XML Schema C++/Tree Mapping User Manual

Revision 1.2.3

July 2006

Table of Contents

Preface

About This Document

This document describes the mapping of W3C XML Schema to the C++ programming language as implemented by CodeSynthesis XSD - an XML Schema to C++ data binding compiler. The mapping represents information stored in XML instance documents as a statically-typed, tree-like in-memory data structure.

Revision 1.2.3
This revision of the manual describes the C++/Tree mapping as implemented by CodeSynthesis XSD version 2.2.0.

This document is available in the following formats: XHTML, PDF, and PostScript.

Copyright and License

Acknowledgements

We would like to thank Karen Arutyunov for his comments on the drafts of this manual.

1 Introduction

Vocabulary-independent APIs for accessing information stored in XML instance documents such as DOM and SAX suffer from a number of drawbacks:

- generic representation of XML instances: elements, attributes, and text
- dynamic typing which leads to errors occurring at run-time rather than at compile-time
- writing non-trivial code using such APIs is a daunting task
- the resulting applications are hard to debug, change, and maintain

In contrast, statically-typed, vocabulary-specific mappings allow you to operate in your domain terms instead of the generic terms of DOM or SAX. Static typing will help you catch errors at compile-time rather than at run-time. Automatic generation of code will free you for more interesting tasks (like doing something useful with the information stored in the instance document) and minimize the effort needed to adopt your applications to changes in the document structure.

A typical application that deals with XML instance documents usually performs the following three steps: it first reads (parses) an XML instance document to an in-memory representation, it then performs some useful computations on that representation which may involve modification of the representation, and finally it may write (serialize) the modified in-memory representation back to XML.

The C++/Tree mapping consists of data types that represent the given vocabulary (Chapter 2, "C++/Tree Mapping"), a set of parsing functions that convert XML instance documents to a tree-like in-memory data structure (Chapter 3, "Parsing"), and a set of serialization functions that convert the in-memory representation back to XML (Chapter 4, "Serialization").

The following section shows how to create a simple application that uses the C++/Tree mapping to parse an XML instance document and then access the resulting in-memory representation.

1.1 Hello World

The following step-by-step guide is based on the *hello* example which can be found in the `examples/cxx/tree/hello` directory of the XSD distribution.

1.1.1 Writing XML Document and Schema Definition

The first thing we need to do is to get an idea about the structure of XML instance documents we are going to process. Our `hello.xml`, for example, could look like this:

```
<?xml version="1.0"?>
<hello xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="hello.xsd">

  <greeting>Hello</greeting>

  <name>sun</name>
  <name>earth</name>
  <name>world</name>

</hello>
```

Then, we can write a Schema definition for the above instance and save it into `hello.xsd`:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:complexType name="hello_type">
    <xsd:sequence>
      <xsd:element name="greeting" type="xsd:string"/>
      <xsd:element name="name" type="xsd:string" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
```

```
  <xsd:element name="hello" type="hello_type"/>

</xsd:schema>
```

1.1.2 Translating Schema Definition to C++

Now we are ready to translate our Schema definition to C++ classes and functions that represent our vocabulary:

```
$ xsd cxx-tree hello.xsd
```

The Schema compiler produces two C++ files: `hello.hxx` and `hello.cxx`. The following code fragment is an approximation of what can be found in `hello.hxx`; it should give you an idea about what gets generated:

```
class hello_type
{
public:
  // greeting
  //
  const string&
  greeting () const;

  string&
  greeting ();

  void
  greeting (const string&);

  // name
  //
  struct name
  {
    typedef sequence<string> container;
    typedef container::iterator iterator;
    typedef container::const_iterator const_iterator;
  };

  const name::container&
  name () const;

  name::container&
  name ();

  void
  name (const name::container&);
```

```
};

std::auto_ptr<hello_type>
hello (const string& uri);
```

1.1.3 Implementing Application Logic

At this point we have all the parts we need to do something useful with the information stored in instance documents (`driver.cxx`):

```
#include "hello.hxx"

int
main (int argc, char* argv[])
{
  auto_ptr<hello_type> h (hello (argv[1]));

  for (hello_type::name::const_iterator i (h->name ().begin ());
       i != h->name ().end ();
       ++i)
  {
    cerr << h->greeting () << ", " << *i << "!" << endl;
  }
}
```

1.1.4 Compiling and Running

Finally, we can compile our application and run it on the instance document:

```
$ g++ -o driver driver.cxx hello.cxx -lxerces-c
$ ./driver hello.xml
Hello, sun!
Hello, moon!
Hello, world!
```

2 C++/Tree Mapping

2.1 Preliminary Information

2.1.1 Identifiers

XML Schema names may happen to be reserved C++ keywords or contain characters that are illegal in C++ identifiers. To avoid C++ compilation problems, such names are changed (escaped) when mapped to C++. If an XML Schema name is a C++ keyword, the "_" suffix is added to it. All character of an XML Schema name that are not allowed in C++ identifiers are replaced with "_".

For example, XML Schema name `try` would be mapped to C++ identifier `try_`. Similarly, XML Schema name `strange.na-me` would be mapped to C++ identifier `strange_na_me`.

2.1.2 Character Type

The code that implements the mapping, depending on the `--char-type` option, is generated using either `char` or `wchar_t` as the character type. In this document code samples use symbol `C` to refer to the character type you have selected when translating your schemas, for example `std::basic_string<C>`.

2.1.3 XML Schema Namespace

The mapping relies on some predefined types, classes, and functions that are logically defined in the XML Schema namespace reserved for the XML Schema language (`http://www.w3.org/2001/XMLSchema`). By default, this namespace is mapped to C++ namespace `xml_schema`. It is automatically accessible from a C++ compilation unit that includes a header file generated from an XML Schema definition.

Note that, if desired, the default mapping of this namespace can be changed as described in Section 2.4, "Mapping for Namespaces".

2.2 Error Handling

The mapping uses the C++ exception handling mechanism as a primary way of reporting error conditions. All exceptions that are specified in this mapping derive from `xml_schema::exception` which itself is derived from `std::exception`:

```
struct exception: virtual std::exception
{
  friend
  std::basic_ostream<C>&
  operator<< (std::basic_ostream<C>& os, const exception& e)
  {
    e.print (os);
    return os;
  }

protected:
  virtual void
  print (std::basic_ostream<C>&) const = 0;
};
```

The exception hierarchy supports "virtual" `operator<<` which allows you to obtain diagnostics corresponding to the thrown exception using the base exception interface. For example:

```
try
{
  ...
}
catch (const xml_schema::exception& e)
{
  cerr << e << endl;
}
```

The following sub-sections describe exceptions thrown by the types that constitute the tree representation. Section 3.3, "Error Handling" of Chapter 3, "Parsing" describes exceptions and error handling mechanisms specific to the parsing functions. Section 4.4, "Error Handling" of Chapter 4, "Serialization" describes exceptions and error handling mechanisms specific to the serialization functions.

2.2.1 `xml_schema::duplicate_id`

```
struct duplicate_id: virtual exception
{
  duplicate_id (const std::basic_string<C>& id);

  const std::basic_string<C>&
  id () const;

  virtual const char*
  what () const throw ();
};
```

The `xml_schema::duplicate_id` is thrown when a conflicting instance of `xml_schema::id` (see Section 2.5, "Mapping for Built-in Data Types") is added to a tree. The offending ID value can be obtained using the `id` function.

2.3 Mapping for `import` and `include`

2.3.1 Import

The XML Schema `import` element is mapped to the C++ Preprocessor `#include` directive. The value of the `schemaLocation` attribute is used to derive the name of the header file that appears in the `#include` directive. For instance:

```
<import namespace="http://www.codesynthesis.com/test"
        schemaLocation="test.xsd"/>
```

is mapped to:

```
#include "test.hxx"
```

Note that you will need to compile imported schemas separately in order to produce corresponding header files.

2.3.2 Inclusion with Target Namespace

The XML Schema `include` element which refers to a schema with a target namespace or appears in a schema without a target namespace follows the same mapping rules as the `import` element, see Section 2.3.1, "Import".

2.3.3 Inclusion without Target Namespace

For the XML Schema `include` element which refers to a schema without a target namespace and appears in a schema with a target namespace (such inclusion sometimes called "chameleon inclusion"), declarations and definitions from the included schema are generated in-line in the namespace of the including schema as if they were declared and defined there verbatim. For example, consider the following two schemas:

```
<-- common.xsd -->
<schema>
  <complexType name="type">
  ...
  </complexType>
</schema>

<-- test.xsd -->
<schema targetNamespace="http://www.codesynthesis.com/test">
  <include schemaLocation="common.xsd"/>
</schema>
```

The fragment of interest from the generated header file for `text.xsd` would look like this:

```
// test.hxx
namespace test
{
  class type
  {
    ...
  };
}
```

2.4 Mapping for Namespaces

An XML Schema namespace is mapped to one or more nested C++ namespaces. XML Schema namespaces are identified by URIs. By default, a namespace URI is mapped to a sequence of C++ namespace names by removing the protocol and host parts and splitting the rest into a sequence of names with '/' as the name separator. For instance:

```
<schema targetNamespace="http://www.codesynthesis.com/system/test">
  ...
</schema>
```

is mapped to:

```
namespace system
{
  namespace test
  {
    ...
  }
}
```

The default mapping of namespace URIs to C++ namespace names can be altered using the `--namespace-regex` option. See the compiler manual for more information.

2.5 Mapping for Built-in Data Types

The mapping of XML Schema built-in data types to C++ types is summarized in the table below.

XML Schema type	Alias in the `xml_schema` namespace	C++ type
anyType and anySimpleType types		
`anyType`	`type`	Section 2.5.2, "Mapping for `anyType`"
`anySimpleType`	`simple_type`	Section 2.5.3, "Mapping for `anySimpleType`"
fixed-length integral types		
`byte`	`byte`	`signed char`
`unsignedByte`	`unsigned_byte`	`unsigned char`
`short`	`short_`	`short`
`unsignedShort`	`unsigned_short`	`unsigned short`
`int`	`int_`	`int`

unsignedInt	unsigned_int	unsigned int
long	long_	long long
unsignedLong	unsigned_long	unsigned long long
arbitrary-length integral types		
integer	integer	long long
nonPositiveInteger	non_positive_integer	long long
nonNegativeInteger	non_negative_integer	long long
positiveInteger	positive_integer	long long
negativeInteger	negative_integer	long long
boolean types		
boolean	boolean	bool
fixed-length floating-point types		
float	float_	float
double	double_	double
arbitrary-length floating-point types		
decimal	decimal	long double
string types		
string	string	type derived from std::basic_string
normalizedString	normalized_string	type derived from string
token	token	type derived from normalized_string
Name	name	type derived from token
NMTOKEN	nmtoken	type derived from token
NMTOKENS	nmtokens	type derived from sequence<nmtoken>
NCName	ncname	type derived from name
language	language	type derived from token
qualified name		
QName	qname	Section 2.5.4, "Mapping for QName"
ID/IDREF types		
ID	id	type derived from ncname

IDREF	idref	Section 2.5.5, "Mapping for IDREF"
IDREFS	idrefs	type derived from sequence<idref>
URI types		
anyURI	uri	type derived from std::basic_string
binary types		
base64Binary	base64_binary	Section 2.5.6, "Mapping for base64Binary and hexBinary"
hexBinary	hex_binary	
date/time types		
date	date	type derived from std::basic_string
dateTime	date_time	type derived from std::basic_string
duration	duration	type derived from std::basic_string
gDay	day	type derived from std::basic_string
gMonth	month	type derived from std::basic_string
gMonthDay	month_day	type derived from std::basic_string
gYear	year	type derived from std::basic_string
gYearMonth	year_month	type derived from std::basic_string
time	time	type derived from std::basic_string
entity types		
ENTITY	entity	type derived from name
ENTITIES	entities	type derived from sequence<entity>

All XML Schema built-in types are mapped to C++ classes that are derived from the xml_schema::simple_type class except where the mapping is to a fundamental C++ type.

The sequence class template is defined in an implementation-specific namespace. It conforms to the sequence interface as defined by the ISO/ANSI Standard for C++ (ISO/IEC 14882:1998, Section 23.1.1, "Sequences"). Practically, this means that you can treat such a sequence as if it was std::vector.

2.5.1 Inheritance from Built-in Data Types

In cases where the mapping calls for an inheritance from a built-in type which is mapped to a fundamental C++ type, a proxy type is used instead of the fundamental C++ type (C++ does not allow inheritance from fundamental types). For instance:

```
<simpleType name="my_int">
  <restriction base="int"/>
</simpleType>
```

is mapped to:

```
class my_int: public fundamental_base<int>
{
  ...
};
```

The `fundamental_base` class template provides a close emulation (though not exact) of a fundamental C++ type. It is defined in an implementation-specific namespace and has the following interface:

```
template <typename X>
class fundamental_base: public simple_type
{
public:
  fundamental_base ();
  fundamental_base (X)
  fundamental_base (const fundamental_base&)

public:
  fundamental_base&
  operator= (const X&);

public:
  operator const X & () const;
  operator X& ();

  template <typename Y>
  operator Y () const;

  template <typename Y>
  operator Y () const;
};
```

2.5.2 Mapping for `anyType`

The XML Schema `anyType` built-in data type is mapped to the `xml_schema::type` C++ class:

```
class type
{
public:
  virtual
  ~type ();

public:
  type ();
  type (const type&);

public:
  type&
  operator= (const type&);

public:
  virtual type*
  _clone () const;

  // DOM association.
  //
public:
  const xercesc::DOMNode*
  _node () const;

  xercesc::DOMNode*
  _node ();
};
```

2.5.3 Mapping for `anySimpleType`

The XML Schema `anySimpleType` built-in data type is mapped to the `xml_schema::simple_type` C++ class:

```
class simple_type: public type
{
public:
  simple_type ();
  simple_type (const simple_type&);

public:
  simple_type&
  operator= (const simple_type&);
```

```
public:
  virtual simple_type*
  _clone () const;
};
```

2.5.4 Mapping for `QName`

The XML Schema `QName` built-in data type is mapped to the `xml_schema::qname` C++ class:

```
class qname: public simple_type
{
public:
  qname (const uri&, const ncname&);
  qname (const qname&);

public:
  qname&
  operator= (const qname&);

public:
  virtual qname*
  _clone () const;

public:
  const uri&
  namespace_ () const;

  const ncname&
  name () const;
};
```

2.5.5 Mapping for `IDREF`

The XML Schema `IDREF` built-in data type is mapped to the `xml_schema::idref` C++ class. This class implements the smart pointer C++ idiom:

```
class idref: public ncname
{
public:
  idref (const C* s);
  idref (const C* s, std::size_t n);
  idref (std::size_t n, C c);
  idref (const std::basic_string<C>&);
  idref (const std::basic_string<C>&,
         std::size_t pos,
         std::size_t n = npos);

public:
```

```
  idref (const idref&);

public:
  virtual idref*
  _clone () const;

public:
  idref&
  operator= (C c);

  idref&
  operator= (const C* s);

  idref&
  operator= (const std::basic_string<C>&)

  idref&
  operator= (const idref&);

public:
  const type*
  operator-> () const;

  type*
  operator-> ();

  const type&
  operator* () const;

  type&
  operator* ();

  const type*
  get () const;

  type*
  get ();

  // Conversion to bool.
  //
public:
  typedef void (idref::*bool_convertible)();
  operator bool_convertible () const;
};
```

The object, `idref` instance refers to, is the immediate container of the matching `id` instance. For example, with the following instance document and schema:

```
<!-- test.xml -->
<root>
  <object id="obj-1" text="hello"/>
  <reference>obj-1</reference>
</root>

<!-- test.xsd -->
<schema>
  <complexType name="object_type">
    <attribute name="id" type="ID"/>
    <attribute name="text" type="string"/>
  </complexType>

  <complexType name="root_type">
    <sequence>
      <element name="object" type="object_type"/>
      <element name="reference" type="IDREF"/>
    </sequence>
  </complexType>

  <element name="root" type="root_type"/>
</schema>
```

The `ref` instance in the code below will refer to an object of type `object_type`:

```
root_type& root = ...;
xml_schema::idref& ref (root.reference ());
object_type& obj (dynamic_cast<object_type&> (*ref));
cout << obj.text () << endl;
```

The smart pointer interface of the `idref` class always returns a pointer or reference to `xml_schema::type`. This means that you will need to manually cast such pointer or reference to its real (dynamic) type before you can use it (unless all you need is the base interface provided by `xml_schema::type`). As a special extension to the XML Schema language, the mapping supports static typing of `idref` references by employing the `refType` extension attribute. The following example illustrates this mechanism:

```
<!-- test.xsd -->
<schema
  xmlns:xse="http://www.codesynthesis.com/xmlns/xml-schema-extension">

  ...

      <element name="reference" type="IDREF" xse:refType="object_type"/>

  ...

</schema>
```

With this modification we do not need to do manual casting anymore:

```
root_type& root = ...;
root_type::reference::type& ref (root.reference ());
object_type& obj (*ref);
cout << ref->text () << endl;
```

2.5.6 Mapping for `base64Binary` and `hexBinary`

The XML Schema `base64Binary` and `hexBinary` built-in data types are mapped to the `xml_schema::base64_binary` and `xml_schema::hex_binary` C++ classes, respectively. The `base64_binary` and `hex_binary` classes support a simple buffer abstraction by inheriting from the `xml_schema::buffer` class:

```
class bounds: public virtual exception
{
public:
  virtual const char*
  what () const throw ();
};

class buffer
{
public:
  typedef std::size_t size_t;

public:
  buffer (size_t size = 0);
  buffer (size_t size, size_t capacity);
  buffer (const void* data, size_t size);
  buffer (const void* data, size_t size, size_t capacity);
  buffer (void* data,
          size_t size,
          size_t capacity,
          bool assume_ownership);

public:
  buffer (const buffer&);

  buffer&
  operator= (const buffer&);

  void
  swap (buffer&);

public:
  size_t
  capacity () const;

  bool
```

```
  capacity (size_t);

public:
  size_t
  size () const;

  bool
  size (size_t);

public:
  const char*
  data () const;

  char*
  data ();

  const char*
  begin () const;

  char*
  begin ();

  const char*
  end () const;

  char*
  end ();
};
```

If the `assume_ownership` argument to the constructor is `true`, the instance assumes the ownership of the memory block pointed to by the `data` argument and will eventually release it by calling `operator delete`. The `capacity` and `size` modifier functions return `true` if the underlying buffer has moved.

The `bounds` exception is thrown if the constructor arguments violate the `(size <= capacity)` constraint.

The `base64_binary` and `hex_binary` classes support the `buffer` interface and perform automatic decoding/encoding from/to the Base64 and Hex formats, respectively:

```
class base64_binary: public simple_type, public buffer
{
public:
  base64_binary (size_t size = 0);
  base64_binary (size_t size, size_t capacity);
  base64_binary (const void* data, size_t size);
  base64_binary (const void* data, size_t size, size_t capacity);
  base64_binary (void* data,
                 size_t size,
                 size_t capacity,
```

```
                bool assume_ownership);

public:
  base64_binary (const base64_binary&);

  base64_binary&
  operator= (const base64_binary&);

  virtual base64_binary*
  _clone () const;

public:
  std::basic_string<C>
  encode () const;
};

class hex_binary: public simple_type, public buffer
{
public:
  hex_binary (size_t size = 0);
  hex_binary (size_t size, size_t capacity);
  hex_binary (const void* data, size_t size);
  hex_binary (const void* data, size_t size, size_t capacity);
  hex_binary (void* data,
              size_t size,
              size_t capacity,
              bool assume_ownership);

public:
  hex_binary (const hex_binary&);

  hex_binary&
  operator= (const hex_binary&);

  virtual hex_binary*
  _clone () const;

public:
  std::basic_string<C>
  encode () const;
};
```

2.6 Mapping for Simple Types

An XML Schema simple type is mapped to a C++ class with the same name as the simple type. The class defines a public copy constructor, a public copy assignment operator, and a public virtual `_clone` function. The `_clone` function is declared `const`, does not take any arguments, and returns a pointer to a complete copy of the instance allocated in the free store. The `_clone` function shall be used to make copies when static type and dynamic type of the instance

may differ (see Section 2.12, "Mapping for `xsi:type` and Substitution Groups"). For instance:

```
<simpleType name="object">
  ...
</simpleType>
```

is mapped to:

```
class object: ...
{
public:
  object (const object&);

public:
  object&
  operator= (const object&);

public:
  virtual object*
  _clone () const;

  ...

};
```

The base class specification and the rest of the class definition depend on the type of derivation used to define the simple type.

2.6.1 Mapping for Derivation by Restriction

XML Schema derivation by restriction is mapped to C++ public inheritance. The base type of the restriction becomes the base type for the resulting C++ class. In addition to the members described in Section 2.6, "Mapping for Simple Types", the resulting C++ class defined a public constructor with the base type as its single argument. For instance:

```
<simpleType name="object">
  <restriction base="base">
    ...
  </restriction>
</simpleType>
```

is mapped to:

```
class object: public base
{
public:
  object (const base&);
  object (const object&);
```

```
public:
  object&
  operator= (const object&);

public:
  virtual object*
  _clone () const;
};
```

2.6.2 Mapping for Enumerations

XML Schema restriction by enumeration is mapped to a C++ class which behaves similar to C++ `enum`. Each XML Schema enumeration element is mapped to a C++ enumerator with the name derived from the `value` attribute and defined in the class scope. In addition to the members described in Section 2.6, "Mapping for Simple Types", the resulting C++ class defined a public constructor that can be called with one of the enumerators as its single argument, a public assignment operator that can be used to assign the value of one of the enumerators, and a public implicit conversion operator to an implementation-specific integral type. For instance:

```
<simpleType name="color">
  <restriction base="string">
    <enumeration value="red"/>
    <enumeration value="green"/>
    <enumeration value="blue"/>
  </restriction>
</simpleType>
```

is mapped to:

```
class color
{
public:
  enum <implementation-specific name>
  {
    red,
    green,
    blue
  };

public:
  color (<implementation-specific name>);
  color (const color&);

public:
  color&
  operator= (<implementation-specific name>);

  color&
  operator= (const color&);
```

```
public:
  virtual color*
  _clone () const;

public:
  operator <implementation-specific name> () const;
};
```

2.6.3 Mapping for Derivation by List

XML Schema derivation by list is mapped to C++ public inheritance from `xml_schema::simple_type` (Section 2.5.3, "Mapping for `anySimpleType`") and a suitable sequence type. The list item type becomes the element type of the sequence. In addition to the members described in Section 2.6, "Mapping for Simple Types", the resulting C++ class defines a public default constructor. For instance:

```
<simpleType name="int_list">
  <list itemType="int"/>
</simpleType>
```

is mapped to:

```
class int_list: public simple_type,
                public sequence<int>
{
public:
  int_list ();
  int_list (const int_list&);

public:
  int_list&
  operator= (const int_list&);

public:
  virtual int_list*
  _clone () const;
};
```

The `sequence` class template is defined in an implementation-specific namespace. It conforms to the sequence interface as defined by the ISO/ANSI Standard for C++ (ISO/IEC 14882:1998, Section 23.1.1, "Sequences"). Practically, this means that you can treat such a sequence as if it was `std::vector`.

2.6.4 Mapping for Derivation by Union

XML Schema derivation by union is mapped to C++ public inheritance from `xml_schema::simple_type` (Section 2.5.3, "Mapping for `anySimpleType`") and `std::basic_string<C>`. In addition to the members described in Section 2.6, "Mapping for Simple Types", the resulting C++ class defines a public constructor with a single argument of type `const std::basic_string<C>&`. For instance:

```
<simpleType name="int_string_union">
  <xsd:union memberTypes="xsd:int xsd:string"/>
</simpleType>
```

is mapped to:

```
class int_string_union: public simple_type,
                        public std::basic_string<C>
{
public:
  int_string_union (const std::basic_string<C>&);
  int_string_union (const int_string_union&);

public:
  int_string_union&
  operator= (const int_string_union&);

public:
  virtual int_string_union*
  _clone () const;
};
```

2.7 Mapping for Complex Types

An XML Schema complex type is mapped to a C++ class with the same name as the complex type. The class defines a public copy constructor, a public copy assignment operator, and a public virtual `_clone` function. The `_clone` function is declared `const`, does not take any arguments, and returns a pointer to a complete copy of the instance allocated in the free store. The `_clone` function shall be used to make copies when static type and dynamic type of the instance may differ (see Section 2.12, "Mapping for `xsi:type` and Substitution Groups"). Additionally, the resulting C++ class defines a public constructor that takes an initializer for each member of the complex type and all its base types that belongs to the One cardinality class (see Section 2.8, "Mapping for Local Elements and Attributes"). For instance:

```
<complexType name="object">
  <sequence>
    <element name="one" type="boolean"/>
    <element name="optional" type="int" minOccurs="0"/>
    <element name="sequence" type="string" maxOccurs="unbounded"/>
  </sequence>
</complexType>
```

is mapped to:

```
class object: xml_schema::type
{
public:
  object (const bool& one);
  object (const object&);

public:
  object&
  operator= (const object&);

public:
  virtual object*
  _clone () const;

  ...

};
```

If an XML Schema complex type is not explicitly derived from any type, the resulting C++ class is derived from `xml_schema::type`. In cases where an XML Schema complex type is defined using derivation by extension or restriction, the resulting C++ base class specification depends on the type of derivation and is described in the subsequent sections.

The mapping for elements and attributes that are defined in a complex type is described in Section 2.8, "Mapping for Local Elements and Attributes".

2.7.1 Mapping for Derivation by Extension

XML Schema derivation by extension is mapped to C++ public inheritance. The base type of the extension becomes the base type for the resulting C++ class.

2.7.2 Mapping for Derivation by Restriction

XML Schema derivation by restriction is mapped to C++ public inheritance. The base type of the restriction becomes the base type for the resulting C++ class. XML Schema elements and attributes defined withing restriction do not result in any definitions in the resulting C++ class. Instead, corresponding (unrestricted) definitions are inherited from the base class. In the future

versions of this mapping, such elements and attributes may result in redefinitions of accessors and modifiers to reflect their restricted semantics.

2.8 Mapping for Local Elements and Attributes

XML Schema element and attribute definitions are called local if they appear withing a complex type definition, an element group definition, or an attribute group definitions.

Local XML Schema element and attribute definitions have the same C++ mapping. Therefore, in this section, local elements and attributes are collectively called members.

While there are many different member cardinality combinations (determined by the `use` attribute for attributes and the `minOccurs` and `maxOccurs` attributes for elements), the mapping divides all possible cardinality combinations into three cardinality classes:

one
- attributes: `use == "required"`
- attributes: `use == "optional"` and has default or fixed value
- elements: `minOccurs == "1"` and `maxOccurs == "1"`

optional
- attributes: `use == "optional"` and doesn't have default or fixed value
- elements: `minOccurs == "0"` and `maxOccurs == "1"`

sequence
- elements: `maxOccurs > "1"`

An optional attribute with a default or fixed value acquires this value if the attribute hasn't been specified in an instance document (see Appendix A, "Default and Fixed Values"). This mapping places such optional attributes to the One cardinality class.

A member is mapped to a public information scope and a set of public accessor and modifier functions. The information scope is a C++ class-scope with the same name as the member. The main purpose of information scopes is to capture type information about members. For example:

```
<complexType name="object">
  <sequence>
    <element name="member" type="int"/>
  </sequence>
</complexType>
```

is mapped to:

```
class object: xml_schema::type
{
public:
  struct member
```

```
  {
    ...
  };

  ...

};
```

The accessor and modifier functions have the same name as the member.

The content of the information scope and signatures of the accessor and modifier functions depend on the member's cardinality class and are described in the following sub-sections.

2.8.1 Mapping for Members with the One Cardinality Class

For the One cardinality class, the information scope contains an alias of the member's type with the name `type` (or `type_` if the member itself is named `type`).

The accessor functions come in constant and non-constant versions. The constant accessor function returns a constant reference to the member and can be used for read-only access. The non-constant version returns an unrestricted reference to the member and can be used for read-write access.

The modifier function expects an argument of type reference to constant of the member's type. The modifier function performs a deep-copy of its argument. For instance:

```
<complexType name="object">
  <sequence>
    <element name="member" type="int"/>
  </sequence>
</complexType>
```

is mapped to:

```
class object: xml_schema::type
{
public:
  // Information scope.
  //
  struct member
  {
    typedef int type;
  };

  // Accessors.
  //
  const int&
  member () const;
```

```
  int&
  member ();

  // Modifier.
  //
  void
  member (const int&);

  ...

};
```

The following code shows how one could use this mapping:

```
void
f (object& o)
{
  int i (o.member ());                    // get
  object::member::type ii (o.member ()); // get

  o.member (2);                           // set
  o.member () = 3;                        // set
}
```

2.8.2 Mapping for Members with the Optional Cardinality Class

For the Optional cardinality class, the information scope contains an alias of the member's type with the name `type` (or `type_` if the member itself is named `type`) and an alias of the container type with the name `container` (or `container_` if the member itself is named `container`).

Unlike accessor functions for the One cardinality class, accessor functions for the Optional cardinality class return references to corresponding containers rather than directly to members. The accessor functions come in constant and non-constant versions. The constant accessor function returns a constant reference to the container and can be used for read-only access. The non-constant version returns an unrestricted reference to the container and can be used for read-write access.

The modifier functions are overloaded for the member's type and the container type. The first modifier function expects an argument of type reference to constant of the member's type. The second expects an argument of type reference to constant of the container type. The modifier functions perform a deep-copy of its argument. For instance:

```
<complexType name="object">
  <sequence>
    <element name="member" type="int" minOccurs="0"/>
  </sequence>
</complexType>
```

is mapped to:

```
class object: xml_schema::type
{
public:
  // Information scope.
  //
  struct member
  {
    typedef int type;
    typedef optional<type> container;
  };

  // Accessors.
  //
  const member::container&
  member () const;

  member::container&
  member ();

  // Modifiers.
  //
  void
  member (const int&);

  void
  member (const member::container&);

  ...

};
```

The `optional` class template is defined in an implementation-specific namespace and has the following interface:

```
template <typename X>
class optional
{
public:
  optional ();

  explicit
  optional (const X&);
```

```
  optional (const optional&);

public:
  optional&
  operator= (const X&);

  optional&
  operator= (const optional&);

  // Pointer-like interface.
  //
public:
  const X*
  operator-> () const;

  X*
  operator-> ();

  const X&
  operator* () const;

  X&
  operator* ();

  typedef void (optional::*bool_convertible) ();
  operator bool_convertible () const;

  // Get/set interface.
  //
public:
  bool
  present () const;

  const X&
  get () const;

  X&
  get ();

  void
  set (const X& y);

  void
  reset ();
};

template <typename X>
bool
operator== (const optional<X>&, const optional<X>&);

template <typename X>
```

```
bool
operator!= (const optional<X>&, const optional<X>&);

template <typename X>
bool
operator< (const optional<X>&, const optional<X>&);

template <typename X>
bool
operator> (const optional<X>&, const optional<X>&);

template <typename X>
bool
operator<= (const optional<X>&, const optional<X>&);

template <typename X>
bool
operator>= (const optional<X>&, const optional<X>&);
```

The following code shows how one could use this mapping:

```
void
f (object& o)
{
  if (o.member ().present ())                    // test
  {
    int i (o.member ().get ());                  // get
    o.member (2);                                // set
    o.member ().set (3);                         // set
    o.member ().reset ();                        // reset
  }

  // Same as above but using pointer notation:
  //
  if (o.member ())                               // test
  {
    int i (*o.member ());                        // get
    o.member (3);                                // set
    *o.member () = 3;                            // set
    o.member () = object::member::container ();  // reset
  }
}
```

2.8.3 Mapping for Members with the Sequence Cardinality Class

For the Sequence cardinality class, the information scope contains an alias of the member's type with the name `type` (or `type_` if the member itself is named `type`), an alias of the container type with the name `container` (or `container_` if the member itself is named `container`), an alias of the iterator type with the name `iterator` (or `iterator_` if the member itself is named `iterator`), and an alias of the constant iterator type with the name `const_iterator`

(or `const_iterator_` if the member itself is named `const_iterator`).

The accessor functions come in constant and non-constant versions. The constant accessor function returns a constant reference to the container and can be used for read-only access. The non-constant version returns an unrestricted reference to the container and can be used for read-write access.

The modifier function expects an argument of type reference to constant of the container type. The modifier function performs a deep-copy of its argument. For instance:

```
<complexType name="object">
  <sequence>
    <element name="member" type="int" minOccurs="unbounded"/>
  </sequence>
</complexType>
```

is mapped to:

```
class object: xml_schema::type
{
public:
  // Information scope.
  //
  struct member
  {
    typedef int type;
    typedef sequence<type> container;
    typedef container::iterator iterator;
    typedef container::const_iterator const_iterator;
  };

  // Accessors.
  //
  const member::container&
  member () const;

  member::container&
  member ();

  // Modifiers.
  //
  void
  member (const member::container&);

  ...

};
```

The `sequence` class template is defined in an implementation-specific namespace. It conforms to the sequence interface as defined by the ISO/ANSI Standard for C++ (ISO/IEC 14882:1998, Section 23.1.1, "Sequences"). Practically, this means that you can treat such a sequence as if it was `std::vector`.

The following code shows how one could use this mapping:

```
void
f (object& o)
{
  object::member::container& c (o.member ());

  // Iteration.
  //
  for (object::member::iterator i (c.begin ()); i != c.end (); ++i)
  {
    int value (*i);
  }

  // Modification.
  //
  c.push_back (10);

  // Setting a new container.
  //
  object::member::container n;
  n.push_back (1);
  o.member (n);
}
```

2.9 Mapping for Global Elements

An XML Schema element definition is called global if it appears directly under the `schema` element. A global element is a valid root of an instance document. As such, it is mapped to a set of overloaded parsing and, optionally, serialization functions with the same name as the element.

The parsing functions read XML instance documents and return corresponding in-memory representations. Their signatures have the following pattern (`type` denotes element's type and `name` denotes element's name):

```
std::auto_ptr<type>
name (....);
```

The process of parsing, including the exact signatures of the parsing functions, is the subject of Chapter 3, "Parsing".

The serialization functions write in-memory representations back to XML instance documents. Their signatures have the following pattern:

```
void
name (<stream type>&, const type&, ....);
```

The process of serialization, including the exact signatures of the serialization functions, is the subject of Chapter 4, "Serialization".

2.10 Mapping for Global Attributes

An XML Schema attribute definition is called global if it appears directly under the `schema` element. A global attribute does not have any mapping.

2.11 Mapping for Anonymous Types

2.11.1 Anonymous Types for Local Elements and Attributes

An XML Schema anonymous type defined for a local element or attribute is mapped to a nested C++ class with an implementation-specific name. Such a class follows standard mapping rules for simple and complex type definitions (see Section 2.6, "Mapping for Simple Types" and Section 2.7, "Mapping for Complex Types"). The only portable way to refer to an anonymous type of a local element or attribute is by using the `type` alias in the member information scope, as described in Section 2.8, "Mapping for Local Elements and Attributes". For example:

```
<complexType name="object">
  <sequence>
    <element name="member">
      <complexType>
        ...
      </complexType>
    </element>
  </sequence>
</complexType>
```

is mapped to:

```
class object: xml_schema::type
{
public:
  struct member
  {
    class <implementation-specific name>
    {
      ...
    };
```

```
    typedef <implementation-specific name> type;
  };

  const member::type&
  member () const;

  member::type&
  member () const;

  void
  member (const member::type&)

  ...

};
```

and can be used like this:

```
void
f (object& o)
{
  object::member::type& m (o.member ());   // get
  o.member (object::member::type (....));  // set
}
```

2.11.2 Anonymous Types for Global Elements

An XML Schema anonymous type defined for a global element is mapped to a C++ class-scope with the same name as the element. Inside this scope, a C++ class is defined with an implementation-specific name that follows standard mapping rules for simple and complex type definitions (see Section 2.6, "Mapping for Simple Types" and Section 2.7, "Mapping for Complex Types"). An alias with the name `type` (or `type_` if the element itself is named `type`) for this implementation-specific name is then defined withing the scope. The only portable way to refer to an anonymous type of a global element is by using the `type` alias. For example:

```
<element name="root">
  <complexType>
    ...
  </complexType>
</element>
```

is mapped to:

```
struct root
{
  class <implementation-specific name>
  {
    ...
```

```
  };

  typedef <implementation-specific name> type;
};

std::auto_ptr<root::type>
root (....);

...
```

and can be used like this:

```
void
f ()
{
  std::auto_ptr<root::type> r (root (....))
}
```

2.11.3 Anonymous Types for Global Attributes

An XML Schema anonymous type defined for a global attribute does not have any mapping.

2.12 Mapping for `xsi:type` and Substitution Groups

The implementation of the mapping supports XML Schema polymorphism features (`xsi:type` and substitution groups). Particularly, this means that the dynamic type of a member may be different from its static type. Consider the following schema definition and instance document:

```
<!-- test.xsd -->
<schema>
  <complexType name="base">
    <attribute name="text" type="string"/>
  </complexType>

  <complexType name="derived">
    <complexContent>
      <extension base="base">
        <attribute name="extra-text" type="string"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="root_type">
    <sequence>
      <element name="item" type="base" maxOccurs="unbounded"/>
    </sequence>
  </complexType>

  <element name="root" type="root_type"/>
```

```
</schema>

<!-- test.xml -->
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <item text="hello"/>
  <item text="hello" extra-text="world" xsi:type="derived"/>
</root>
```

In the resulting in-memory representation, the container for the `root::item` member will have two elements: the first element's type will be `base` while the second element's (dynamic) type will be `derived`.

The `_clone` virtual function should be used instead of copy constructors to make copies of members that might use polymorphism:

```
void
f (root& r)
{
  for (root::item::const_iterator i (r.item ().begin ());
       i != r.item ().end ()
       ++i)
  {
    std::auto_ptr<base> c (i->_clone ());
  }
}
```

2.13 Mapping for `any`, `anyAttribute`, and Mixed Content Models

XML Schema `any`, `anyAttribute`, and mixed content models do not have direct C++ mapping. Instead, information in XML instance documents, corresponding to these constructs, is accessed using generic DOM nodes that can optionally be associated with nodes of the statically-typed tree. See Section 3.2, "Flags and Properties" for more information.

3 Parsing

This chapter covers various aspects of parsing XML instance documents in order to obtain corresponding tree-like in-memory representations.

Each global XML Schema element in the form:

```
<element name="name" type="type"/>
```

is mapped to thirteen overloaded C++ functions in the form:

```
// Read from a URI or a local file.
//

std::auto_ptr<type>
name (const std::basic_string<C>& uri,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());

std::auto_ptr<type>
name (const std::basic_string<C>& uri,
      xml_schema::error_handler&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());

std::auto_ptr<type>
name (const std::basic_string<C>& uri,
      xercesc::DOMErrorHandler&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());


// Read from std::istream.
//

std::auto_ptr<type>
name (std::istream&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());

std::auto_ptr<type>
name (std::istream&,
      xml_schema::error_handler&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());

std::auto_ptr<type>
name (std::istream&,
      xercesc::DOMErrorHandler&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());


std::auto_ptr<type>
name (std::istream&,
      const std::basic_string<C>& id,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());

std::auto_ptr<type>
```

```
name (std::istream&,
      const std::basic_string<C>& id,
      xml_schema::error_handler&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());

std::auto_ptr<type>
name (std::istream&,
      const std::basic_string<C>& id,
      xercesc::DOMErrorHandler&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());


// Read from InputSource.
//

std::auto_ptr<type>
name (const xercesc::DOMInputSource&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());

std::auto_ptr<type>
name (const xercesc::DOMInputSource&,
      xml_schema::error_handler&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());

std::auto_ptr<type>
name (const xercesc::DOMInputSource&,
      xercesc::DOMErrorHandler&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());


// Read from DOM.
//

std::auto_ptr<type>
name (const xercesc::DOMDocument&,
      xml_schema::flags = 0,
      const xml_schema::properties& = xml_schema::properties ());
```

You can choose between reading an XML instance from a local file, URI, `std::istream`, `xercesc::DOMInputSource`, or a pre-parsed DOM instance in the form of `xercesc::DOMDocument`.

3.1 Initializing the Xerces-C++ Runtime

Some parsing functions expect you to initialize the Xerces-C++ runtime while others initialize and terminate it as part of their work. The general rule is as follows: if a function has any arguments or return a value that is an instance of a Xerces-C++ type, then this function expects you to initialize the Xerces-C++ runtime. Otherwise, the function initializes and terminates the runtime for you. Note that it is legal to have nested calls to the Xerces-C++ initialize and terminate functions as long as the calls are balanced.

You can instruct parsing functions that initialize and terminate the runtime not to do so by passing the `xml_schema::flags::dont_initialize` flag (see Section 3.2, "Flags and Properties").

3.2 Flags and Properties

Parsing flags and properties are the last two arguments of every parsing function. They allow you to fine-tune the process of instance validation and parsing. Both arguments are optional.

The following flags are recognized by the parsing functions:

`xml_schema::flags::keep_dom`
: Keep association between DOM nodes and the resulting tree nodes.

`xml_schema::flags::dont_validate`
: Do not validate instance documents against schemas.

`xml_schema::flags::dont_initialize`
: Do not initialize the Xerces-C++ runtime.

You can pass several flags by combining them using the bit-wise OR operator. For example:

```
using xml_schema::flags;

std::auto_ptr<type> r (
  name ("test.xml", flags::keep_dom | flags::dont_validate));
```

Keeping association with DOM nodes is useful for dealing with type-less content such as mixed content models, `any`/`anyAttribute`, and `anyType`/`anySimpleType`. You can access DOM nodes using the `_node` functions defined in the `xml_schema::type` class (see Section 2.5.2, "Mapping for `anyType`"). Note that since DOM nodes "out-live" the parsing function call, you need to initializing the Xerces-C++ runtime before calling one of the parsing functions with the `keep_dom` flag and terminate it after the in-memory representation is destroyed (see Section 3.1, "Initializing the Xerces-C++ Runtime").

By default, validation of instance documents is turned on even though parsers generated by XSD do not assume instance documents are valid. They include a number of checks that prevent construction of inconsistent in-memory representations. This, however, does not mean that an instance document that was successfully parsed by the XSD-generated parsers is valid per the corresponding schema. If an instance document is not "valid enough" for the generated parsers to construct consistent in-memory representation, one of the exceptions defined in `xml_schema` namespace is thrown (see Section 3.3, "Error Handling").

For more information on the Xerces-C++ runtime initialization refer to Section 3.1, "Initializing the Xerces-C++ Runtime".

The `xml_schema::properties` class allows you to programmatically specify schema locations to be used instead of those specified with the `xsi::schemaLocation` and `xsi::noNamespaceSchemaLocation` attributes in instance documents. The interface of the `properties` class is presented below:

```
class properties
{
public:
  void
  schema_location (const std::basic_string<C>& namespace_,
                   const std::basic_string<C>& location);
  void
  no_namespace_schema_location (const std::basic_string<C>& location);
};
```

Note that all locations are relative to an instance document unless they are URIs. For example, if you want to use a local file as your schema, then you will need to pass `file:///absolute/path/to/your/schema` as the location argument.

3.3 Error Handling

As discussed in Section 2.2, "Error Handling", the mapping uses the C++ exception handling mechanism as its primary way of reporting error conditions. However, to handle recoverable parsing and validation errors and warnings, a callback interface maybe preferred by the application.

To better understand error handling and reporting strategies employed by the parsing functions, it is useful to know that the transformation of an XML instance document to a statically-typed tree happens in two stages. The first stage, performed by Xerces-C++, consists of parsing an XML document into a DOM instance. For short, we will call this stage the XML-DOM stage. Validation, if not disabled, happens during this stage. The second stage, performed by the generated parsers, consist of parsing the DOM instance into the statically-typed tree. We will call this stage the DOM-Tree stage. Additional checks are performed during this stage in order to prevent construction of inconsistent tree which could otherwise happen when validation is disabled, for

example.

All parsing functions except the one that operates on a DOM instance come in overloaded triples. The first function in such a triple reports error conditions exclusively by throwing exceptions. It accumulates all the parsing and validation errors of the XML-DOM stage and throws them in a single instance of the `xml_schema::parsing` exception (described below). The second and the third functions in the triple use callback interfaces to report parsing and validation errors and warnings. The two callback interfaces are `xml_schema::error_handler` and `xercesc::DOMErrorHandler`. For more information on the `xercesc::DOMErrorHandler` interface refer to the Xerces-C++ documentation. The `xml_schema::error_handler` interface is presented below:

```
class error_handler
{
public:
  struct severity
  {
    enum value
    {
      warning,
      error,
      fatal
    };
  };

  virtual bool
  handle (const std::basic_string<C>& id,
          unsigned long line,
          unsigned long column,
          severity,
          const std::basic_string<C>& message) = 0;

  virtual
  ~error_handler ();
};
```

The `id` argument of the `error_handler::handle` function identifies the resource being parsed (e.g., a file name or URI).

By returning `true` from the `handle` function you instruct the parser to recover and continue parsing. Returning `false` results in termination of the parsing process. An error with the `fatal` severity level results in termination of the parsing process no matter what is returned from the `handle` function. It is safe to throw an exception from the `handle` function.

The DOM-Tree stage reports error conditions exclusively by throwing exceptions. Individual exceptions thrown by the parsing functions are described in the following sub-sections.

3.3.1 `xml_schema::parsing`

```
struct error
{
  error (const std::basic_string<C>& id,
         unsigned long line,
         unsigned long column,
         const std::basic_string<C>& message);

  const std::basic_string<C>&
  id () const;

  unsigned long
  line () const;

  unsigned long
  column () const;

  const std::basic_string<C>&
  message () const;
};

std::basic_ostream<C>&
operator<< (std::basic_ostream<C>&, const error&);


struct errors: std::vector<error>
{
};

std::basic_ostream<C>&
operator<< (std::basic_ostream<C>&, const errors&);


struct parsing: virtual exception
{
  parsing ();
  parsing (const errors&);

  const errors&
  errors () const;

  virtual const char*
  what () const throw ();
};
```

The `xml_schema::parsing` exception is thrown if there were parsing or validation errors reported during the XML-DOM stage. If no callback interface was provided to the parsing function, the exception contains a list of errors accessible using the `errors` function. The usual conditions when this exception is thrown include malformed XML instances and, if validation is

turned on, invalid instance documents.

3.3.2 xml_schema::expected_element

```
struct expected_element: virtual exception
{
  expected_element (const std::basic_string<C>& name,
                    const std::basic_string<C>& namespace_);


  const std::basic_string<C>&
  name () const;

  const std::basic_string<C>&
  namespace_ () const;


  virtual const char*
  what () const throw ();
};
```

The xml_schema::expected_element exception is thrown when an expected element is not encountered by the DOM-Tree stage. The name and namespace of the expected element can be obtained using the name and namespace_ functions respectively.

3.3.3 xml_schema::unexpected_element

```
struct unexpected_element: virtual exception
{
  unexpected_element (const std::basic_string<C>& encountered_name,
                      const std::basic_string<C>& encountered_namespace,
                      const std::basic_string<C>& expected_name,
                      const std::basic_string<C>& expected_namespace)


  const std::basic_string<C>&
  encountered_name () const;

  const std::basic_string<C>&
  encountered_namespace () const;


  const std::basic_string<C>&
  expected_name () const;

  const std::basic_string<C>&
  expected_namespace () const;
```

```
  virtual const char*
  what () const throw ();
};
```

The `xml_schema::unexpected_element` exception is thrown when an unexpected element is encountered by the DOM-Tree stage. The name and namespace of the encountered element can be obtained using the `encountered_name` and `encountered_namespace` functions respectively. If an element was expected instead of the encountered one, its name and namespace can be obtained using the `expected_name` and `expected_namespace` functions respectively. Otherwise these functions return empty strings.

3.3.4 `xml_schema::expected_attribute`

```
struct expected_attribute: virtual exception
{
  expected_attribute (const std::basic_string<C>& name,
                      const std::basic_string<C>& namespace_);


  const std::basic_string<C>&
  name () const;

  const std::basic_string<C>&
  namespace_ () const;


  virtual const char*
  what () const throw ();
};
```

The `xml_schema::expected_attribute` exception is thrown when an expected attribute is not encountered by the DOM-Tree stage. The name and namespace of the expected attribute can be obtained using the `name` and `namespace_` functions respectively.

3.3.5 `xml_schema::unexpected_enumerator`

```
struct unexpected_enumerator: virtual exception
{
  unexpected_enumerator (const std::basic_string<C>& enumerator);

  const std::basic_string<C>&
  enumerator () const;

  virtual const char*
  what () const throw ();
};
```

The `xml_schema::unexpected_enumerator` exception is thrown when an unexpected enumerator is encountered by the DOM-Tree stage. The enumerator can be obtained using the `enumerator` functions.

3.3.6 `xml_schema::no_type_info`

```
struct no_type_info: virtual exception
{
  no_type_info (const std::basic_string<C>& type_name,
                const std::basic_string<C>& type_namespace);


  const std::basic_string<C>&
  type_name () const;

  const std::basic_string<C>&
  type_namespace () const;


  virtual const char*
  what () const throw ();
};
```

The `xml_schema::no_type_info` exception is thrown when there is no type information associated with a type specified by the `xsi:type` attribute. This exception is thrown by the DOM-Tree stage. The name and namespace of the type in question can be obtained using the `type_name` and `type_namespace` functions respectively. Usually, catching this exception means that you haven't linked the code generated from the schema defining the type in question with your application.

3.3.7 `xml_schema::not_derived`

```
struct not_derived: virtual exception
{
  not_derived (const std::basic_string<C>& base_type_name,
               const std::basic_string<C>& base_type_namespace,
               const std::basic_string<C>& derived_type_name,
               const std::basic_string<C>& derived_type_namespace);


  const std::basic_string<C>&
  base_type_name () const;

  const std::basic_string<C>&
  base_type_namespace () const;


  const std::basic_string<C>&
  derived_type_name () const;
```

```
  const std::basic_string<C>&
  derived_type_namespace () const;


  virtual const char*
  what () const throw ();
};
```

The `xml_schema::not_derived` exception is thrown when a type specified by the `xsi:type` attribute is not derived from the expected base type. This exception is thrown by the DOM-Tree stage. The name and namespace of the expected base type can be obtained using the `base_type_name` and `base_type_namespace` functions respectively. The name and namespace of the offending type can be obtained using the `derived_type_name` and `derived_type_namespace` functions respectively.

3.4 Reading from a Local File or URI

Using a local file or URI is the simplest way to parse an XML instance. For example:

```
using std::auto_ptr;

auto_ptr<type> r1 (name ("test.xml"));
auto_ptr<type> r2 (name ("http://www.codesynthesis.com/test.xml"));
```

3.5 Reading from `std::istream`

When using an `std::istream` instance, you may also pass an optional resource id. This id is used to identify the resource (for example in error messages) as well as to resolve relative paths. For instance:

```
using std::auto_ptr;

{
  std::ifstream ifs ("test.xml");
  auto_ptr<type> r (name (ifs, "test.xml"));
}

{
  std::string str ("..."); // Some XML fragment.
  std::istringstream iss (str);
  auto_ptr<type> r (name (iss));
}
```

3.6 Reading from `xercesc::DOMInputSource`

Reading from a `xercesc::DOMInputSource` instance is similar to the `std::istream` case except the resource id is maintained by the `DOMInputSource` object. For instance:

```
xercesc::StdInInputSource is;
xercesc::Wrapper4InputSource wis (is, false);

std::auto_ptr<type> r (name (wis));
```

3.7 Reading from DOM

Reading from a `xercesc::DOMDocument` instance allows you to setup a custom XML-DOM stage. Things like DOM parser reuse, schema pre-parsing, and schema caching can be achieved with this approach. For more information on how to obtain DOM representation from an XML instance refer to the Xerces-C++ documentation.

4 Serialization

This chapter covers various aspects of serializing a tree-like in-memory representation to DOM or XML. In this regard, serialization is complimentary to the reverse process of parsing a DOM or XML instance into an in-memory representation which is discussed in Chapter 3, "Parsing". Note that generation of the serialization code is optional and should be explicitly requested with the `--generate-serialization` option. See the compiler manual for more information.

Each global XML Schema element in the form:

```
<xsd:element name="name" type="type"/>
```

is mapped to eight overloaded C++ functions in the form:

```
// Serialize to std::ostream.
//
void
name (std::ostream&,
      const type&,
      const xml_schema::namespace_infomap&,
      const string& encoding = "UTF-8",
      xml_schema::flags = 0);

void
name (std::ostream&,
      const type&,
      const xml_schema::namespace_infomap&,
      xml_schema::error_handler&,
      const string& encoding = "UTF-8",
```

```
      xml_schema::flags = 0);

void
name (std::ostream&,
      const type&,
      const xml_schema::namespace_infomap&,
      xercesc::DOMErrorHandler&,
      const string& encoding = "UTF-8",
      xml_schema::flags = 0);


// Serialize to XMLFormatTarget.
//
void
name (xercesc::XMLFormatTarget&,
      const type&,
      const xml_schema::namespace_infomap&,
      const string& encoding = "UTF-8",
      xml_schema::flags = 0);

void
name (xercesc::XMLFormatTarget&,
      const type&,
      const xml_schema::namespace_infomap&,
      xml_schema::error_handler&,
      const string& encoding = "UTF-8",
      xml_schema::flags = 0);

void
name (xercesc::XMLFormatTarget&,
      const type&,
      const xml_schema::namespace_infomap&,
      xercesc::DOMErrorHandler&,
      const string& encoding = "UTF-8",
      xml_schema::flags = 0);


// Serialize to DOM.
//
xml::auto_ptr<xercesc::DOMDocument>
name (const type&,
      const xml_schema::namespace_infomap&,
      xml_schema::flags = 0);

void
name (xercesc::DOMDocument&,
      const type&,
      xml_schema::flags = 0);
```

You can choose between writing XML to `std::ostream` or `xercesc::XMLFormatTarget` and creating a DOM instance in the form of `xercesc::DOMDocument`. Serialization to `ostream` or `XMLFormatTarget` requires a considerably less work while serialization to DOM provides for greater flexibility.

4.1 Initializing the Xerces-C++ Runtime

Some serialization functions expect you to initialize the Xerces-C++ runtime while others initialize and terminate it as part of their work. The general rule is as follows: if a function has any arguments or return a value that is an instance of a Xerces-C++ type, then this function expects you to initialize the Xerces-C++ runtime. Otherwise, the function initializes and terminates the runtime for you. Note that it is legal to have nested calls to the Xerces-C++ initialize and terminate functions as long as the calls are balanced.

You can instruct serialization functions that initialize and terminate the runtime not to do so by passing the `xml_schema::flags::dont_initialize` flag (see Section 4.3, "Flags").

4.2 Namespace Infomap and Character Encoding

When a document being serialized uses XML namespaces, prefix-namespace associations need to be established. Also, if you would like the resulting instance document to contain the `schemaLocation` or `noNamespaceSchemaLocation` attributes, you will need to provide namespace-schema associations. The `xml_schema::namespace_infomap` class is used to capture this information:

```
struct namespace_info
{
  namespace_info ();
  namespace_info (const string& name, const string& schema);

  string name;
  string schema;
};

// Map of namespace prefix to namespace_info.
//
struct namespace_infomap: public std::map<string, namespace_info>
{
};
```

Consider the following associations as an example:

```
xml_schema::namespace_infomap map;

map["t"].name = "http://www.codesynthesis.com/test";
map["t"].schema = "test.xsd";
```

This map, if passed to one of the serialization functions, could result in the following XML fragment:

```
<?xml version="1.0" ?>
<t:name xmlns:t="http://www.codesynthesis.com/test"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.codesynthesis.com/test test.xsd">
```

As you can see, the serialization function automatically added namespace mapping for the `xsi` prefix. You can change this by providing your own prefix:

```
xml_schema::namespace_infomap map;

map["xsn"].name = "http://www.w3.org/2001/XMLSchema-instance";

map["t"].name = "http://www.codesynthesis.com/test";
map["t"].schema = "test.xsd";
```

This could result in the following XML fragment:

```
<?xml version="1.0" ?>
<t:name xmlns:t="http://www.codesynthesis.com/test"
        xmlns:xsn="http://www.w3.org/2001/XMLSchema-instance"
        xsn:schemaLocation="http://www.codesynthesis.com/test test.xsd">
```

Another bit of information that you can pass to the serialization functions is the character encoding method that you would like to use. Common values for this argument are `"USASCII"`, `"ISO8859-1"`, `"UTF-8"`, `"UTF-16BE"`, `"UTF-16LE"`, `"UCS-4BE"`, and `"UCS-4LE"`. The default encoding is `"UTF-8"`. For more information on encoding methods see the "Character Encoding" article from Wikipedia.

4.3 Flags

Serialization flags are the last argument of every serialization function. They allow you to fine-tune the process of serialization. The flags argument is optional.

The following flags are recognized by the serialization functions:

`xml_schema::flags::dont_initialize`
: Do not initialize the Xerces-C++ runtime.

For more information on the Xerces-C++ runtime initialization refer to Section 4.1, "Initializing the Xerces-C++ Runtime".

4.4 Error Handling

As with the parsing functions (see Section 3.3, "Error Handling"), to better understand error handling and reporting strategies employed by the serialization functions, it is useful to know that the transformation of a statically-typed tree to an XML instance document happens in two stages. The first stage, performed by the generated code, consist of building a DOM instance from the statically-typed tree . For short, we will call this stage the Tree-DOM stage. The second stage, performed by Xerces-C++, consists of serializing the DOM instance into the XML document. We will call this stage the DOM-XML stage.

All serialization functions except the two that serialize into a DOM instance come in overloaded triples. The first function in such a triple reports error conditions exclusively by throwing exceptions It accumulates all the serialization errors of the DOM-XML stage and throws them in a single instance of the `xml_schema::serialization` exception (described below). The second and the third functions in the triple use callback interfaces to report serialization errors and warnings. The two callback interfaces are `xml_schema::error_handler` and `xercesc::DOMErrorHandler`. The `xml_schema::error_handler` interface is described in Section 3.3, "Error Handling". For more information on the `xercesc::DOMErrorHandler` interface refer to the Xerces-C++ documentation.

The Tree-DOM stage reports error conditions exclusively by throwing exceptions. Individual exceptions thrown by the serialization functions are described in the following sub-sections.

4.4.1 `xml_schema::serialization`

```
struct serialization: virtual exception
{
  serialization ();
  serialization (const errors&);

  const errors&
  errors () const;

  virtual const char*
  what () const throw ();
};
```

The `xml_schema::errors` class is described in Section 3.3.1, "`xml_schema::parsing`". The `xml_schema::serialization` exception is thrown if there were serialization errors reported during the XML-DOM stage. If no callback interface was provided to the serialization function, the exception contains a list of errors accessible using the `errors` function.

4.4.2 `xml_schema::unexpected_element`

The `xml_schema::unexpected_element` exception is described in Section 3.3.3, "`xml_schema::unexpected_element`". It is thrown by the serialization functions during the Tree-DOM stage if the root element name of the provided DOM instance does not match with the name of the element this serialization function is for.

4.4.3 `xml_schema::no_namespace_mapping`

```
struct no_namespace_mapping: virtual exception
{
  no_namespace_mapping (const std::basic_string<C>& namespace_);

  const std::basic_string<C>&
  namespace_ () const;

  virtual const char*
  what () const throw ();
};
```

The `xml_schema::no_namespace_mapping` exception is thrown during the Tree-DOM stage if a namespace is encountered for which a prefix-namespace mapping hasn't been provided. The namespace in question can be obtained using the `namespace_` function.

4.4.4 `xml_schema::no_prefix_mapping`

```
struct no_prefix_mapping: virtual exception
{
  no_prefix_mapping (const std::basic_string<C>& prefix);

  const std::basic_string<C>&
  prefix () const;

  virtual const char*
  what () const throw ();
};
```

The `xml_schema::no_prefix_mapping` exception is thrown during the Tree-DOM stage if a namespace prefix is encountered for which a prefix-namespace mapping hasn't been provided. The namespace prefix in question can be obtained using the `prefix` function.

4.4.5 `xml_schema::xsi_already_in_use`

```
struct xsi_already_in_use: virtual exception
{
  virtual const char*
  what () const throw ();
};
```

The `xml_schema::xsi_already_in_use` exception is thrown during the Tree-DOM stage if the `xsi` prefix is already in use and no user-defined prefix-namespace mapping has been provided for the `http://www.w3.org/2001/XMLSchema-instance` namespace.

4.5 Serializing to `std::ostream`

In order to serialize to `std::ostream` you will need an in-memory representation, an output stream and a namespace infomap. For instance:

```
// Obtain the in-memory representation.
//
std::auto_ptr<type> r = ...

// Prepare namespace mapping and schema location information.
//
xml_schema::namespace_infomap map;

map["t"].name = "http://www.codesynthesis.com/test";
map["t"].schema = "test.xsd";

// Write it out.
//
name (std::cout, *r, map);
```

Note that the output stream is treated as a binary stream. This becomes important when you use a character encoding that is wider than 8-bit `char`, for instance UTF-16 or UCS-4. For example, things will most likely break if you try to serialize to `std::ostringstream` with UTF-16 or UCS-4 as an encoding. This is due to the special value, `'\0'`, that will most likely occur as part of such serialization and it won't have the special meaning assumed by `std::ostringstream`.

4.6 Serializing to `xercesc::XMLFormatTarget`

Serializing to an `xercesc::XMLFormatTarget` instance is similar the `std::ostream` case. For instance:

```
using std::auto_ptr;

// Obtain the in-memory representation.
//
auto_ptr<type> r = ...

// Prepare namespace mapping and schema location information.
//
xml_schema::namespace_infomap map;

map["t"].name = "http://www.codesynthesis.com/test";
```

```
map["t"].schema = "test.xsd";

using namespace xercesc;

XMLPlatformUtils::Initialize ();

{
  // Choose a target.
  //
  auto_ptr<XMLFormatTarget> ft;

  if (argc != 2)
  {
    ft = auto_ptr<XMLFormatTarget> (new StdOutFormatTarget ());
  }
  else
  {
    ft = auto_ptr<XMLFormatTarget> (
      new LocalFileFormatTarget (argv[1]));
  }

  // Write it out.
  //
  name (*ft, *r, map);
}

XMLPlatformUtils::Terminate ();
```

Note that we had to initialize the Xerces-C++ runtime before we could call this serialization function.

4.7 Serializing to DOM

The mapping provides two overloaded functions that implement serialization to a DOM instance. The first creates a DOM instance for you and the second serializes to an existing DOM instance. While serializing to a new DOM instance is similar to serializing to `std::ostream` or `xercesc::XMLFormatTarget`, serializing to an existing DOM instance requires quite a bit of work from your side. You will need to set all the namespace mapping attributes as well as the `schemaLocation` and/or `noNamespaceSchemaLocation` attributes. The following listing should give you an idea about what needs to be done:

```
// Obtain the in-memory representation.
//
std::auto_ptr<type> r = ...

using namespace xercesc;
using namespace xsd::cxx;

XMLPlatformUtils::Initialize ();
```

```
{
  DOMImplementation* impl (
    DOMImplementationRegistry::getDOMImplementation (
      xml::string ("LS").c_str ()));


  // Create a DOM instance.
  //
  xml::auto_ptr<DOMDocument> doc (
    impl->createDocument (
      //
      // Root element namespace.
      //
      xml::string ("http://www.codesynthesis.com/test").c_str (),
      //
      // Root element name. Note that the namespace prefix is
      // automatically associated with the root element namespace
      // above.
      //
      xml::string ("t:name").c_str (),
      //
      // Document type object.
      //
      0));


  // Set namespace mapping and schema location attributes.
  //
  DOMElement* root (doc->getDocumentElement ());

  root->setAttributeNS (
    xml::string ("http://www.w3.org/2000/xmlns/").c_str (),
    xml::string ("xmlns:xsi").c_str (),
    xml::string ("http://www.w3.org/2001/XMLSchema-instance").c_str ());

  root->setAttributeNS (
    xml::string ("http://www.w3.org/2001/XMLSchema-instance").c_str (),
    xml::string ("xsi:schemaLocation").c_str (),
    xml::string ("http://www.codesynthesis.com/test test.xsd").c_str ());


  // Serialize to DOM.
  //
  name (*doc, *r);


  // Get an instance of DOMWriter.
  //
  xml::auto_ptr<DOMWriter> writer (impl->createDOMWriter ());
```

```
  // Plug in an error handler.
  //
  xml::error_handler<char> eh;
  writer->setErrorHandler (&eh);


  // Set some nice features if the writer supports them.
  //
  if (writer->canSetFeature (XMLUni::fgDOMWRTDiscardDefaultContent, true))
    writer->setFeature (XMLUni::fgDOMWRTDiscardDefaultContent, true);

  if (writer->canSetFeature (XMLUni::fgDOMWRTFormatPrettyPrint, true))
    writer->setFeature (XMLUni::fgDOMWRTFormatPrettyPrint, true);


  // Create a format target that will receive the resulting
  // XML stream from the writer.
  //
  std::auto_ptr<XMLFormatTarget> ft;

  if (argc != 2)
  {
    ft = std::auto_ptr<XMLFormatTarget> (new StdOutFormatTarget ());
  }
  else
  {
    ft = std::auto_ptr<XMLFormatTarget> (
      new LocalFileFormatTarget (argv[1]));
  }

  // Write to the format target.
  //
  writer->writeNode (ft.get (), *doc);
}

XMLPlatformUtils::Terminate ();
```

For more information on how to create and serialize a DOM instance refer to the Xerces-C++ documentation.

Appendix A — Default and Fixed Values

The following table summarizes the effect of default and fixed values (specified with the `default` and `fixed` attributes, respectively) on attribute and element values. The `default` and `fixed` attributes are mutually exclusive. It is also worthwhile to note that the fixed value semantics is a superset of the default value semantics.

<table>
<tr><th></th><th></th><th colspan="2">default</th><th colspan="2">fixed</th></tr>
<tr><td rowspan="4">element</td><td rowspan="2">not present</td><td>optional</td><td>required</td><td>optional</td><td>required</td></tr>
<tr><td>not present</td><td>invalid instance</td><td>not present</td><td>invalid instance</td></tr>
<tr><td>empty</td><td colspan="2">default value is used</td><td colspan="2">fixed value is used</td></tr>
<tr><td>value</td><td colspan="2">value is used</td><td colspan="2">value is used provided it’s the same as fixed</td></tr>
<tr><td rowspan="4">attribute</td><td rowspan="2">not present</td><td>optional</td><td>required</td><td>optional</td><td>required</td></tr>
<tr><td>default value is used</td><td>invalid schema</td><td>fixed value is used</td><td>invalid instance</td></tr>
<tr><td>empty</td><td colspan="2">empty value is used</td><td colspan="2">empty value is used provided it’s the same as fixed</td></tr>
<tr><td>value</td><td colspan="2">value is used</td><td colspan="2">value is used provided it’s the same as fixed</td></tr>
</table>