SOX Tutorial

Table of Contents

1 Introduction

This document covers most of the basic and advanced features in the **SOX** language, including namespaces, inheritance and polymorphism. It contains explanations of SOX features, advice on writing schemas and instances, as well as sample schemas and instances, and is intended to serve as help in developing schemas. It provides enough information about the language to enable a reader to write SOX schemas, and instances of schemas.

The pre-requisite for this document is some familiarity with **XML**. Familiarity with SOX is optional. The current version of XML is described in the **XML 1.0 specification**. The current version of SOX is described in the **SOX 2.0 specification**. The specification documents can be found at:

http://www.w3.org/TR/REC-xml
http://www.w3.org/TR/NOTE-SOX/

This document is intended both for novice and experienced schema authors who want to learn more about the features of the language. For more detailed information about all the available SOX features, please see the SOX specification, titled *Schema for Object-Oriented XML 2.0*.

2 An Introduction to SOX Schemas and Instances

Two concepts are vital to the SOX language, a **SOX schema** and an **instance document**:

- **SOX schema** - defines structure rules in the form of **elementtype** definitions and **datatype** definitions. The schema is written in XML format and conforms to a **DTD** called "**schema.dtd**". The schema must be a valid instance of the DTD. It will have only one root element, which is of type schema. In the schema element, elementtype definitions and datatype definitions can take place.
- **Instance document** - an XML instance of a SOX schema. The instance is written in XML format and must conform to a schema or a set of schemas. It may only have one **root element**, and that element must be defined in a schema that the instance has access to. That root element, and all of its valid content, must in turn conform to their specific structure rules.

2.1 The Basics of a SOX schema

A schema provides a way to do a lot of the basic declarations possible in XML, but makes the declarations easier to both read and write. A SOX schema is expressed in XML format, which means it must be both **well-formed XML**, and **valid XML** according the schema DTD. The declarations contained in a schema enable a structured way of containing data.

A SOX schema provides the basic **XML datatypes**, as well as an added number of **SOX datatypes**. It adds the possibility of using these datatypes in element content in addition to attributes. That means that any value of intrinsic type appearing anywhere in a document can be type checked. In addition, SOX provides a means for user-defined datatypes, which extend the intrinsic, pre-defined datatypes. This means that a SOX schema writer can put additional constrains on datatypes to suit his or her needs.

2.2 The Basics of an Instance Document

An instance document must conform to structure rules set up in one or several schemas. This set of rules makes the XML instance document very useful for data storage, since content must be present exactly in the order as stated in the schema, and with all required data present, in order for the document to be **valid**. A SOX **parser** such as **cxp** can tell if the document is a valid instance or not. In addition, the data undergoes validity checks for type and constraints, saving a developer of a data-consuming application a lot of work.

3 Document Type Declarations

All SOX documents, and the XML document instances, must start with identifying information, that states what type of document will follow. This information tells a receiving application the kind of documnt it is receiving, and also enables the application to verify that the document is, in fact, one that it can process.

A SOX schema must have an **XML version tag**, as well as a **DOCTYPE declaration**.
An instance document must have a **soxtype declaration**.

3.1 XML Version and DOCTYPE Declaration

A SOX schema is described in XML format, and is a valid XML document that conforms to the schema DTD. The SOX schema should therefore always begin with the XML version tag. A DOCTYPE declaration should immediately follow the XML version. The declaration describes the type of the document that, in the case of SOX schemas, is schema. It also gives the location of the DTD to which the schema conforms, preceded by the SYSTEM keyword. These two lines look the same in all Commerce One SOX schemas:

Example 3.1:

```
<?xml version="1.0"?>
<!DOCTYPE schema SYSTEM "urn:x-commerceone:document:com:commerceone:xdk:xml:schema.dtd$1.0">
```

As reflected in this example, the current version of XML is 1.0.

3.2 The Instance soxtype Declaration

Just as the SOX schema needs a DOCTYPE declaration to specify what DTD it conforms to, the instance that conforms to a schema needs a soxtype declaration to specify what schema it is instantiating. The format for the soxtype tag is very simple:

Example 3.2:

```
<?soxtype urn:x-commerceone:document:sample:xdk:sox:sample.sox$1.0?>
```

The tag contains the keyword **soxtype**, which signals to a SOX processor, such as **cxp**, that this is an instance of a SOX schema. Next follows the URI of the schema that has been instantiated. It must correspond to the URI attribute of the schema element, as described in 4.1. See the URI working draft for more information on the format of URI's.

4 The Basic SOX Schema

Any SOX schema will, aside from an XML version and a DOCTYPE declaration, contain one root element: the **schema element**. The schema element start and end tags will be the wrapper of all other definitions in the document.

4.1 The Schema Element

The schema start tag must always contain a URI attribute. The URI attribute defines the **namespace** of the schema, and is expressed in a URI format. See the URI working draft for more information on the format of URI's.
The namespace is a unique identifier of the schema, which may also be used in determining the schema's physical location. Here is an example of how the schema element would be used:

Example 4.1:

```
<schema uri="urn:x-commerceone:document:sample:xdk:sox:sample.sox$1.0">

   . . .

</schema>
```

The ellipsis, "…", symbolizes the content of the schema element.

Commerce One Implementation specific note:
Since the Commerce One implementation uses the URI of the schema to determine the physical location of the schema, a strict formula has to be followed in constructing the URI, if the schema is to be used with Commerce One tools or software:

1. The URI must always start with "`urn:x-commerceone:document:`".

2. Determine what the root of your schema tree is. This is a location in the file hierarchy that all the schemas are located underneath. The root is represented as (ROOT) in the examples below.

3. From the root, determine the path to the file itself. That path will be the next part in the URI, with the file separator replaced with a colon ":". In example 4.1, the path to the file is "`(ROOT)/sample/xdk/sox/`". Note that an extra level is added to the physical path of the file, which is not reflected here. See step 5.

4. The next part of the URI, separated from the path by a colon, ":", is the name of the file. In the example above, the name of the file is "`sample.sox`".

5. The last part of the URI is the version. The version is separated from the rest of the URI by a "$". In example 4.1, the version is 1.0. Currently, the implementation only allows version 1.0.

   The version is represented in the physical location of the schema, as an extra directory level in the path. This directory is the last directory in the path, and is effectively the directory the schema is located in. That is, the schema must be physically located in a directory representing the version. The version is modified before being used in the path, by adding an "n" before the version, and substituting the period, ".", with an underscore, "_". Version 1.0 would therefore become "n1_0" in the physical path of the file. The version part of the path is not reflected in the part of the URI that is derived from the path, see step 3. In the example above, the schema would be located in "`(ROOT)/sample/xdk/sox/n1_0`".

The complete physical path to the file represented by the URI above would therefore be:

```
(ROOT)/sample/xdk/sox/n1_0/sample.sox
```

4.2 Elementtype Definitions

In your schema element, you can specify elementtype definitions. They are definitions of structures in your document. Any elementtype that has been specified in a schema can be instantiated as a root element in an instance of that schema. An elementtype can have five different kinds of content models: **empty**, **string**, **element**, **sequence** or **choice**.

4.2.1 Empty Content Model

An **empty content model** means that the element can contain no data or elements. It is useful when the presence of an element is optional and the element's presence is significant. See section 4.2.6: Occurrence. An empty content model can also be useful when all of the data is contained in attributes. See section 4.2.7: Attributes.

Example 4.2:

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

Note the naming convention of the element in example 4.2. It is generally a good idea to name your objects in **Camel Case** style, where the name begins with an upper case letter, and each new word in the name also begins with an upper case letter. This makes the names easy to read.

> **Commerce One Implementation specific note:**
> In addition to making the names easier to read, this naming convention also creates class names and method names with good Java style in the java beans that the SOX compiler generates from the SOX schemas.

An instance of an elementtype with an empty content model can be expressed in two ways, with an open tag and close tag with no content in between, or an empty tag. Note that in the following examples, a forward slash signifies the end, or closure, of an element. Also note the difference in use of the forward slash in the following examples:

Example 4.3:

```
<NoContent></NoContent>
```

In this element instance of `NoContent`, there is first an opening tag, immediately followed by a closing tag. The closing tag has the exact same name as the opening tag, but has a forward slash before the name, to signify that it is a closing tag. The lack of content between the tags means that this is an empty element. The same effect can be achieved with the empty tag:

Example 4.4:

```
<NoContent/>
```

The empty tag is different from the start and end tags, in that it has the forward slash indicating closure, but it is appended to the tag name instead of preceding it. The empty tag functions as both a start tag and an end tag combined. It accomplishes exactly the same thing as the two tags in example 4.3.

4.2.2 String Content Model

The **string content model** essentially means that the element can only contain data, and no elements. The string itself can be of a specific datatype, which can be intrinsic or user-defined. See Appendix A: Intrinsic Datatypes for more information on the intrinsic datatypes available. This approach is useful when you have a very simple structure, which will only contain one piece of text data.

Example 4.5:

```
<elementtype name="StringContent">
      <model>
            <string/>
      </model>
</elementtype>
```

Example 4.5 creates an element with a string content model with a datatype of string. If no datatype has been provided the type will always default to string. Some possible instances of this elementtype would be:

Example 4.6:

```
<StringContent>This is merely string content</StringContent>
```

Example 4.7:

```
<StringContent></StringContent>
```

With a string datatype, such as **string**, **NMTOKEN** or **NMTOKENS**, it is possible to have an instance like the one in example 4.7, where no value has been provided. In fact, you could even have an instance that has an empty tag. This might look odd, but it is the same thing as a string with a length of 0 characters, and is completely valid.

Here is an example of how to set the datatype of the string content model:

Example 4.8:

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

A sample instance of this elementtype could look like:

Example 4.9:

```
<IntegerStringContent>123</IntegerStringContent>
```

In this case, you can not create an instance with no content, which would not be a valid integer. In example 4.7, a string of 0 characters is perfectly valid; but in example 4.9, the datatype has been set to be integer, and an integer must always have a value. The same is true for most of the intrinsic datatypes.

4.2.3 Element Content Model

An **element content model** means that the content of the defined element is restricted to be only one type of element. Any content model that is not empty must be contained within a model tag. The type of the element can either be an **intrinsic datatype**, (see Appendix A: Intrinsic Datatypes), **user defined datatype** or another **elementtype**. If the type is a datatype, then the element must also have a name, if the type is an elementtype the name is optional:

Example 4.10:

```
<elementtype name="ElementContent">
      <model>
            <element type="string" name="StringContent"/>
      </model>
</elementtype>
```

An instance of the above elementtype could look like this:

Example 4.11:

```
<ElementContent>
      <StringContent>This is my string content</StringContent>
</ElementContent>
```

Notice the difference between example 4.11 and example 4.6. In 4.11we get two levels of tags before we reach the text data. This is because the `ElementContent` element only has element content, and it is that element content that in turn has string content. Because the content is of type string, `StringContent` could have empty content, just as the element in example 4.7.

Next let's create an element content model with content that consists of another defined elementtype. Let's use the `NoContent` element from example 4.2:

Example 4.12

```
<elementtype name="ElementContentTwo">
	<model>
		<element type="NoContent"/>
	</model>
</elementtype>
```

An instance of this elementtype could look like this:

Example 4.13:

```
<ElementContentTwo>
	<NoContent/>
</ElementContentTwo>
```

As suspected, example 4.13 looks just like example 4.11, except that the inner element can not have any content. Just like that example, we have one element being contained inside the other. As stated above, when the type is another elementtype, the name value is optional. In this case we did not rename the contained elementtype, but happens if we do?

Example 4.14:

```
<elementtype name="ElementContentThree">
	<model>
		<element type="ElementContent" name="ElementName"/>
	</model>
</elementtype>
```

The elementtype in example 4.14 differs from the elementtype in example 4.12 because, it has a name value set. How will the instances differ from each other?

Example 4.15:

```
<ElementContentThree>
	<ElementName>
		<ElementContent>Here is another string</ElementContent>
	</ElementName>
</ElementContentThree>
```

In example 4.15 we can see that if the type is another elementtype, and a different name is assigned to it, the element in question gets two surrounding tags instead of one. The outer tag is the new name that has been assigned to the elementtype, and the inner tag is the actual type of the elementtype. This approach enables polymorphism, which is discussed in section 7: Polymorphism.

4.2.4 Sequence Content Model

The **sequence content model** can express a sequence in which a number of elements should appear in the instance. The content of the sequence can be elements or **nested sequences** or **nested choices**. There always has to be at least two content items in a sequence. The simplest sequence only contains two **element** elements:

Example 4.16:

```
<elementtype name="SequenceContent">
      <model>
            <sequence>
                  <element type="string" name="StringContent"/>
                  <element type="int" name="IntegerContent"/>
            </sequence>
      </model>
</elementtype>
```

A valid instance of the elementtype in example 4.16 could look like this:

Example 4.17:

```
<SequenceContent>
      <StringContent>This is a string in a sequence</StringContent>
      <IntegerContent>123</IntegerContent>
</SequenceContent>
```

In the instance of a sequence content model, the elements have to appear exactly in the order they were declared in the elementtype sequence. They must not appear out of order. The sequence model is a way to guarantee that the data appear exactly in the order it was specified.

Note that the sequence tags are not reflected in the instance. They are only a way to describe the structure of a document.

A more complex model can for example contain nested sequences:

Example 4.18:

```
<elementtype name="SequenceContentToo">
      <model>
            <sequence>
                  <element type="boolean" name="BooleanContent"/>
                  <sequence>
                        <element type="float" name="FloatContent"/>
                        <element type="date" name="DateContent"/>
                  </sequence>
            </sequence>
      </model>
</elementtype>
```

Example 4.18 introduces three new intrinsic datatypes, boolean, float and date. Refer to Appendix A: Intrinsic Datatypes for more information on these and other available intrinsic types. A valid instance of the elementtype in example 4.18 would look like this:

Example 4.19:

```
<SequenceContentToo>
      <BooleanContent>true</BooleanContent>
      <FloatContent>123.123</FloatContent>
      <DateContent>19990101</DateContent>
</SequenceContentToo>
```

Again, note that the sequence tags are not reflected in the instance, not even the nested sequence. This does not mean they are not useful; in fact, the ability to nest sequences and choices can be extremely useful for creating very precise complex content rules.

4.2.5 Choice Content Model

A **choice content model** is similar to the sequence in that it lists a number of elements, sequences or choices. Instead of describing a structure, they outline what options are allowed in the instance. In the actual instance, only one of the specified options is selected. As with a sequence, the choice must contain at least two items that can be **elements**, **nested sequences** or **nested choices**. Again, the simplest model is one with just two elements:

Example 4.20:

```
<elementtype name="ChoiceContent">
      <model>
            <choice>
                  <element type="string" name="StringContent"/>
                  <element type="int" name="IntegerContent"/>
            </choice>
      </model>
</elementtype>
```

One valid instance of the elementtype in example 4.20 could be:

Example 4.21:

```
<ChoiceContent>
      <StringContent>This is a string in a choice</StringContent>
</ChoiceContent>
```

In the instance in example 4.21, the `StringContent` element has been chosen. Since only one element can be chosen, it would not be valid to have more than one element from the choice. Now let's try choosing the `IntegerContent` element instead:

Example 4.22:

```
<ChoiceContent>
      <IntegerContent>123</IntegerContent>
</ChoiceContent>
```

Next let's try and make a more complex choice content model, by using a nested sequence as one of the options:

Example 4.23:

```
<elementtype name="ChoiceContentToo">
      <model>
            <choice>
                  <element type="boolean" name="BooleanContent"/>
                  <sequence>
                        <element type="float" name="FloatContent"/>
                        <element type="date" name="DateContent"/>
                  </sequence>
            </choice>
      </model>
</elementtype>
```

In example 4.23, we have the option between choosing either the `BooleanContent` element, or the sequence containing the `FloatContent` and `DateContent` elements. First, let's try the simpler instance case:

Example 4.24:

```
<ChoiceContentToo>
      <BooleanContent>true</BooleanContent>
</ChoiceContentToo>
```

Example 4.24 is just as straightforward as in example 4.21 and 4.22. Now let's try the other instance case:

Example 4.25:

```
<ChoiceContentToo>
      <FloatContent>123.123</FloatContent>
      <DateContent>19990101</DateContent>
</ChoiceContentToo>
```

Note that in the case of example 4.25, both elements in the sequence have been selected. That is because the sequence they were contained in was the selected option, not the elements themselves. Only the outermost structures contained in the choice can be selected when creating an instance. `FloatContent` by itself would not have been a valid choice since `DateContent` would be missing from the selected sequence.

4.2.6 Occurs

Elements, nested sequences or nested choices can have occurrence specifications. String elements may not have an occurrence, because they do not define any named tags. Therefore there would be no way to tell where one string ends and another starts. This does not apply to elements of type string, which have an enclosing tag and can have multiple occurrences, just as any other element.

Another disallowed case is an outermost sequence or choice, that is, a sequence or choice contained directly in the model tag. These content models may not have an occurrence at the present time.
An occurrence specification can specify that a certain object can be optional, and/or can be allowed more than once. There are four different ways of specifying an occurrence: ?, +, * or N,M. The default occurrence is 1. That is, if no occurrence value has been specified, as has been the case with all of our examples so far, then the object must be present, and may only occur once.

4.2.6.1 ? Occurrence

An occurs value of "**?**" specifies that an element, nested sequence or nested choice is optional, and that it may or may not appear once. First let's try the simplest case, that of an **element content model** (see section 4.2.3):

Example 4.26:

```
<elementtype name="OptionalContent">
      <model>
            <element type="string" name="StringContent" occurs="?"/>
      </model>
</elementtype>
```

In example 4.26 we specify that the element `StringContent` may or may not appear inside the element `OptionalContent`. First, let's try with the element present:

Example 4.27:

```
<OptionalContent>
      <StringContent>This element is optional</StringContent>
</OptionalContent >
```

Example 4.28:

```
<OptionalContent>
</OptionalContent>
```

What we end up with in example 4.28 is an empty element. In fact, we can use an empty tag, and that is legal.

Next, let's try a slightly more complex elementtype:

Example 4.29:

```
<elementtype name="OptionalSequenceContent">
      <model>
            <sequence>
                  <element type="boolean" name="BooleanContent"/>
                  <sequence occurs="?">
                        <element type="float" name="FloatContent"/>
                        <element type="date" name="DateContent"/>
                  </sequence>
            </sequence>
      </model>
</elementtype>
```

In example 4.29, the `BooleanContent` element is required, but the sequence that follows it is optional. First, let's try with all elements present:

Example 4.30:

```
<OptionalSequenceContent>
      <BooleanContent>
      <FloatContent>123.321</FloatContent>
      <DateContent>19991231</DateContent>
</OptionalSequenceContent>
```

Next, let's omit the optional sequence:

Example 4.31:

```
<OptionalSequenceContent>
      <BooleanContent>
</OptionalSequenceContent>
```

4.2.6.2 + Occurrence

An occurs value of "+" specifies that an element, nested sequence or nested choice may be present more than once, but always have to appear at least once. This gives you the option of having more than one of the same element, but still enforce that it appears in an instance. First let's try a simple case, with only one repeatable element:

Example 4.32:

```
<elementtype name="RepeatableContent">
      <model>
            <element type="string" name="StringContent" occurs="+"/>
      </model>
</elementtype>
```

Even though the elementtype in example 4.32 has a single element content, a valid instance can now contain many elements, as long as they are all of `StringContent` type:

Example 4.33:

```
<RepeatableContent>
      <StringContent>This is the first occurrence</StringContent>
      <StringContent>This is the second</StringContent>
      <StringContent>I can have as many as I like</StringContent>
      <StringContent>But I think this is enough</StringContent>
</RepeatableContent
```

Having only one content element is still perfectly valid:

Example 4.34:

```
<RepeatableContent>
      <StringContent>Having only one is fine</StringContent>
</RepeatableContent>
```

A more complex case involves having repeatable nested sequences or choices:

Example 4.35:

```
<elementtype name="RepeatableSequenceContent">
      <model>
            <sequence>
                  <element type="boolean" name="BooleanContent"/>
                  <sequence occurs="+">
                        <element type="float" name="FloatContent"/>
                        <element type="date" name="DateContent"/>
                  </sequence>
            </sequence>
      </model>
</elementtype>
```

An instance of the elementtype in example 4.35 must contain one, and only one of the `BooleanContent` element, but it can contain the nested sequence once or repeated several times:

Example 4.36:

```
<RepeatableSequenceContent>
      <BooleanContent>false</BooleanContent>
      <FloatContent>123.0</FloatContent>
      <DateContent>19950228</DateContent>
      <FloatContent>0.0</FloatContent>
      <DateContent>18971225</DateContent>
      <FloatContent>5729.0001</FloatContent>
      <DateContent>20000101</DateContent>
</RepeatableSequenceContent>
```

Note that, in example 4.36, the nested sequence must always have all of its content present, and in the right order, but can be repeated any number of times.

4.2.6.3 * Occurrence

An occurs value of "*****" specifies that an element, nested sequence or nested choice is optional, but may appear multiple times. This is useful when an element's presence isn't required, but you would like it to be able to appear any number of times:

Example 4.37:

```
<elementtype name="OptionalMultipleContent">
      <model>
            <element type="string" name="StringContent" occurs="*"/>
      </model>
</elementtype>
```

An instance of the elementtype in example 4.37 could have any number of the `StringContent` element:

Example 4.38:

```
<OptionalMultipleContent>
      <StringContent>Or you can have many</StringContent>
      <StringContent>As many as you like</StringContent>
      <StringContent>Any amount you feel like<StringContent>
      <StringContent>Or none at all</StringContent>
</OptionalMultipleContent>
```

At the same time, a perfectly valid instance of the elementtype in example 4.37 could have no `StringContent` content elements at all:

Example 4.39:

```
<OptionalMultipleContent/>
```

In this case, we have even made the `OptionalMultipleContent` element an empty tag. Since it does not contain any content, this is valid.

Let's create a more complex example where we use a "*" occurrence in a nested choice:

Example 4.40:

```
<elementtype name="OptionalMultipleChoiceContent">
      <model>
            <sequence>
                  <element type="boolean" name="BooleanContent"/>
                  <choice occurs="*">
                        <element type="float" name="FloatContent"/>
                        <element type="date" name="DateContent"/>
                  </sequence>
            </sequence>
      </model>
</elementtype>
```

This schema allows an instance to choose from the available options, any number of times, from zero to an infinite number. As usual, `BooleanContent` is a required element.

Let's create a valid instance of this elementtype:

Example 4.41:

```
<OptionalMultipleChoiceContent>
      <BooleanContent>false</BooleanContent>
      <DateContent>19770717</DateContent>
      <DateContent>19721219</DateContent>
      <FloatContent>5729.0001</FloatContent>
      <DateContent>20010101</DateContent>
</OptionalMultipleChoiceContent>
```

Compare the instance in example 4.41 to example 4.36, which contains a nested sequence. In 4.36, the sequence can appear any number of times, but the sequence must always be complete. Example 4.41 is much less restricted, since any of the elements can be chosen each time. A valid instance can contain only `FloatContent` elements, or only `DateContent` elements, in addition to the required `BooleanContent` element.

Another valid use would be to omit the choice completely, since it has an occurs of "*":

Example 4.42:

```
<OptionalMultipleChoiceContent>
      <BooleanContent>true</BooleanContent>
</OptionalMultipleChoiceContent>
```

4.2.6.4 N,M occurrence

An occurs value of **N,M**, (where N and M specifies numeric values), specifies an occurrence range of an element, nested sequence or nested choice. This is useful when you want multiple occurrences, but still want to limit how many objects are allowed, or when you want a multiple minimum occurrence, or a specific number of one specific object. First, let us try setting a range for an elements' occurrence:

Example 4.43:

```
<elementtype name="OneToThreeContent">
      <model>
            <element type="string" name="StringContent" occurs="1,3"/>
      </model>
</elementtype>
```

Example 4.43 specifies that a valid instance of `OneToThreeContent` can contain between one and three instances of the `StringContent` element. Let's try to have two:

Example 4.44:

```
<OneToThreeContent>
      <StringContent>String one</StringContent>
      <StringContent>String two</StringContent>
</OneToThreeContent>
```

You can also specify an unlimited maximum occurrence using the "*" specifier as the second value:

Example 4.45:

```
<elementtype name="TwoToManyContent">
      <model>
            <element type="int" name="IntContent" occurs="2,*"/>
      </model>
</elementtype>
```

In example 4.45, we have specified that `TwoToManyContent` must contain at least two, and up to any number, of instances of the `IntContent` element. Here is a possible instance of the elementtype:

Example 4.46:

```
<TwoToManyContent>
      <IntContent>1</IntContent>
      <IntContent>2</IntContent>
      <IntContent>3</IntContent>
      <IntContent>4</IntContent>
</TwoToManyContent>
```

Next let's try to use the N,M occurrence to specify a specific number of valid occurrences:

Example 4.47:

```
<elementtype name="FourContent">
      <model>
            <element type="boolean" name="BooleanContent"
            occurs="4,4"/>
      </model>
</elementtype>
```

A valid instance of `FourContent` in example 4.47 must contain four, and only four, instances of `BooleanContent`. This way we can very easily constrain a multiple occurrence of an element, or a nested sequence or choice for that matter.

A valid instance would look like this:

Example 4.48:

```
<FourContent>
      <BooleanContent>true</BooleanContent>
      <BooleanContent>true</BooleanContent>
      <BooleanContent>false</BooleanContent>
      <BooleanContent>true</BooleanContent>
</FourContent>
```

> **Commerce One Implementation specific note:**
> The Commerce One implementation of the SOX parser currently does not reinforce the exact values in an N,M occurrence, but will treat a 0,M as an occurrence of * and an N,M as an occurrence of +.

4.2.7 Attributes

In addition to putting data as content in tags, you can also specify **attributes**, just as you can in XML. Attributes are contained in the start tag, and consist of a name, and equal sign, and a quoted value. An attribute's type can be of an intrinsic datatype or a user defined datatype, but never an elementtype. An element can have multiple attributes.

Often an attribute is used to describe an element, or the content of the element. There are no strict guidelines as to when to use attributes and when to use elements. Generally the attribute is thought to have an "is-a" relation to the element, as opposed to the "has-a" relation of the element content. For example, a `Person` element could have `Age` and `Nationality` attributes, whereas it could contain `Clothing` and `Car` elements. In this case the person is of a certain nationality and age, something that relates to who that person is, whereas the items that person owns are less tied to that specific person, and are probably better suited to be full fledged elements of their own, with their own attributes and content.

Another good rule is if the data is relatively small, such as a few characters long, it can often make sense to use an attribute to cut down the size of the instance considerably.

In addition, a major advantage of attributes is that you can specify an attribute to have a fixed or a default value, which is not possible for data in an element.

An attribute definition should contain a presence specification element, which specifies the presence information for the attribute. Four different presence modes exist: **required**, **implied**, **default**, **fixed**:

Required - the attribute must be present in the instance.
Implied - The attribute is optional in the instance.
Default - The attribute has a default value. If no value is specified in the instance, then the parsing application behaves as if the attribute was specified, with the default value provided in the schema.
Fixed - It is an error for the attribute to have any other value in the instance than the one specified in the schema. The attribute does not have to be explicitly present in the instance, but the parsing application will behave as if the value is present, with the fixed value.

If you do not provide a presence mode, the attribute presence will default to **implied**.

Let's create an elementtype with a few attributes:

Example 4.49:

```
<elementtype name="Person">
      <model>
            <element datatype="NMTOKENS" name="Name"/>
      </model>
      <attdef name="age" datatype="int">
            <required/>
      </attdef>
      <attdef name="occupation" datatype="string">
            <implied/>
      </attdef>
      <attdef name="language" datatype="NMTOKEN">
            <default>English</default>
      </attdef>
      <attdef name="species" datatype="string">
            <fixed>human</fixed>
      </attdef>
</elementtype>
```

Example 4.49 uses all four different presence modes. The datatypes NMTOKEN and NMTOKENS are intrinsic datatypes. See Appendix A: Intrinsic Datatypes. Here is a valid instance of the above elementtype:

Example 4.50:

```
<Person age="28" occupation="engineer" language="Chinese" species="human">
      <Name>John F. Smith</Name>
</Person>
```

In example 4.50 four attributes are specified. Note that attribute values must always be quoted. Also note that the value of the attribute species conforms to the fixed value specified in example 4.46. Any other value would make this instance invalid. Now let's try and omit those attributes that are not required:

Example 4.51:

```
<Person age="43">
      <Name>Doris Baumgartner</Name>
</Person>
```

In example 4.51, only the required attribute `age` is provided. This means that Doris does not have an occupation, and her language will default to English. Her species will always be fixed to human, whether it is provided or not.

4.3 Data Type Definitions

The SOX schema syntax provides you with the ability to define your own datatypes. This is very useful when you want to put restrictions on the allowable values for an attribute or element. Datatypes can never be instantiated by themselves, but have to be used as types for elements or attributes.
There are three ways to define a datatype, as an **enumeration**, a **scalar** or a **varchar**.

4.3.1 Enumeration Datatype Definition

An **enumeration** is a way to give an enumeration of allowable values of any intrinsic or user defined datatype. A list of options is provided in the datatype definition:

Example 4.52:

```
<datatype name="Color">
      <enumeration datatype="NMTOKEN">
            <option>Red</option>
            <option>Blue</option>
            <option>Green</option>
            <option>Yellow</option>
            <option>Orange</option>
            <option>Purple</option>
            <option>Black</option>
            <option>White</option>
            <option>Grey</option>
      </enumeration>
</datatype>

<datatype name="TrafficLightColor">
      <enumeration datatype="Color">
            <option>Red</option>
            <option>Green</option>
            <option>Yellow</option>
      </enumeration>
</datatype>

<elementtype name="TrafficLight">
      <model>
            <element name="State" type="TrafficLightColor"/>
      </model>
      <attdef name="CasingColor" datatype="Color">
            <required/>
      </attdef>
</elementtype>
```

In the first datatype we define an enumeration of NMTOKENs called `Color`. The enumeration lists a number of allowed values in **option** tags. These values are the only valid ones in the instance. The second datatype is a specialized subset of the first enumeration called `TrafficLightColor`. It puts further constraints on the `Color` datatype. It cannot add a value that does not exist in `Color`, since that would not be a valid color value. Next, since a datatype can not be instantiated, we create an elementtype called `TrafficLight` that uses both of these datatypes. One of the datatypes is used in an element, and the other in an attribute. A valid instance of the elementtype would look like this:

Example 4.53:

```
<TrafficLight CasingColor="Grey">
      <State>Green</State>
</TrafficLight>
```

As you can see, enumerations can be very useful, because you can specify a very specific set of values that you will accept as being valid for an element or an attribute. Unless the values provided in an instance conforms to that set of values, it will not be valid.

4.3.2 Scalar Datatype Definition

A **scalar** is useful when you want to use a numeric value, but want to restrict the range, number of decimals and/or number of digits. A scalar has to be of type number, float, int, byte, long, double or any subtype thereof. That means its type can be another scalar that is of one of those types.

Example 4.54:

```
<datatype name="Price">
      <scalar datatype="float" decimals="2"/>
</datatype>

<datatype name="MovieTicketPrice">
      <scalar datatype="Price" digits="1" maxvalue="8.50"
            minvalue="1.50" maxexclusive="false"
            minexclusive="false"/>
</datatype>

<elementtype name="MovieTicket">
      <model>
            <element name="MovieTitle" type="string"/>
      </model>
      <attdef name="TicketPrice" datatype="MovieTicketPrice">
            <required/>
      </attdef>
</elementtype>
```

In example 4.54 we have defined a scalar that can be used to express a generic price. The only restriction it has is that of the number of decimals, so we can express any price that deals with a whole number of cents.

Next, a subtype of price is defined, called `MovieTicketPrice`. It is defined specifically to express the price of a movie ticket. It always has to be more restrictive than its type, in this case `Price`. It will inherit any restrictions from its supertype, and is therefore not allowed to loosen up the supertype restrictions. This means that in this case, `MovieTicketPrice` is not allowed to have a `decimals` value that is higher than that of the same value in `Price`, which is `MovieTicketPrice`'s supertype.

In this case we do not try to change the decimals value. However, we add restrictions to the new datatype. We specify that the value may only have one digit with the **digits** attribute (not including the decimals). We also set **minvalue** and **maxvalue** to constrain the scalar to be a value between 1.50 and 8.50. Finally, we define that the minimum and maximum values are both valid values, by setting both **minexclusive** and **maxexclusive** to false. This means that we are not excluding those values from being valid. It should be noted that `MovieTicketPrice` uses all the possible constraints available for a scalar. **decimals** is used indirectly, through inheritance from `Price`, and the rest are used directly by `MovieTicketPrice`.

In example 4.54, we have also defined an elementtype called `MovieTicket` that uses the `MovieTicketPrice` datatype. A valid instance could look like this:

Example 4.55:

```
<MovieTicket TicketPrice="6.50">
      <MovieTitle>Gone With the Wind</MovieTitle>
</MovieTicket>
```

4.3.3 Varchar Datatype Definition

A **varchar** datatype restricts string types to have a maximum length. Varchars can be of type string, NMTOKEN, NMTOKENS, ID, IDREF, IDREFS or another varchar, and must specify the **maxlength** attribute. Any instance of the datatype is not allowed to exceed the specified maximum length.

Example 4.56:

```
<datatype name="LimitedString">
      <varchar maxlength="50" datatype="string"/>
</datatype>

<datatype name="TitleString">
      <varchar maxlength="25" datatype="LimitedString"/>
</datatype>

<datatype name="PhoneString">
            <varchar maxlength="16" datatype="LimitedString"/>
</datatype>

<elementtype name="BusinessCard">
      <model>
            <sequence>
                  <element name="Name" type="string"/>
                  <element name="Title" type="TitleString"/>
                  <element name="Phone" type="PhoneString"/>
                  <element name="Motto" type="LimitedString"/>
            </sequence>
      </model>
</elementtype>
```

In example 4.56, we have decided to create a SOX schema as a template for the information that goes on a business card. We define three varchars, two of which derive from the first varchar. The first is called `limitedString`. It is used for those employees who wish to put a motto on their card, but in order to be able to fit the motto on the card, we have the constraint that a motto can have a maximum length of 50 characters. We also limit the length of the title they can put on their card. The phone number has a limit to its length, which corresponds to the length of a normally formatted phone number. Now we can ensure that the employee information will fit onto the card.

An instance could look like this:

Example 4.57:

```
<BusinessCard>
      <Name>Harold Hodgemeier</Name>
      <Title>Regional Coordinator</Title>
      <Phone>1 (800) 123-4567</Phone>
      <Motto>To boldly go where no man has gone before...</Motto>
</BusinessCard>
```

4.3.4 Anonymous Datatype Definitions

In addition to defining enumerations, scalars and varchars as reusable datatypes, you can also make an anonymous definition inside of an attribute definition. The datatype can then only be used for that attribute, and cannot be reused, as opposed to regularly defined datatypes. It is important to note that if a datatype is defined in the attribute definition, then the attribute definition itself may not specify its datatype attribute.

This would serve no purpose, and would probably contradict what is being specified inside the attribute definition.

Here is an example of anonymously defined datatypes:

Example 4.58:

```
<elementtype name="Car">
      <empty/>
      <attdef name="Color">
            <enumeration datatype="NMTOKEN">
                  <option>Red</option>
                  <option>Blue</option>
                  <option>Silver</option>
                  <option>Black</option>
                  <option>White</option>
                  <option>Brown</option>
                  <option>Green</option>
            </enumeration>
            <required/>
      </attdef>
      <attdef name="Registration">
            <varchar maxlength="7" datatype="string"/>
            <implied/>
      </attdef>
      <attdef name="Mileage>
            <scalar datatype="double" digits="6" decimals="1"
                  minvalue="0"/>
            <required/>
      </attdef>
</elementtype>
```

In example 4.58 we have defined an anonymous enumeration, scalar and varchar, each in its own attribute definition in an elementtype. An instance could look like this:

Example 4.59:

```
<Car color="Blue" registration="MY CAR" mileage="72881.0"/>
```

As you can see in example 4.59, the attributes are used just as if the anonymous datatypes had been specified as regular datatypes.

4.4 Complete SOX Schema

Now let’s create a complete SOX schema. This example shows the structure for information about a film. Don't forget the XML version and DOCTYPE declaration at the top of the schema!

Example 4.60:

```
<?xml version="1.0"?>
<!DOCTYPE schema SYSTEM
  "urn:x-commerceone:document:com:commerceone:xdk:xml:schema.dtd$1.0">

<schema uri="urn:x-commerceone:document:sample:xdk:sox:Film.sox$1.0">

    <elementtype name="Film">
        <model>
            <sequence>
                <element name="Director" type="Crew" occurs="?"/>
                <element name="Actor" type="Crew" occurs="*"/>
            </sequence>
        </model>
        <attdef name="Name" datatype="string">
            <required/>
        </attdef>
        <attdef name="Genre" datatype="FilmGenre">
            <required/>
        </attdef>
        <attdef name="Length" datatype="int">
            <required/>
        </attdef>
        <attdef name="ReleaseYear" datatype="Year">
            <implied/>
        </attdef>
    </elementtype>

    <elementtype name="Crew">
        <model>
            <element name="PreviousFilm" type="FilmSummary"
                occurs="*"/>
        </model>
        <attdef name="Name" datatype="string">
            <required/>
        </attdef>
        <attdef name="Gender">
            <enumeration datatype="NMTOKEN">
                <option>male</option>
                <option>female</option>
            </enumeration>
            <required/>
        </attdef>
    </elementtype>
```

Continued on next page…

Example 4.60 continued:

```
    <elementtype name="FilmSummary">
        <empty/>
        <attdef name="Name" datatype="string">
            <required/>
        </attdef>
        <attdef name="ReleaseYear" datatype="Year">
            <required/>
        </attdef>
    </elementtype>

    <datatype name="FilmGenre">
        <enumeration datatype="string">
            <option>Comedy</option>
            <option>Drama</option>
            <option>Sci-fi</option>
            <option>Thriller</option>
            <option>Action</option>
            <option>Western</option>
        </enumeration>
    </datatype>

    <datatype name="Year">
        <scalar datatype="int" digits="4" minvalue="1880"/>
    </datatype>
</schema>
```

Here is a sample instance of the Film Schema. Don't forget the soxtype declaration at the top of the instance!

Example 4.61:

```
<?soxtype urn:x-commerceone:document:sample:xdk:sox:Film.sox$1.0?>

<Film Name="Gone With the Wind" Genre="Drama" Length="222"
    ReleaseYear="1939">
    <Director>
        <Crew Name="Victor Fleming" Gender="male">
            <PreviousFilm>
                <FilmSummary Name="Wizard of Oz" ReleaseYear="1939"/>
            </PreviousFilm>
            <PreviousFilm>
                <FilmSummary Name="Treasure Island"
                    ReleaseYear="1934"/>
            </PreviousFilm>

        </Crew>
    </Director>
    <Actor>
        <Crew Name="Clark Gable" Gender="male">
            <PreviousFilm>
                <FilmSummary Name="Misfits" ReleaseYear="1961"/>
            </PreviousFilm>
            <PreviousFilm>
                <FilmSummary Name="It Happened One Night"
                    ReleaseYear="1961"/>
            </PreviousFilm>
        </Crew>
    </Actor>
    <Actor>
        <Crew Name="Vivien Leigh" Gender="female">
            <PreviousFilm>
                <FilmSummary Name="A streetcar named Desire"
                    ReleaseYear="1951"/>
            </PreviousFilm>
        </Crew>
    </Actor>
</Film>
```

5 Namespaces

A schema exists in a **namespace**, defined by the URI attribute on the schema element of that schema. To use definitions from another schema, you should import that schema's namespace into the current schema. This is very similar to importing java classes. It enables you to reuse elementtypes or datatypes, and it promotes modular schema writing. If you intend to create complex or reusable schemas and definitions, it is highly recommended that you use namespace imports.

After you import the definitions into your schema, they still retain their own namespace, and must always be referred to with a **prefix** associated with that namespace. The prefix enables you to redefine the same name in different schemas, and still be able to reuse the definitions without name collisions.

5.1 Importing SOX Schemas

In order to be able to use definitions in a schema, you first need to import the schema that you want to use. This is done with the **namespace element**. The namespace element associates a prefix with a namespace.

Example 5.1:

```
<namespace prefix="pre"
namespace="urn:x-commerceone:document:sample:xdk:sox:sample.sox$1.0"/>
```

5.2 Using Elementtypes and Datatypes from Imported Schemas

Now that a prefix has been associated with a namespace, the definitions from the imported namespace can be used freely, as long as they are used with the defined prefix. First we define a schema that has generic definitions:

Example 5.2:

```
<?xml version="1.0"?>
<!DOCTYPE schema SYSTEM
  "urn:x-commerceone:document:com:commerceone:xdk:xml:schema.dtd$1.0">
<schema uri=
"urn:x-commerceone:document:sample:xdk:sox:Container.sox$1.0">

    <elementtype name="AluminumCan">
        <empty/>
    </elementtype>

    <elementtype name="GlassBottle">
        <empty/>
    </elementtype>

    <elementtype name="PaperCup">
        <model>
            <sequence>
                <element type="Lid" occurs="?"/>
                <element type="Straw" occurs="?"/>
            </sequence>
        </model>
    </elementtype>

    <elementtype name="Lid">
        <empty/>
    </elementtype>

    <elementtype name="Straw">
        <empty/>
        <attdef name="Striped" datatype="boolean">
            <default>false</default>
        </attdef>
    </elementtype>

</schema>
```

Next we define a schema that imports and reuses elements from the Container schema:

Example 5.3:

```
<?xml version="1.0"?>
<!DOCTYPE schema SYSTEM
  "urn:x-commerceone:document:com:commerceone:xdk:xml:schema.dtd$1.0">

<schema uri="urn:x-commerceone:document:sample:xdk:sox:Beverage.sox$1.0">

   <namespace prefix="containers" namespace=
      "urn:x-commerceone:document:sample:xdk:sox:Container.sox$1.0"/>

   <elementtype name="Beverage">
      <model>
          <sequence>
             <element name="Name" type="string"/>
             <choice>
                <element name="Can" type="AluminumCan"
                   prefix="containers"/>
                <element type="GlassBottle" prefix="containers"/>
                <element type="PaperCup" prefix="containers"/>
             </choice>
          </sequence>
      </model>
      <attdef name="Volume" datatype="float">
          <required/>
      </attdef>
      <attdef name="VolumeUnit" datatype="Unit">
          <default>fluid ounces</default>
      </attdef>
      <attdef name="Price" datatype="float">
          <required/>
      </attdef>
      <attdef name="Carbonated" datatype="boolean">
            <default>true</default>
      </attdef>
  </elementtype>

  <datatype name="Unit">
      <enumeration datatype="NMTOKENS">
          <option>fluid ounces</option>
          <option>milliliters</option>
          <option>centiliters</option>
          <option>liters</option>
      </enumeration>
  </datatype>

</schema>
```

Importing and using the elements from `Container.sox` is pretty obvious. A prefix is given in an attribute for any types defined in a different schema, and that prefix is defined in the namespace import tag.

The interesting part comes when we try to make an instance of a schema that uses namespaces. How does this look?

Example 5.4:

```
<?soxtype urn:x-commerceone:document:sample:xdk:sox:Beverage.sox$1.0?>

<Beverage Volume="12" Price="0.99" Carbonated="true">
   <Name>Coca Cola</Name>
      <other:PaperCup xmlns:other=
       "urn:x-commerceone:document:sample:xdk:sox:Container.sox$1.0">
         <other:Lid/>
         <other:Straw Striped="true"></other:Straw>
      </other:PaperCup>
</Beverage>
```

In example 5.4, the Beverage element has been defined in the **current namespace**, that is, the namespace that is given in the soxtype definition at the top of the instance. Therefore, anything that is defined in the `Beverage` schema, can be used without namespaces, as usual. The `PaperCup` element has been defined in the Container namespace however, so its namespace must first be defined in attribute format, where the name is the xmlns keyword, followed by a colon ":" and the prefix. In this case we have chosen the prefix "other". The prefix does not have to be the same as that defined in the schema. The value of the `xmlns:other` attribute is the namespace that the prefix maps to. Any elements that have been defined in the Container namespace must now be used with the "other" prefix and a colon, pre-pended to the element name. This is true for both opening and closing tags.

But what if an imported elementtype is given a new name in the content model in the importing schema? The definition of the element is still in the imported schema, but the new name has been defined in the current namespace. Let's take a look at an example:

Example 5.5:

```
<?soxtype urn:x-commerceone:document:sample:xdk:sox:Beverage.sox$1.0?>

<Beverage Volume="8" Price="0.45" Carbonated="true" xmlns:other=
       "urn:x-commerceone:document:sample:xdk:sox:Container.sox$1.0">
   <Name>Tab</Name>
      <Can>
          <other:AluminumCan/>
      </Can>
</Beverage>
```

The answer is that while `AluminumCan` needs to have the prefix of the namespace it has been defined in, the new name `Can` is defined in the current namespace, and therefore should not have a prefix.

In example 5.5, the definition of the prefix has been placed in the root element, which is perfectly legal. The prefix-namespace association will still be valid for throughout the current namespace, that is, the current schema. If an instance contains many namespace definitions, it can be a good idea to place them all inside the Root element, because it makes them easier to find, and it makes the instance easier to read.

5.3 Multi-Level Imports

The previous namespace examples were fairly straight forward, but what if we import from multiple namespaces, and some of those namespaces in turn have imported other namespaces? Let's add a new schema:

Example 5.6:

```
<?xml version="1.0"?>
<!DOCTYPE schema SYSTEM
  "urn:x-commerceone:document:com:commerceone:xdk:xml:schema.dtd$1.0">
<schema uri="urn:x-commerceone:document:sample:xdk:sox:Snack.sox$1.0">

    <elementtype name="Snack">
        <model>
           <choice>
              <element type="PopCorn"/>
              <element name="Candy" type="ChocolateBar"/>
              <element name="Fruit" type="FruitEnum"/>
           </choice>
        </model>
    </elementtype>

    <elementtype name="PopCorn">
        <empty/>
        <attdef name="LowFat" datatype="boolean">
           <implied/>
        </attdef>
    </elementtype>

    <elementtype name="ChocolateBar">
        <model>
           <element name="Name" type="string"/>
        </model>
    </elementtype>

    <datatype name="FruitEnum">
        <enumeration datatype="NMTOKEN">
            <option>Apple</option><option>Banana</option>
            <option>orange</option><option>mango</option>
            <option>grapes</option><option>melon</option>
        </enumeration>
    </datatype>

</schema>
```

The schema in example 5.6 defines a `Snack` element, which in turn will contain one out a number of choices of specific snacks.

Next we define a `RefreshmentOrder` which imports both the `Snack` schema and `Beverage` schema (which in turn imports the `Container` schema):

Example 5.7:

```
<?xml version="1.0"?>
<!DOCTYPE schema SYSTEM
  "urn:x-commerceone:document:com:commerceone:xdk:xml:schema.dtd$1.0">

<schema uri=
"urn:x-commerceone:document:sample:xdk:sox:RefreshmentOrder.sox$1.0">

   <namespace prefix="bev" namespace=
       "urn:x-commerceone:document:sample:xdk:sox:Beverage.sox$1.0"/>

   <namespace prefix="snack" namespace=
       "urn:x-commerceone:document:sample:xdk:sox:Snack.sox$1.0"/>

   <elementtype name="RefreshmentOrder">
      <model>
          <sequence>
             <element type="BeverageOrder" occurs="*"/>
             <element type="SnackOrder" occurs="*"/>
          </sequence>
      </model>
      <attdef name="Charge">
          <enumeration datatype="string">
             <option>cash</option>
             <option>check</option>
             <option>credit</option>
          </enumeration>
          <required/>
      </attdef>
   </elementtype>

   <elementtype name="BeverageOrder">
      <model>
         <element name="BeverageSpec" prefix="bev" type="Beverage"/>
      </model>
      <attdef name="Quantity" datatype="int">
         <required/>
      </attdef>
   </elementtype>

   <elementtype name="SnackOrder">
      <model>
         <element prefix="snack" type="Snack"/>
      </model>
      <attdef name="Quantity" datatype="int">
         <required/>
      </attdef>
   </elementtype>
</schema>
```

A namespace import made in one schema is not visible in any other schema. Although `RefreshmentOrder.sox` imports `Beverage.sox`, it does not have access to any definitions in `Container.sox`, without explicitly importing that schema itself.

An instance of the `RefreshmentOrder` schema could look like this:

Example 5.8:

```
<?soxtype
urn:x-commerceone:document:sample:xdk:sox:RefreshmentOrder.sox$1.0 ?>

<RefreshmentOrder Charge="credit"
   xmlns:bev="urn:x-commerceone:document:sample:xdk:sox:Beverage.sox$1.0"

   xmlns:cnt=
   "urn:x-commerceone:document:sample:xdk:sox:Container.sox$1.0"

   xmlns:snk="urn:x-commerceone:document:sample:xdk:sox:Snack.sox$1.0">

      <BeverageOrder Quantity="100">
         <BeverageSpec>
            <bev:Beverage Volume="355" VolumeUnit="milliliters"
                  Price="1.29">
               <bev:Name>Juice Squeeze Ruby Grapefruit</bev:Name>
               <cnt:GlassBottle/>
            </bev:Beverage>
         </BeverageSpec>
      </BeverageOrder>
      <BeverageOrder Quantity="250">
         <BeverageSpec>
            <bev:Beverage Volume="12" Price="0.65">
               <bev:Name>Coca Cola</bev:Name>
               <bev:Can>
                  <cnt:AluminumCan/>
               </bev:Can>
            </bev:Beverage>
         </BeverageSpec>
      </BeverageOrder>
      <SnackOrder Quantity="50">
         <snk:Snack>
            <snk:Fruit>Apple</snk:Fruit>
         </snk:Snack>
      </SnackOrder>
      <SnackOrder Quantity="125">
         <snk:Snack>
            <snk:Candy>
               <snk:ChocolateBar>
                  <snk:Name>Almond Joy</snk:Name>
               </snk:ChocolateBar>
            </snk:Candy>
         </snk:Snack>
      </SnackOrder>
      <SnackOrder Quantity="25">
         <snk:Snack>
            <snk:PopCorn LowFat="false"/>
         </snk:Snack>
      </SnackOrder>
</RefreshmentOrder>
```

In example 5.8 you can see that all elements have a prefix pointing to the namespace they were defined in. The only exception are the elements that were defined in the current namespace, which do not need to have a prefix.

Note that none of the attributes has a prefix. They do not need prefixes, as they can only be in the same namespace as the element they are contained in. Only element names have prefixes, and always the prefix corresponding to the namespace they were defined in.

Observe the difference between a `BeverageOrder` and `SnackOrder`. The `BeverageOrder` has assigned a new name to the `Beverage` element, and therefore gets an additional tag level in the current namespace. Each element belongs to the namespace it was defined in. The `BeverageOrder`s are especially interesting, as their content stems from three different namespaces, the current namespace, the `Beverage` namespace, and the `Container` namespace.

If the prefixes are made descriptive enough, and in addition are sufficiently different from each other, it is quite easy to tell the origin of an element.

5.4 Using a Default Namespace in an Instance

As you have probably noticed, using namespaces can seem quite cumbersome if most of the elements you are using have been imported from other schemas. For example, if you import and use large elements, a good alternative is to use a **default namespace** declaration in the instance. A default namespace is used with an element, to declare which namespace that element, and all of its content, will default to. That means that the element with the default namespace declaration, and all of its content elements, will be assumed to be from the declared namespace. These elements now no longer need a prefix, unless they stem from a different namespace than the default namespace.

The default namespace is declared by using an attribute called xmlns, which is used in the element you want to associate the default namespace with. You will recognize this keyword from previous examples, but notice that in this case there is no colon, and no prefix attached to it. The value of the xmlns attribute is the namespace that should be default for the element you are using it with. Let's try this with a trimmed down version of the instance of `RefreshmentOrder` from example 5.8:

Example 5.9:

```
<?soxtype
urn:x-commerceone:document:sample:xdk:sox:RefreshmentOrder.sox$1.0 ?>

<RefreshmentOrder Charge="credit"
   xmlns:cnt=
   "urn:x-commerceone:document:sample:xdk:sox:Container.sox$1.0">

      <BeverageOrder Quantity="250">
         <BeverageSpec>
            <Beverage Volume="12" Price="0.65" xmlns=
   "urn:x-commerceone:document:sample:xdk:sox:Beverage.sox$1.0">
               <Name>Coca Cola</Name>
               <Can>
                  <cnt:AluminumCan/>
               </Can>
            </Beverage>
         </BeverageSpec>
      </BeverageOrder>
      <SnackOrder Quantity="125">
         <Snack xmlns=
               "urn:x-commerceone:document:sample:xdk:sox:Snack.sox$1.0">

            <Candy>
               <ChocolateBar>
                  <Name>Almond Joy</Name>
               </ChocolateBar>
            </Candy>
         </Snack>
      </SnackOrder>
</RefreshmentOrder>
```

`Snack` instances each declare a default namespace for themselves and all of their content. Note the beverage container `AluminumCan` in the `Beverage` Instance. It still needs to have a prefix, because it does not belong to the default namespace declared for `Beverage`. If you would like to use an element from the current namespace, then it also would have to be used with a prefix, and that prefix would have to be associated with the namespace of the current schema.

6 Inheritance

The SOX inheritance feature allows you to define elementtypes that inherit structure from other elementtypes. With inheritance, a previously defined elementtype, the parent elementtype can be extended by another elementtype, the child elementtype. This enables you to create new elementtypes by extending existing elementtypes in order to add additional structures or attributes.

Instances of the extended elements can also be used polymorphically, that is, as if they were instances of the original elements that were extended. This will be discussed further in section 7: Polymorphism.

6.1 Extending an Existing Element

Only elementtypes that have sequence content models may be extended. The reason for this is that the extending element is essentially appending new content to the end of the existing. A string content model does not provide any structure to add to, as its model means that only a text string will be contained in the element. A choice in turn does not have an ideal structure either, since its content model means that only one of the elements in the model will actually be used. Adding structure to such a model would violate the model itself. The element content model is considered to be a sequence content model with one element in the sequence, and the empty content model is considered to be a sequence of 0 elements, so it is perfectly legal to extend both of these content models. They are both considered to be special cases of a sequence.

Example 6.1:

```
<?xml version="1.0"?>
<!DOCTYPE schema SYSTEM
  "urn:x-commerceone:document:com:commerceone:xdk:xml:schema.dtd$1.0">

<schema uri="urn:x-commerceone:document:sample:xdk:sox:Rooms.sox$1.0">
   <elementtype name="Room">
      <model>
         <sequence>
            <element name="Width" type="int"/>
            <element name="Length" type="int"/>
            <element name="Height" type="int"/>
            <choice occurs="*">
               <element type="Carpet"/>
               <element type="Window"/>
               <element type="Door"/>
            </choice>
         </sequence>
      </model>
   </elementtype>
   <elementtype name="Carpet">
      <empty/>
      <attdef name="Length">
         <enumeration datatype="string">
            <option>short</option>
            <option>medium</option>
            <option>long</option>
         </enumeration>
      </attdef>
   </elementtype>
```

Continued on next page…

Example 6.1 continued:

```
    <elementtype name="Opening">
       <empty/>
       <attdef name="Direction">
          <enumeration datatype="NMTOKEN">
             <option>south</option>
             <option>north</option>
             <option>west</option>
             <option>east</option>
          </enumeration>
       </attdef>
    </elementtype>
    <elementtype name="Window">
       <extends type="Opening">
          <append>
             <element name="WindowType" type="OpeningType"/>
          </append>
          <attdef name="MosquitoNet" datatype="boolean">
             <default>true</default>
          </attdef>
          <attdef name="Blinds" datatype="boolean">
             <default>true</default>
          </attdef>
       </extends>
    </elementtype>
    <datatype name="OpeningType">
       <enumeration datatype="string">
          <option>sliding</option>
          <option>opening</option>
       </enumeration>
    </datatype>
    <elementtype name="Door">
       <extends type="Opening">
          <attdef name="DoorType">
             <enumeration datatype="NMTOKEN">
                <option>single</option>
                <option>double</option>
             </enumeration>
             <default>single</default>
          </attdef>
       </extends>
    </elementtype>
    <elementtype name="BedRoom">
       <extends type="Room">
          <append>
             <element type="Closet" occurs="*"/>
          </append>
       </extends>
    </elementtype>
    <elementtype name="Closet">
       <empty/>
       <attdef name="WalkIn" datatype="boolean">
          <implied/>
       </attdef>
    </elementtype>
    <elementtype name="LivingRoom">
       <extends type="Room"/>
    </elementtype>

 </schema>
```

In the schema in example 6.1, two different elementtypes are extended. The elementtype `Opening` is extended by both `Window`, that adds elements and attribute definitions to the empty content model of `Opening`, and `Door`, that only adds an attribute. The element `Room` is extended by the element `BedRoom`, which appends an element to the sequence content model. The `Closet` element will be appended at the end, after the choice in `Room`. `Room` is also extended by the elementtype `LivingRoom`, which does not add any content structure to the `Room` elementtype, but is perfectly valid. This is a way to create an elementtype that is distinctly different from `Room`, even though it does not add anything to it.

An instance of `LivingRoom` could look like this:

Example 6.2:

```
<?soxtype urn:x-commerceone:document:sample:xdk:sox:Rooms.sox$1.0?>

<LivingRoom>
   <Width>12</Width>
   <Length>15</Length>
   <Height>9</Height>
   <Carpet Length="medium"/>
   <Window Direction="south" Blinds="false">
      <WindowType>sliding</WindowType>
   </Window>
   <Window Direction="east" Blinds="true" MosquitoNet="false">
      <WindowType>opening</WindowType>
   </Window>
   <Door Direction="west" DoorType="double"></Door>
</LivingRoom>
```

In example 6.2, the elements that extended `Opening` now have `Opening`'s attribute, as well as some attributes of their own. The attributes from both elementtypes work just the same way, regardless of where they were defined.

Next let's look at an instance of `BedRoom` which adds to elementtypes to `Room`:

Example 6.3:

```
<?soxtype urn:x-commerceone:document:sample:xdk:sox:Rooms.sox$1.0?>

<BedRoom>
   <Width>9</Width>
   <Length>12</Length>
   <Height>8</Height>
   <Carpet Length="long"/>
   <Window Direction="east">
      <WindowType>sliding</WindowType>
   </Window>
   <Door Direction="west" DoorType="single"/>
   <Closet WalkIn="true"/>
   <Closet/>
</BedRoom>
```

When creating extended elementtypes, keep in mind that an elementtype can only extend one other elementtype, but that elementtype may in turn be extending other elementtypes.

6.2 Extending an Element from a Different Namespace

It is also possible to extend elementtypes from other namespaces than the current. In this example we first define an element in one namespace:

Example 6.4:

```
<?xml version="1.0"?>
<!DOCTYPE schema SYSTEM
  "urn:x-commerceone:document:com:commerceone:xdk:xml:schema.dtd$1.0">

<schema uri="urn:x-commerceone:document:sample:xdk:sox:Ticket.sox$1.0">

   <elementtype name="Ticket">
      <model>
         <sequence>
            <element name="Price" type="TicketPrice"/>
            <element name="Time" type="datetime"/>
            <element name="Location" type="string"/>
         </sequence>
      </model>
      <attdef name="PrePaid" datatype="boolean">
         <required/>
      </attdef>
   </elementtype>

   <datatype name="TicketPrice">
      <scalar datatype="float" decimals="2" minvalue="0"/>
   </datatype>

</schema>
```

Note the use of the SOX datatype **datetime** in example 6.4. See Appendix A: Intrinsic Datatypes for further reference on this and other intrinsic datatypes.

Next we extend the first elementtype in two other namespaces. First we use it in a schema called `MovieTicket`:

Example 6.5:

```
<?xml version="1.0"?>
<!DOCTYPE schema SYSTEM
  "urn:x-commerceone:document:com:commerceone:xdk:xml:schema.dtd$1.0">

<schema uri=
"urn:x-commerceone:document:sample:xdk:sox:MovieTicket.sox$1.0">

<namespace prefix="tick"
namespace="urn:x-commerceone:document:sample:xdk:sox:Ticket.sox$1.0"/>

   <elementtype name="MovieTicket">
      <extends prefix="tick" type="Ticket">
         <append>
            <element name="Title" type="string"/>
            <element name="Screen" type="int"/>
         </append>
         <attdef name="Discount">
            <enumeration datatype="NMTOKEN">
               <option>Student</option>
               <option>Senior</option>
               <option>Matinee</option>
               <option>child</option>
            </enumeration>
         </attdef>
      </extends>
   </elementtype>

</schema>
```

Let us use the same Ticket schema in a different schema called `ConcertTicket`:
Example 6.6:

```
<?xml version="1.0"?>
<!DOCTYPE schema SYSTEM
  "urn:x-commerceone:document:com:commerceone:xdk:xml:schema.dtd$1.0">

<schema uri=
"urn:x-commerceone:document:sample:xdk:sox:ConcertTicket.sox$1.0">

<namespace prefix="ns1"
namespace="urn:x-commerceone:document:sample:xdk:sox:Ticket.sox$1.0"/>

   <elementtype name="ConcertTicket">
      <extends prefix="ns1" type="Ticket">
         <append>
            <element name="Band" type="string"/>
            <element name="OpeningAct" type="string" occurs="*"/>
         </append>
         <attdef name="Seating">
            <enumeration datatype="NMTOKEN">
               <option>General</option>
               <option>Reserved</option>
               <option>Box</option>
            </enumeration>
            <required/>
         </attdef>
         <attdef name="SeatNumber" datatype="string">
            <implied/>
         </attdef>
      </extends>
   </elementtype>
```

Let's see what the use of different namespaces do to the `MovieTicket` instance:
Example 6.8:

```
<?soxtype
urn:x-commerceone:document:sample:xdk:sox:MovieTicket.sox$1.0?>

<MovieTicket Discount="Student" PrePaid="false"
xmlns:gen="urn:x-commerceone:document:sample:xdk:sox:Ticket.sox$1.0">
   <gen:Price>4.50</gen:Price>
   <gen:Time>19990726T19:30:00</gen:Time>
   <gen:Location>AMC Mercado</gen:Location>
   <Title>Matrix</Title>
   <Screen>11</Screen>
</MovieTicket>
```

Just as with the previous namespace examples, elements have the prefix of the namespace where they are defined. The attributes are of special interest here, because one was defined in the parent, and one in the child, but neither has a prefix. Attributes never have a prefix, regardless of where they were defined.

As an added example, here follows an instance of `ConcertTicket`:

Example 6.9:

```
<?soxtype
urn:x-commerceone:document:sample:xdk:sox:ConcertTicket.sox$1.0?>

<ConcertTicket Seating="Reserved" SeatNumber="A35" PrePaid="true"
xmlns:parent="urn:x-commerceone:document:sample:xdk:sox:Ticket.sox$1.0">
   <parent:Price>39.50</parent:Price>
   <parent:Time>19990802T21:00:00</parent:Time>
   <parent:Location>Shoreline Amphitheatre</parent:Location>
   <Band>B-52s</Band>
   <OpeningAct>The Nobodys</OpeningAct>
</ConcertTicket>
```

So as you can see, the generic structure of `Ticket` can now easily be reused for two very different ticket types. If inheritance is used the right way, it can save a lot of work and mark-up, and it can also help describe relationships between elementtypes.

7 Polymorphism

The SOX language supports polymorphism, which means that when an elementtype specifies a specific elementtype in its content model, a subtype of this elementtype may be used instead in the instance. The subtype will always contain all the content that the parent type must have, regardless of what additional content it might add.

7.1 Using Polymorphic Elements from the same Namespace

The simplest form of polymorphism is when a supertype and a subtype exist in the same namespace. In a content model, the supertype is specified, but in the actual instance, the sub type can be used instead:

Example 7.1:

```
<?xml version="1.0"?>
<!DOCTYPE schema SYSTEM
  "urn:x-commerceone:document:com:commerceone:xdk:xml:schema.dtd$1.0">

<schema uri=
"urn:x-commerceone:document:sample:xdk:sox:FruitSalad.sox$1.0">

   <elementtype name="FruitSalad">
      <model>
         <sequence>
            <element name="BaseFruit" type="Fruit"/>
            <element type="Fruit" occurs="+"/>
         </sequence>
      </model>
   </elementtype>
   <elementtype name="Fruit">
      <model>
         <element name="Name" type="string"/>
      </model>
      <attdef name="Presentation">
         <enumeration datatype="string">
            <option>sliced</option>
            <option>diced</option>
            <option>peeled</option>
            <option>whole</option>
         </enumeration>
      </attdef>
   </elementtype>
   <elementtype name="Apple">
      <extends type="Fruit">
         <append>
            <element name="Color" type="string"/>
         </append>
      </extends>
   </elementtype>
   <elementtype name="Banana">
      <extends type="Fruit">
         <attdef name="Ripeness">
            <enumeration datatype="NMTOKEN">
               <option>green</option>
               <option>yellow</option>
               <option>speckled</option>
               <option>brown</option>
            </enumeration>
         </attdef>
      </extends>
   </elementtype>
</schema>
```

Let's look at how a polymorphic instance of the schema in example 7.1 might look:

Example 7.2:

```
<?soxtype urn:x-commerceone:document:sample:xdk:sox:FruitSalad.sox$1.0?>

<FruitSalad>
   <BaseFruit>
      <Apple Presentation="diced">
         <Name>Granny Smith</Name>
         <Color>Green</Color>
      </Apple>
   </BaseFruit>
   <Banana Ripeness="yellow" Presentation="sliced">
      <Name>Plantain</Name>
   </Banana>
   <Fruit Presentation="diced">
      <Name>Mango</Name>
   </Fruit>
   <Fruit Presentation="whole">
      <Name>Cherries</Name>
   </Fruit>
   <Apple Presentation="sliced">
      <Name>Fuji</Name>
      <Color>Yellow</Color>
   </Apple>
</FruitSalad>
```

In example 7.2, note the case of the first element in the content model, `BaseFruit`. Even though the content is a polymorphic element, the new name assignment is still used. In the case of every polymorphic fruit above, the sub type is used as if it was a Fruit element, which it essentially is.

7.2 Using Polymorphic Elements from Different Namespaces

Now let's define a schema that uses the rooms from example 6.1, to create a description of a house. As one might suspect a house consists of a number of rooms. In addition to the rooms we have already defined in the `Rooms` schema, let's add another type of room, a `BathRoom`:

Example 7.3:

```
<?xml version="1.0"?>
<!DOCTYPE schema SYSTEM
  "urn:x-commerceone:document:com:commerceone:xdk:xml:schema.dtd$1.0">

<schema uri="urn:x-commerceone:document:sample:xdk:sox:House.sox$1.0">

   <namespace prefix="room" namespace=
      "urn:x-commerceone:document:sample:xdk:sox:Rooms.sox$1.0"/>

   <elementtype name="House">
      <model>
         <element prefix="room" type="Room" occurs="2,*"/>
      </model>
   </elementtype>

   <elementtype name="BathRoom">
      <extends prefix="room" type="Room">
         <append>
            <element name="Facility" type="WaterFacility" occurs="+"/>
         </append>
      </extends>
   </elementtype>

   <datatype name="WaterFacility">
      <enumeration datatype="string">
         <option>Toilet</option>
         <option>Shower</option>
         <option>Sink</option>
         <option>Bathtub</option>
      </enumeration>
   </datatype>

</schema>
```

Now let's create an instance of a House that uses rooms from two namespaces polymorphically:

Example 7.4:

```
<?soxtype urn:x-commerceone:document:sample:xdk:sox:House.sox$1.0?>

<House xmlns:room=
"urn:x-commerceone:document:sample:xdk:sox:Rooms.sox$1.0">
   <room:BedRoom>
      <room:Width>20</room:Width>
      <room:Length>12</room:Length>
      <room:Height>12</room:Height>
      <room:Carpet Length="medium"/>
      <room:Window Direction="west">
         <room:WindowType>sliding</room:WindowType>
      </room:Window>
      <room:Window Direction="north">
         <room:WindowType>sliding</room:WindowType>
      </room:Window>
      <room:Door Direction="south" DoorType="single"/>
      <room:Door Direction="east" DoorType="single"/>
      <room:Closet/>
      <room:Closet/>
   </room:BedRoom>
   <room:LivingRoom>
      <room:Width>20</room:Width>
      <room:Length>18</room:Length>
      <room:Height>12</room:Height>
      <room:Window Direction="south" MosquitoNet="false">
         <room:WindowType>opening</room:WindowType>
      </room:Window>
      <room:Window Direction="south" MosquitoNet="false">
         <room:WindowType>opening</room:WindowType>
      </room:Window>
      <room:Window Direction="east" MosquitoNet="false">
         <room:WindowType>opening</room:WindowType>
      </room:Window>
      <room:Door Direction="west" DoorType="double"></room:Door>
      <room:Door Direction="north" DoorType="double"></room:Door>
   </room:LivingRoom>
   <BathRoom>
      <room:Width>20</room:Width>
      <room:Length>18</room:Length>
      <room:Height>12</room:Height>
      <room:Door Direction="west" DoorType="single"/>
      <Facility>Toilet</Facility>
      <Facility>Sink</Facility>
      <Facility>Bathtub</Facility>
   </BathRoom>
</House>
```

As you can see in example 7.4, the Room objects from the two different namespaces, can mingle freely. The only thing that matters is that all present elements are valid, in that they directly or indirectly extend the specified element.

7.3 Using Polymorphic Elements that are Not Accessible from the Current Namespace

A more complicated, but by no means unusual, scenario for polymorphism, is when the polymorphic elementtypes that are used in the instance, are not defined in, nor imported to, the schema that is defined in the soxtype element. One reason why this would happen is when you use an existing schema, but want to extend an elementtype that is contained in the content model of that schema. Your extended element is not defined in the root schema, nor is it imported into it, but you still want to use the extended element polymorphically. In order to be able to do so, you have to import the schema with the extended definition into the instance, so that the software can access your new definition. This is done with the import element, which must come before the root of your schema:

Example 7.5:

```
<?import urn:x-commerceone:document:sample:xdk:sox:Sample.sox$1.0?>
```

To demonstrate the use of the import element, let's use the Ticket schemas from the previous section to create a Ticket Purchase Order:

Example 7.6:

```
<?xml version="1.0"?>
<!DOCTYPE schema SYSTEM
  "urn:x-commerceone:document:com:commerceone:xdk:xml:schema.dtd$1.0">

<schema uri=
"urn:x-commerceone:document:sample:xdk:sox:TicketPurchase.sox$1.0">

<namespace prefix="tick" namespace=
"urn:x-commerceone:document:sample:xdk:sox:Ticket.sox$1.0"/>

   <elementtype name="TicketPurchase">
      <model>
         <element type="Ticket" prefix="tick" occurs="+"/>
      </model>
   </elementtype>

</schema>
```

Now we have defined a schema that can take a number of Ticket instances, but since `Ticket` is hardly very useful in itself in a purchase order, we will want to use extended versions of Ticket that actually provide more detailed information as to what kind of `Ticket` is ordered.

Let's create a `TicketPurchase` instance that uses `MovieTickets` and `ConcertTickets` polymorphically. Since the `TicketPurchase` schema only knows about the `Ticket` schema, we will have to import the other two schemas in order to be able to access the definitions of the extended tickets:

Example 7.7:

```
<?soxtype
urn:x-commerceone:document:sample:xdk:sox:TicketPurchase.sox$1.0?>

<?import urn:x-commerceone:document:sample:xdk:sox:MovieTicket.sox$1.0?>
<?import
urn:x-commerceone:document:sample:xdk:sox:ConcertTicket.sox$1.0?>

<TicketPurchase xmlns:mov=
"urn:x-commerceone:document:sample:xdk:sox:MovieTicket.sox$1.0"
   xmlns:con=
"urn:x-commerceone:document:sample:xdk:sox:ConcertTicket.sox$1.0"
   xmlns:tic="urn:x-commerceone:document:sample:xdk:sox:Ticket.sox$1.0">

   <con:ConcertTicket PrePaid="false" Seating="General">
      <tic:Price>22.50</tic:Price>
      <tic:Time>19991231T18:00:00</tic:Time>
      <tic:Location>Times Square</tic:Location>
      <con:Band>Rolling Stones</con:Band>
      <con:OpeningAct>Willy Nelson</con:OpeningAct>
   </con:ConcertTicket>
   <mov:MovieTicket PrePaid="false">
      <tic:Price>6.50</tic:Price>
      <tic:Time>19990811T14:15:00</tic:Time>
      <tic:Location>Century 23</tic:Location>
      <mov:Title>Mystery Men</mov:Title>
      <mov:Screen>2</mov:Screen>
   </mov:MovieTicket>
</TicketPurchase>
```

As example 7.7 shows, you still have to assign a prefix for each namespace with an xmlns attribute. Even though you have explicitly imported the namespace, you still have to declare what namespace an element was declared in.

Appendix A: Intrinsic Datatypes

These are all of the intrinsic datatypes defined in SOX. Some have been defined in XML, and some have been added in SOX. They can be used as datatypes for attributes or elements, or be used in user defined datatypes.

boolean	A binary datatype. The valid values are either "true" or "false"
string	A text string with any number of characters.
URI	A Universal Resource Identifier which is essentially an address to some resource. Can for example be mapped to a file in the local file system, a URL on the internet. For more information on URI's, see the URI working draft: http://www.w3.org/Addressing/URL/URI_Overview.html
number	An infinite precision number which may be preceded by a "-" or "+", and which may contain one decimal point.
float	A single precision floating point number in the range -3.40282347*10E38 to 3.40282347*10E38.
double	A double precision floating point number in the range -1.17549435*10E308 to 1.17549435*10E308.
int	An integer in the range -2,147,483,648 to 2,147,483,647.
long	An integer in the range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
byte	An integer in the range -128 to 127.
ID	An ID is a way to give an object a unique identification in an instance. This means that an ID value in an instance cannot be equal to any other ID value in that instance. ID's are referred to by an IDREF or IDREFS object. See the XML 1.0 spec for more information.
IDREF	AN IDREf is used to refer to an ID object. An instance of an IDREF object have to contain a value that is identical to an ID value that exists in the same instance as the IDREF object.
IDREFS	One or several IDREF values separated by white space.
NMTOKEN	An NMTOKEN is a text string that has a limited set of allowed characters. Digits and letters are allowed, as well as period, ".", dash, "-", colon, ":", underscore, "_", as well as some other special characters. See the XML 1.0 specification for more information.
NMTOKENS	One or several NMTOKEN values separated by white space.
date	A date in the format of: YYYYMMDD. This means that the year comes first, specified with 4 digits, then a two-digit month, and last a two-digit day. This means that if the month or day only has one digit, then that digit should be preceded by a zero, "0", to make the value take up two digits. For example, 4/1/99, that is, the first of April 1999, would be expressed as: 19990401.

time	A time, specified with hours, minutes and seconds in the format HH:MM:SS. HH is the hours, expressed in two digits, MM is the minutes expressed in two digits, and SS is the seconds, expressed in two digits. The hours are specified in the range of 0-23, which means that the time is expressed in "military hours", tat is without am or pm. 9:07 am would be expressed as 09:07:00, whereas 9:07 pm would be 21:07:00. The time datatype also has the option of specifying an offset from GMT, in which case the format is HH:MM:SS(+/-)HH:MM. For example, if the time 5:03 pm is specified, being 4 hours ahead of GMT, then the value would be: 17:03:00+04:00
datetime	A date and time combined. The format is: YYYYMMDDTHH:MM:SS. First comes a date datatype value, next the character "T", which separates the date from the time, and then follows a time datatype value. As an example, 12:43:27 noon on the last of December 1999 would be expressed as: 19991231T12:43:27 As with the time datatype, an offset from GMT can be specified, in which case the format is: YYYYMMDDTHH:MM:SS(+/-)HH:MM. The previous example, 7 hours behind GMT, would be: 19991231T12:43:27-07:00

Appendix B: Glossary

? occurrence	An element, nested sequence or nested choice specifying an occurs value of "?" is optional and may appear once, or not at all.
*** occurrence**	An element, nested sequence or nested choice specifying an occurs value of "*" is optional and may appear multiple times.
+ occurrence	An element, nested sequence or nested choice specifying an occurs value of "+" must appear, but may appear more than once.
camel case	A naming style from java and C++, where a name will contain of one or several words. Each new word is started with a capital letter. An example would be: ThisIsACamelCasedName. It is a recommended naming style for SOX elements and datatypes.
choice content model	A content model of an elementtype that specifies a number of elements, one of which may appear in an instance of the elementtype.
current namespace	The namespace of the current schema.
cxp	Commerce One's SOX schema and XML parser
datatype definition	An element in a schema that defines a user-defined datatype. The datatype can be used in attribute definitions or element definitions.
decimals	An attribute used in scalar to specify the maximum number of allowed decimals
default	A presence specified for an attribute that has a default value. If the attribute is not present in the instance, the default value will be used instead.
default namespace	A namespace declared for an element in an instance, which defines the namespace for that element and all of its contents. This enables a user to avoid prefixes in cases where large elements are not in the current namespace.
digits	An attribute used in scalar to specify the maximum number of allowed digits (not including decimals)
DOCTYPE declaration	A tag identifies the document type of the current document. Must be present in all SOX schemas containing the keyword schema and the URI to the current schema.dtd
DTD	**D**ocument **T**ype **D**efinition. Defines a set of structures for XML documents.
elementtype	An element in a schema that defines an element structure to be used in an instance document.
element content model	A content model of an elementtype that specifies that only one type of element may be present in the instance of the elementtype.
element element	An element used in a content model to specify a content element. The type attribute on element specifies what type the element should be of in the instance, and can be of a datatype or element type.
empty content model	A content model of an elementtype that specifies that no content may be present in the instance of the elementtype.

enumeration	An element that specifies a user defined datatype that consist of an enumeration of valid values.
fixed	A presence specified for an attribute that has a fixed value. No other value may be specified in the instance.
implied	A presence specified for an attribute that is optional in the instance.
inheritance	A feature of SOX allowing an elementtype to derive structure from a previously defined elementtype.
instance document	An XML instance of a SOX schema. The instance is written in XML format and must conform to a schema or a set of schemas. It may only have one root element.
maxlength	An attribute used in varchar to specify the maximum length of a string in the instance.
maxexclusive	An attribute used in scalar to specify if the maximum value specified is or is not allowed in the instance.
maxvalue	An attribute used in scalar to specify the maximum value allowed in the instance.
minvalue	An attribute used in scalar to specify the minimum value allowed in the instance.
minexclusive	An attribute used in scalar to specify if the minimum value specified is or is not allowed in the instance.
namespace	A unique identifier of a schema. Since all schemas reside in their own namespace that is separate from all other namespaces, they can refer to other namespaces without any name collisions.
nested choice	A choice element that is nested inside another sequence or choice element.
nested sequence	A sequence element that is nested inside another sequence or choice element.
N,M occurrence	An element, nested sequence or nested choice specifying an occurs value of N,M specifies the valid range of occurrences of that object.
option	An element used in an enumeration element to specify a valid value for the enumeration.
parser	A processor that parses documents.
polymorphism	Subtypes of elements can appear in an instance whenever the presence of their supertype is specified.
prefix	A name associated with a namespace, used with an object in a schema or an instance to specify that that object is from the associated namespace. In the schema the prefix is used as the value for the prefix attribute, in the instance the prefix is pre-pended to the name of an element, separated from the element with a colon, ":".
required	A presence specified for an attribute that must appear in the instance.
root element	The outermost set of tags in an XML document that contains all other tags. There may only be one root element in each document, except for any version and document type elements.

scalar	An element that specifies a user defined datatype that defines a number type datatype with various constraints.
schema.dtd	A DTD that all SOX schemas must conform to.
schema element	The root element inside a SOX schema that defines that schema's namespace, as well as wraps all definitions in that schema.
sequence content model	A content model of an elementtype that constrains the content of the elementtype's instance to the specified sequence. The elements must appear in the correct sequence, with all required elements present.
SOX	A schema language expressed in XML format that provides a more powerful way of defining an XML structure than a DTD. Some of the features that make it more advantageous are: a large number of datatypes that can be used both in attributes and element content, inheritance and polymorphism.
SOX 2.0 specification	A language specification of the current version of the SOX language. Can be found at: http://www.w3.org/TR/NOTE-SOX/
SOX schema	Defines structure rules in the form of elementtype definitions and datatype definitions. The schema is written in XML format and conforms to a DTD called "schema.dtd".
soxtype declaration	A tag that must be present in all XML instances of SOX schemas. It consists of a processing instruction that identifies the document as being an instance of a SOX schema.
string content model	A content model of an elementtype that specifies that only text content may be present in the instance of the elementtype.
occurs	A way to specify how many times an element, nested sequence or nested choice may appear in a content model.
URI	**U**niform **R**esource **I**dentifier. Consists of an address to a resource. How that address is resolved depends on the specific scheme. See the current URI working draft for more information: http://www.w3.org/Addressing/URL/URI_Overview.html
valid	Meaning that the document is conforms to the constraints specified in the document it claims to conform to. That document could for example be a DTD or a SOX schema. See the XML 1.0 specification or the SOX 2.0 specification for more details on validity.
varchar	An element that specifies a user defined datatype that specifies a string datatype with a maximum length.
well-formed	Meaning that an XML document conforms to the well-formedness constraints set forth in the XML 1.0 specification.
XML	E**x**tensible **M**ark-up **L**anguage. A mark-up language ideal for storing data in a human readable form. Provides means to define tags and structures which enables a highly customizable way of storing data.

XML 1.0 specification	A language specification of the current version of the XML language. Can be found at: http://www.w3.org/TR/REC-xml
XML version tag	A tag that identifies the current version of XML. Sample format is: `<?xml version="1.0"?>` This tag must be present in all documents that consist of XML content that should be parseable by an XML processor. It must be present in SOX schemas.