but no upper maximum

We will call an occurs where N1 is not equal to N2 an *indefinite occurs*. The degenerate case of "0,0" is allowed and means exactly 0 repetitions, which is treated the same as if the declaration did not occur.

NOTE Values of the form "**N1,N2**" and "**N1,***" are not currently supported. They will be treated as an **occurs** of "**+**" if **N1** is greater than 0, or as a "*****" otherwise.

In the following example, the definition of the content model for a **list** element type specifies that it contains a minimum of 2 and a maximum of 9 **item** elements.

```
<elementtype name="list">
    <model>
        <element type = "item" occurs="2,9"/>
    </model>
</elementtype>
```

A valid instance of the above would be:

```
<list><item/><item/></list>
```

7.2.4 Choice content model atom

The **choice** atom defines a content model to comprise one of a set of choices of **element**, **choice** or **sequence** content models. The relevant DTD fragment is:

```
<!ELEMENT choice
```

```
               ((element|choice|sequence),
                (element|choice|sequence)+) >
<!ATTLIST choice
                name       NMTOKEN   #IMPLIED
                occurs     CDATA     #IMPLIED >
```

As with **element**, the **occurs** attribute specifies the number of repetitions, and it can take the same values as defined earlier.

In the following example, the **dl** element type's content model specifies that either a single **dt** or a single **dd** element is allowed.

```
<elementtype name="dl">
    <model>
        <choice>
            <element type="dt"/>
            <element type="dd"/>
        </choice>
    </model>
</elementtype>
```

A valid instance would be:

```
<dl><dt/></dl>
```

or

```
<dl><dd/></dl>
```

but not:

```
<dl><dd/><dt/></dl>
```

7.2.5 Sequence content model atom

The sequence atom defines a content model to consist of the specified **element**, **choice** or **sequence** content models appended together in the order specified. The relevant DTD fragment is:

```
<!ELEMENT sequence
               ((element|choice|sequence),
                (element|choice|sequence)+) >
<!ATTLIST sequence
                name       NMTOKEN   #IMPLIED
                occurs     CDATA     #IMPLIED >
```

As before the **occurs** attribute specifies the number of repetitions of the entire **sequence**, and it can take the same values as defined earlier.

In the following example, the **dl** element type's content model specifies that a single **dt** followed by a single **dd** is allowed.

```
<elementtype name="dl">
    <model>
        <sequence>
            <element type="dt"/>
            <element type="dd"/>
        </sequence>
    </model>
</elementtype>
```

A valid instance would be:

```
<dl><dt/><dd/></dl>
```

7.2.6 Combining content model atoms

The various content model atoms defined above may be combined to allow the definition of complex content models.

For example, the **dl** element type's content model below specifies that a **dh** is followed by two or more **dt** or **dd** elements.

```
<elementtype name="dl">
    <model>
        <sequence>
            <element type="dh"/>
            <choice occurs="2,*">
                <element type="dt"/>
                <element type="dd"/>
            </choice>
        </sequence>
    </model>
</elementtype>
```

A valid instance would be:

```
<dl><dh/><dt/><dd/><dd/><dt/></dl>
```

7.3 Validity Issues

The **occurs** attribute may not occur on the outermost **sequence** or **choice** in an **elementtype** definition. This is the **sequence** or **choice** immediately contained within **model**. The outermost **sequence** or **choice** within the **model** must occur exactly once.

The value of the name attribute (if any) given to an element, choice, or sequence, must be unique within the innermost enclosing construct. For example, the following is legal:

```
<sequence>
  <element name = “a” type = “string”/>
  <element name = “b” type = “int”/>
</sequence>
```

while the following is not:

```
<sequence>
  <element name = “a” type = “string”/>
  <element name = “a” type = “int”/>
</sequence>
```

Likewise, the following is valid:

```
<sequence>
  <element name = “a” type = “string”/>
  <sequence name = “c”>
    <element name = “a” type = “string”/>
    <element name = “c“ type = “int”/>
  </sequence>
  <element name = “b” type = “string”/>
</sequence>
```

---

8 Elementtype inheritance

As in object-oriented inheritance, an element may specialize (or subclass) from another element by inheriting its structure and then adding on to its content model. Inheritance is specified using the **extends** construct, and the relevant DTD fragment is:

```
<!ELEMENT extends (append?, attdef*)>
<!ATTLIST extends
               prefix    NMTOKEN     #IMPLIED
                 type       NMTOKEN   #REQUIRED >
<!ELEMENT append
                   (element|choice|sequence)+>
```

The **type** attribute refers to the base element type that is being extended, and the structure of the **append** atom has the same contents as that of **model**. The base type must be already defined. The contents of the **append** element are added to the end of the parent's content model (the outermost **sequence**). Note that the **append** element has been made optional. This makes it possible to declare semantically distinct element types whose structures remain the same as that of some common parent.

In the following example, the element type **datednote** has the content model of the element type it extends (**note**) with an appended date (using the intrinsic date datatype). The **multinote** element type polymorphyically can use either..

```
<elementtype name="note">
    <model><element type = “p” occurs = “+”></model>
</elementtype>

<elementtype name="datednote">
    <extends type="note">
        <append>
            <element type="date" name = “adate”>
            <element type=”time” name = “atime” occurs = “?”/>
        </append>
    </extends>
</elementtype>

<elementtype name = “multinote”>
  <element type = “note” occurs = “+”/>
</elementtype>
```

The following is a valid instance of **multinote:**

```
<multinote><note><p>This is a plain note</p></note><datednote><p>This is a
dated note</p><adate>19981209</adate><atime>10:23:32</atime></datednote></
multinote>
```

8.1 Interaction of namespaces and extension in document instances

SOX permits an elementtype in one namespace to extend an elementtype in another Schema. In order to work correctly with the current draft of the W3C Namespace Recommendation, each element in an instance belongs to the namespace in which it was declared. In the above example, if **note** were declared in schema **Foo.sox** and both **datednote** and **multinote** in **Bar.sox**, then the following would be an appropriately prefixed version of the above example:

```
<bar:multinote xmlns:foo="Foo.sox" xmlns:bar="Bar.sox">
<foo:note><foo:p>This is a plain note</foo:p></foo:note>
<bar:datednote><foo:p>This is a dated note</foo:p>
<bar:adate>19981209</bar:adate><bar:atime>10:23:32</bar:atime>
</bar:datednote></bar:multinote>
```

A prefered solution would be to consider local names as being in the namespace of the elementtype they are declared in (or its subtypes) and therefore need not be prefixed. This would be similar to the treatment of attributes.

8.2 Validity issues

In order to support extensibility, each elementtype must be either a choice or a sequence. By default:

1. an **`empty`** elementtype is considered to be an empty sequence.
2. An elementtype with a model of just **`element`** is considered to be a sequence.
3. An elementtype with a model of **`string`** is considered to be a choice.

There are some constraints placed on the form of content models to avoid ambiguous models and interminable documents. These are an extension of similar restriction in XML 1.0.

For any possible path in the parse tree from an element to a descendant of itself there must be an intervening optional node (?, *, or indefinite occurs) or intevening choice node with at least two children. This assures that infinite documents are not required.

For any optional node in the parse tree (?, *, +, indefinite occurs, or choice), none of the descendants of elements in its first set may be in its follow set.

These conditions continue to hold when extending an elementtype.

Due to the desire to maintain substitutability of extended elementtypes with their base type, SOX 2.0 does not allow the extension of elementtypes whose content model is choice. In order to maintain substitutability, any element extending a choice would need to be a subtype of something already in the choice, so it would already be valid in the parent type.

When extending a sequence with a occurs of indefinite extent, the resulting content model must be checked for ambiguity with itself. In other words, if the parent model was *x**, where *x* is some content model, then the content model *xx* (*x* followed immediately by *x*) must have been unambiguous. If the child model is now *x+e* (i.e., *x* plus some additional element), then the content model *(x+e)(x+e)* must still be unambiguous.

Within an **`element,`** the value of the **`name`** attribute is scoped to the surrounding **`elementtype.`** It neither creates nor prevents the creation of a top level definition using the same name. However, in order to maintain

backwards compatibility with XML 1.0, the binding between **name** and **type** must be global. In other words, once a name has been used with a particular type, it cannot be used with another type. We expect this restriction to be relaxed in a future version.

---

9 Datatypes

Along the lines of being able to define element types, XML Schema provides the means to define and refer to datatypes. XML Schema defines a set of intrinsic datatypes, listed below, from which user-defined datatypes may be derived. Some of these have been refered to elsewhere in this document. It also (currently) provides the **datatype** element for actually defining new datatypes. The appropriate DTD fragment is:

```
<!ELEMENT datatype
                (explain?, (enumeration|scalar|varchar)) >
<!ATTLIST datatype
                 name        NMTOKEN   #REQUIRED >
```

The value of the name attribute specifies the name of the new datatype. This must be unique across all the datatypes defined for this schema. The three operators, **enumeration**, **scalar**, and **varchar** (all further described below), each derive a new datatype from an existing datatype. The existing datatype can be either one of the intrisic types, or some other user-defined datatype, whether in this schema or another. Both **scalar** and **varchar** have some restrictions on which datatypes they can extend, as described below.

9.1 Intrinsic datatypes

The intrinsic datatypes define the domains of the atomic data units in XML Schema documents. The list below includes those datatypes defined intrinsic to this version of the Schema language.

boolean

A value of either “true” or “false”

string

A sequence of characters.

Format: X*

URI

A sequence of characters forming a valid URI:

number

An infinite precision number.

Format: [+-][0-9]*[‘.’[0-9]*]?

float

A single precision floating point number in the range -3.40282347 * 10E38 to 3.40282347 * 10E38

Format: [+-][0-9]*[’.’[0-9]*]?

double

A double precision floating point number in the range -1.17549435 * 10E308 to 1.17549435 * 10E308

Format: [+-][0-9]*[’.’?[0-9]*]?

int

An integer in the range -2,147,483,648 to 2,147,483,647

Format: [+-]?[0-9]*

long

An integer in the range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Format: [+-]?[0-9]*

byte

An integer in the range -128 to 127

Format: [+-]?[0-9]*

ID

An ID is a name which must be unique within the instance and names the node for which it is defined. The value of an ID cannot be reused within the instance.

Format: NMTOKEN production of XML 1.0 specification.

IDREF

An IDREF must contain the value of an ID string declared elsewhere in the document.

IDREFS

A list of whitespace delimited IDREF values.

NMTOKEN

A string corresponding to the values of the NMTOKEN production of the XML 1.0 specification.

NMTOKENS

A list of whitespace delimited NMTOKEN values.

date

A date including month, day, and year

Format: YYYYMMDD

time

A time accurate to the nearest second with optional offset from GMT

Format: HH:MM:SS[[+-]HH:MM]?

datetime

A combination Date and Time. Note the presence of a "T" character between the date and time portions, and the use of colons to separate hours, minutes, and seconds. These are as per ISO 8601.

Format: YYYYMMDD**T**HH**:**MM**:**SS[[+-]HH:MM]? (the first MM is Months, the other two are minutes)

9.2 Enumeration datatypes

XML Schema documents provide a mechanism for defining enumerations to constrain attribute or element string content. An **`enumeration`** datatype is a finite set of values enumerated by the **`option`** elements inside the **`enumeration`** element.

```
<!ELEMENT enumeration
               (explain?, option)+ >
<!ATTLIST enumeration
                prefix        NMTOKEN #IMPLIED
                datatype      NMTOKEN   #REQUIRED >

<!ELEMENT option (#PCDATA)* >
```

The **`datatype`** attribute of the enumeration specifies the instrinsic (see list above) or user-defined datatype (i.e., other **`enumeration`**, **`varchar`**, or **`scalar`**) being refined. If the datatype is not defined in this Schema, then the prefix of the appropriate schema must be specified.

Each **`option`** has a value representing a valid value for the **`datatype`** being extended.

The following example demonstrates the definition and use of an enumeration datatype:

```
<datatype name="colortype">
    <enumeration datatype="NMTOKEN">
        <option>Red</option>
        <option>Blue</option>
        <option>Green</option>
```

```
    </enumeration>
</datatype>

<elementtype name="car">
    <empty/>
    <attdef name="color" datatype="colortype"><required/></attdef>
</elementtype>

<elementtype name="bus">
    <model>
        <element name="color" type="colortype">
    </model>
</elementtype>
```

The following are valid instances:

```
<car color="Red"/>

<bus><color>Blue</color></bus>
```

9.3 Scalar datatypes

Scalar datatypes are used for creating subtypes of **number**. The relevant DTD fragment is:

```
<!ELEMENT scalar EMPTY >
<!ATTLIST scalar
              prefix         NMTOKEN     #IMPLIED
              datatype       NMTOKEN   "number"
              digits         CDATA     #IMPLIED
              decimals       CDATA     #IMPLIED
              minvalue       CDATA     #IMPLIED
              maxvalue       CDATA     #IMPLIED
              minexclusive   (true | false)  "false"
              maxexclusive   (true | false)  “false" >
```

Digits specifies the maximum number of digits for the integral part of the number, **decimals** specifies the maximum number of digits for the fraction part of the number. The following constraints must hold:

- If the datatype is a subtype of **int**, **long**, or **byte**, then **decimals** must be unspecified or 0.
- **Minvalue** specifies the low end of the range of possible values. **Maxvalue** specifies the high end. **Minvalue** must be less than or equal to **maxvalue**.
- If **minvalue** (respectively, **maxvalue**) is unspecified, then it is either the minimum (resp. maximum) possible value given the specified values of digits and decimals, or it is the value specified by the parent datatype

Both **minvalue** and **maxvalue** must be specifiable with the given values for **digits** and **decimals**, or must be expressible as proper values for the parent class.

- **Minexclusive** and **maxexclusive** determine whether the minimum number, respectively largest number, in the range is included.
- The values for **digits** and **decimals** must be non-negative integers. Both must be less than or equal to the values specified (if any) for the parent class.
- The value of the datatype must be either one of the intrinsic scalar datatypes (**byte**, **long**, **int**, **float**, **double**, **number**), or of a scalar datatype derived from one of these intrinsics.

An example scalar would be:

```
<scalar digits = “4” decimals = “3” minvalue = “-9999” maxvalue = “8888”
        minexclusive=”true”/>
```

The following are valid: -9998.999, 8887.999, 0.0, 8888.

The following are invalid: -9999, 8888.001.

The **number** datatype covers all rational numbers which can be finitely specified as a decimal (i.e., it does not cover infinitely repeating decimals, such as 1/9, or irrational reals, such as π). The other intrinsic classes are designed to be easily mappable to existing datatypes in common programming languages.

9.4 Varchar types

Varchar, adapted from SQL, is for specifying string types with fixed maximum length.

```
<!ELEMENT varchar EMPTY>
<!ATTLIST varchar
                prefix NMTOKEN #IMPLIED
                datatype NMTOKEN "string"
                 maxlength CDATA #REQUIRED>
```

The value of **maxlength** must be a non-negative integer.

An example varchar use is:

```
<datatype name = “var”>
  <varchar maxlength = “4”/>
</datatype>
<elementtype name = “wrap”>
  <model>
    <string datatype = “var”/>
```

```
  </model>
</elementtype>
```

A valid instance would be:

```
<wrap>abc</wrap>
```

as would:

```
<wrap>abcd</wrap>
```

An invalid instance would be:

```
<wrap>abcde</wrap>
```

9.5 Validity issues

Float, **double**, **int**, **long**, and **byte** are all predefined scalar types. This means they can be used as base types for any new definition of a **scalar** and can be referenced as datatypes. The value of **maxlength** in varchar must be greater than or equal to 0. Note that a length of 0 implies an empty string. The datatype of a **varchar** must be either another **varchar** or one of **string**, **NMTOKEN**, **NMTOKENS**, **ID**, **IDREF**, or **IDREFS** .

The names of the intrinsic types are reserved by SOX. It is an error to define a datatype or elementtype using the name of one of the intrinsic types.

---

10 Attribute definitions

Attribute definitions in XML Schema documents may be defined as part of the element type definition. An attribute definition has a name and a type, and must include a presence element. The relevant fragment of the DTD is:

```
<!ELEMENT attdef
               (explain?, (enumeration | scalar | varchar)?,
                 (required|implied|default|fixed)?)>
<!ATTLIST attdef
               name       NMTOKEN   #REQUIRED
               prefix     NMTOKEN     #IMPLIED
               datatype   NMTOKEN   #IMPLIED>
```

The name of the attribute is defined by the value of **name**, and it may be of a certain **datatype**. It must be unique among the attributes for this elementtype. If a value is not given for the **datatype** attribute and the **attdef** does not contain an **enumeration**, **scalar**, or **varchar**, then a datatype of **string** is assumed.

An attribute value's presence in an instance may be specified as **default**, **fixed**, **required** or **implied,** as in the XML specification. If no value is given for the presence, then it defaults to **implied**. In all cases, the value of an attribute must be valid for its datatype. The DTD fragment for the presence elements is:

```
<!ELEMENT default
              (#PCDATA) >
<!ELEMENT fixed
              (#PCDATA) >
<!ELEMENT required
               EMPTY >
<!ELEMENT implied
               EMPTY >
```

A **default** presence indicates that the value of the attribute is automatically set to the default value if none is specified. In an instance, if the attribute is defined to have another value, the default is ignored.

A **fixed** presence indicates that the value of the attribute is assumed to be the fixed value if no value is specified. The attribute may also be explicitly assigned exactly this fixed value. In an instance, if the attribute is defined to have a different value, this signals a fatal error.

An empty value can be specified for **default** or **fixed.**

A **required** presence indicates that in an instance, whenever the parent element appears, this attribute must be assigned a value.

And finally, an **implied** presence indicates that this attribute is entirely optional and no default value is specified for the case where it is not present.

In XML Schema documents, unlike XML DTDs, enumerations may be specified for any attribute type. This information will be lost if an XML DTD is generated from an XML Schema document, except for attributes of type **NMTOKEN**, indicating a name token. If there is an **enumeration** specified for an attribute and also a **fixed** or **default** value, then that **fixed** or **default** value must be a member of the **enumeration**.

Thus, example attribute definitions (within the definition of an **elementtype**, of course) might be:

```
<elementtype name="car">
    <empty/>
    <attdef name = “owner” datatype = “string”/>
    <attdef name="color">
        <enumeration datatype="NMTOKEN">
            <option>Red</option>
            <option>Blue</option>
```

```
            <option>Green</option>
        </enumeration>
        <required/>
    </attdef>
</elementtype>
```

An instance corresponding to this definition would look like:

```
<car color="Blue" owner = “John Smith”/>
```

10.1 Validity constraints

It is a fatal error for the **datatype** attribute of the **attdef** element to have a value if there is an enclosed **enumeration**, **scalar**, or **varchar.**

---

11 Including schema files

An externally-defined schema file whose definitions belong to the same namespace may be pulled in and parsed with the current schema definition. This is accomplished using the **join** construct. The relevant fragment of the DTD is:

```
<!ELEMENT join
                (explain?)>
<!ATTLIST join
                 datatype   NMTOKEN   #FIXED "schema"
                 public     CDATA     #IMPLIED
                 system     CDATA     #REQUIRED>
```

The **datatype** attribute is fixed as only schemas may be included for now. The **public** attribute is the public identifier of the file as defined in the XML 1.0 specification. The **system** attribute is the URI of the file containing the schema definition to be lexically included. The entity manager must resolve this URI.

A **join**ed file is read only once by the parser. The parser determines identity between files by comparison of the values of the **system** attribute. It is not illegal for two **join** elements to reference the same URI, but one will be ignored. It is a user error to use two different URIs which ultimately map to the same file. Note that reading a fragment twice will cause an error as the schema processor will create all its definitions twice.

11.1 Validity issue

The joined file must belong to the same namespace as the joining one, as identified by the URI attribute of the root elements.

In the current implementation, both **namespace** and **join** elements use URIs to point to external files, and processing a namespace involves retrieving the physical file referenced by the **namespace** element. It is not an error for a **join** element to reference this file again, however it is a runtime error for that file to actually be retrieved a second time. In other words, no fragment for a schema is to be processed more than once.

---

12 SOX DTD

```
<!-- ********************************************************** -->
<!-- SOX DTD     -->
<!-- PUBLIC "-//Commerce One Inc.//DTD SOX 2.0//EN" -->
<!-- SYSTEM "schema.dtd" -->

<!-- Copyright:      Commerce One Inc., 1997, 1998, 1999

     Date created:   17 Dec 1997
      Date revised:     03 June 1999
     Version:        2.0
-->

<!-- ********************************************************** -->

<!-- ********************************************************** -->
<!-- Schema for Object Oriented XML
     ****************************************************** -->
<!-- ********************************************************** -->

<!ENTITY % htmltext SYSTEM "htmltext.ent">
%htmltext;

<!ELEMENT schema (intro?, (datatype | elementtype |
                           join | comment | namespace)*) >
<!ATTLIST schema
                prefix     NMTOKEN   #IMPLIED
                 uri        CDATA     #REQUIRED
                soxlang-version  NMTOKEN  #FIXED “V0.2.2”>


<!-- ********************************************************** -->
<!-- ELEMENTS  ********************************************** -->
<!-- ********************************************************** -->
```

```
<!-- An Element Type definition requires a name.
     It is defined to extend a named element,
     as an instance of a named element,
     as an EMPTY or ANY element with optional attribute definitions,
     or with a content model with optional attribute definitions. -->

<!ELEMENT elementtype
               (explain?, (extends | ((empty|model), (attdef)*)))>
<!ATTLIST elementtype
                name       NMTOKEN   #REQUIRED >

<!ELEMENT empty EMPTY >

<!-- ********************************************************** -->
<!-- MODEL  ***************************************************** -->
<!-- ********************************************************** -->

<!ELEMENT model
              (string|element|choice|sequence)>

<!ELEMENT extends (append?, attdef*)>
<!ATTLIST extends
                prefix NMTOKEN #IMPLIED
                type       NMTOKEN   #REQUIRED >

<!ELEMENT append
        (element|choice|sequence)+>

<!ELEMENT element EMPTY >
<!ATTLIST element
                prefix     NMTOKEN     #IMPLIED
                  type        NMTOKEN   #REQUIRED
                name       NMTOKEN   #IMPLIED
                 occurs     CDATA     #IMPLIED >

<!ELEMENT string EMPTY >
<!ATTLIST string
                prefix NMTOKEN #IMPLIED
                datatype   NMTOKEN "string" >

<!ELEMENT choice
              ((element|choice|sequence),
```

```
                (element|choice|sequence)+) >
<!ATTLIST choice
                name        NMTOKEN   #IMPLIED
                occurs      CDATA     #IMPLIED >

<!ELEMENT sequence
               ((element|choice|sequence),
                (element|choice|sequence)+) >
<!ATTLIST sequence
                name        NMTOKEN   #IMPLIED
                occurs      CDATA     #IMPLIED >

<!-- replacement for "include" -->
<!ELEMENT join
               (explain?)>
<!ATTLIST join
                datatype    NMTOKEN   #FIXED "schema"
                public      CDATA     #IMPLIED
                system      CDATA     #REQUIRED>

<!-- ********************************************************** -->
<!-- ATTRIBUTES  ****************************************** -->
<!-- ********************************************************** -->

<!-- An attribute definition has a name and datatype, and must have
     a presence element "required|implied|default|fixed" included.
     It may have a namespace associated with it, or inherit
     enumeration is supposed to define the domain of acceptable valu
     -->

<!ELEMENT attdef
               (explain?, (enumeration | scalar | varchar)?,
                 (required|implied|default|fixed)?)>
<!ATTLIST attdef
                name        NMTOKEN   #REQUIRED
                prefix      NMTOKEN   #IMPLIED
                datatype    NMTOKEN   #IMPLIED>

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

```
<!ELEMENT required
                EMPTY >
<!ELEMENT implied
                EMPTY >

<!-- ********************************************************** -->
<!-- DATATYPE  ********************************************** -->
<!-- ********************************************************** -->
<!ELEMENT datatype
               (explain?, (enumeration|scalar|varchar)) >
<!ATTLIST datatype
                name       NMTOKEN   #REQUIRED >

<!ELEMENT enumeration
              (explain?, option)+ >
<!ATTLIST enumeration
                prefix        NMTOKEN #IMPLIED
                datatype      NMTOKEN   #REQUIRED >

<!ELEMENT option (#PCDATA)* >

<!ELEMENT scalar EMPTY >
<!ATTLIST scalar
                prefix         NMTOKEN #IMPLIED
                datatype       NMTOKEN   "number"
                digits         CDATA     #IMPLIED
                decimals       CDATA     #IMPLIED
                minvalue       CDATA     #IMPLIED
                maxvalue       CDATA     #IMPLIED
                minexclusive   (true | false)     "false"
                maxexclusive   (true | false)     "false" >

<!ELEMENT varchar EMPTY>
<!ATTLIST varchar
                 prefix NMTOKEN #IMPLIED
                 datatype NMTOKEN "string"
                  maxlength CDATA #REQUIRED>


<!-- ********************************************************** -->
<!-- COMMENT  ********************************************** -->
<!-- ********************************************************** -->
```

```
<!ELEMENT comment
              (#PCDATA)>


<!-- Namespaces -->
<!ELEMENT namespace  (explain?) >
<!ATTLIST namespace
          prefix      NMTOKEN   #REQUIRED
          namespace   CDATA     #REQUIRED >
```

---

13 Appendix B: htmltext.ent - HTML element types for explain

```
<!-- ********************************************************** -->
<!-- HTML Text: SOX uses HTML element types for convenience.       -->
<!-- ********************************************************** -->
<!-- Copyright:     Commerce One Systems Inc., 1997, 1998
     Date created:  17 Dec 1997
     Date revised:  01 Mar 1999
     Version:       1.0 -->
<!-- ********************************************************** -->

<!ENTITY % html.nonheading  " table | p | bq | pre | ol | ul | dl" >

<!ENTITY % html.text  "#PCDATA| a | abbr | b | big | br
| cite | code | em | i | img
| q | small | span | strike | strong
| sub | sup | tt | u " >

<!ENTITY % html.heading.text  "#PCDATA| a | abbr | b | big | br | cite |
code | em
| i | img  | q | small | span | strike | strong | sub | sup | tt | u " >


<!ENTITY % html.block "h1|h2|h3|h4|h5|h6|%html.nonheading;" >

<!-- ********************************************************** -->
<!-- The intro element type is used to introduce a schema.  It contains a
     general description of the purpose and use of the schema’s document
     type or components. -->
```

```
<!ELEMENT intro     ((%html.block;)+) >

<!-- The explain element is use to document a component within a schema. -->
<!ELEMENT explain  (title?, synopsis?, (%html.block;)+) >

<!-- The title is used to give a human readable title to some type name. -->
<!ELEMENT title    (%html.text;)* >

<!-- The synopsis is used to give a purpose or synopsis to the thing being
     explained.  It is a single paragraph of text. -->
<!ELEMENT synopsis (%html.text;)* >

<!-- ************************************************************ -->
<!ELEMENT h1   (%html.heading.text;)* >
<!ELEMENT h2   (%html.heading.text;)* >
<!ELEMENT h3   (%html.heading.text;)* >
<!ELEMENT h4   (%html.heading.text;)* >
<!ELEMENT h5   (%html.heading.text;)* >
<!ELEMENT h6   (%html.heading.text;)* >

<!-- ************************************************************ -->
<!ELEMENT b    (#PCDATA)* >
<!ELEMENT br   EMPTY >
<!ELEMENT big  (#PCDATA)* >
<!ELEMENT i    (#PCDATA)* >
<!ELEMENT small
               (#PCDATA)* >
<!ELEMENT sub  (#PCDATA)* >
<!ELEMENT sup  (#PCDATA)* >
<!ELEMENT strike
               (#PCDATA)* >
<!ELEMENT tt   (#PCDATA)* >
<!ELEMENT u    (#PCDATA)* >

<!ELEMENT abbr (#PCDATA)* >
<!ELEMENT cite (#PCDATA)* >
<!ELEMENT code (#PCDATA)* >
<!ELEMENT em   (#PCDATA)* >
<!ELEMENT q    (#PCDATA)* >
<!ELEMENT span (#PCDATA)* >
<!ELEMENT strong
               (#PCDATA)* >
```

```
<!-- ********************************************************** -->
<!ELEMENT a    (%html.text;)* >
<!ATTLIST a
                name       CDATA     #IMPLIED
                href       CDATA     #IMPLIED
                title      CDATA     #IMPLIED >

<!-- ********************************************************** -->
<!ELEMENT img  (explain?) >
<!ATTLIST img
                src        CDATA     #REQUIRED
                alt        CDATA     #REQUIRED
                longdesc   CDATA     #IMPLIED
                usemap     CDATA     #IMPLIED >

<!-- ********************************************************** -->
<!ELEMENT pre  (%html.text;)* >
<!ATTLIST pre
                xml:space (preserve) #REQUIRED >

<!-- ********************************************************** -->
<!ELEMENT p    (%html.text;)* >
<!ELEMENT bq   (%html.text;)* >

<!ELEMENT ol   (lh?, li+) >
<!ELEMENT ul   (lh?, li+) >
<!ELEMENT lh   (%html.heading.text;)* >
<!ELEMENT li   (%html.text;|%html.block;)* >

<!ELEMENT dl   (dh?,(dt,dd)+) >
<!ELEMENT dh   (%html.heading.text;)* >
<!ELEMENT dt   (%html.text;|%html.block;)* >
<!ELEMENT dd   (%html.text;|%html.block;)* >

<!-- ********************************************************** -->
<!ELEMENT table
               (thead?, tbody) >
<!ATTLIST table
                cols       CDATA     #IMPLIED
                width      CDATA     #IMPLIED
                height     CDATA     #IMPLIED
```

```
                align      (left|center|right|justify) #IMPLIED
                valign     (top | middle | bottom | baseline) #IMPLIED
                vspace     CDATA     #IMPLIED
                hspace     CDATA     #IMPLIED
                cellpadding CDATA    #IMPLIED
                cellspacing CDATA    #IMPLIED
                border     CDATA     #IMPLIED
                frame     (box|void|above|below|hsides|vsides|lhs|rhs)
#IMPLIED
                rules     (none|groups|rows|cols|all) #IMPLIED >

<!ELEMENT thead
               (tr)+ >
<!ATTLIST thead
                align     (left|center|right|justify)  #IMPLIED
                valign    (top|middle|bottom|baseline) #IMPLIED >

<!ELEMENT tbody
              (tr)+ >
<!ATTLIST tbody
                align     (left|center|right|justify)  #IMPLIED
                valign    (top|middle|bottom|baseline) #IMPLIED >

<!ELEMENT tr   (th | td)+ >
<!ATTLIST tr
                align     (left|center|right|justify)        #IMPLIED
                valign    (top | middle | bottom | baseline) #IMPLIED >

<!ELEMENT th   (%html.text;|%html.block;)* >
<!ATTLIST th
                colspan    CDATA     #IMPLIED
                rowspan    CDATA     #IMPLIED
                width      CDATA     #IMPLIED
                height     CDATA     #IMPLIED
                align     (left|center|right|justify)        #IMPLIED
                valign    (top | middle | bottom | baseline) #IMPLIED >

<!ELEMENT td   (%html.text;|%html.block;)* >
<!ATTLIST td
                colspan    CDATA     #IMPLIED
                rowspan    CDATA     #IMPLIED
                width      CDATA     #IMPLIED
```

```
              height     CDATA     #IMPLIED
              align     (left|center|right|justify)        #IMPLIED
              valign    (top | middle | bottom | baseline) #IMPLIED >


<!-- ********************************************************** -->
```