NISTIR 8337

An Integrated Set of XML Tools – User Guide

Thomas R. Kramer
Zeid Kootbally
Craig Schlenoff

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8337

National Institute of Standards and Technology
U.S. Department of Commerce

NISTIR 8337

An Integrated Set of XML Tools – User Guide

Thomas R. Kramer
Catholic University of America

Zeid Kootbally
University of Southern California

Craig Schlenoff
Intelligent Systems Division
Engineering Laboratory, NIST

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8337

December 2020

U.S. Department of Commerce
Wilbur L. Ross, Jr., Secretary

National Institute of Standards and Technology
Walter Copan, NIST Director and Undersecretary of Commerce for Standards and Technology

National Institute of Standards and Technology Interagency or Internal Report 8337
Natl. Inst. Stand. Technol. Interag. Intern. Rep. 8337, 128 pages (December 2020)

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8337

Abstract

This user guide describes an integrated s C++ software tools for manipulating XML (eXtensible Markup Language) schemas and XML instance files. The tools are:

- xmlSchemaParser - parses and reformats XML schema files, and prints type derivation hierarchies
- xmlInstanceParserGenerator - generates C++ classes and a YACC/Lex instance file parser from an XML schema – it's a software tool that writes software tools
- orphanFinder - identifies unused types and undefined types in one or a set of XML schema files
- xmlSchemaAttributeConverter - rewrites an XML schema file, converting attributes to elements

All of the tools are invoked by giving a command in a terminal window.

The tools were built at the National Institute of Standards and Technology. The xmlSchemaParser and the xmlInstanceParserGenerator were originally built in support of the Agility Performance of Robotic Systems project. They have been greatly extended since that time. The orphanFinder and xmlSchemaAttributeConverter are more recent.

Keywords: *automatic*, *C++, information m generator, schema, software, tool, XML, XSDL, YACC, Lex*

Contents

Figures

1 Introduction

This document is a user manual for a set of four software tools that process XML files. The manual is intended for people who want to use the tools and for programmers who want to understand and/or modify the code of the tools. To read this manual one should be familiar with XML files and know a little bit about XML Schema Definition Language (XSDL). If you are not familiar with XSDL, read section 16 after reading this introduction.

1.1 Overview

The tools are:

xmlSchemaParser

- reads one XML schema file, stopping if an error is found
- stores the contents of the file in an abstract syntax tree built in terms of a set of C++ classes that represent XSDL
- reprints the file as directed by the user

xmlInstanceParserGenerator

- reads an XML schema embodied in a top-level XML schema file and any additional XML schema files connected to it by include directives, stopping if an error is found
- generates a YACC file for a parser of XML instance files that conform to the schema
- generates a Lex scanner used by the YACC parser
- generates a set of C++ classes representing the schema, with separate C++ header (.hh) and implementation (.cc) files for each XML schema file

orphanFinder

- reads one or a set of XML schema files (as directed by the command that starts it)
- lists types that are defined but not used in any of the schema files
- lists types that are used but not defined in any of the schema files

xmlSchemaAttributeConverter

- reads an XML schema embodied in a top-level XML schema file and any additional XML schema files connected to it by include directives, stopping if an error is found
- for each input file, prints a new schema file in which attributes have been converted to elements, so that the information content is unchanged

This manual also describes how to compile and use the code generated by the xmlInstanceParserGenerator.

The different tools have differing levels of functionality for handling XSDL schemas and XML instance files conforming to schemas. The level of functionality of each tool is described in the section about that tool.

1.2 Typographical Conventions

From this point on, reserved words from XSDL are set `in this font`, as are sample files. File names are set in this font. User commands are set **in this font**. C++ code (including variable, field and function/method names) is set in this font. When the word *domain* is set in italics, it stands for a particular information domain.

If any tool is called incorrectly, it prints a message describing how to call it. In such messages optional arguments are enclosed in square brackets. For example, `[-x]` indicates that `-x` is optional.

1.3 Previous Report

A report, *Software Tools for XML to OWL Translation*, NISTIR 8068, was published in June 2015. It was less detailed than the current document, focused primarily on the XML to OWL tools, and did not describe in detail the other tools that existed at the time, all of which have been upgraded since then. The orphanFinder and xmlSchemaAttributeConverter tools described in this report were developed after that report appeared. The earlier report described related research and similar tools. This report focuses on describing the tools built by the authors.

1.4 Organization of This Manual

Section 2 describes how the tools work together and gives typical scenarios of using the tools.

Section 3 gives an example of using the tools.

Sections 4 through 8 describe how to use each tool, including examples of commands. Tool users should read some or all of those sections. Each tool, if invoked with no arguments, prints a message about how to invoke it.

Section 9 gives an overview of the software of the tools. The section includes Makefiles for building:

- the XML tools
- an example XML instance file parser

Sections 10 through 15 describe details of the code for the tools. These sections are for C++ programmers who wish to modify the tools.

Section 16 gives a brief overview of XML and XSDL. Read that section first if you are not familiar with XML and XSDL. Section 17 provides a brief introduction to YACC. Section 18 describes how the tools have been used, up to 2020. Section 19 describes testing of the tools, and section 20 discusses future work on the tools.

2 How the Tools Work Together

Figure 1 shows the tools, the file types the tools manipulate, and the connections among them. The tools all run from a command shell; they have no graphical user interfaces. This makes them independent of any operating system. A python graphical user interface for the xmlInstanceParserGenerator was built in 2013 but has not been maintained.

The file *domain*.xsd on the figure is an information model file. An information model shows how instances of information should be structured. For example, a point might be modeled in an information model as x, y, and z coordinate values. The files *domain*Instance.xml on the right side of the figure are instance files that contain specific data instances that conform to an information model. For example, a specific point in an instance file might be (7, 2, -3), corresponding to the x, y, z model. Many instance files may correspond to a given information model.

The subject matter area of an information model is called its domain. All the tools except those on the right side of Figure 1 are domain independent. Each tool will work with any XML `schema` that meets that tool's restrictions on the usage of the XSDL. The restrictions vary among the tools.

Two of the four domain independent tools (xmlInstanceParserGenerator and xmlSchemaAttributeConverter) read and process all files in a set of XML `schema` files connected by inclusion. The orphanFinder does not deal with inclusion but does process multiple files. By telling it to process all the files in a set, the same results are obtained as would be obtained if it did process connected files. The xmlSchemaParser processes only one file at a time.

The tools on the right side of Figure 1 are domain dependent. They take as input only XML instance files in the domain for which the tools were generated.

A typical tool-using scenario for a user interested in XML is:

- An XML `schema` model file, *domain*.xsd, is built or otherwise acquired. There is no representation of building it on Figure 1.

- Optionally, the xmlSchemaParser is used to check that *domain*.xsd is valid (Figure 1, Arrow C pointing up). Errors in the `schema` file cause the xmlSchemaParser to exit. Pointless legal constructions such as empty sequences cause the xmlSchemaParser to issue warnings. The `schema` file is pretty-printed with a different name (Figure 1, Arrow C pointing down). Optionally, the `documentation` nodes are removed or reformatted. Optionally, a derivation hierarchy of types defined in the file is printed (Figure 1, Arrow A).

- Optionally, the orphanFinder is used to examine a set of XML `schema` files headed by *domain*.xsd and find (1) defined types that are not used and (2) undefined types that are used (Figure 1, Arrow G). These are printed to the monitor (Figure 1, Arrow H).

- The *domain*.xsd file is processed by the xmlInstanceParserGenerator (Figure 1, Arrow D) to generate code for parsing XML instance files that conform to *domain*.xsd (Figure 1, Arrow B). This includes generating C++ classes equivalent to the types defined in *domain*.xsd.

- A Domain Instance XML Parser is compiled from the code.

- An XML instance file, *domain*Instance.xml, conforming to *domain*.xsd is built or otherwise acquired. There is no representation of building it on Figure 1.

- Optionally, the Domain Instance XML Parser is used to check that *domain*Instance.xml conforms to *domain*.xsd (Figure 1, Arrow E pointing up) and pretty-print the file with a different name (Figure 1, Arrow E pointing down).

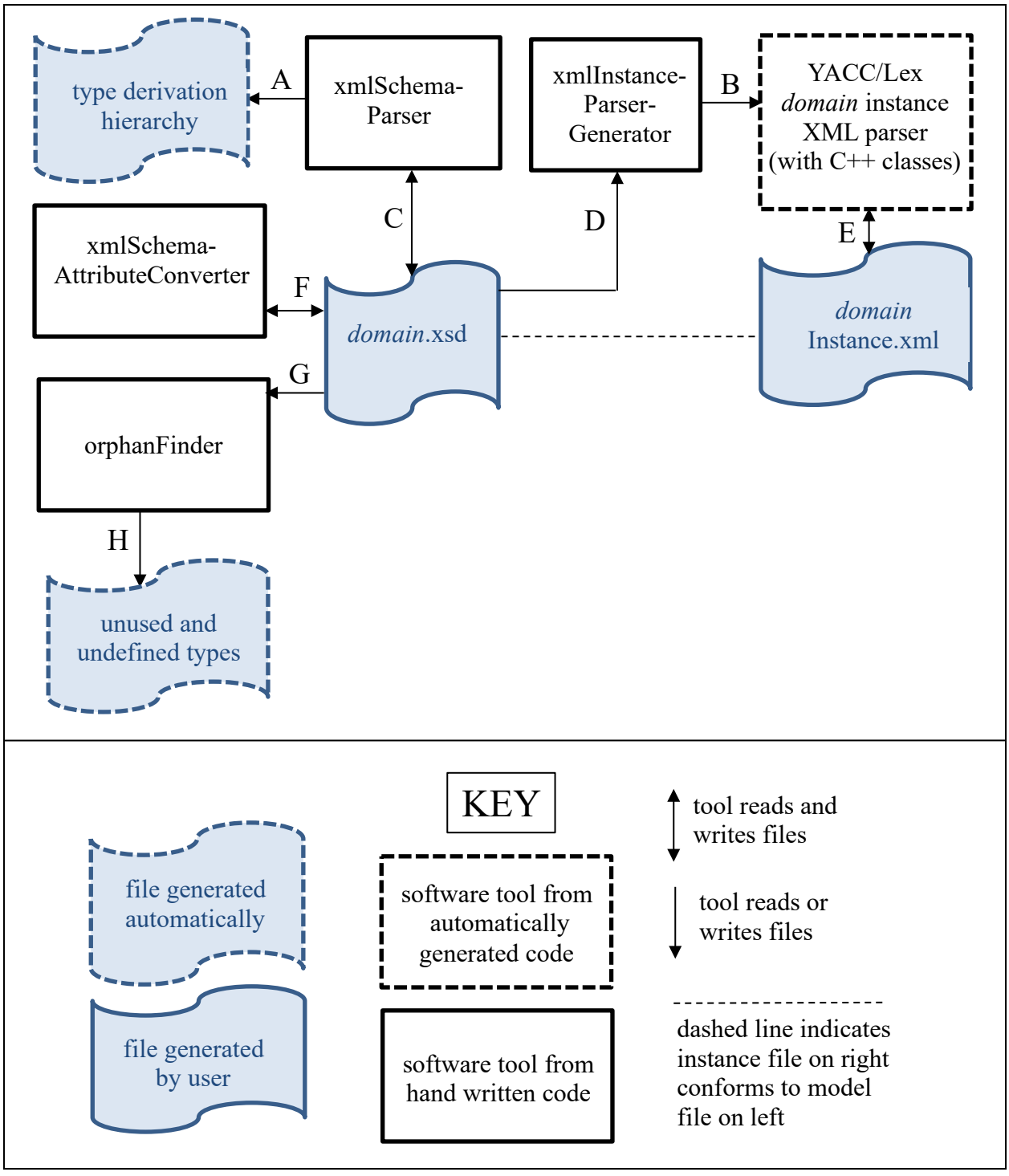

Figure 1 Tools Work Together

The xmlSchemaAttributeConverter may be used to read a `schema` file containing `attributes` and write a `schema` file representing the same information model in which

all the attributes have been changed to elements. This is represented by Arrow F on Figure 1 and described in Section 7.

The steps of the scenario that build code are taken only once for a given set of XML schema files, but the step that deals with instance files (Figure 1, Arrow E) may be repeated many times.

3 An Example

To show how the various tools work, the simple schema shown in Figure 2 and Figure 3 that models two-dimensional points, vectors, and lines will be used along with the XML instance file shown in Figure 4 as an example. The instance file conforms to the schema. To show how the tools deal with a set of files connected by include, the example schema has been divided into two schema files: line.xsd and primitives.xsd. On line 7, line.xsd includes primitives.xsd. Files produced by tools from these files are shown in figures or in the annexes.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">
<!-- This file defines BaseType, PointType and VectorType -->

  <xs:complexType name="BaseType" abstract="true">
    <xs:sequence>
      <xs:element name="Name" type="xs:ID"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="PointType">
    <xs:complexContent>
      <xs:extension base="BaseType">
        <xs:sequence>
          <xs:element name="X" type="xs:decimal">
          <xs:annotation>
            <xs:documentation>
The X element is the X coordinate of the point.
            </xs:documentation>
          </xs:annotation>
          </xs:element>
          <xs:element name="Y" type="xs:decimal"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="VectorType">
    <xs:complexContent>
      <xs:extension base="BaseType">
        <xs:sequence>
          <xs:element name="X" type="xs:decimal"/>
          <xs:element name="Y" type="xs:decimal"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

Figure 2 XML Schema File primitives.xsd – 2D Point and Vector

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This file defines the root element Line and the LineType -->
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">
  <xs:include schemaLocation="primitives.xsd"/>

  <xs:element name="Line"
    type="LineType">
    <xs:annotation>
      <xs:documentation>
        Root element
      </xs:documentation>
    </xs:annotation>
  </xs:element>

  <xs:complexType name="LineType">
    <xs:complexContent>
      <xs:extension base="BaseType">
        <xs:sequence>
          <xs:element name="Point"
          type="PointType"/>
          <xs:element name="Vector"
          type="VectorType"/>
        </xs:sequence>
        <xs:attribute name=”color”
          type=”xs:token”/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

</xs:schema>
```

Figure 3 XML Schema File line.xsd – 2D Line Model

```
<?xml version="1.0" encoding="UTF-8"?>
<Line
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="../schema/line.xsd"
  color=”green”>
  <Name>Line_1</Name>
  <Point>
  <Name>Point_1</Name>
    <X>0</X>
    <Y>0</Y>
  </Point>
  <Vector>
    <Name>Vector_1</Name>
    <X>0</X>
    <Y>1</Y>
  </Vector>
</Line>
```

Figure 4 Instance File line1.xml – Instance of 2D Line

4 Using the xmlSchemaParser

Using the xmlSchemaParser is indicated by Arrows A and C on Figure 1.

4.1 What the xmlSchemaParser Does

The xmlSchemaParser reads and writes XML `schema` files. It handles almost all of XSDL. When it runs, it reads an input file, stores it in terms of a C++ class model of XML `schema`s, and writes it in a file with almost the same name as the input file; “echo” is appended to the file name. The output file is formatted to be easily readable for humans who can read XSDL files directly. While it runs, the xmlSchemaParser prints what it is reading in the command window in which it is running. If there is any syntax error, the xmlSchemaParser stops reading at the point where the first error occurred, prints an error message, and exits; no output file is generated. The xmlSchemaParser also checks for some pointless constructs and issues warnings, as described below.

The xmlSchemaParser (when invoked with the `-h` option) will also print the type derivation hierarchy of the types defined in the `schema`. If a type in the `schema` is derived from some other base type (by `extension` or `restriction`), the base type is higher in the hierarchy than the derived type. If the base type is from an `included`

schema, the xmlSchemaParser will not know whether the base type is derived from some other type, and if so, that other type will not be shown in the hierarchy.

The -d option may be used to reformat or remove documentation nodes.

If the xmlSchemaParser is invoked with the -c option, it will preserve comments. Otherwise, comments will be removed.

If the xmlSchemaParser is invoked with the -k option, it will remove key, keyref, and unique declarations. Otherwise, those items will be kept in the output file.

4.2 Invoking the xmlSchemaParser

If the xmlSchemaParser is invoked with no arguments, as shown below, it prints examples of valid invocations.

bin/xmlSchemaParser

```
usage: xmlSchemaParser [-d <docPrint>] [-c] [-h] [-k] <schema>
Optional arguments may be given in any order.
<docPrint> controls documentation printing and
must be asIs, indent, left, or none
-c means keep comments included in the input schema
-h means generate a class inheritance hierarchy
-k means do not print key, keyref, or unique
<schema> is the XML schema file to read
Example 1: bin/xmlSchemaParser plan.xsd
Example 2: bin/xmlSchemaParser -d indent plan.xsd
Example 3: bin/xmlSchemaParser -k -d asIs -c plan.xsd
Example 4: bin/xmlSchemaParser -c -h plan.xsd
defaults are asIs documentation, remove comments
do not make inheritance hierarchy, keep id constraints
```

Using the XML schema file primitives.xsd shown in Figure 2, suppose the following command is given from the directory containing that file:

bin/xmlSchemaParser -d indent -h primitives.xsd

An output file named primitives.xsdecho will be created with the schema pretty-printed in it as shown in Annex A. Since the **-c** flag was not used, the comment on the sixth line of the file does not appear in the output. Since the **-h** flag was used, a (very simple) hierarchy file named primitivesHierarchy.txt will be printed as shown in Figure 5. It indicates that PointType and VectorType are derived from BaseType. Each derived type is listed below its parent type and indented more deeply than the parent type.

```
BaseType
   PointType
   VectorType
```

Figure 5 Type Hierarchy File

The meanings of the choices for printing `documentation` nodes are:

- **asIs** means do not change the white space in `documentation` nodes.

- **indent** means pretty-print `documentation` nodes so that each line begins indented (by using space characters) two spaces more than `<documentation>` and ends after at most 75 characters (including the spaces) have been printed on the line. So that the user may put formatted text such as lists into `documentation` nodes, however, a `documentation` node is not reformatted if all of the following conditions are met:
  - Each line of the documentation is indented at least as much as it would be if it were pretty-printed.
  - At least one line is indented exactly two spaces more than it would be if it were pretty-printed.
  - No line extends beyond 75 characters, including the spaces at the beginning.

  Also, to give the user leeway to format manually, a `documentation` node is not reformatted if all of the following conditions are met:
  - Each line of the documentation is indented exactly as much as it would be if it were pretty-printed.
  - No line extends beyond 75 characters, including the spaces at the beginning.

- **left** means collapse all the white space so that words are separated by just one space. This is beneficial when a `schema` is going to be fed to a tool that generates fancy documentation, since such tools may keep the existing white space and add more.

- **none** means remove all `documentation` nodes entirely along with the enclosing `annotation` nodes. This is useful for studying the structure of a `schema` file or for reducing its size before further processing that does not deal with `documentation` nodes. Note that any `appinfo` will also be removed.

The **–k** option, which removes `key`, `keyref`, and `unique` constraints, is useful for dealing with large `schema`s with many of those items. Commonly available commercial XML tools may have difficulty with such `schema`s – for example being unable to load if constraint checking is turned on and taking a very long time to load even if constraint checking is off. Hence, it may be convenient to remove constraints so that a commercial tool may be used readily.

4.3 Warnings Issued

The xmlSchemaParser prints warnings for eight situations. Each of these situations is legal XSDL but accomplishes nothing useful while making the `schema` more complex. In each case, the warning message includes the number of occurrences.

- An empty `documentation` node is found.
- An empty `sequence` is found.
- An empty `all` is found.
- An empty `choice` is found.
- A `choice` node is found with only one choice.
- A `sequence` is found nested inside another `sequence`, and the enclosing `sequence` has nothing else in it.
- A `choice` is found nested inside a `sequence`, and the enclosing `sequence` has nothing else in it.
- A `name` is found with white space inside the quotes surrounding the name.

4.4 Limitations of the xmlSchemaParser

The xmlSchemaParser can handle most constructs allowed in an XML `schema` file, but it does have the following limitations.

4.4.1 One schema file

The executable xmlSchemaParser deals with only one `schema` file. If the file `include`s any other `schema` files, they are not read.

4.4.2 Comment Location Limited

The xmlSchemaParser cannot handle comments in the middle of a type definition, `element` definition, or other construct. Any comments must be between such items.

4.4.3 Constraints Not Checked for Validity

Although the xmlSchemaParser reads and writes `unique`s, `key`s, and `keyref`s, it does not check that they are valid.

4.4.4 Prefix Required for XML Schema Namespace

The XML schema namespace http://www.w3.org/2001/XMLSchema must be assigned a prefix. The prefix is usually xs or xsd, although some other prefix may be used.

4.4.5 Only One User-defined Namespace

There may be only one user-defined namespace (i.e., a namespace other than the XML schema namespace and the schema instance namespace). The user-defined namespace may have both an empty prefix and a non-empty prefix. Redeclarations of the XML schema namespace inside elements and types are allowed but are not included in the output schema.

4.4.6 Not All Qnames

Some names in XML (and XSDL) may have prefixes separated from the rest of the name by a colon. These are called Qnames. In the generator, some Qnames do not have prefixes implemented.

4.4.7 Not All XSDL

The following XSDL constructs are not implemented and will cause a parse error if included in the schema being parsed.

- restriction types: fractionDigits, totalDigits, whiteSpace
- redefine
- anyAttribute
- wildcards
- block
- notation

4.4.8 Limited Regular Expressions

Not all regular expressions that may appear in patterns allowed by XSDL are allowed. All regular expression syntax as described on pages 180-200 of [1] and in https://www.regular-expressions.info/xml.html is supported other than the following:

- block escapes (for example, \p{IsBasicLatin})
- multi-character escapes (which have the form \p{*XX*})
- Unicode character representations (which have the form &#*XXXX*;).

4.4.9 Final Not Enforced

The final construct is parsed, but the restrictions on type derivation it makes are not enforced.

4.4.10 Data Types Not Checked

The xmlSchemaParser is not checking that data types used in a `schema` are valid data types. For example, if a data type of `noSuchType` were used in a `schema`, the xmlSchemaParser would not signal an error or warning.

5 Using the xmlInstanceParserGenerator

Using the xmlInstanceParserGenerator is indicated by Arrows D and B on Figure 1.

5.1 What the xmlInstanceParserGenerator Does

The xmlInstanceParserGenerator reads an XML `schema` file and all other XML `schema` files connected to it by `include`. It also, optionally, reads an existing C++ header file. It generates code files.

5.1.1 Parse Schema

The xmlInstanceParserGenerator uses the same parser as the xmlSchemaParser to read `schema` files and store them in terms of C++ classes. However, unlike the xmlSchemaParser, the xmlInstanceParserGenerator:

- parses all `schema` files connected to a top-level `schema` file by `include`s.
- checks that all data types are either XSDL basic (i.e., built-in) data types or defined data types.

5.1.2 Generate Code Files

The files that the xmlInstanceParserGenerator generates from the *domain*.xsd XML `schema` file (where *domain* may be any name allowed by XSDL and C++) are:

- *domain*.lex – a Lex file for a lexical scanner used by the YACC parser.
- *domain*.y – a YACC file for a parser for XML files in the domain.
- *domain*Classes.hh – a C++ header file defining classes for the domain. Each class has one to three constructors, a destructor, and a printSelf function (for a `complexType`) or two printSelf functions (for a `simpleType`)[1]. For most classes there are a number of data fields. Data fields are always pointers. The header file may also have "get" and "set" functions for accessing values of protected fields.
- *domain*Classes.cc – a C++ code file implementing the classes.
- *domain*Parser.cc – a C++ code file with a main program.

[1] See section 11.4 for more information about the printSelf functions.

The *domain* parser, Lex, and YACC files are generated only if the `schema` file whose name is used in the command that starts the generator has a root `element` at the beginning.

If the XML `schema` file on which the xmlInstanceParserGenerator is operating `include`s one or more other XML `schema` files, a pair of *domain*Classes C++ files is generated for each additional `schema` file, but there is still only one Lex file, one YACC file, and one main *domain* parser file.

The executable *domain* parser that is generated from *domain*Parser.cc is a simple application of the rest of the code (parser and C++ classes). The rest of the code may be reused in other applications.

All output code files are carefully formatted to be human readable – if the reader is familiar with the language in which the file is written.

5.1.2.1 Constructors

One of the constructors for every class takes no arguments (unless `xsi:type` is allowed as described below). That constructor sets the values of all the data fields to a null pointer. If the type from which the class was constructed is a `complexType` with no contents, then there are no data fields and only this one constructor is defined.

If the type from which the class was constructed is a `complexType` with a `sequence` or `sequence`s, a second constructor is defined that takes values for all the `element`s of the `sequence`(s) as arguments.

If the type is as described in the preceding paragraph and also has `attribute`s, a third constructor is defined that has arguments that are values for all the `attribute`s and `element`s.

If `xsi:type` is allowed, the constructors just described also have printTypp as an additional argument at the end of the arguments. If there would otherwise be no arguments, printTypp has a default value of 0 so that the constructor can be called with no arguments.

Second constructors for classes representing other XSDL constructs (`choice`, `enumeration`, etc.) or lists are different. Examples are provided in the in-line documentation of the code and, hence, in the HTML documentation.

5.1.2.2 Destructor

The destructor that is generated for each class normally has the effect of deleting all the data fields of the class. However, if the code is compiled with NODESTRUCT defined, the data fields are not deleted[2].

5.1.3 Save User Changes to C++ Header

The xmlInstanceParserGenerator is able to preserve changes made manually to the automatically generated *domain*Classes.hh header file if the input `schema` is modified and the header file is regenerated. If the arguments to the command that starts the xmlInstanceParserGenerator include **-h *domain*Classes.hh**, where *domain*Classes.hh is the manually changed header file, any allowed changes in the old header file will be transcribed into the corresponding positions in the new header file that is generated. Details are given in Section 5.5.

5.1.4 Pattern Restrictions

XML Schema Definition Language includes a method of restricting string-like data values in instance files by specifying the character pattern that the data must have. Regular expression syntax is used to describe the `pattern restriction`s. When the generator parses a `schema` file containing a `pattern restriction`, it is checked that the regular expression in the `restriction` is a valid regular expression. How this is done is described in section 10.3.2.

The generator writes code to check that an instance of data subject to a `pattern restriction` conforms to the `pattern`. The code that is written calls the boost regular expression parser [2]. When an instance file is being checked, the `pattern` and the data are passed to the boost parser. Boost regular expressions are almost a superset of XSDL `pattern` regular expressions, but there are minor differences. For example `$` is a special character in boost but not in XSDL. If boost cannot handle the `pattern`, a warning message is printed saying that the `pattern` is not being checked. If boost can check the `pattern`, the data is checked. If the check fails, an error message is printed and the instance parser exits.

Since boost is used, if the `schema` used to build an instance parser includes `pattern restriction`s, boost must be linked in when the instance parser is compiled. The generator always puts a #include <boost/regex.hpp> line in the .cc file it generates for a `schema` file. That can be deleted if the file has no `pattern restriction`s.

[2] See section 5.3.2.4 for more on destructors.

5.2 Invoking the xmlInstanceParserGenerator

If the xmlInstanceParserGenerator is invoked with no arguments or if the invocation has incorrect arguments, the generator prints a message containing instructions on how to invoke it and examples of valid invocations. The message is shown below.

```
usage: xmlInstanceParserGenerator [-a <app include prefix>] [-f
getset] [-h <header>] [-i <include prefix>] [-o <output type>]
[-p <prefix>] -s <schema> [-x]
arguments may be given in any order
<schema> is the XML schema file to read
<header> is the existing header file
<output type> is one of: file, string, or macros
<prefix> is the prefix to use in YACC and Lex files
<include prefix> is the prefix for these header files
<app include prefix> is the prefix for application header files
-f getset means generate "get" and "set" access functions for
   fields in the classes, and make the fields protected
-x means allow xsi:type in instance files

Example 1: xmlInstanceParserGenerator -s plan.xsd -x -o macros
Example 2: xmlInstanceParserGenerator -p yypl -f getset -s plan.xsd
Example 3: xmlInstanceParserGenerator -s plan.xsd -h planClasses.hh -f getset
Example 4: xmlInstanceParserGenerator -i generator/ -a crcl/ -s plan.xsd
```

The command arguments (**-x** or a flag-name pair) may be given in any order. The **-s schema** argument is required. All other arguments are optional. The base name of generated files is the `schema` name with the ".xsd" removed. In the examples above, the base name for files will be "plan".

The command to start the generator should be given while working in the directory containing the `schema`. The generated files will be placed in that directory.

In all of the examples above, the name of the `schema` file being processed is plan.xsd. However, if plan.xsd `include`s other `schema` files, those files and any `schema` files they `include` (and so on) will also be processed.

5.2.1 App Include Prefix (-a)

The app include prefix is added to the beginning of paths in #include statements in code that is generated. For instance, in `example 4` above, the base name used for code files will be "plan" and the app include prefix is "crcl/", so the statement #include crcl/planClasses.hh will appear in planClasses.cc, plan.y, and plan.lex.

5.2.2 Get and Set (-f)

By default, all fields of the generated C++ classes are public, so they may be accessed directly (with object.field or object→field). However, if **-f getset** is included in the command that starts the generator, the fields of classes are protected and "get" and "set" functions are defined for obtaining and setting the values of fields. These are used with code such as object.getfield() and object.setfield(value) or object→getfield() and object→setfield(value).

5.2.3 Header Update (-h)

As described in Section 5.1.3, the **-h** flag is used with the name of an existing automatically generated header file in which manual changes have been made. That header file must be for an earlier version of the `schema` being processed. In `Example 3` above the existing header file is named planClasses.hh. Manually made changes that are specially marked in the existing file will be transcribed into the newly generated file. Details are given in Section 5.5.

5.2.4 Include Prefix (-i)

The include prefix is added to the beginning of the path for xmlSchemaInstance.hh in #include statements in the XXXClasses.cc files that are generated. For instance, in `Example 4` above, which has `-i generator/`, the planClasses.cc file will have #include generator/xmlSchemaInstance.hh. The xmlSchemaInstance.hh file is needed because it declares classes for the basic XML data types.

5.2.5 Output (-o)

From a logical point of view, an XML instance file is just a string of characters. The characters may appear in a file or in a string. A user might prefer one or the other.

If the **-o** flag is not used, the code generated by the generator puts output into a file.

If the **-o** flag is used, it controls whether the code generated by the generator puts output into a file or into a string. There are three choices.

- If "file" follows the **-o** flag, the code generated by the generator puts output into a file.
- If "string" follows the **-o** flag, the code generated by the generator puts output into a string.
- If "macro" follows the **-o** flag, the code generated by the generator contains macros that allow output to go into either a file or a string, as determined by a flag used when compiling the code.

The generated code is easiest to read when it puts output in a file.

The input to an automatically generated parser may also come from either a file or a string. The default is file input, but by compiling with STRINGIN defined, the input will be taken from a string. No generator argument is needed.

5.2.6 Prefix (-p)

The **-p** flag with a <prefix> (such as `yypl` shown in `Example 2` above) refers to the prefix for global symbols used by YACC and Lex files. This prefix has a default value of `yy`. With that default value, the parsing function that is built is called yyparse. If two or more parsers are to be included in a process, each parsing function (and all the other global symbols) must have a different name. In `Example 2`, the parsing function that is generated will be named yyplparse, and all the other global symbol names used by YACC and Lex will also start with yypl.

5.2.7 Xsi:type (-x)

The **-x** flag means that `xsi:type` declarations should be allowed with `element`s. The rules for what is allowed in an XML instance file that conforms to an XML `schema` provide that `xsi:type` may always be used with `element`s that are of `schema`-defined type.

Some `schema`s are written in such a way that an instance file cannot be written without using `xsi:type`. Such `schema`s must be processed with the **-x** flag.

However, by using `substitutionGroup`s, it is usually feasible, regardless of the information content of a `schema`, to build the `schema` so that `xsi:type` is never required in an instance file. Since `xsi:type` is unnecessary characters requiring parsing in instance files conforming to such `schema`s, the users of such `schema`s may adopt a business rule that `xsi:type` may not be used. That is the default behavior of a parser generated by the generator; the parser will signal an error if `xsi:type` is encountered. By disallowing `xsi:type`, the YACC file will be significantly smaller. With one international standard, the YACC file is reduced from 165592 lines to 103793 lines.

Further information about how the generator deals with `xsi:type` is given in section 11.6.4.

5.3 Processing Generated Files

After the xmlInstanceParserGenerator has finished running, further processing builds a Domain Instance XML Parser from the code that has been generated. Using the parser is described in section 8.

5.3.1 Processing by Flex and Bison

The flex Lex processor [3] is used to generate the C++ file *domain*Lex.cc automatically from domain.lex. The bison YACC processor [4] is used to generate *domain*YACC.cc and

*domain*YACC.hh automatically from *domain*.y. Calls to flex and bison are included in the Makefile of section 9.3.2.

5.3.2 Compiling

The four (or more) .cc files are then compiled and linked by using a Makefile in any operating system that uses standard Makefiles (see 9.3.2 for an example) or by using Visual Studio for MS Windows.

The parsing function (default name yyparse) is used in the executable domain instance parser that is created by compiling. The parsing function may also be used in any program that uses input from an XML instance file that conforms to *domain*.xsd. For such a program, all the object files created from code written by the xmlInstanceParserGenerator (except for *domain*Parser.cc) must be linked in when the program is compiled.

A variety of behaviors of domain parsers is available depending on

- options chosen when generating the code for the parser
- compiler flags used when compiling the code
- the version of the object file made from xmlSchemaInstance.cc that is linked with the parser code.

XML has a number of basic datatypes such as `string` and `integer`, as described in [1] and [5]. These datatypes are modeled in xmlSchemaInstance.hh and xmlSchemaInstance.cc. An object file built from those files must also be linked into the executable domain instance parser or any program that creates XML instance files for the domain. By using compile flags, four versions of the object file can be compiled, depending on whether `xsi:type` is allowed and whether output should go to a string. More information about this is given in section 5.3.2.3 and section 14.

5.3.2.1 Echo While Parsing – or Not

By default, the input (an XML instance file) to a domain parser will be echoed as it is read. That is often helpful since if there is a parse error, the parser stops where the error occurs. A user can get a good idea of where the problem lies by looking at the last text that was echoed.

A user might not want to have the input echoed. That can be achieved by using the NO_ECHO flag when the lex.cc file is compiled.

5.3.2.2 Taking Input from a File or a String

The underlying parser of a domain parser can take input from either a file or a string. This is determined entirely by a compiler flag.

If an executable domain parser is compiled with STRINGIN defined, it will read the input file into a string and then parse it from the string; the underlying parser takes input from a string in this case. Otherwise it will parse directly from the file.

5.3.2.3 Printing Output to a File or a String

The printing routines of a domain parser will print either to a file or to a string. This is determined by all three bulleted items above.

5.3.2.3.1 *Always to a File*

If the code was generated with no **-o** option or if **-o file** was used, domain parser output will go to a file, and no compiler flag for output should be used when compiling the domain parser. If the **-x** flag was used when the code was generated, xmlSchemaInstanceXsi.o (xmlSchemaInstance.cc compiled with USEXSITYPE defined) must be linked in. Otherwise, xmlSchemaInstance.o must be linked in.

5.3.2.3.2 *Always to a String*

If the code was generated with **-o string**, domain parser output will go to a string, and the flag STRINGOUT must be used when compiling the domain parser. In this case printing to the string will include line endings but no other extra whitespace. If the **-x** flag was used when the code was generated, xmlSchemaInstanceStrXsi.o (xmlSchemaInstance.cc compiled with USEXSITYPE and STRNGOUT defined) must be linked in. Otherwise, xmlSchemaInstancStr.o must be linked in.

5.3.2.3.3 *To Either String or File*

If the code was generated with **-o macros**, it contains macros that will output to either a file or a string, depending on how the code is compiled.

If the flag STRINGOUT is used when compiling the domain parser, output will be sent to a string. In this case, if the **-x** flag was used when the code was generated, xmlSchemaInstanceStrXsi.o (xmlSchemaInstance.cc compiled with USEXSITYPE and STRINGOUT defined) must be linked in. If the **-x** flag was not used when the code was generated, xmlSchemaInstanceStr.o (xmlSchemaInstance.cc compiled with STRINGOUT defined) must be linked in.

If the flag STRINGOUT is not used when compiling the domain parser, output will be sent to a file. In this case, if the **-x** flag was used when the code was generated, xmlSchemaInstanceXsi.o (xmlSchemaInstance.cc compiled with USEXSITYPE defined) must be linked in. If the **-x** flag was not used when the code was generated, xmlSchemaInstance.o must be linked in.

5.3.2.4 Destructors and Clearing Memory

In many applications, it will be desirable to process a series of XML instance files conforming to the same `schema`. In such applications, it will be desirable (or necessary)

to remove the abstract syntax tree representing one file before going on to the next file. The code that is generated implements two methods for doing that. The choice is determined by a compiler flag. If it is not necessary to clean abstract syntax trees out of memory while an application is running, it does not matter which way the flag is set when the code is compiled. In that case the memory will be recovered automatically when the application exits. The rest of this section assumes it is necessary to reclaim memory while the application is running.

5.3.2.4.1 *First Method*

In the first method, implemented by default, all objects should be created using new, for example:

```
point * pointA;
point A = new point;
line->firstPoint = pointA;
```

This method is used when the automatically generated parser is used to read files. Hence, if an application uses the parser to read an existing file and then modifies the file, the application programmer must use this method. This method may also be used when generating instance file trees programmatically.

The destructors that are generated in the C++ classes delete all of the fields of the class. By doing that, all memory for a tree of class instances is freed by deleteing the root of the tree.

Since all fields are pointers, if any object in the tree has been referenced more than once, there will be an error when the tree is deleted. One method of avoiding that is to remove all references but one to an object before deleteing the tree. Having to keep track of references to an object, however, would be a significant burden for the programmer. A much simpler method for dealing with this is not to create multiple references to any object. This is feasible because an XML file is a tree with no cross links. To implement this, it may be necessary to make copies of objects, but that is easy to do. For example, two references to an object named cmm1 might be created as follows.

```
ReferenceType * cmm1Ref1 = new ReferenceType();
cmm1Ref1->setval(cmm1->getid());

ReferenceType * cmm1Ref2 = new ReferenceType();
*cmm1Ref2 = *cmm1Ref1;
```

Making multiple copies in this situation (handling objects with the same content) is what the parser does automatically, and thus is not unreasonably wasteful of memory.

5.3.2.4.2 *Second Method*

In the second method, all objects should be made using automatic variables, for example:

```
point pointA;
line.firstPoint = &pointA;
```

This method may be used when generating trees programmatically.

The second method is implemented by using the flag NODESTRUCT when compiling the .cc files corresponding to the `schema` files. Using that flag stops the destructors from deleteing the fields of the class. In this method, all objects disappear automatically when they go out of scope, and the memory they used is recovered automatically (because that's how C++ works).

5.3.2.4.3 *Recommendation*

In experimenting with the methods, using the first method with no multiple references to the same object appeared to work the most smoothly. Users may wish to try different methods to see what works best in particular applications.

5.4 Example – Generating from Line.xsd

Using the XML `schema` file line.xsd shown in Figure 3, suppose the following command is given from the directory containing that file:

bin/xmlInstanceParserGenerator -f getset -x -s line.xsd

Seven files will be generated, five for line.xsd itself and two for primitives.xsd (shown in Figure 2), which is `included` by line.xsd:

- primitivesClasses.hh - Annex B
- lineClasses.hh - Annex C
- primitivesClasses.cc - Annex D
- lineClasses.cc - Annex E
- lineParser.cc - Annex F
- line.y - Annex G
- line.lex - Annex H

The first four of those will implement "get" and "set" functions. The line.lex and line.y files will allow `xsi:type`.

The Makefile for this example is shown in section 9.3.2.

5.5 Saving Header File Changes Made Manually

A well-known drawback with automatically generated code is that a user may modify the code, but the modifications will be lost if the code is regenerated. As mentioned earlier, the xmlInstanceParserGenerator is able to preserve some types of changes made manually to an existing automatically generated header file, if the header file is being regenerated. The code generated by the xmlInstanceParserGenerator is formatted to be

human-readable and is relatively straightforward, so that making changes manually is feasible.

The xmlInstanceParserGenerator is able to preserve manually-made modifications if the modifications are additions to the classes – entirely new classes or new fields or functions in existing classes. In C++, this is not an insurmountable problem for implementation (.cc) files because new implementation code can be put into a separate file that is linked in during compilation. For header (.hh) files, however, it is not possible to write a separate header file that modifies an existing class; any changes have to be in a single header file.

Two types of manual changes to header files can be preserved. First, immediately after the list of #includes near the top of the file, a two-slashes-style (i.e., //) comment line may be inserted followed by more #includes. Second, immediately before the right curly brace that closes each class declaration, a // style comment line may be inserted followed by any lines that are syntactically correct in that position (for example, a field declaration or a constructor declaration).

To accomplish the transcription of changes in class declarations, when the xmlInstanceParserGenerator starts, it reads the old header file and builds a std::map from class names to std::lists of character arrays containing the changes. When the new header is being printed, just before the printing of each class ends, if the std::map has an entry for the class, the contents of the std::list of changes for that class are copied into the new header file. At the same time, “done” is put at front of the std::list of changes to indicate that the changes for that class have been transcribed. After the new header file has been generated, the change std::map is checked to be sure all changes are marked done. If a change is not marked done, that implies that a class defined in the old header file is not present in the new one, and a warning message is printed.

Any manually written code implementing changes in the header file, such as a new constructor, should be put into a separate .cc file, not into *domain*Classes.cc. Otherwise, the changes will be lost if *domain*Classes.cc is regenerated.

Augmenting C++ code by adding fields and functions to classes to support building an application is frequently done, so being able to preserve manual changes to header files is valuable.

5.6 Limitations of the xmlInstanceParserGenerator

The xmlInstanceParserGenerator has the following limitations. A few other statements allowed in XSDL not listed here cannot be handled but are expected to be encountered rarely. Those that have been identified are noted in the in-line documentation of the generator with the word “FIX”. There may be other limitations not yet discovered.

5.6.1 Limitations Inherited from xmlSchemaParser

The generator uses the parse function from the xmlSchemaParser. Hence, except as noted in Section 5.1.1, it has all of the limitations of the xmlSchemaParser, as given in Section 4.4.

5.6.2 Type Definitions Must Be At Schema Level

The generator handles only `schema`s in which all type definitions are at the `schema` level; that ensures that every type has a name. This is not a major limitation because embedded type definitions can easily be moved to the top level in a way that lets the revised `schema` handle exactly the same instance files as the original `schema`. New types are defined, but no new `element`s are introduced.

5.6.3 No Code to Check Some Constraints

While the generator does write code that checks many types of constraints, some types of constraints can be parsed for which no enforcing code is generated. Specifically, the generator does not generate code to verify that an instance file satisfies `fixed`, `unique`, `key`, or `keyref` constraints in the `schema`.

5.6.4 Several Constructs Not Handled

The generator is not able to generate code dealing with following constructs that the xmlSchemaParser is able to parse. If one of the constructs is encountered, the xmlInstanceParserGenerator prints a “cannot handle” error message and exits.

- `complexContent restriction`
- `simpleContent restriction`
- `enumeration`s of numbers
- `restriction`s of strings other than `enumeration`s and `pattern`s
- `all`
- `a choice with no items`
- `a sequence with no items`
- `an extension of an extension that adds a choice`
- `a non-element in an element group sequence`
- `maxOccurs or minOccurs of an element group reference`
- `maxOccurs or minOccurs of a sequence`
- `duplicate elements in a sequence or choice`
- `an optional item in a choice (which is pointless)`
- `an element group with no sequence (which is pointless)`

5.6.5 Some Basic Data Types Not Handled

The generator handles 26 of XSDL’s basic (i.e., built-in) data types. However, the `date`, `time`, and `dateTime` data types are handled as strings, and the following data types

are not handled: `byte`, `ENTITY`, `ENTITIES`, `hexBinary`, `IDREFS`, `language`, `Name`, `NCName`, `NMTOKENS`, `normalizedString`, `NOTATION`, `QName`.

5.6.6 Names Not Guaranteed to be Unique

C++ gives the dash (-) and period (.) characters special meaning, so they cannot be used in C++ names. They can be used in XSDL names. For use in C++ code, the generator changes dashes and periods in names used in the `schema` to underscores. If a `schema` contains two names that are identical except that one has an underscore where the other has a dash or a period (e.g., half-track and half_track), an error will occur. If duplicate type names result, the xmlInstanceParserGenerator will print a "duplicate class name" error message and exit. If duplicate field names in a class result, compiler errors will occur.[3]

Names in a `schema` that contain underscores and are similar can result in duplicate names in the generated code. For example, if the `element` foo_bar has type baz and the `element` foo has type bar_baz, the name foo_bar_baz may be generated twice for different uses, leading to processing errors.

5.6.7 Prefixes ignored

The generator does not consider the prefix when keeping track of the names of `simpleType`s and `complexType`s defined in the `schema`. Hence, no type definitions in the `schema` may have the same name as a basic type. For example, a `simpleType` definition with the name `co:string` would not work because it is too similar to the basic `xs:string`.

5.6.8 Not All Patterns Handled

Because, as described in section 4.4.8, not all regular expressions allowed by XSDL are parsable, the generator cannot handle all character `pattern`s.

6 Using the orphanFinder

Using the orphanFinder is indicated by Arrows G and H on Figure 1.

6.1 What the orphanFinder Does

The orphanFinder examines one or many `schema` files and finds:

- types that are used but are not defined
- types that are defined but are not used

[3] C++ reserved words also cannot be used as names. The generator changes names that are C++ reserved words by appending "_AltNaym" to the name.

XSDL allows defining `simpleType`s and `complexType`s that are not used anywhere. This is pointless, but it is easy for it to occur in a complex `schema` that is revised several times. Using a type that is not defined is also an easy mistake to make – by misspelling the name, for example.

6.2 Invoking the orphanFinder

If the orphanFinder is invoked with no arguments, it prints the following message explaining how to make a correct invocation.

```
Usage: orphanFinder <file> <file> ...
Where each <file> is the name of an XML schema file
```

Only the named files are examined; chains of `include`s are not followed. For example, the file line.xsd shown in Figure 3 `include`s the file primitives.xsd shown in Figure 2 and uses types defined in primitives.xsd.

Because the orphanFinder does not parse `schema` files, it is a good idea to use the xmlSchemaParser first on any `schema`s to be fed to the orphanFinder. The file echoed by the xmlSchemaParser can be used as input to the orphanFinder if the original `schema` file is valid but not in good format.

6.3 Examples of Using the orphanFinder

If the command **orphanFinder line.xsd** is given, the following is printed in the command window.

```
DEFINED DATA TYPES NOT USED

USED DATA TYPES NOT DEFINED
BaseType - line.xsd
PointType - line.xsd
VectorType - line.xsd
```

This indicates that there are no defined data types that are not used but that there are three data types that are used but not defined.

On the other hand, if the command **orphanFinder line.xsd primitives.xsd** is given, the following is printed.

```
DEFINED DATA TYPES NOT USED
```

```
USED DATA TYPES NOT DEFINED
```

This indicates that there are no defined data types that are not used and no data types that are used but not defined.

As shown above, the name of the `schema` file is printed on each line of the output that names a type. If an undefined data type is found that is used in more than one `schema` file, the data type name is printed only once, and each file name is printed on a separate line.

6.4 Limitations of the orphanFinder

The orphanFinder does not use the parse function from the xmlSchemaParser (or any other parser), so it does not have the limitations of the xmlSchemaParser. It works by looking for text patterns. Hence, any sort of error in a `schema` other than using an undefined type will not be detected.

The orphanFinder requires that all `complexType` and `simpleType` definitions be at the top level of a `schema` starting on a new line.

The orphanFinder requires that the XML `schema` prefix be `xs`.

7 Using the xmlSchemaAttributeConverter

Using the xmlSchemaAttributeConverter is indicated by Arrow F on Figure 1.

7.1 What the xmlSchemaAttributeConverter Does

The xmlSchemaAttributeConverter reads an XML `schema` file, checks the syntax of the file, builds an abstract syntax tree, changes all `attribute`s in the tree to `element`s, and reprints the file from the modified tree. The name of the reprinted file is the name of the original file with "NoAtt" added before the ".xsd" at the end.

If the `schema` file has `include`s, each `include`d `schema` file is also converted (and so on, so that the entire tree or graph of `include`d files is converted).

All references to global `attribute`s and global `attributeGroup`s are replaced by `element`s with types. The global `attribute`s and global `attributeGroup`s are removed. Local `attribute`s are also removed after being replaced by `element`s.

If a `complexType` with `simpleContent` extends a basic type or `simpleType` by adding `attribute`s, it is converted to a `complexType` with `complexContent`. The

new type is not an `extension` of the original type. Hence, the original type cannot be used as the head of a `substitutionGroup` that includes the new type or its descendants.

Because the default for `attribute`s is to be optional, while the default for `element`s is to be required, if an `attribute` is `required`, the corresponding `element` has no `maxOccurs` or `minOccurs`, while if an `attribute` is not `required`, the `element` has `minOccurs="0"`.

`Documentation` nodes are reprinted indented but with no other change. In particular, if a `documentation` node describes what it is documenting as an `attribute`, that documentation remains unchanged in the converted file and becomes misleading.

7.2 Invoking the xmlSchemaAttributeConverter

If the xmlSchemaAttributeConverter is invoked with no arguments, it prints the following message explaining how to make a correct invocation.

```
usage: xmlSchemaAttributeConverter <schema>
<schema> is the XML schema file to read
Example: xmlSchemaAttributeConverter plan.xsd
```

7.3 Example of Using the xmlSchemaAttributeConverter

If the command **xmlSchemaAttributeConverter line.xsd** is given, the files lineNoAtt.xsd and primitivesNoAtt.xsd will be written. The primitivesNoAtt.xsd file is identical to primitives.xsd since it does not deal with `attribute`s. The file lineNoAtt.xsd shown in Annex I, however, differs from line.xsd in the obvious way that the two lines declaring the color `attribute` have been removed, and there is one more `element` line. The new color `element` appears after the other `element`s.

The lineNoAtt.xsd file can be processed by the other tools like any other `schema` file.

7.4 Limitations of the xmlSchemaAttributeConverter

The xmlSchemaAttributeConverter uses the parse function from the xmlSchemaParser, so it has the same limitations, as given in Section 4.4.

`Key`s, `keyref`s, and other constraints that reference `attribute`s are not being converted to referencing `element`s. Hence, such constructs become invalid in the output file.

8 Using Domain Instance XML Parsers

Using a *Domain* Instance XML Parser (*domain* parser, for short) is indicated by Arrow D on Figure 1. A *domain* parser is created by compiling the output of the xmlInstanceParserGenerator as indicated by Arrow B on Figure 1 and described in Section 5.3. An example Makefile is shown in section 9.3.2.

8.1 What a Domain Parser Does

A *domain* parser reads and writes XML instance files intended to conform to the *domain*.xsd information model. The name of an output file (if there is only one output file – see section 8.2.2) is the name of the input file with "1" appended. If there are no errors, the contents of the output file are identical to the contents of the input file except, possibly, for:

- the location white space (spaces, tabs, newlines, carriage returns).
- the use of exponential notation and number of decimal places in numbers that are not whole numbers, which may be specified in the call to the *domain* parser (see section 8.2.1 for details).

The methods of handling the printing of numbers (as described in section 8.2.1) make it feasible in many cases for the output file to be identical to the input file. Without those methods, a number such as 3.5 in the input file might be changed to something like 3.49999999999 in the output file.

The output file is formatted for human readability, so *domain* parsers are useful for changing instance files from unreadable to readable.

Tabs are not used in output files, so input and output files that appear to be identical may be found to differ by utilities that compare files. Trailing white space on lines of input files will also cause differences.

A *domain* parser requires strict conformance of instance files to the syntax implied by the *domain*.xsd `schema`. A *domain* parser (by default) prints what it is reading in the command window in which it is running. Echoing input while reading can be turned off by a compiler flag as described in Annex H. If there is any syntax error, the *domain* parser stops reading at the point where the first error occurred, prints an error message, and exits; no output file is generated.

Domain parsers check that all values of the XML basic `ID` type in an instance file are unique and that every `IDREF` value is the value of an `ID`. They also check all the `restriction`s on data (such as `pattern`s) that the xmlInstanceParserGenerator can handle.

Domain parsers use C++ definitions of the XML basic data types given in the files xmlSchemaInstance.hh and xmlSchemaInstance.cc.

8.2 Invoking a Domain Parser

If a domain parser is invoked with no arguments, it prints a message explaining how to make a correct invocation. For example, if the domain parser "lineParser" generated from line.xsd is called with no arguments, it prints:

```
Usage: lineParser <file name> [-n|N <places>] [-f|F <format>] [<times>]
<places> and <times> are integers; <format> is f, e, or E
Example 1: bin/lineParser dFile.xml
Example 2: bin/lineParser dFile.xml 2
Example 3: bin/lineParser dFile.xml -n 5
Example 4: bin/lineParser dFile.xml -F e 2
Example 5: bin/lineParser dFile.xml -n 6 -F E
```

8.2.1 Numbers with Decimal Points

A domain parser's C++ representation of XML numbers with decimal points (`double`, `float`, and `decimal`) all have fields for (1) the number of decimal places to use when printing (places) and (2) the format to use (format). Those fields are populated when a number is read from a file and their values are determined both by the format in the file and the call to the parser. The values of the places and format fields may also be set programmatically.

In the call to the parser, the [-n|N <places>] and [-f|F <format>] determine how `double`, `float`, and `decimal` are printed in the echoed file. In the following text, DFD means a number of one of those types, and DF means `double` and `float` numbers. <places> is a non-negative integer. <format> is one of f, e, or E and their meaning is as given for printf in the C++ standard (floating point, exponential with lower case e, exponential with upper case E).

If -N is used, all DFDs are printed with <places> decimal places.

If -n is used, DFDs are printed out with the same number of decimal places as they had when they were read in.

If neither -n nor -N is used, DFDs are printed out with the same number of decimal places as they had when they were read in.

The -f and -F options have no effect on `decimal` numbers (since `decimal` numbers may not use exponential notation).

If -F is used all DFs are printed with the given <format>.

If -f is used, DFs are printed out using exponential notation (e or E) if they had exponential notation when they were read in, but otherwise are printed with the given <format>.

If neither -f nor -F is used, DFs are printed out using exponential notation (e or E) if they had exponential notation when they were read in, but otherwise are printed with the f format.

Examples: Suppose a number in dFile.xml is 23.14159265.
1. If the command is **lineParser dFile.xml**, the number is printed as 23.14159265 regardless of which type of number with a decimal point it is.
2. If the command is **lineParser dFile.xml –N 6**, the number is printed as 23.141592 regardless of which type of number with a decimal point it is.
3. If the command is **lineParser dFile.xml –N 5 –F e**, the number is printed as 2.31415E1 if it is `double` or `float`, but as 23.14159 if it is `decimal`.

8.2.2 The Times Argument

The <times> argument gives the number of times the file should be parsed. The argument exists to help test for memory leaks and is not expected to be useful otherwise. The number of output files is the value of <times>, and their names are the name of the input file, with 1, 2, … appended.

8.3 Example of Running a Domain Parser

If the command **lineParser line1.xml** is given, the file line1.xml1 will be printed, and it will be identical to line1.xml as shown in Figure 4 except for a small amount of white space.

8.4 Limitations of Domain Parsers

Domain parsers do not check conformance of instance files to any `key` and `keyref` constraints that may be present in *domain*.xsd.

9 Software Overview

9.1 Code

The code for the tools is written in C++, YACC, and Lex. Modifying the code requires good command of those languages in addition to knowledge of XSDL and XML instance files.

As mentioned previously, the source code for the four hand-written tools (all of which take one or more XML `schema` files as input) is primarily in C++. All of them except the orphanFinder use xmlSchemaClasses.hh and xmlSchemaClasses.cc (classes for representing

XSDL structures), xmlSchema.y (the YACC parser for `schema` files), and xmlSchema.lex (the lexer used by the YACC parser). To deal with XSDL `pattern` constraints, the xmlSchemaParser and the xmlInstanceParserGenerator also use a second YACC-Lex parser built from pattern.y and pattern.lex. Each of the four tools has a C++ file dedicated to its particular job in addition to the other files. The largest of those is xmlInstanceParserGenerator.cc at over 21,000 lines.

9.1.1 Code Documentation

The hand-written code for the tools is heavily documented to provide additional information for programmers. Each C++ function is introduced by a comment section that gives “Returned Value” and “Called By” plus a description of what the function does. Some comment sections include examples. Where the code has dealt with a thorny problem, a description of the problem and how the code deals with it may be included. Much of the text of sections 10 through 15.3 is copied from the in-line documentation.

The in-line documentation is formatted to be processable by the Doxygen documentation generator (https://www.doxygen.nl). Four Doxygen configuration files (one for each tool) are included in the code distribution for generating HTML documentation. The HTML documentation files themselves are also included in the distribution. The HTML may be regenerated by a command such as **doxygen DoxyGenerator** (for the xmlInstanceParserGenerator). Programmers may prefer using the HTML documentation to reading the documentation in the code. The HTML has more information than the code files, such as links to parent types and derived types.

9.1.2 Code Formatting

All of the hand-written code is carefully formatted (with the help of the emacs editor running in the appropriate mode) so that it is easily readable by a programmer. With very few exceptions, lines do not have more than 80 characters.

To make the code easier to follow, the C++ function definitions in the hand-written .cc files are arranged in alphabetical order. The .hh files are not alphabetical because of the need to define a parent class before any of its child classes. Items in xmlSchema.y are also in alphabetical order.

9.2 Differences in Approach Among Tools

Three of the four tools use the same YACC/Lex XML `schema` parser to read in XML `schema` files and build an abstract syntax tree for each file using the classes defined in xmlSchemaClasses.hh and xmlSchemaClasses.cc.

The source code for the automatically generated domain instance XML parsers is described in section 15.

9.2.1 xmlInstanceParserGenerator

The source code for the xmlInstanceParserGenerator defines a generator class containing all the functions needed for the tool as well as a set of variables for data about the XML `schema` being processed. In the xmlInstanceParserGenerator, if `include` commands are used in the `schema`, so that more than one `schema` file is to be processed, a separate instance of the generator class is created for each `include`d file. Each generator uses the YACC/Lex XML `schema` parser to read in the XML `schema` file for the generator.

The other three tools do not use a generator class.

9.2.2 xmlSchemaParser

The xmlSchemaParser processes only one `schema` file. It uses the YACC/Lex XML `schema` parser and the xmlSchemaClasses to build a model of the input `schema` file. The rest of the xmlSchemaParser prints out the model in a format selected by the user.

9.2.3 xmlSchemaAttributeConverter

The xmlSchemaAttributeConverter uses the YACC/Lex XML `schema` parser and the xmlSchemaClasses to build a model of each input `schema` file in a group of `schema` files connected by `include`. It extracts data from all of the models and then processes the extracted data.

9.2.4 orphanFinder

The orphanFinder does not use the YACC/Lex XML `schema` parser. However, it does process all files in a set of `schema` files connected by `include`. It does string processing.

9.3 Makefiles

The tools may be compiled in Linux and similar systems using a Makefile. As may be seen below, the Makefiles are all straightforward.

9.3.1 Makefile for Purely XML Tools

The following is the Makefile for the XML tools. Since building a domain instance parser requires using one of four versions of an object file for xmlSchemaInstance (as described in section 5.3.2), this Makefile also compiles them.

```
LINCOMPILE = g++ -c -v -g -Wall
LINLINK = g++ -v
INCLUD = -Isource -Iinclude
HHDIR = include
bin/orphanFinder: ofiles/orphanFinder.o
     $(LINLINK) -o $@ ofiles/orphanFinder.o
bin/xmlInstanceParserGenerator : ofiles/xmlInstanceParserGenerator.o \
                                 ofiles/xmlSchemaYACC.o              \
                                 ofiles/xmlSchemaLex.o               \
                                 ofiles/xmlSchemaClasses.o           \
                                 ofiles/patternLex.o                 \
                                 ofiles/patternYACC.o
     $(LINLINK) -o $@ ofiles/xmlInstanceParserGenerator.o  \
                      ofiles/xmlSchemaYACC.o               \
                      ofiles/xmlSchemaLex.o                \
                      ofiles/xmlSchemaClasses.o            \
                      ofiles/patternLex.o                  \
                      ofiles/patternYACC.o
bin/xmlSchemaAttributeConverter : ofiles/xmlSchemaAttributeConverter.o \
                                  ofiles/xmlSchemaYACC.o               \
                                  ofiles/xmlSchemaLex.o                \
                                  ofiles/xmlSchemaClasses.o            \
                                  ofiles/patternLex.o                  \
                                  ofiles/patternYACC.o
     $(LINLINK) -o $@ ofiles/xmlSchemaAttributeConverter.o \
                      ofiles/xmlSchemaYACC.o               \
                      ofiles/xmlSchemaLex.o                \
                      ofiles/xmlSchemaClasses.o            \
                      ofiles/patternLex.o                  \
                      ofiles/patternYACC.o
bin/xmlSchemaParser : ofiles/xmlSchemaParser.o  \
                      ofiles/xmlSchemaYACC.o    \
                      ofiles/xmlSchemaLex.o     \
                      ofiles/xmlSchemaClasses.o \
                     ofiles/patternLex.o        \
                     ofiles/patternYACC.o
     $(LINLINK) -o $@ ofiles/xmlSchemaParser.o  \
                      ofiles/xmlSchemaYACC.o    \
                      ofiles/xmlSchemaLex.o     \
                      ofiles/xmlSchemaClasses.o \
                      ofiles/patternLex.o       \
                      ofiles/patternYACC.o
ofiles/orphanFinder.o : source/orphanFinder.cc
     $(LINCOMPILE) -o $@ source/orphanFinder.cc
ofiles/xmlInstanceParserGenerator.o :
                       $(HHDIR)/xmlInstanceParserGenerator.hh \
                       source/xmlInstanceParserGenerator.cc   \
                       $(HHDIR)/xmlSchemaClasses.hh
     $(LINCOMPILE) $(INCLUD) -o $@ source/xmlInstanceParserGenerator.cc
ofiles/patternLex.o : source/patternLex.cc \
                      source/patternYACC.hh
     $(LINCOMPILE) -o $@ source/patternLex.cc
```

```
ofiles/patternYACC.o : source/patternYACC.cc \
                       source/patternYACC.hh
     $(LINCOMPILE) -o $@ source/patternYACC.cc
ofiles/xmlSchemaAttributeConverter.o : $(HHDIR)/xmlSchemaClasses.hh \
                                       source/xmlSchemaAttributeConverter.cc
     $(LINCOMPILE)  $(INCLUD) -o $@ source/xmlSchemaAttributeConverter.cc
ofiles/xmlSchemaClasses.o : $(HHDIR)/xmlSchemaClasses.hh \
                            source/xmlSchemaClasses.cc
     $(LINCOMPILE) $(INCLUD) -o $@ source/xmlSchemaClasses.cc
ofiles/xmlSchemaInstance.o : source/xmlSchemaInstance.cc \
                             $(HHDIR)/xmlSchemaInstance.hh
     $(LINCOMPILE) $(INCLUD) -o $@ source/xmlSchemaInstance.cc
ofiles/xmlSchemaInstanceXsi.o : source/xmlSchemaInstance.cc \
                              $(HHDIR)/xmlSchemaInstance.hh
     $(LINCOMPILE) $(INCLUD) -o $@ -DUSEXSITYPE source/xmlSchemaInstance.cc
ofiles/xmlSchemaInstanceStr.o : source/xmlSchemaInstance.cc \
                                $(HHDIR)/xmlSchemaInstance.hh
     $(LINCOMPILE) $(INCLUD) -o $@ -DSTRINGOUT source/xmlSchemaInstance.cc
ofiles/xmlSchemaInstanceStrXsi.o : source/xmlSchemaInstance.cc \
                                   $(HHDIR)/xmlSchemaInstance.hh
     $(LINCOMPILE) $(INCLUD) -o $@ -DSTRINGOUT -DUSEXSITYPE source/xmlSchemaInstance.cc
ofiles/xmlSchemaLex.o : source/xmlSchemaLex.cc\
                        source/xmlSchemaYACC.hh
     $(LINCOMPILE) $(INCLUD) -o $@ source/xmlSchemaLex.cc
ofiles/xmlSchemaParser.o : $(HHDIR)/xmlSchemaClasses.hh \
                           source/xmlSchemaParser.cc
     $(LINCOMPILE)  $(INCLUD) -o $@ source/xmlSchemaParser.cc
ofiles/xmlSchemaYACC.o : source/xmlSchemaYACC.cc \
                         source/xmlSchemaYACC.hh
     $(LINCOMPILE) $(INCLUD) -o $@ source/xmlSchemaYACC.cc
source/patternLex.cc : source/pattern.lex    \
                       source/patternYACC.cc
     flex -L -t  -Pyyre source/pattern.lex > source/patternLex.cc
source/patternYACC.cc : source/pattern.y
     bison -d -l -p yyre -o $@ source/pattern.y
source/xmlSchemaLex.cc : source/xmlSchema.lex       \
                         source/xmlSchemaYACC.cc    \
                          $(HHDIR)/xmlSchemaClasses.hh
     flex -L -t source/xmlSchema.lex > source/xmlSchemaLex.cc
source/xmlSchemaYACC.cc : source/xmlSchema.y        \
                          $(HHDIR)/xmlSchemaClasses.hh
     bison -d -l -o $@ source/xmlSchema.y
```

9.3.2 Makefile for Example lineParser

The following Makefile is for the lineParser example. It may be used to build lineParser and lineParserStr. See section 5.4, Annex B, Annex C, Annex D, Annex E, Annex F, Annex G, and Annex H for more information about the example.

```
LINCOMPILE = g++ -c -v -g -Wall -DUSEXSITYPE
LINLINK = g++ -v
XTOOLSHH = ../../../tools/include
```

```
XTOOLSO = ../../../tools/ofiles
INCLUD = -Isource -I ../../../tools/include

bin/lineParser : ofiles/lineParser.o          \
                 ofiles/lineClasses.o         \
                 ofiles/lineLex.o             \
                 ofiles/lineYACC.o            \
                 ofiles/primitivesClasses.o   \
                 $(XTOOLSO)/xmlSchemaInstanceXsi.o
     $(LINLINK) -o $@ ofiles/lineParser.o          \
                      ofiles/lineClasses.o         \
                      ofiles/lineLex.o             \
                      ofiles/lineYACC.o            \
                      ofiles/primitivesClasses.o   \
                      $(XTOOLSO)/xmlSchemaInstanceXsi.o

bin/lineParserStrIn : ofiles/lineParserStrIn.o    \
                      ofiles/lineClasses.o        \
                      ofiles/lineLexStrIn.o       \
                      ofiles/lineYACC.o           \
                      ofiles/primitivesClasses.o  \
                      $(XTOOLSO)/xmlSchemaInstanceXsi.o
        $(LINLINK) -o $@  ofiles/lineParserStrIn.o   \
                          ofiles/lineClasses.o       \
                          ofiles/lineLexStrIn.o      \
                          ofiles/lineYACC.o          \
                          ofiles/primitivesClasses.o \
                          $(XTOOLSO)/xmlSchemaInstanceXsi.o

ofiles/primitivesClasses.o : source/primitivesClasses.hh      \
                             $(XTOOLSHH)/xmlSchemaInstance.hh \
                             source/primitivesClasses.cc
     $(LINCOMPILE) -o $@ $(INCLUD) source/primitivesClasses.cc

ofiles/lineClasses.o : source/lineClasses.hh            \
                       source/lineYACC.hh               \
                       $(XTOOLSHH)/xmlSchemaInstance.hh \
                       source/primitivesClasses.hh      \
                       source/lineClasses.cc
     $(LINCOMPILE) -o $@ $(INCLUD) source/lineClasses.cc

ofiles/lineLex.o : source/lineLex.cc  \
                   source/lineYACC.hh
     $(LINCOMPILE) -DNO_ECHO -o $@ $(INCLUD) source/lineLex.cc

ofiles/lineLexStrIn.o : source/lineLex.cc  \
                        source/lineYACC.hh
        $(LINCOMPILE) -DSTRINGIN -DNO_ECHO -o $@ $(INCLUD) source/lineLex.cc

ofiles/lineParser.o : source/lineParser.cc
     $(LINCOMPILE) -o $@ $(INCLUD) source/lineParser.cc
```

```
ofiles/lineParserStrIn.o : source/lineParser.cc
        $(LINCOMPILE) -DSTRINGIN -o $@ $(INCLUD) source/lineParser.cc

ofiles/lineYACC.o : source/lineClasses.hh             \
                    source/lineYACC.hh                \
                    $(XTOOLSHH)/xmlSchemaInstance.hh  \
                    source/primitivesClasses.hh       \
                    source/lineYACC.cc
     $(LINCOMPILE) -o $@ $(INCLUD) source/lineYACC.cc

source/lineYACC.hh : source/lineYACC.cc

source/lineYACC.cc : source/line.y
     bison -d -l -o $@ source/line.y

source/lineLex.cc : source/line.lex
     flex -L -t source/line.lex > source/lineLex.cc
```

10 xmlSchemaParser Details

Using the xmlSchemaParser is described in section 4. That section also describes the options for the parser. This section gives further details about the parser.

The xmlSchemaParser uses a YACC-Lex parser. It runs in $\mathcal{O}(N^2)$ time where N is the number of lines in the `schema` file. The parsing step populates an abstract syntax tree in terms of a C++ model of an XSDL `schema` file.

As shown in section 9.3.1, building the xmlSchemaParser requires seven source files:

- xmlSchemaParser.cc
- xmlSchemaClasses.cc
- xmlSchemaClasses.hh
- xmlSchema.y
- xmlSchema.lex
- pattern.y
- pattern.lex

10.1 xmlSchemaParser.cc

The xmlSchemaParser.cc file (382 lines) is an application of the underlying YACC/Lex parser. What it does is covered thoroughly in section 4.

The main function does the following:

1. Checks that the number of arguments is reasonable, sets the inFileName, and calls processArguments to deal with the rest of the arguments.

2. Opens the file named inFileName. If the file does not open, it prints an error message and exits.

3. Parses the file named inFileName and closes the file. The parsing builds a parse tree with the root at xmlSchemaFile. The parsing also may set the following warning counters:

- emptyAlls (`all` nodes with nothing inside)
- emptyDocs (`documentation` nodes with an empty string)
- emptySeqs (`sequence`s with nothing inside)
- emptyChos (`choice`s with nothing inside)
- oneChos (`choice`s with one `choice`)
- nestedSeqs (`sequence`s with nothing but a `sequence` inside)
- seqWithChos (`sequence`s with nothing but a `choice` inside)
- whiteNames (names that have leading or trailing white space).

4. Opens the echo file and prints the `schema` into it from the xmlSchemaFile parse tree. The process of printing the tree builds the globalInheritanceMap and the globalIsSubtype.

5. If XmlSchemaFile::printHierarchy is true, goes through the globalIsSubtype std::map and calls printKids for each type that is in the std::map, has kids, and is NOT derived from some other type. That causes the inheritance hierarchy (as seen by the `schema`) to be printed.

6. If any of the warning counters is non-zero, prints the number of occurrences of each type of warning.

10.2 xmlSchemaClasses.cc and xmlSchemaClasses.hh

The xmlSchemaClasses.cc (4500 lines) and xmlSchemaClasses.hh (2673 lines) files define the classes that represent XSDL. The top-level class is named XmlCppBase, and all but a very few other classes derive from it. For many of the classes, a section of the XSDL standard [6] giving a BNF production for the XSDL construct is included in the documentation.

Some of the classes include a boolean mock field. That is not used by the xmlSchemaParser but is used in the xmlInstanceParserGenerator, which creates mock `element`s to deal with nested `sequence`s and `choice`s. In the xmlSchemaParser, the value of the mock fields is always false.

10.3 xmlSchema.y

The xmlSchema.y YACC file has a little over 4900 lines.

10.3.1 Comments

As noted in section 4.2, one of the options of the xmlSchemaParser is to reprint comments.

The xmlSchema.y YACC file saves all allowed comments (not to be confused with `annotation`s) for possible reprinting. Whether the comments are reprinted depends on the user's choice. The use of comments is restricted from what is normally allowed in an XML `schema`, as follows.

1. Multiple comments may appear immediately after the first line of the file (which gives the XML version).

2. Multiple comments may appear immediately before a `key` or `keyref` (`annotation`s are not allowed there).

3. A single comment may appear instead of or immediately before an `annotation`. As a result:

3A. where a single `annotation` is allowed, there may be any of:

- an `annotation`
- a single comment (of any length)
- a comment followed by an `annotation`.

3B. where multiple `annotation`s are allowed, there may be multiple comments, possibly mixed with `annotation`s.

10.3.2 Checking Pattern Regular Expressions

As mentioned in section 5.1.4, the parser checks that any regular expression used in a `pattern restriction` is valid. It does this by passing the expression as a char * string to a `pattern` regular expression parser. That parser is a separate YACC/Lex parser, the source files for which are pattern.lex and pattern.y. These files are used in building the xmlSchemaParser as shown in the Makefile of section 9.3.1. The data types for which `pattern` regular expressions are checked this way are `xs:string`, `xs:token`, `xs:ID`, `xs:IDREF`, and `xs:NMTOKEN`.

10.3.3 doXmlXXXAttributes functions

In an XML `schema`, many of the specifications (e.g., `name`, `type`, `id`, `minOccurs`, `maxOccurs`) are given as `attribute`s. Since the `attribute`s may occur in any order, writing YACC rules to recognize all orders would be prohibitively complex. Hence, for each XSDL structure (`attribute`, `element`, `complexType`, etc.) a std::list of the `attribute`s and their values is collected in the YACC rule for the structure. Then a function named doXml*XXX*Attributes (where *XXX* is the structure name) is called to check the validity of the `attribute`s and set the values of the C++ fields in an instance of the structure. The collection of `attribute`s is represented by a C++

std::list<XmlAttribPair *>. The term XmlAttribPair is something of misnomer since it has three fields: name, pref (prefix), and val (value).

The doXml*XXX*Attributes functions take up more than half of the lines of the xmlSchema.y file.

10.3.4 YACC Types and Rules

Most of the YACC types match a C++ class from the xmlSchemaClasses and have names matching the class name but starting with a lower case letter. For example, the YACC type xmlComplexType matches the C++ XmlComplexType class. Most of the rules for a YACC type call the corresponding doXML*XXX*Attributes function. For example, the rules for xmlComplexType call doXmlComplexTypeAttributes.

10.3.5 Bison Conflicts

The YACC file makes one shift/reduce and two reduce/reduce conflicts when processed by bison, but these do not affect the functioning of the parser. The conflicts are caused by the way XML comments are handled. Comments in xmlSchema.y identify the productions causing the conflicts.

11 xmlInstanceParserGenerator Details

Using the xmlInstanceParserGenerator is described in section 5.

If the larger of the number of `complexType`s and the number of `simpleType`s in a `schema` file is N, the time taken by the xmlInstanceParserGenerator is $\mathcal{O}(N^2)$.

As shown in section 9.3.1, building the xmlInstanceParserGenerator requires eight source files:

- xmlInstanceParserGenerator.cc
- xmlInstanceParserGenerator.hh
- xmlSchemaClasses.cc
- xmlSchemaClasses.hh
- xmlSchema.y
- xmlSchema.lex
- pattern.y
- pattern.lex

The last six of these are also used in the xmlSchemaParser, as described in section 10.

The xmlInstanceParserGenerator.cc file is over 21,000 lines long, and the xmlInstanceParserGenerator.hh file has over 700 lines. Because the xmlInstanceParserGenerator generates five or more files in three different languages, it is by far the most complex of the tools described in this manual. For a `schema` up to

several thousand lines long, however, the xmlInstanceParserGenerator runs in a fraction of a second on an ordinary desktop or laptop computer.

11.1 How the xmlInstanceParserGenerator Runs

The main function of the xmlInstanceParserGenerator goes through the following stages. The descriptions here provide an overview of the stages but not enough detail to understand the nuts and bolts; only examining the xmlInstanceParserGenerator.hh and xmlInstanceParserGenerator.cc files (and/or the HTML documentation for the generator) is adequate for that.

11.1.1 Initialize

To initialize, the main function:

- declares a primary generator variable (of class type generator) and several (global) variables needed by every generator.
- checks the arguments in the command to run. If they are bad, a usage message is printed and the executable quits.
- calls readOldHeader to read the existing header (only if the command arguments include the **–h** option followed by the name of an existing C++ header file corresponding to the top-level `schema` file). That populates a changeMap which holds additions to make to the new top-level C++ header file while it is being generated.
- sets the output switches
- sets some of the generator variables according to the arguments and sets other generator variables to point at the corresponding global variables.
- uses the parser to read the `schema` file whose name is a command argument and build a parse tree, then saves the major parts of the parse tree in variables of the generator.
- puts pointers to other variables into the appropriate fields of the generator.
- records the names of `include`d files in a global std::list (includeds) of files `include`d for all generators.

11.1.2 Process Included Schema files

Next, the parent generator calls its processIncludes method, passing it a pointer to the global includeds std::list. The processIncludes method goes through the includedSchemas std::list in the parse tree built by the parent generator. For each `include`d `schema` file that has not already been parsed, the processIncludes method:

- creates a new generator class instance
- copies pointers to shared data fields from the parent generator into the new generator
- parses the `include`d `schema`
- adds the name of the `include`d `schema` to the global includeds and the parent generator's includedSchemas

- calls (recursively) the processIncludes method of the new generator
- calls the buildClassesIncluded method of the new generator
- adds a pointer to the new generator to the subordinates std::list of the parent generator.

The two hierarchies in Figure 6 below are an example. The one on the left represents a hierarchy of `include`s. The one on the right represents a generator hierarchy that might result from those `include`s. Capital letters represent `schema` files on the left and generators on the right. On the left, any given `schema` file may be named in several `include`s. On the right, there is only one generator per `schema` file.

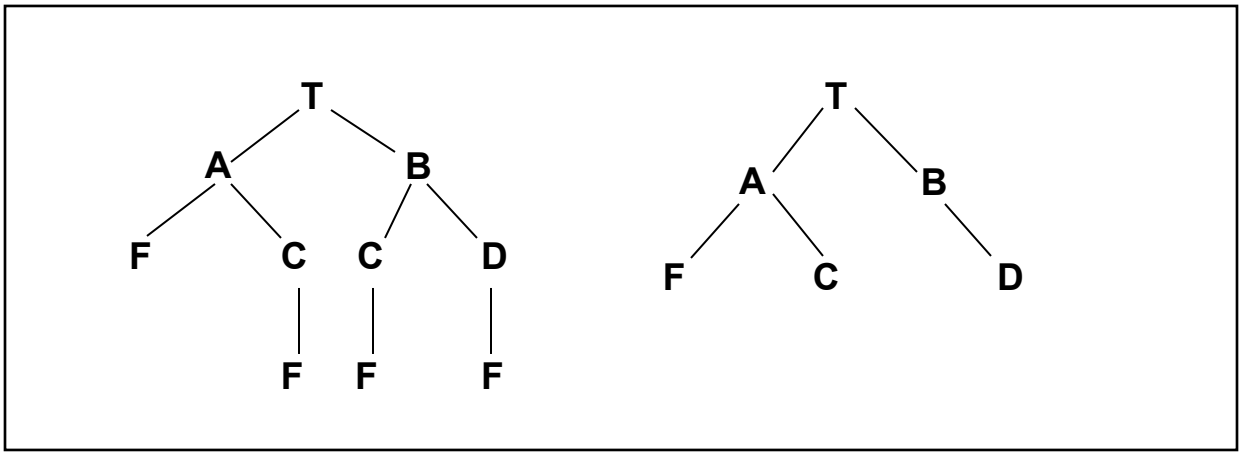

Figure 6 Hierarchies of Includes and Generators – Example

11.1.3 Build Classes

Then the primary generator calls its buildClasses method.

The xmlInstanceParserGenerator has to deal with the difficulty that an XML `schema` may have several `element`s defined at the top level, for each of which a conforming instance file may be written, but an XML instance file conforming to an XML `schema` has only one top-level `element`. The xmlInstanceParserGenerator generates a parser for only one top-level `element`. The xmlInstanceParserGenerator expects a top-level `element` to be the first thing after the header of a `schema`, and that is the `element` for which a parser is generated.

The buildClasses method records the first top-level `element` if there is one (calling it top), and then goes through all the top-level items of the `schema` and gathers information about them.

11.1.4 Build Information

In the next step, the primary generator calls its buildInformation method, which builds information needed to print C++, Lex and YACC files by calling methods that each build a particular kind of information. In particular, it:

- flattens any nested `substitutionGroup`s
- calls buildAllKidsEtc to build the C++ class inheritance hierarchy
- calls buildAllElementInfo
- calls buildAllAttributes
- calls buildDescendants
- calls buildUsesEndTag.

If there is a top `element` (so that a YACC-Lex parser is to be generated), buildInformation also calls methods that generate information need for printing the YACC and Lex files, namely:

- markElementUsed
- checkElementInfoDuplicatesUsed
- buildElementSubstitutes
- checkElementInfoDuplicatesProdBase
- buildYaccUnionElementPairs
- buildXsiTypeNamesAll (if `xsi:type` is allowed)
- buildYaccRulesEnd.

11.1.5 Print Everything

Next to last, the primary generator calls its printSelf method, which calls the printSelf method of each of the subordinates of the primary generator. The subordinates each call the printSelf method of their subordinates, and so on. After telling the subordinates to print themselves, the printSelf method calls either printTop (if the generator has a top `element`, which can happen only for the primary generator) or printNotTop (for the primary generator if it has no top `element` and for all other generators).

The printTop method:

- calls printCppHeader to print the header file for the C++ classes derived from the types defined in the top `schema` file. While the subordinate methods of printCppHeader are running, several of them call printCppHeaderChanges to put in text from the changeMap.
- calls printCppCode to print the implementation of the classes
- calls printYacc to print the YACC parser file for an instance file conforming to the `schema`
- calls printLex to print the lexical scanner used by the YACC file
- calls printParser to print the parser application.

The printNotTop method:

- calls printCppHeader to print the header file for the C++ classes derived from the types defined in the `schema` file
- calls printCppCode to print the implementation of the classes.

The C++ header files and the C++ code files produced by the generator are clear and nicely formatted. The C++ parser files produced by the generator are nicely formatted, but they are a little hard to read because

- the main program that runs the parser takes a variety of arguments
- the main program can run the parser multiple times
- input to the parse function might come from a string or a file.

Good formatting for instance files is also implemented in the PRINTSELF functions in the C++ class code that is generated. For example, if an instance of a type has multiple `attribute`s, the second and any successive `attribute`s are each printed on a separate line. This is implemented in the printCppCodePrintAttribs function and in the functions that call it.

The YACC and Lex files produced by the generator are displeasing to the eye. YACC has strange syntax, uses special characters to separate sections, uses $n in C++ code, and mixes C++ code with production code. Lex has strange syntax, uses regular expressions, and mixes C++ code with regular expression code. The net effect is that YACC and Lex files are inevitably unattractive. A human could not improve the formatting significantly.

The portions of the generator code that write files are also unattractive. That is inescapable for code that writes formatted code. For example, to print a `"` in a file written by code generated by the generator, the `"` must be doubly escaped in the generator code like so `\\\"`. The printing of code is done using C-style print commands since they enable fine control more easily than C++ print commands. The formatting of the YACC file is done in two stages. The rules in the file are constructed as a std::list of pairs of strings by many buildYacc*XXX* functions. The string pairs are alphabetized and unduplicated. Then the file is printed by printYacc*XXX* functions.

11.1.6 Finish

The final action of main function of the xmlInstanceParserGenerator is to call the reviewChanges method of the generator class. This reviews the texts in the changeMap to see if all of them have been marked done. If any is not so marked, reviewChanges prints a warning message.

11.2 Code Coordination

In the xmlInstanceParserGenerator code, it is often the case that symbols that must be the same or closely related are spread across the code that the generator writes, and

the generator code does not make this obvious. For example, the XML `schema` TruckLoadingPlan.xsd defines a `TruckLoadingPlan element` whose type is `TruckLoadingPlanType`. When the generator produces files for this `schema`, symbols containing some form of "TruckLoadingPlan" are used as follows.

1. The C++ header file is named TruckLoadingPlanClasses.hh, and in that file:
   a. TruckLoadingPlanType is listed in the class declarations at the beginning of the file.
   b. The TruckLoadingPlanType class is declared in the middle of the file.
2. The C++ code file is named TruckLoadingPlanClasses.cc and the TruckLoadingPlanType class is defined in the middle of the file.
3. The YACC file is named TruckLoadingPlan.y, and in that file:
   a. The union definition in the second section includes a TruckLoadingPlanTypeVal variable which is a pointer to an instance of the TruckLoadingPlanType class.
   b. The YACC type declarations in the third section include a declaration that TruckLoadingPlanTypeVal is the type of the y_TruckLoadingPlanType production.
   c. The YACC tokens that comprise the fourth section include TruckLoadingPlanEND and TruckLoadingPlanSTART.
   d. The YACC rule for the y_TruckLoadingPlanType production is given in the fifth section.
4. The Lex file is named TruckLoadingPlan.lex, and in that file the Lex rules for recognizing TruckLoadingPlanEND and TruckLoadingPlanSTART are given.

The generator includes code-writing code that ensures that all these items are coordinated. However, although the in-line documentation of the generator code describes some of the required coordination, it is not obvious in the code itself. Any programmer modifying the code should be aware that when any code-writing section of the generator is changed, it may be necessary to change some or all of the other sections that deal with the same sort of item.

11.3 xmlInstanceParserGenerator Methods

The generator class for the xmlInstanceParserGenerator has over 240 methods and over 60 data fields. As mentioned in section 11.1.2, a separate instance of the generator class is used for each XML `schema` file that is processed. Most of the methods fall into one group or another, as follows.

Most of the data building is done before any files are printed, but some data (mostly data regarding what has been printed) is built while printing is in progress.

11.3.1 Build YACC

Over 50 methods, all of whose names start with buildYacc, are used to create the data used to print the YACC file.

11.3.2 Print YACC

Another 19 methods, whose names start with printYacc, print the YACC file.

11.3.3 Print C++ Header Files

C++ header (i.e., .hh) files are printed by 33 methods whose names begin with printCppHeader. Some of them, printCppHeaderSequenceArgs, for example, are also called by the methods that print the .cc files. The generator class has pointer variables named ccFile and hhFile for the FILEs to print in. To use printCppHeader methods to print in a .cc file, the hhFile variable is simply set temporarily to point to the .cc file.

11.3.4 Print C++ Code Files

The .cc files are printed by 49 methods whose names begin with printCppCode.

11.3.5 Print Lex

Half a dozen methods, whose names start with printLex, are used to print the Lex file.

11.3.6 Other Data Builders

Fifteen methods whose names start with build (but not buildYacc) help to build various data structures used in the xmlInstanceParserGenerator. Another 13 methods, whose names start with enter, put data into std::lists or std::maps.

11.3.7 Data Finders

There are nine methods whose names start with find that look in a std::list or std::map for data with a particular name or characteristic.

11.4 Printing to a File or a String

As discussed in section 5.3.2, the generator code is written so that the automatically generated XML instance file parsers can take input from either a string or a file and can write to either a string or a file. The choices for writing are:

- call functions that write to a file
- call functions that write to a string
- use macros that write to either a string or a file depending on how they are compiled.

This is implemented in the generator code by using function pointers that point to functions for writing different types of item. There are nine different types of item, and for each type of item there are three functions and a function pointer. Each of the functions returns a string to be printed in the code generated by the generator. In the functions that write macros the nine items are:

PRINTSELF – a function that tells an object to print itself with “>” (for `element`s)
PRINTSELFDECL – a declaration of the PRINTSELF function
OPRINTSELF – a function that tells an object to print itself without “>” (for `attribute`s)

OPRINTSELFDECL– a declaration of the OPRINTSELF function
PRINTNAMEDECL – a declaration of a function that tells an object to print its name
XFPRINTF – a function that prints to a file or string
SPACESPLUS – a function that increases the number of spaces in the spaces variable
SPACESMINUS – a function that decreases the number of spaces in the spaces variable
SPACESZERO – a function that prints the spaces variable

For example, for the function that tells an object to print itself, the code is:

```
const char * macroprintself() {return "PRINTSELF";}
const char * fileprintself() {return "printSelf(outFile)";}
const char * stringprintself() {return "printSelf(outStr, remain, N)";}
const char * (*prself)(void);
```

According to the value of **-o** option used in calling the generator, the function pointer for each of the nine items is set to point at one of the three functions. For example, prself = macroprintself; if **-o macros** was used.

The macros are defined and explained at the beginning of the file xmlSchemaInstance.hh. They have definitions that are equivalent to the string printing functions if STRINGOUT is defined or equivalent to the file printing functions if not.

The three SPACES macros do nothing when writing to a string but help provide proper indenting when writing to a file.

11.5 Lists, Maps, and Sets

11.5.1 Uses of Lists, Maps, and Sets

Several std::lists and std::maps of things are used by the generator as follows.

std::lists that are C++ fields of the generator class

- allAttributeNames (used in printing Lex and YACC)
- allElementInfos (of all `schema` files, populated only in top generator)
- classes (of `schema` file of This generator)
- contents1 (of `schema` file of This generator)
- contents2 (of `schema` file of This generator)
- endRules (YACC rules to be put in the fifth section of the YACC file)
- includedSchemas (the `include`d `schema` files of a `schema` file)
- moreIncludes (used for preserving #includes in changed C++ header files)
- startEndNames (names used in START and END tokens)
- subordinates (the subordinate generators of This generator)
- typePairs (types to go into the YACC types)

- unionPairs (items to go into the YACC union)
- xsiTypeNames (names of types that may be used with `xsi:type`)

std::maps that are C++ fields of the generator class (all use std::string as the key)

- allComplex (used widely, includes `complexType`s in all `schema`s)
- allSimple (used widely, includes `simpleType`s in all `schema`s)
- attributeGroupRefables (of all `schema` files)
- attributeLonerRefables (of all `schema` files)
- changeMap (changes in changed C++ header files)
- descends (descendants of used types plus the type)
- elementGroups (all `element group`s)
- elementInfoDuplicates (duplicate elementInfos)
- elementRefables (of all `schema` files, shared by all generators)

A number of std::lists are used in XML `schema` classes as described in xmlSchemaClasses.hh.

A number of std::lists, std::maps, and std::sets are transient in generator functions.

11.5.2 List Terminology

Specific terminology is used in the documentation to distinguish among four types of list:
1. occurrence list - oList (for multiple occurrences of an `element`)
2. XML simple `list` - sList
3. lists in C++ code - std::list
4. lists in YACC - YACC list

To avoid confusion, sequences of lines in a file (such as the class declarations that appear at the beginning of a header file) are not called lists.

For oLists, a std::list of the type of the `element` is constructed. This is consistent through the C++ files and the YACC files that are generated. There does not appear to be any down-side to handling oLists that way. For sLists also, a std::list of the type of the `element` is constructed.

C++ names related to sLists and oLists are of the form XXXLisd. YACC names related to sLists and oLists are of the form LiztXXX. The odd spellings are used to avoid conflicts among automatically generated names. For example, a line from the YACC union might be:

```
  XmlIntegerLisd *          LiztXmlIntegerVal;
```

and a line from the list of %type might be

```
  %type <LiztXmlIntegerVal>          y_LiztIntElement_XmlInteger_u
```

11.6 Issues Handled in the xmlInstanceParserGenerator

Small, medium, and large issues arose in building the xmlInstanceParserGenerator and had to be overcome. This section discusses some of the larger issues and how they were overcome.

11.6.1 Keeping Track of Which Elements and Defined Types Are Used

XSDL does not require that all defined types be used. When one `schema` file `include`s another, if the second `schema` file is a utility file used by more than one other `schema` file, the second `schema` file may contain types that are not used. If YACC productions are written for those types and/or the `element`s in them, bison will issue warnings. Bison checks that every production but the top-level production (which is for the Top C++ XmlElementLocal) is used in the tree whose root is the top-level production.

To deal with this, the procedure described below walks the tree before YACC is generated and marks types and `element`s that are in the tree. Then YACC is generated only for those items that have been marked as being in the tree.

The C++ model of every non-basic type (i.e., every XmlSimpleType and XmlComplexType) has an integer-valued used field that is initially set to zero. When a `simpleType` or `complexType` is used as the type of an `element`, the used field of its model is set to 2. When a `simpleType` is used as the item type of a simple `list` and its model's used field is zero, the field is set to 1. In addition, the code keeps track of whether the basic types (e.g., `integer`) are used, by having an integer has*XXX* field for each of the basic types. has*XXX* is originally set to 0, is set to 1 if has*XXX* is zero and the *XXX* type is used as the type of a `list` element, and is set to 2 if the *XXX* type is used as the type of an `element`.

A production is printed in the YACC file for every `simpleType` or `complexType` T whose used is non-zero. If the value of used is 2 and `xsi:type` is allowed, a production is also printed for every derived type of T (since the derived type may be used in place of T via `xsi:type`). An `xsi:type` recognizer is printed in the Lex file for every `simpleType` T whose used is 2 (and for every derived type of T).

A production is printed in the YACC file for every basic type *XXX* whose has*XXX* is not zero. If the value of has*XXX* is 2 and `xsi:type` is allowed, a production is also printed for every derived type of *XXX*. Also if `xsi:type` is allowed, an `xsi:type` recognizer is printed in the Lex file for every basic type *XXX* whose has*XXX* is 2 (and for every derived type of *XXX*).

The marking of types is done by a markElementUsed function and four subordinates. The markComplexTypeUsed subordinate calls markElementUsed recursively so that the two functions walk the tree. The process starts by calling markElementUsed on the top level `element`.

Types used only for attribute values are not marked because attributes are not parsed by YACC productions. Attributes of a type (other than xsi:type) are parsed by the badAttributes function of the type, which is called by a YACC production.

11.6.2 Choice, Mock Types, and Mock Elements

11.6.2.1 Plain Choice

C++ classes are created for handling an XmlComplexType containing an XmlChoice of XML elements as follows. Suppose the XML name of the XmlComplexType is *XXX*.

- A C++ class named *XXX* is created to represent the XmlComplexType. It has a C++ field named pairs, which is a std::list of pointers to *XXX*ChoicePairs. It may have other C++ fields which are pointers to the C++ equivalents of XML attributes.
- A union named *XXX*Val is defined. Each line of the union represents one of the elements of the XmlChoice. The names in the union are the names of the elements. The types in the union are the C++ equivalents of the types of the elements.
- A class named *XXX*ChoicePair is defined. It has two fields. One is named *XXX*Type and its value is a whichOne. The other is named *XXX*Value and its value is an *XXX*Val.
- An enum named whichOne is defined in the *XXX*ChoicePair class. The values of the enum are made by appending E to the names of the elements.

If the XML type of an XML element is *XXX*, then the C++ type of the C++ field representing that element is *XXX*.

The *XXX* class has the pairs field defined as a std::list because if maxOccurs for the XmlChoice is unbounded or greater than 1, a std::list is required. If maxOccurs is 1 or is not given (which means maxOccurs is 1), the std::list length is 1. If maxOccurs is 0, the std::list is empty.

If an XML choice and the elements in it both have maxOccurs greater than one, parsing an instance in YACC may be ambiguous. This has not been handled. If it occurs, bison will announce that a conflict exists.

11.6.2.2 Mock Types and Mock Elements

Mock types and mock elements are generated to handle:

- a sequence containing a sequence or a choice
- a choice containing a sequence or a choice

Here is an example of a choice in a sequence.

```
<xs:complexType name="ShirtType">
  <xs:sequence>
    <xs:choice>
      <xs:element name="number"
```

```
        type="xs:integer"
        minOccurs="0"/>
      <xs:element name="name"
        type="xs:string"
        minOccurs="0"/>
    </xs:choice>
    <xs:element name="size"
      type="SizeType"/>
  </xs:sequence>
</xs:complexType>
```

This is handled in the generator's internal model of the `complexType` (that is built when the `schema` is read) by creating a mock type and a mock `element` for the `choice` and replacing the `choice` with the mock `element`. After the changes, the internal model of the example is what it would be if the following had been in the `schema`.

```
<xs:complexType name="ShirtType">
  <xs:sequence>
    <xs:element name="ShirtType_1001"
      type="ShirtType_1001_Type"/>
    <xs:element name="size"
      type="SizeType"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="ShirtType_1001_Type"
  <xs:choice>
    <xs:element name="number"
      type="xs:integer"
      minOccurs="0"/>
    <xs:element name="name"
      type="xs:string"
      minOccurs="0"/>
  </xs:choice>
</xs:complexType>
```

However, both the `ShirtType_1001 element` and the `ShirtType_1001_Type complexType` have been marked with a mock boolean flag set to true. When the C++ code, Lex code, and YACC code are generated for the instance file parser, the mock is checked and the code is written to do the right thing for reading and writing instance files conforming to the original `schema`. The number 1001 in the names above is the value of a mockCount master counter. The counter starts at 1001 and is incremented each time its value is used. The mock type and the mock `element` that uses it both use the same number.

A `sequence` inside a `choice` or another `sequence` is handled similarly by defining a type containing the inner `sequence` and replacing the inner `sequence` with a mock

element in the outer sequence or choice. The type of the mock element is the new type. The new type does not need to be marked as being mock.

The creation of mock items occurs in buildElementInfo before anything is printed. buildElementInfo works by going through the classes std::list from front to back. The class for each mock type that is created is added at the end of the classes std::list during the process of going through the std::list. Hence, each of the mock classes is also processed. The effect is to recursively dig into any nests of choices and sequences. When buildElementInfo has finished executing, no nesting remains, and mock elements appear in the places where there were nested sequences and choices.

11.6.3 Handling Optional Elements

Optional elements (those with minOccurs="0") that may occur at most once are handled in YACC by allowing the optional element instance to be empty and returning a null pointer in that case. In the C++ class model of a type, since all C++ fields corresponding to elements are pointers, nothing special needs to be done to handle an element that occurs 0 or 1 times. If the element instance does not occur, a null pointer is used.

Optional elements that may occur more than once (maxOccurs="unbounded" or maxOccurs is a number greater than 1) are handled by making a std::list. The std::list is never null, even if there is no instance of the element; in that case the std::list is empty.

11.6.4 Handling xsi:type

As mentioned in section 5.2.7, in an XML instance file conforming to an XML schema, every instance of a complexType or simpleType used as the value of an element may include an xsi:type declaration, whether it needs it or not. If a type D is derived from any sort of type P (basic, complex, or simple), then anywhere P may be used as the type of an element, D may be used as the type of the element as long as there is an xsi:type declaration.

By default, the generator does not allow using xsi:type. However, by including **-x** in the call to the generator, using xsi:type is enabled. The generator has a boolean xsiTypeAllowed field that is false by default but true if **-x** is used. The generator tests the value of xsiTypeAllowed in many (27) places.

Some of the more important things the generator does to implement using xsi:type are listed below. Starred items happen only if xsiTypeAllowed is set to true.

1. If USEXSITYPE is defined when xmlSchemaInstance.hh,cc is compiled, then every C++ class generated by the generator that represents an XSDL type (basic, complex, or simple) has a printTypp field. This is implemented by having all C++ classes (other than std::list classes) representing types be derived directly or indirectly from XmlTypeBase,

which is defined in xmlSchemaInstance.hh,cc and has the printTypp field if USEXSITYPE is defined. The basic types are not allowed to use `xsi:type` in an instance file, but since `restriction`s of basic types may use `xsi:type`, it is convenient to have the printTypp field in the basic types. Classes for std::lists of basic types are derived from XmlBasicListBase, which also has the printTypp field if USEXSITYPE is defined.

2*. The xsiTypeNames field of the generator is used to collect the names of all classes for which a YACC production is needed that looks for an `xsi:type` declaration. The documentation of buildXsiTypeNamesAll in xmlInstanceParserGenerator.cc describes how this is done.

3*. printLexXsiTypes prints a line in the Lex file for each type in xsiTypeNames.

4*. printYaccXsiTypeTokens prints a token line in the YACC file for each type in xsiTypeNames.

5. buildDescendants builds a std::list of the descendants of every type that has descendants.

6*. buildYaccElementXsiRules composes a rule that includes `xsi:type` for each descendant of each type used as the value of an `element`.

7. The PRINTSELF functions written into the C++ code file by the generator include printing an `xsi:type` declaration if printTypp is set to true.

8*. The constructors written into the C++ header and code files by the generator include a printTypp field.

11.6.5 Handling Ref

In C++ terms, the `ref` mechanism in XSDL lets you define one or more named class fields outside of any class definition. Defining and using `ref`s in XML is a lot like #define in C or C++. In XML, instead of an explicit indication that something is being #defined, the definition is simply put at the top level of the `schema`. Everything defined at the top level (except for types and the first item if it is an `element`) is really like a #define. To use the field(s), you use `ref="`*`fieldName`*`"`.

`Ref` may be used with `element`s, `element group`s, `attribute`s, and `attributeGroup`s. It is the only way to use `element group`s and `attributeGroup`s.

For `element`s, `ref` is handled as follows. The handling of `ref` for XML `attribute`s is described in section 11.6.6 (next).

1. The XmlElementRefable class is defined. Every top-level `element` in the XML `schema` file is an XmlElementRefable in the C++ code. If the first entry is an `element`, it is parsed as an XmlElementRefable but it is replaced in the abstract syntax tree by an equivalent XmlElementLocal.

2. A std::map named elementRefables of XmlElementRefables for the entire set of XML `schema` files being processed is built. The std::map is accessed by the `element name`.

3. The elementRefables std::map for a generator is populated in buildClasses or buildClassesIncluded after all `schema` files have been parsed. buildClasses and buildClassesIncluded call enterElementRefable to put XmlElementRefables into the std::map.

4. When buildElementInfoSchema runs, if an elementLocal has a `ref`,
   - findElementRefable is called to find the `ref` in the elementRefables,
   - the `ref` field of the elementLocal is replaced by a name field whose value is the name of the `ref` (and a newName is inserted also), and
   - a typ field is inserted whose value is the type of the `ref`.

After all the `ref`s have been replaced, everything runs as if the `ref`s were never there. This is analogous to what the preprocessor does in C++.

Example:

Suppose the following XmlElementRefable is defined in the `schema`:

```
<xs:element name="SignificantDigits"
  type="xs:integer"/>
```

and elsewhere the following XmlElementLocal appears:

```
<xs:element ref="SignificantDigits"/>
```

Then in the generator, the C++ model of the XmlElementLocal is revised to be the same as if the `schema` had the following instead:

```
<xs:element name="SignificantDigits"
     type="xs:integer"/>
```

This approach will not work when `key/keyref` pairs are embedded in `ref`d `element`s. Generating code to check `key` and `keyref` is not currently implemented.

11.6.6 Handling XML Attributes

In the xmlInstanceParserGenerator, XML `attribute`s may be included with an XmlComplexExtension, an XmlSimpleContentExtension, or an XmlOtherContent. To make the text clearer, in this section "attribute owner" means any of those types.

There are several XML features that make it difficult to deal with XML `attribute`s, namely:

- There are several ways `attribute`s can be associated with an attribute owner:
  - by putting in one or more `attribute` definitions – An `attribute` definition that may be put in is defined in an XmlAttributeLoner.
  - by putting in one or more `ref`s to `attribute` definitions – The reference is put into an XmlAttributeLoner instead of having a definition in the XmlAttributeLoner. An `attribute` definition that may be referenced is called an XmlAttributeLonerRefable and may occur only at the top level of the `schema`. Every top-level `attribute` is an XmlAttributeLonerRefable.
  - by putting in one or more references to an `attributeGroup` – A reference to an `attributeGroup` is called an XmlAttributeGroupRef. The `attributeGroup` itself is called an XmlAttributeGroupRefable and may occur only at the top level of the `schema`. Every top level `attributeGroup` is an XmlAttributeGroupRefable.

- `Ref`s may be nested in both single `attribute`s and `attributeGroup`s (i.e., an `attribute ref` may name an `attribute` that has an `attribute ref`).

- In an XML instance file, `attribute`s may be given in any order.

XML `attribute`s are modeled in the C++ classes by having each XML `attribute` be a separate C++ field.

The semantics of `attribute`s are the same regardless of how they get into an attribute owner. That is, `attribute`s mean the same thing as if they were all put in as individual `attribute` definitions. The generator uses this fact to make std::lists called newAttribs consisting of XmlAttributeLoners that have definitions. Each attribute owner has a newAttribs std::list. To make the process of building those std::lists easier, XmlAttributeGroupRefable also has a newAttribs std::list. The newAttribs std::lists are kept in alphabetical order.

To deal with the first two items above, after the XML `schema` has been parsed, first the newAttribs of all XmlAttributeGroupRefables are built and any XmlAttributeLonerRefable that has a `ref` has the `ref` replaced by a definition. Then the newAttribs std::lists of all attribute owners are built. In the newAttribs for an attribute owner, any XmlAttributeLoner with a `ref` is represented by an XmlAttributeLoner with a definition, and any XmlAttributeGroupRef is represented by copying in its newAttribs. C++ generation for an

attribute owner does not take place until its newAttribs std::list has been built. During C++ generation, the newAttribs std::list is used.

The third bullet above makes it difficult to generate an instance file parser. Fortunately, all `attribute` values are strings. This makes it feasible to define the AttributePair class, which is (1) a std::string that represents the `attribute` name and (2) a std::string that represents the value. All `attribute`s are given one after the other in the instance file, so the instance file parser handles them by first making a std::list of AttributePairs, second making an instance of the class that has null pointers for all the `attribute`s, and third calling a badAttributes checking function belonging to the instance. The checking function checks that the `attribute` names and values are all legitimate and inserts the values in the instance.

Printing `attribute`s into instance files is straightforward.

11.6.7 Handling Chains of Simple Restrictions

A simple `restriction` is a type that is created by restricting a basic data type or another simple `restriction`. For example, the `NotTooSmallIntegerType` might be created by a `restriction` on the basic `integer` type requiring the `integer` to be at least 3 and the `JustRightIntegerType` might then be created by a `restriction` on the `NotTooSmallIntegerType` requiring the value to be not more than 7.

Where the `schema` has a chain of type `restriction`s of this sort, the C++ code that is generated has a matching class inheritance chain. In the example, the NotTooSmallIntegerType class would be derived from XmlInteger, and the JustRightIntegerType class would be derived from the NotTooSmallIntegerType class.

Each derived class in the chain inherits two C++ fields from the C++ class for the basic type: (1) val, which has a built-in C++ type, and (2) a bool bad which will be set to true if val violates the constraints it is supposed to follow (see section 14.1). The derived class also has a function that checks that the `restriction`(s) for its type is met. For each derived type, the constructor that takes a string argument first passes the string to the constructor for its parent. If bad is not true when the parent constructor is finished, the derived type checks its `restriction`(s) and sets bad to true if it is violated. That way, all the applicable `restriction`s get checked. When a derived type is printing itself, it first checks its `restriction`(s) and then calls the printer for its parent only if its `restriction`s are met. The value is ultimately printed by the printer for the basic type, after all the `restriction`s in the chain have been checked and none has been violated.

11.6.8 Handling SubstitutionGroups

A `substitutionGroup` in a `schema` is a set of `element`s defined at the top level of the `schema` that may be substituted for another `element` called the head of the `substitutionGroup`. The head of the `substitutionGroup` must also be defined at the top level and, therefore, can only be used by `ref`. The types of the members of the

substitutionGroup must be derived from the type of the head of the substitutionGroup. For example, using the types from the previous section, suppose the Chair element of type integer is the head of a substitutionGroup whose members are NotTooSmallChair of type NotTooSmallIntegerType and JustRightChair of type JustRightIntegerType. Suppose also that the schema has a HouseType defined as follows:

```
<complexType name="HouseType">
  <sequence>
    <element ref="Chair"
       minOccurs="2" maxOccurs="6"/>
  </sequence>
</complexType>
```

Then an instance of HouseType as the value of a House element might be:

```
<House>
  <NotTooSmallChair>4</NotTooSmallChair>
  <Chair>2</Chair>
  <NotTooSmallChair>10</NotTooSmallChair>
  <JustRightChair>5</JustRightChair>
</House>
```

Notice that any of the substitute elements can appear and the head of the substitutionGroup can, too.

In the C++ code, an element defined at the top level of a schema that is not the root element is modeled as an XmlElementRefable, and one of fields of an XmlElementRefable is substitutes, which is a pointer to a std::list of the other XmlElementRefables that can be substituted for it. Where a local element uses ref to refer to one of the top level elements, the substitutes std::list of the XmlElementRefable corresponding to the top level element is copied into the XmlElementLocal representing the local element.

SubstitutionGroups can be nested. For example, suppose the JustRightChair were in the substitutionGroup of NotTooSmallChair and were not explicitly in the substitutionGroup of Chair. Then JustRightChair would implicitly be in the substitutionGroup of Chair, and the House instance above would still be valid. To deal with nested substitutionGroups, a flattenSubstitutes function adds the implicit substitutes of every XmlElementRefable to its substitutes std::list.

SubstitutionGroups complicate the YACC rules. Wherever the head of a substitutionGroup might be used, the YACC rules must look also for all of the substitutes. To deal with this, a production whose name ends in "substituteType" is written that contains a subrule for each of the substitutes. For the Chair element, that name would be y_Chair_substituteType. That production would also contain a subrule for Chair.

When printing an element that may be any of the substitutes for the head of a substitutionGroup, it is first necessary to find which substitute is being used. Then it is checked that the correct type is used and the printSelf function for that type is called. This is done in the printCppCodePrintElementSubstits function.

11.6.9 Handling Simple Lists

The YACC productions the generator writes that handle simple lists and their extensions or restrictions (call all such things SListOids) are similar to the normal YACC productions the generator writes for lists in that:

- one choice is the list followed by a list item. The action for this choice adds the item to end of the list.
- the other choice is for the first item in the list. The action for this choice calls a list constructor that takes one list item. However, the rule for this choice always includes the ">" immediately before the beginning of the first list item. If there are attributes, the rule also includes the attributes before the ">", and the action deals with the attributes.

The list items may be either basic types or restrictions of basic types. In order that appropriate checks are made on the individual items, the production for the SListOid looks for items of the type named in the SList from which the SListOid is derived (not the underlying simple type of the items).

The C++ header and code files for SListOids have constructors that take one list item.

11.6.10 Checking Restrictions on XML Simple List Types

The rules for the chain of C++ classes that represent a chain of XML restrictions of a simple list (sList) of simple XML data type are as follow. These are similar to the rules for the basic data types.

1. The bottom level of the chain is an sList of [a basic or simple datatype that is not an sList]. There are no restrictions on this bottom level sList. This sList has a bad field that is inherited if the element is basic or is assigned by code if the element is simple. The printSelf function that does the actual printing is at this level.

2. The next level up has one or more restrictions on the list, so it has a checking function that checks the restrictions of this level. This level inherits the bad field from the bottom level. The printSelf function calls the checking function and exits if it returns true (indicating a restriction check failed); then it calls the printSelf function at the bottom level.

3. The third level up (and higher levels) inherits the bad field from the next level down. This level will normally have more restrictions (although a restriction that does not restrict except by documentation is allowed by XSDL). This level has a checking

function that first calls the checking function from the next level down and then, if bad is false, checks the restrictions of this level. The printSelf function calls the checking function at this level and exits if it returns true (indicating a restriction check failed); then it calls the printSelf function at the bottom level.

During file parsing, every simple list is built one element at a time, regardless of the level. If the list type is the type of an element, the rule for the element calls the checking function for the list. The YACC parser calls yyerror (which exits) if bad is true after the checking function is called.

11.6.11 ComplexType usesEndTag

The usesEndTag field of an XmlComplexType is used to help decide what to put at the end of an instance of a complexType in an instance file. The value of usesEndTag may be 0, 1, or 2.

If the value is 2, the instance is printed with an end tag on a new line -- for instance, the tree in the code snippet below.

```
<tree>
  <color saturation="0.5">green</color>
</tree>
```

If the value is 1, the instance is printed with an end tag on the same line as the start tag -- for instance the color in the code snippet above.

If the value is 0, the instance is printed with /> rather than an end tag at the end. For instance:

```
<Gizmo color="blue"/>
```

The usesEndTag field is initialized to 0 in both of the constructors for an XmlComplexType. If the type has a sequence or a choice (so that it can have elements), usesEndTag is changed to 2 early in processing. If the type is descended from an ancestor whose usesEndTag is 2, its usesEndTag is later reset to 2. Otherwise, if an ancestor is basic or simple (so that a value is required), usesEndTag is reset to 1.

If a complexType has no elements, an instance of an element of that type must end in /> (with no end tag) whether or not the complexType has attributes.

If a complexType has elements that are all optional and an instance uses none of them, the instance may end either with an end tag or with />, even though usesEndTag has been set to 2. The printer always uses the end tag in this case. The parser requires an extra production to deal with /> instead of an end tag. The extra production is in the instance, not in the type and uses ENDWHOLEITEM to represent />.

11.6.12 YACC Names

As may be seen in Annex G, YACC requires names for YACC types, tokens, and productions. The generator uses the following conventions for these names.

11.6.12.1 YACC Type Names

YACC type names are defined in the union definition near the beginning of the YACC file and are used in the type declarations that follow the union definition. All YACC type names end with "Val". Where a YACC type name is for a std::list, the YACC type name is of the form "LiztXXXVal".

11.6.12.2 YACC Token Names

YACC token names represent constants. There are five subgroups of YACC tokens:

- tokens for key words common to all or most `schema`s, such as BAD, ENDITEM, and XMLVERSION
- tokens for `attribute`s in the `schema` header common to all or most `schema`s, such as xmlnsATTR and xsiSchemaLocationATTR
- start and end tokens for each `element` name, such as LineSTART and LineEND
- a token for each `attribute` name for `attribute`s not in the `schema` header, such as colorATTR
- tokens for `xsi:type` declarations such as PointTypeXSIDECL, included only if xsiTypeAllowed is true

11.6.12.3 YACC Production Names

All YACC production names produced by the generator start with "y_".

All YACC production names for `element`s that may occur more than once start with "y_Lizt".

Most YACC production names for `element`s are constructed using either the prodBase field or the prodListBase field of XmlElementLocal. Those fields are built in the makeProdBase function. See the documentation of that function for details.

If an `element` has a `minOccurs` greater than 1 or a (not `unbounded`) `maxOccurs` greater than 1, a production name ending in "_Check" for a production that checks the size of the `element` list against the bounds is generated.

For `element`s that head `substitutionGroup`s, a production whose name includes "_substitutesType" is generated.

11.6.13 Preventing Memory Leaks in Domain Parsers

A system for preventing memory leaks in domain parsers is implemented. How this system is implemented in the code for the domain parsers is described in section 15.2. The primary C++ structure used in the implementation is a std::map called the yyUnrefMap (where the yy prefix may be something else). The generator code that writes the code for managing the yyUnrefMap is in five printYacc*XXX* functions and in the YACC actions written by 26 buildYacc*XXX* functions.

11.6.14 Ideas not Implemented

11.6.14.1 Abbreviation of Names

The generator has been observed to produce some long YACC names (over 80 characters) for `schema`s that use long names. C++ and bison can handle the long names, but they may make it impossible to keep line length under 80 characters. In some earlier versions of the generator there was an option to abbreviate names. The user would provide the number of characters to keep and the rest of the name would be replaced by an underscore and the value of the mockCount (described in section 11.6.2.2). This was removed for two reasons:

- Implementing it fully would have required building and using a std::map between the short name and the long name.
- The names that resulted made the code hard for a user to understand, especially the YACC.

11.6.14.2 Access Function Style

Currently, to have the generator generate access functions, the **-f** option must be used followed by “getset” (see section 5.2.2). It is planned that a future version of the generator will allow **-f** to be followed by "overload". In that case the access functions would be XXX() for getting the value of the XXX field and XXX(*value*) for setting the value. That style of access function is in common use, and users may prefer it.

12 xmlSchemaAttributeConverter Details

Using the xmlSchemaAttributeConverter is described in section 7.

As shown in section 9.3.1, building the xmlSchemaAttributeConverter requires seven source files:

- xmlSchemaAttributeConverter.cc
- xmlSchemaClasses.cc
- xmlSchemaClasses.hh
- xmlSchema.y
- xmlSchema.lex
- pattern.y
- pattern.lex

The xmlSchemaAttributeConverter is relatively straightforward software. The xmlSchemaAttributeConverter.cc file, at a little over 1300 lines, is less than a tenth the size of the xmlInstanceParserGenerator.cc file. There are no thorny issues or subtleties. The in-line documentation covers details adequately.

13 orphanFinder Details

Using the orphanFinder is described in section 6.

The orphanFinder is the simplest of the XML tools. The orphanFinder.cc file is less than 600 lines long and the executable is built from only that file.

The main function calls readSchema repeatedly to read all the `schema` files given as arguments, and then it calls checkNames to produce the output. The readSchema function is a straightforward state machine. The in-line documentation gives a readily understandable description of how it works.

14 xmlSchemaInstance Details

As previously noted and shown in the Makefile of section 9.3.2, domain parsers require linking with xmlSchemaInstance.o. That is compiled from the files xmlSchemaInstance.hh and xmlSchemaInstance.cc. Those files define and implement C++ classes that represent XML basic (i.e., built-in) data types.

14.1 C++ Classes

For each basic XML data type (such as `positiveInteger`) a C++ class is declared and implemented (in xmlSchemaInstance.hh and xmlSchemaInstance.cc, respectively). The class has a name starting with Xml (such as XmlPositiveInteger). The value of the data is stored in a val field. A bad field indicates whether the data violates the restrictions required by the XML data type. A checking function (whose name ends with IsBad) checks the restrictions.

On parsing, the YACC parser calls the constructor which calls the checking function and sets the value of bad. The YACC parser checks the value of bad. On printing, the printer runs the checking function and stops the process if the value is bad. Inside an executing program the value of val may be changed to violate the constraints, but it will not be possible to print a file containing the bad value.

For most basic data types there are two constructors. One constructor takes no arguments, sets bad to true, and sets val to something innocuous (such as 0 or “”). The other constructor takes a char * string argument, interprets it as a printed value that needs to be read, reads it, and sets bad and val. For number types there is a third constructor that takes a number of the correct type as an argument and sets val to that number.

The basic types all have the printTypp field if using `xsi:type` is allowed, but printTypp must always be 0 for a basic type, since `element`s whose value is of basic type may not use `xsi:type`. The basic type constructors, therefore, do not take a printTyppIn argument.

A C++ class for a std::list of each basic XML data type (with a name ending in *Lisd* such as XmlPositiveIntegerLisd) is also declared and defined in xmlSchemaInstance.hh and xmlSchemaInstance.cc.

14.2 Printing Basic Data Types for Element or Attribute

Every XML basic (i.e., built-in) data type may be used as the value of an `attribute` or the value of an `element`. When used as the value of an `element`, a > precedes the printed value, but when used as the value of an `attribute` there is no >. To deal with this, every basic data type has two print methods: PRINTSELFDECL for `element`s (with >) and OPRINTSELFDECL (without >) for `attribute`s. Those are macros as described in section 11.4.

14.3 Checking ID and IDREF

The `ID` and `IDREF` basic XML data types have special rules. Every instance of an `ID` must be different than every other instance, and every instance of `IDREF` (a reference to an `ID`) must match an `ID`.

The restriction on `ID`s is implemented relatively simply in xmlSchemaInstance.cc. A std::set named allIDs is maintained. Whenever an `ID` is read during parsing, it is checked that the `ID` is not already in allIDs, and when the constructor for an XmlID runs, it puts the name into allIDs.

The check for `IDREF` is a little more complicated since checking `IDREF`s against `ID`s must be postponed until an entire data file has been read and all `ID`s have been recorded. A std::set named allIDREFs is maintained. Every time a new XmlIDREF is created, its val is recorded in allIDREFs. The xmlIDREFIsBad function checks that the value of an `IDREF` is found in allIDs. That function runs whenever an `IDREF` is printed. The executable domain parsers generated by the xmlInstanceParserGenerator print the file, so every `IDREF` is checked. An application using the underlying parser can check allIDREFs against allIDs without printing by running the idMissing method of XmlIDREF.

15 Domain Instance Parser Details

15.1 Re-running a Domain Parser

To use a domain parser to parse more than once in a program, the lexer needs to be reset to be run again, regardless of whether or not there is a parse error. To handle that there is a yyStartAnew (the yy prefix is a variable) global variable shared by the YACC

parser and the Lex lexer. The Lex file has the following code for restarting. The calling program must set yyStartAnew to a non-zero value in order to restart parsing.

```
if (yyStartAnew)
  {
   yyrestart(yyin);
   yyStartAnew = 0;
   yyReadData = 0;
   yyReadDataList = 0;
  }
```

15.2 Preventing Memory Leaks in Domain Parsers

Most processes that use a parser built by the xmlInstanceParserGenerator will probably not want to exit if the parser has an error. However, if there is an error during parsing, a lot of memory will have been used in building the parse tree. Typically, there will be several branches of various sizes that have been built but are not yet joined into a tree. That memory would be leaked if it were not deleted while recovering from the failed parsing attempt. Memory obtained in the lexer by malloc or alloc is released during parsing using free where appropriate, so only new memory needs to be handled in case of error.

In the parsers built by the xmlInstanceParserGenerator, the disconnected branches of the partially built parse tree are saved in a std::map called the yyUnrefMap (the yy prefix is actually a variable). As disconnected branches of the parse tree are built, they are added to the yyUnrefMap. As smaller branches are connected together to form a larger branch they are removed from the yyUnrefMap. Thus, unless there are branches that are long lists, the yyUnrefMap never gets very big. Consequently, it never takes long to put a branch in or take one out.

For example, In the following code from a YACC file, the branch $$ is created from the branch $3. The $3 branch must have previously been disconnected since it is being connected, so it was in the yyUnrefMap. The newly created branch, $$ is put into the yyUnrefMap, and the newly connected branch, $3, is removed from the yyUnrefMap.

```
{$$ = new XmlHeaderForAShirt($3);
 yyUnrefMap[$$] = $$;
 if ($3) yyUnrefMap.erase($3);
}
```

If parsing is completed without error, the parser checks that there is only one `element` in the yyUnrefMap, which is the root of the parse tree. As long as the parser runs without error, another file can be parsed. The only additional step needed to be taken to parse another file is to empty the yyUnrefMap of that one member. If there is a parse error of any sort, yyerror is called (after freeing any memory created in the lexer). In yyerror (which returns 1), all of the branches in the yyUnrefMap are deleted.

Here are two examples from a YACC file. In the first example, $3 is tested and is freed whether or not there is an error. In the second example, $1 is freed before $$ is tested.

```
if (strcmp($3, "1.0"))
  {
   free($3);
   return yyerror("version number must be 1.0");
  }
free($3);

{$$ = new BoxOrientType($1);
 yyUnrefMap[$$] = $$;
 free($1);
 if ($$->bad)
   return yyerror("bad BoxOrientType");
}
```

The yyUnrefMap is defined as a global variable that is a std::map. The std::map key type and value type are both pointer to XmlSchemaInstanceBase. When destructors are called on objects in the std::map, since all types are polymorphic and are descendants of XmlSchemaInstanceBase, the correct destructor is called for the actual type. Also, there is no problem with putting instances into the std::map.

To allow for testing for memory leaks, the parsers that are generated take an optional final argument, which is a positive integer indicating how many times to parse the file being parsed. Checking may be done by running the parser inside valgrind with the final argument set to 2 or more on a file known to contain an error. For example:

valgrind -v --leak-check=yes ../bin/SkusParser skusError.xml 2

15.3 Lists in Domain Parsers

As noted in section 15.2, to support memory leak prevention, every branch of a parse tree must be descended from XmlSchemaInstanceBase. Where there are branches of the parse tree that are lists, std::list cannot be used directly since std::list is not descended from XmlSchemaInstanceBase. Hence, for every type of list that might occur, a class is defined that is derived from both the std::list for that type and XmlSchemaInstanceBase. For the basic XML data types, the special std:list classes are declared in xmlSchemaInstance.hh and defined in xmlSchemaInstance.cc. The AttributePairList class is also included in those files.

In the case of `restriction`s of simple `list`s there is a minor problem since a simple `list` cannot be restricted directly in XML Schema Definition Language. A `simpleType` must be declared that is an alias for a simple `list`, and then the `simpleType` can be restricted.

To deal with that:

- Each simple `list` type in xmlSchemaInstance.hh has a constructor that takes the simple `list` type as an argument and copies it.

- In the generated C++ code, each class in a C++ hierarchy that parallels a derivation hierarchy of XML `restriction` types of a simple `list` has a constructor that takes the simple `list` type as an argument and passes it to the constructor for the parent type.

- In the generated YACC code, there is a production S for the simple `list`, and a production D for each C++ type in the derivation hierarchy that is the type of an `element`. The production D uses S. The actual `list` is copied from S into D as a result of the hierarchy of calls to parent type constructors.

- If an intermediate type in the hierarchy is not used as the type of an `element`, no production is written for that type.

- If a restricted `element` type R is defined in an `include`d `schema` A and a `list` of R is defined in an including `schema` B, when the C++ code for A is being written, a check is made of whether code should be written for the `list` type for R. This is done by having a needList field in XmlSimpleRestriction whose value is originally false and checking it while writing code for A. This handles the case where another `schema` C `include`s A and also defines a `list` of R. However, it requires that the needList field of R be set to true whenever any `schema` (including A) defines a `list` of R.

16 XML and XSDL

This section briefly describes XSDL models in Subsection 16.1 and XML instance files in Subsection 16.2. Full descriptions may be found for XSDL in [7], [6], [5], and [1], for XML in [8], and for XML data files conforming to an XML `schema` in [9].

An XSDL file for a small complete model is shown in Figure 2 and Annex I . An XML file for a small instance file conforming to the model is shown in Annex J.

16.1 XML Schemas

XSDL is an object-oriented information modeling language. A model written in XSDL is called an XML `schema`. Data members may be represented in the model as `element`s. The contents of a `schema` normally include a root `element` and a number of type definitions. Objects are modeled as instances of `complexType`s that may have `element`s. XSDL also includes basic (i.e., built-in) data types such as `ID`, `integer`, and `string` and supports specializations of basic data types in `simpleType`s. The

line.xsd schema file shown in Figure 3 illustrates how a two-dimensional Line might be modeled in XSDL using PointType and VectorType.

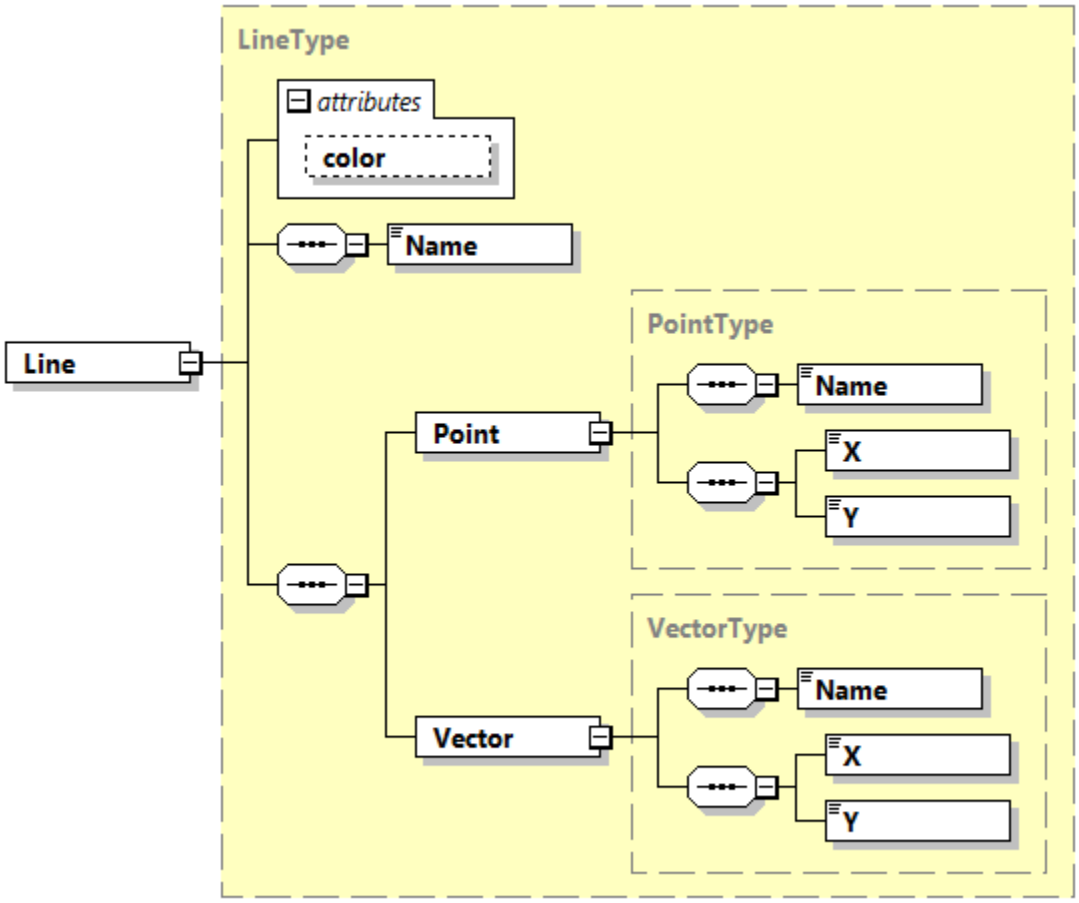

Figure 7 Structure of the Line Model

A graphical view of the line.xsd XML schema is given in Figure 7. In the figure, elements are shown as white rectangles. Three of the four complexTypes (LineType, PointType, and VectorType) are depicted as large shaded rectangles surrounded by dashed lines. The BaseType is not shown because it is never used as the value of an element. The irregular octagons are connectors joining a parent element to the elements in its type. Each type in the figure has two connectors because each of them is an extension of the BaseType and inherits the Name element from it. The color attribute is shown in a white box at the top of the figure. The outline of the box is dashed because the color attribute is optional.

One complexType (child) may be derived from another (parent) by extending or restricting the parent. Restrictions of complexTypes are verbose in XSDL and are

difficult to model in C++, which does not provide a method of restricting a derived class. Hence, restrictions of complexTypes are not allowed in schemas used with the tools. Extensions usually add elements. The child has all the elements of its parent plus any that are added by the extension. XSDL does not provide any method for a child type to have two parent types. In modeling terms, that means multiple inheritance is not possible. In the schema file above, the BaseType, which provides the Name element, is the parent of the other three types.

The scope of element names in XSDL is local to the type in which the element appears. In the example above, for instance, both Point and Vector have X and Y elements.

XSDL attributes are semantically identical to XSDL elements[4]; both are fields of a complexType. Attributes and elements differ in several ways, including:

- The value of an attribute in an instance file is given in a string (e.g., color="blue"), while the value of an element is given inside tags (e.g., <Name>Jack</Name>)
- An attribute can appear at most once in an instance of a type and is optional by default, while an element can appear zero to many times as determined by maxOccurs and MinOccurs of the element; an element is optional if minOccurs="0"; if neither is given, the element must appear exactly once.

XSDL provides for using prefixes to implement separate namespaces. XSDL allows multiple schema files in a single namespace (or no namespace).

16.2 XML Instance Files Conforming to XML Schemas

Under the XML standards, an XML instance file conforming to an XML schema must be in a different format than the schema and must contain different sorts of statements. An XML statement naming the XML schema file to which an instance file corresponds is normally given near the beginning of the instance file. Many different instance files may correspond to the same schema.

The form of an instance file is a tree in which instances of the elements of each type are textually inside the instance of the type.

The line1.xml XML instance file shown in Figure 4 conforms to the line.xsd XML schema. Names of elements in the schema become XML tags in the instance file (e.g.,

[4] It is often said that attributes should contain metadata while elements should contain data, but metadata is pretty much indistinguishable from data.

<Point>). The line1.xml file models a line that passes through the origin and lies on the Y axis.

In XSDL, there is a rule that a valid instance of a complexType must have valid instances of the required elements of the type in the order given in the schema, and elements are required unless explicitly made optional in the schema. Thus, for example, the Line_1 instance of LineType shown in Figure 4 is valid since it has a valid Name element followed by a valid Point element followed by a valid Vector element. If it did not have those valid elements in that order, it would not be valid.

17 YACC Basics

To get started with YACC, see [3] or [4].

17.1 Arrangement of a YACC file

An example YACC file is shown in Annex G.

The first section of a YACC file is C++ code set off by %{ and %}. It is optional.

The second section of a YACC file is a C++ union definition, starting with %union.

The third section of a YACC file is a set of type declarations, each marked %type.

The fourth section of a YACC file is set of token declarations, each marked %token.

The fifth section of a YACC file is the rules, as described below, set off by %% and %%.

The sixth section of a YACC file is C++ code. It is optional.

17.2 YACC Rules

The next-to-last section of a YACC file, which is usually the largest section of a YACC file, is a set of rules. For the user's convenience, the generator arranges the YACC rules in alphabetical order. Here is an example of a YACC rule. Line numbers have been added.

```
1.  y_LiztPerson_PersonType_1_u :
2.          y_LiztPerson_PersonType_1_u y_Person_PersonType_1_u
3.          {$$ = $1;
4.           $$->push_back($2);
5.           if ($2) yyUnrefMap.erase($2);
6.          }
7.        | y_Person_PersonType_1_u
8.          {$$ = new PersonTypeLisd($1);
```

```
9.          yyUnrefMap[$$] = $$;
10.          if ($1) yyUnrefMap.erase($1);
11.         }
12.       ;
```

The "rule" is all twelve lines above. A rule consists of a left-hand side, a colon, and a right-hand side (everything after the colon). The left-hand side has only one symbol, called the "production name" of the rule. The right-hand side has a semicolon at the end and consists of pairs (which we will call "subrules") of sets of lines. In the example above, there are two subrules. If there is more than one subrule, the subrules are separated by vertical bars. The first set of lines in a subrule is called the "definition". In the example above, each definition has only one line, but in general, a definition may have several lines. The second set of lines is YACCified C++ code starting with a left curly brace and ending with a right curly brace. The second set is called the "action". The actions in the example above each take up four lines.

YACC allows the same production name to be used more than once. For example, the snippet of YACC above could have been written as two rules, each with the same production name and one subrule. The generator uses each production name only once and has multiple subrules if necessary.

When a YACC parser runs, it keeps track of what it has read so far. That establishes a context that tells the parser what might come next (a token, a value (e.g., a number or string), or a production name). Based on what is read next, the parser changes its context. Also, whenever a series of items is read that matches the definition part of a subrule, in addition to changing its context, the parser executes the C++ action part of the subrule. In the YACC files built by the generator, the action is always (1) add something to the abstract syntax tree that is being built, and (2) adjust the yyUnrefMap.

18 Use of the Tools

18.1 xmlSchemaParser Use

The xmlSchemaParser has been used extensively at NIST to help develop the Quality Information Framework (QIF), an American National Standard developed by the Digital Metrology Standards Consortium. The part of the QIF model written in XSDL consists of 23 XML `schema` files totaling over 100,000 lines that make up one `schema`. The xmlSchemaParser's ability to remove `key`s and `keyref`s and its ability to reformat or remove `documentation` nodes have been important. Commercial XML tools that produce `schema` documentation automatically cannot handle QIF's `key`s and `keyref`s. Reformatting `documentation` nodes is not found in commercial tools that handle `schema` files.

18.2 xmlInstanceParserGenerator Use

The xmlInstanceParserGenerator was used in the NIST Agility Performance of Robotic Systems project. It also was used in the Systems Integration for Additive Manufacturing

(SIAM) project of the NIST Measurement Science for Additive Manufacturing Program and at the Georgia Tech Research Institute. Most recently it was used to produce C++ classes and a parser for QIF; the code may found at https://github.com/QualityInformationFramework/qif-community/tree/master/bindings.

18.3 orphanFinder Use

The orphanFinder has been used in QIF development and in SIAM development.

18.4 xmlSchemaAttributeConverter Use

The xmlSchemaAttributeConverter has been used by the authors in connection with standards work.

19 Testing the Tools

Tests have been built for testing the xmlSchemaParser, the xmlInstanceParserGenerator, and the domain parsers built from code produced by the xmlInstanceParserGenerator. These tests are in the testCases directory, which contains automated regression tests as described in the following subsections.
The testCases directory contains three subdirectories:

- testCasesFile – 30 cases; code is generated without macros, access functions, or `xsi:type`; the *domain* parser that is generated writes to a file
- testCasesFileAcc – 30 cases; code is generated with access functions, but no macros and no `xsi:type`; the *domain* parser that is generated writes to a file
- testCasesMacXsi – 38 cases; code is generated with macros and `xsi:type`, but no access functions; the *domain* parser that is generated writes to a file or a string according to how it is compiled.

Each subdirectory also contains four regression tests for the generator in the form of bash scripts. For the 30 cases in the testCasesFile and testCasesFileAcc subdirectories, all three subdirectories use the same `schema`.

The testCasesMacXsi directory has more cases because five of them will not run without `xsi:type` and because a few are for the xmlSchemaParser only (the generator cannot handle those). The testCasesMacXsi directory has an additional regression test named regressionTestParse for the xmlSchemaParser. It tests all of the schemas in the testCasesMacXsi directory.

19.1 Test Case Files

Each test case is in a separate subdirectory. The test case subdirectories for the generator all have the structure shown below (with occasional text files thrown in):

bin data Makefile ofiles schema source

For each test case:

The schema directory contains one to many XML schema files, some subset of which may be connected by includes and (with one exception) form a complete schema. Some of the schema files were written to test specific capabilities of the generator, and some are from other projects. Except for test cases for the parser only, the schema files all have all types defined at schema (top) level since the generator requires that.

The source directory contains C++, YACC, and Lex files generated by the xmlInstanceParserGenerator from the schema files in the schema directory, plus C++ files generated by bison and Lex from the YACC and Lex files.

The ofiles directory contains C++ object files compiled from the C++ source code files.

The bin directory contains one or more executables built from the ofiles and xmlSchemaInstance.o. There may be more than one executable because they can read from either a file or a string, depending on compiler flag settings. Also, in the testCasesMacXsi directory, some executables are compiled so that they write to a string.

The data directory contains one to many XML instance files that conform to the schema. It usually also contains one to a few non-conforming instance files. Non-conforming instance files serve to test the error detection and the error recovery capability of the executable.

The subdirectories of testCasesMacXsi for testing only the xmlSchemaParser have only a schema subdirectory.

19.2 regressionTestGenerate

The regressionTestGenerate test script checks that running the xmlInstanceParserGenerator on the schema files for all test cases produces source code identical to the saved source code.

When this test runs, if any generated file differs from the saved file, the test stops and the difference between the two files is displayed. The person running the test can then decide what to do about it.

On a Dell XPS laptop computer[5] running cygwin (https://www.cygwin.com/), the complete test for testCasesMacXsi takes about 24 seconds if there are no discrepancies between generated and saved files.

If a change is made in the xmlInstanceParserGenerator, running regressionTestGenerate and updating the test files can be very tedious since a single small change might affect all the test cases.

[5] Certain commercial/open source software and tools are identified in this manual in order to explain our research. Such identification does not imply recommendation or endorsement by the authors or NIST, nor does it imply that the software tools identified are necessarily the best available for the purpose

19.3 regressionTestCompile

The regressionTestCompile script uses the Makefile in each of the test cases to compile the executable parser from the source code. If the source code has not changed, no compiling is done.

If compiling fails for any test case, the test stops. The person running the test can then decide what to do about it.

19.4 regressionTestExecute

The regressionTestExecute script checks that in each test case:

- Running the executable domain parser on conforming XML instance files produces output identical to the input (or a reformatted version of the input).
- Running the executable domain parser on non-conforming XML instance files produces the expected error messages.
- Running the executable on conforming XML instance files has no memory leaks.
- Running the executable on an non-conforming XML instance file has no memory leaks when two attempts to parse the file are made.

The memory leak tests use valgrind [10], which runs on Linux systems but not Windows systems, so the Windows version of regressionTestExecute omits the memory testing.

19.5 regressionTest

The regressionTest test script runs the other three regression tests in order.

20 Future Work

There is an enormous amount that could, in principle, be done to improve the XML tools.

A number of places in the code are labelled “FIX”. It would be useful to fix those places.

It would be useful to attempt to make the generators that run in $\mathcal{O}(N^2)$ time run in $\mathcal{O}(N \log N)$ time. This might be done by using std::maps rather than std::lists. Some effort has already gone into doing that. More is feasible, but it is not clear that every function currently requiring $\mathcal{O}(N^2)$ time can be changed to run in $\mathcal{O}(N \log N)$ time.

21 Acknowledgements

The work reported in this manual was funded in part by grant number 70NANB12H143 from the National Institute of Standards and Technology to the Catholic University of America that supported Dr. Kramer's work.

Dr. Kootbally acknowledges support for this work under grant 70NANB19H009 from the National Institute of Standards and Technology to the University of Southern California.

Work on timing tests for the tools using python was done in January 2014 by Mr. Benjamin Marks.

The initial work on changing the xmlInstanceParserGenerator code to provide an option for access functions as described in section 5.2 was done in the summer of 2014 by Mr. Chris Lawler.

The initial work on changing the xmlSchemaParserCode so that it generates a type derivation hierarchy as described in section 4 was done in January 2017 by Mr. David Zuckerman.

Work on checking the code coverage triggered by the test cases was done in January 2020 by Mr. Jonah Langlieb.

Annex A **XML schema file primitives.xsdecho**

This is the XML `schema` file primitives.xsdecho, generated as described in Section 4.2. It is a pretty-printed version of primitives.xsd, with all comments removed.


```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">

  <xs:complexType name="BaseType"
    abstract="true">
    <xs:sequence>
      <xs:element name="Name"
        type="xs:ID"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="PointType">
    <xs:complexContent>
      <xs:extension base="BaseType">
        <xs:sequence>
          <xs:element name="X"
            type="xs:decimal">
            <xs:annotation>
              <xs:documentation>
                The X element is the X coordinate of the point.
              </xs:documentation>
            </xs:annotation>
          </xs:element>
          <xs:element name="Y"
            type="xs:decimal"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="VectorType">
    <xs:complexContent>
      <xs:extension base="BaseType">
        <xs:sequence>
```

```
          <xs:element name="X"
            type="xs:decimal"/>
          <xs:element name="Y"
            type="xs:decimal"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

</xs:schema>
```

Annex B C++ File primitivesClasses.hh

This is the C++ header file primitivesClasses.hh, generated by the xmlInstanceParserGenerator, as described in Section 5.4.

```
/* ************************************************************** */

#ifndef PRIMITIVES_HH
#define PRIMITIVES_HH
#include <stdio.h>
#include <list>
#include <xmlSchemaInstance.hh>

/* ************************************************************** */

class BaseType;
class PointType;
class VectorType;

/* ************************************************************** */
/* ************************************************************** */

class BaseType :
  public XmlTypeBase
{
public:
  BaseType(
    const char * printTyppIn = 0);
  BaseType(
    XmlID * NameIn,
    const char * printTyppIn);
  ~BaseType();
  void printSelf(FILE * outFile);

  XmlID * getName();
  void setName(XmlID * NameIn);

protected:
  XmlID * Name;
};

/* ************************************************************** */

class PointType :
  public BaseType
```

```
{
public:
  PointType(
    const char * printTyppIn = 0);
  PointType(
    XmlID * NameIn,
    XmlDecimal * XIn,
    XmlDecimal * YIn,
    const char * printTyppIn);
  ~PointType();
  void printSelf(FILE * outFile);

  XmlDecimal * getX();
  void setX(XmlDecimal * XIn);
  XmlDecimal * getY();
  void setY(XmlDecimal * YIn);

protected:
  XmlDecimal * X;
  XmlDecimal * Y;
};

/* ************************************************************** */

class VectorType :
  public BaseType
{
public:
  VectorType(
    const char * printTyppIn = 0);
  VectorType(
    XmlID * NameIn,
    XmlDecimal * XIn,
    XmlDecimal * YIn,
    const char * printTyppIn);
  ~VectorType();
  void printSelf(FILE * outFile);

  XmlDecimal * getX();
  void setX(XmlDecimal * XIn);
  XmlDecimal * getY();
  void setY(XmlDecimal * YIn);

protected:
  XmlDecimal * X;
  XmlDecimal * Y;
};

/* ************************************************************** */

#endif // PRIMITIVES_HH
```

Annex C C++ File lineClasses.hh

This is the C++ header file lineClasses.hh, generated by the xmlInstanceParserGenerator, as described in Section 5.4.

```
/* ************************************************************** */

#ifndef LINE_HH
#define LINE_HH
#include <stdio.h>
#include <list>
#include <xmlSchemaInstance.hh>
#include "primitivesClasses.hh"

/* ************************************************************** */

class LineFile;
class LineType;
class XmlHeaderForLine;

/* ************************************************************** */
/* ************************************************************** */

class LineFile :
  public XmlTypeBase
{
public:
  LineFile();
  LineFile(
    XmlVersion * versionIn,
    XmlHeaderForLine * headerIn,
    LineType * LineIn);
  ~LineFile();
  void printSelf(FILE * outFile);
  XmlVersion * getversion();
  void setversion(XmlVersion * versionIn);
  XmlHeaderForLine * getheader();
  void setheader(XmlHeaderForLine * headerIn);
  LineType * getLine();
  void setLine(LineType * LineIn);

protected:
  XmlVersion * version;
  XmlHeaderForLine * header;
  LineType * Line;
};

/* ************************************************************** */
 class LineType :
  public BaseType
{
public:
```

```
  LineType(
    const char * printTyppIn = 0);
  LineType(
    XmlID * NameIn,
    PointType * PointIn,
    VectorType * VectorIn,
    const char * printTyppIn);
  LineType(
    XmlID * NameIn,
    XmlToken * colorIn,
    PointType * PointIn,
    VectorType * VectorIn,
    const char * printTyppIn);
  ~LineType();
  void printSelf(FILE * outFile);
  bool badAttributes(AttributePairLisd * attributes);

  XmlToken * getcolor();
  void setcolor(XmlToken * colorIn);

  PointType * getPoint();
  void setPoint(PointType * PointIn);
  VectorType * getVector();
  void setVector(VectorType * VectorIn);

protected:
  XmlToken * color;
  PointType * Point;
  VectorType * Vector;
};

/* ************************************************************** */
 class XmlHeaderForLine :
  public XmlSchemaInstanceBase
{
public:
  XmlHeaderForLine();
  XmlHeaderForLine(
    XmlString * XmlnsNoPrefixIn,
    XmlStringLisd * XmlnsiWithPrefixIn,
    SchemaLocation * locationIn);
  ~XmlHeaderForLine();
  void printSelf(FILE * outFile);
  bool badAttributes(AttributePairLisd * attributes);
  XmlString * getXmlnsNoPrefix();
  void setXmlnsNoPrefix(XmlString * XmlnsNoPrefixIn);
  XmlStringLisd * getXmlnsiWithPrefix();
  void setXmlnsiWithPrefix(XmlStringLisd * XmlnsiWithPrefixIn);
  SchemaLocation * getlocation();
  void setlocation(SchemaLocation * locationIn);

  XmlToken * getcolor();
```

```
  void setcolor(XmlToken * colorIn);

protected:
  XmlString * XmlnsNoPrefix;
  XmlStringLisd * XmlnsiWithPrefix;
  SchemaLocation * location;
  XmlToken * color;
};

/* ************************************************************** */

#endif // LINE_HH
```

Annex D C++ File primitivesClasses.cc

This is the C++ code file primitivesClasses.cc, generated by the xmlInstanceParserGenerator, as described in Section 5.4.

```
/* ************************************************************ */

#include <stdio.h>             // for printf, etc.
#include <string.h>            // for strdup
#include <stdlib.h>            // for exit
#include <list>
#include <boost/regex.hpp>
#include <xmlSchemaInstance.hh>
#include "primitivesClasses.hh"

#define INDENT 2

/* ************************************************************ */
/* ************************************************************ */

/* class BaseType

*/

BaseType::BaseType(
 const char * printTyppIn)
{
  Name = 0;
  printTypp = printTyppIn;
}

BaseType::BaseType(
 XmlID * NameIn,
 const char * printTyppIn)
{
  Name = NameIn;
  printTypp = printTyppIn;
}

BaseType::~BaseType()
{
  #ifndef NODESTRUCT
  delete Name;
  #endif
}

void BaseType::printSelf(FILE * outFile)
{
  fprintf(outFile, ">\n");
  doSpaces(+INDENT, outFile);
  doSpaces(0, outFile);
  fprintf(outFile, "<Name");
```

```
  Name->printSelf(outFile);
  fprintf(outFile, "</Name>\n");
  doSpaces(-INDENT, outFile);
}

XmlID * BaseType::getName()
{return Name;}

void BaseType::setName(XmlID * NameIn)
{Name = NameIn;}

/* ************************************************************** */

/* class PointType

*/

PointType::PointType(
 const char * printTyppIn) :
  BaseType(printTyppIn)
{
  X = 0;
  Y = 0;
}

PointType::PointType(
 XmlID * NameIn,
 XmlDecimal * XIn,
 XmlDecimal * YIn,
 const char * printTyppIn) :
  BaseType(
    NameIn,
    printTyppIn)
{
  X = XIn;
  Y = YIn;
}

PointType::~PointType()
{
  #ifndef NODESTRUCT
  delete X;
  delete Y;
  #endif
}

void PointType::printSelf(FILE * outFile)
{
  fprintf(outFile, ">\n");
  doSpaces(+INDENT, outFile);
  doSpaces(0, outFile);
  fprintf(outFile, "<Name");
```

```
  Name->printSelf(outFile);
  fprintf(outFile, "</Name>\n");
  doSpaces(0, outFile);
  fprintf(outFile, "<X");
  X->printSelf(outFile);
  fprintf(outFile, "</X>\n");
  doSpaces(0, outFile);
  fprintf(outFile, "<Y");
  Y->printSelf(outFile);
  fprintf(outFile, "</Y>\n");
  doSpaces(-INDENT, outFile);
}

XmlDecimal * PointType::getX()
{return X;}

void PointType::setX(XmlDecimal * XIn)
{X = XIn;}

XmlDecimal * PointType::getY()
{return Y;}

void PointType::setY(XmlDecimal * YIn)
{Y = YIn;}

/* ************************************************************** */

/* class VectorType

*/

VectorType::VectorType(
 const char * printTyppIn) :
  BaseType(printTyppIn)
{
  X = 0;
  Y = 0;
}

VectorType::VectorType(
 XmlID * NameIn,
 XmlDecimal * XIn,
 XmlDecimal * YIn,
 const char * printTyppIn) :
  BaseType(
    NameIn,
    printTyppIn)
{
  X = XIn;
  Y = YIn;
}
```

```
VectorType::~VectorType()
{
  #ifndef NODESTRUCT
  delete X;
  delete Y;
  #endif
}

void VectorType::printSelf(FILE * outFile)
{
  fprintf(outFile, ">\n");
  doSpaces(+INDENT, outFile);
  doSpaces(0, outFile);
  fprintf(outFile, "<Name");
  Name->printSelf(outFile);
  fprintf(outFile, "</Name>\n");
  doSpaces(0, outFile);
  fprintf(outFile, "<X");
  X->printSelf(outFile);
  fprintf(outFile, "</X>\n");
  doSpaces(0, outFile);
  fprintf(outFile, "<Y");
  Y->printSelf(outFile);
  fprintf(outFile, "</Y>\n");
  doSpaces(-INDENT, outFile);
}

XmlDecimal * VectorType::getX()
{return X;}

void VectorType::setX(XmlDecimal * XIn)
{X = XIn;}

XmlDecimal * VectorType::getY()
{return Y;}

void VectorType::setY(XmlDecimal * YIn)
{Y = YIn;}

/* ************************************************************** */
```

Annex E C++ File lineClasses.cc

This is the C++ code file lineClasses.cc, generated by the xmlInstanceParserGenerator, as described in Section 5.4.

```
/* ************************************************************
*/

#include <stdio.h>             // for printf, etc.
```

```
#include <string.h>            // for strdup
#include <stdlib.h>            // for exit
#include <list>
#include <boost/regex.hpp>
#include <xmlSchemaInstance.hh>
#include "lineClasses.hh"

#define INDENT 2

/* ********************************************************************** */
/* ********************************************************************** */

/* class LineFile

*/

LineFile::LineFile()
{
  version = 0;
  header = 0;
  Line = 0;
}

LineFile::LineFile(
  XmlVersion * versionIn,
  XmlHeaderForLine * headerIn,
  LineType * LineIn)
{
  version = versionIn;
  header = headerIn;
  Line = LineIn;
}

LineFile::~LineFile()
{
  #ifndef NODESTRUCT
  delete version;
  delete header;
  delete Line;
  #endif
}

void LineFile::printSelf(FILE * outFile)
{
  version->printSelf(outFile);
  fprintf(outFile, "<Line\n");
  header->printSelf(outFile);
  Line->printSelf(outFile);
  fprintf(outFile, "</Line>\n");
}

XmlVersion * LineFile::getversion()
```

```
{return version;}

void LineFile::setversion(XmlVersion * versionIn)
{version = versionIn;}

XmlHeaderForLine * LineFile::getheader()
{return header;}

void LineFile::setheader(XmlHeaderForLine * headerIn)
{header = headerIn;}

LineType * LineFile::getLine()
{return Line;}

void LineFile::setLine(LineType * LineIn )
{Line = LineIn;}

/* ********************************************************** */

/* class LineType

*/

LineType::LineType(
 const char * printTyppIn) :
  BaseType(printTyppIn)
{
  color = 0;
  Point = 0;
  Vector = 0;
}

LineType::LineType(
 XmlID * NameIn,
 PointType * PointIn,
 VectorType * VectorIn,
 const char * printTyppIn) :
  BaseType(
    NameIn,
    printTyppIn)
{
  color = 0;
  Point = PointIn;
  Vector = VectorIn;
}

LineType::LineType(
 XmlID * NameIn,
 XmlToken * colorIn,
 PointType * PointIn,
 VectorType * VectorIn,
 const char * printTyppIn) :
```

```
  BaseType(
    NameIn,
    printTyppIn)
{
  color = colorIn;
  Point = PointIn;
  Vector = VectorIn;
}

LineType::~LineType()
{
  #ifndef NODESTRUCT
  delete color;
  delete Point;
  delete Vector;
  #endif
}

void LineType::printSelf(FILE * outFile)
{
  if (color)
    {
      fprintf(outFile, "\n");
      fprintf(outFile, "  color=\"");
      color->oPrintSelf(outFile);
      fprintf(outFile, "\"");
    }
  fprintf(outFile, ">\n");
  doSpaces(+INDENT, outFile);
  doSpaces(0, outFile);
  fprintf(outFile, "<Name");
  Name->printSelf(outFile);
  fprintf(outFile, "</Name>\n");
  doSpaces(0, outFile);
  fprintf(outFile, "<Point");
  if (Point->printTypp)
    {
      fprintf(outFile, " xsi:type=\"%s\"", Point->printTypp);
    }
  Point->printSelf(outFile);
  doSpaces(0, outFile);
  fprintf(outFile, "</Point>\n");
  doSpaces(0, outFile);
  fprintf(outFile, "<Vector");
  if (Vector->printTypp)
    {
      fprintf(outFile, " xsi:type=\"%s\"", Vector->printTypp);
    }
  Vector->printSelf(outFile);
  doSpaces(0, outFile);
  fprintf(outFile, "</Vector>\n");
  doSpaces(-INDENT, outFile);
```

```
}

bool LineType::badAttributes(
 AttributePairLisd * attributes)
{
  std::list<AttributePair *>::iterator iter;
  AttributePair * decl;
  bool returnValue;

  returnValue = false;
  for (iter = attributes->begin(); iter != attributes->end(); iter++)
    {
      decl = *iter;
      if (decl->name == "color")
        {
          XmlToken * colorVal;
          if (color)
            {
              fprintf(stderr, "two values for color in LineType\n");
              returnValue = true;
              break;
            }
          colorVal = new XmlToken(decl->val.c_str());
          if (colorVal->bad)
            {
              delete colorVal;
              fprintf(stderr, "bad value %s for color in LineType\n",
                      decl->val.c_str());
              returnValue = true;
              break;
            }
          else
            color = colorVal;
        }
      else
        {
          fprintf(stderr, "bad attribute in LineType\n");
          returnValue = true;
          break;
        }
    }
  for (iter = attributes->begin(); iter != attributes->end(); iter++)
    {
      delete *iter;
    }
  attributes->clear();
  if (returnValue == true)
    {
      delete color;
      color = 0;
    }
  return returnValue;
```

```
}

XmlToken * LineType::getcolor()
{return color;}

void LineType::setcolor(XmlToken * colorIn)
{color = colorIn;}

PointType * LineType::getPoint()
{return Point;}

void LineType::setPoint(PointType * PointIn)
{Point = PointIn;}

VectorType * LineType::getVector()
{return Vector;}

void LineType::setVector(VectorType * VectorIn)
{Vector = VectorIn;}

/* ************************************************************** */

XmlHeaderForLine::XmlHeaderForLine()
{
  XmlnsNoPrefix = 0;
  XmlnsiWithPrefix = 0;
  location = 0;
  color = 0;
}

XmlHeaderForLine::XmlHeaderForLine(
  XmlString * XmlnsNoPrefixIn,
  XmlStringLisd * XmlnsiWithPrefixIn,
  SchemaLocation * locationIn)
{
  XmlnsNoPrefix = XmlnsNoPrefixIn;
  XmlnsiWithPrefix = XmlnsiWithPrefixIn;
  location = locationIn;
  color = 0;
}

XmlHeaderForLine::~XmlHeaderForLine()
{
  #ifndef NODESTRUCT
  std::list<XmlString *>::iterator iter;

  if (XmlnsiWithPrefix)
    {
      for (iter = XmlnsiWithPrefix->begin();
           iter != XmlnsiWithPrefix->end(); iter++)
        {
          delete *iter;
```

```
        }
      delete XmlnsiWithPrefix;
    }
  delete XmlnsNoPrefix;
  delete location;
  delete color;
  #endif
}

// The fields other than XmlnsNoPrefix, XmlnsiWithPrefix, and location
// belong also to the top level type and are printed by its PRINTSELF.

void XmlHeaderForLine::printSelf(FILE * outFile)
{
  std::list<XmlString *>::iterator iter;

  if (XmlnsNoPrefix)
    {
      fprintf(outFile, "  xmlns=\"%s\"\n", XmlnsNoPrefix->val.c_str());
    }
  if (XmlnsiWithPrefix)
    {
      for (iter = XmlnsiWithPrefix->begin();
           iter != XmlnsiWithPrefix->end(); iter++)
        {
          fprintf(outFile, "  xmlns:%s\"\n", (*iter)->val.c_str());
        }
    }
  if (location)
    {
      fprintf(outFile,
              "  xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-
instance\"\n");
      location->printSelf(outFile);
    }
}

bool XmlHeaderForLine::badAttributes(
 AttributePairLisd * attributes)
{
  std::list<AttributePair *>::iterator iter;
  AttributePair * decl;
  bool returnValue;
  bool hasXsi;
  int stop;
  int n;

  returnValue = false;
  hasXsi = false;
  char buffer[NAMESIZE];
  for (iter = attributes->begin(); iter != attributes->end(); iter++)
    {
```

```
decl = *iter;
if (decl->name == "xmlns")
  {
    if (XmlnsNoPrefix)
      {
        fprintf(stderr,
         "two values for no colon xmlns in XmlHeaderForLine\n");
        returnValue = true;
        break;
      }
    XmlnsNoPrefix = new XmlString(decl->val.c_str());
  }
else if (decl->name == "xmlns:")
  {
    strncpy(buffer, decl->val.c_str(), NAMESIZE);
    if ((buffer[0] == 'x') &&
        (buffer[1] == 's') &&
        (buffer[2] == 'i') &&
        (isspace(buffer[3]) || (buffer[3] == '=')))
      {
        stop = strlen(buffer);
        if (hasXsi)
          {
            fprintf(stderr, "two values for xmlns:xsi\n");
            returnValue = true;
            break;
          }
        n = 3;
        if (buffer[n] != '=')
          { // find the = if not already there -- must be one
            for (n = 4; ((n < stop) && (buffer[n] != '=')); n++);
          } // next find the "
        for (n++; ((n < stop) && (buffer[n] != '"')); n++);
        if (strcmp(buffer+n+1,
                   "http://www.w3.org/2001/XMLSchema-instance"))
          {
            fprintf(stderr, "xmlns:xsi must be "
             "\"http://www.w3.org/2001/XMLSchema-instance\"\n");
            returnValue = true;
            break;
          }
        else
          {
            hasXsi = true;
          }
      }
    else
      {
        if (XmlnsiWithPrefix == 0)
          {
            XmlnsiWithPrefix = new XmlStringLisd;
          }
```

```
        XmlnsiWithPrefix->push_back(new XmlString(buffer));
      }
  }
else if (decl->name == "xsi:schemaLocation")
  {
    if (location)
      {
        fprintf(stderr,
               "two values for location in XmlHeaderForLine\n");
        returnValue = true;
        break;
      }
    location = new SchemaLocation("xsi", decl->val.c_str(), true);
  }
else if (decl->name == "xsi:noNamespaceSchemaLocation")
  {
    if (location)
      {
        fprintf(stderr,
               "two values for location in XmlHeaderForLine\n");
        returnValue = true;
        break;
      }
    location = new SchemaLocation("xsi", decl->val.c_str(), false);
  }
else if (decl->name == "color")
  {
    XmlToken * colorVal;
    if (color)
      {
        fprintf(stderr,
                "two values for color in XmlHeaderForLine\n");
        returnValue = true;
        break;
      }
    colorVal = new XmlToken(decl->val.c_str());
    if (colorVal->bad)
      {
        delete colorVal;
        fprintf(stderr,
                "bad value %s for color in XmlHeaderForLine\n",
                decl->val.c_str());
        returnValue = true;
        break;
      }
    else
      color = colorVal;
  }
else
  {
    fprintf(stderr, "bad attribute in XmlHeaderForLine\n");
    returnValue = true;
```

```
          break;
        }
    }
  for (iter = attributes->begin(); iter != attributes->end(); iter++)
    {
      delete *iter;
    }
  attributes->clear();
  if (returnValue == true)
    {
      delete XmlnsNoPrefix;
      XmlnsNoPrefix = 0;
      delete XmlnsiWithPrefix;
      XmlnsiWithPrefix = 0;
      delete location;
      location = 0;
      delete color;
      color = 0;
    }
  if (location && !hasXsi)
    {
      fprintf(stderr, "xsi namespace used but not declared\n");
      returnValue = true;
    }
  return returnValue;
}

XmlString * XmlHeaderForLine::getXmlnsNoPrefix()
{return XmlnsNoPrefix;}

void XmlHeaderForLine::setXmlnsNoPrefix(XmlString * XmlnsNoPrefixIn)
{XmlnsNoPrefix = XmlnsNoPrefixIn;}

XmlStringLisd * XmlHeaderForLine::getXmlnsiWithPrefix()
{return XmlnsiWithPrefix;}

void XmlHeaderForLine::setXmlnsiWithPrefix
(XmlStringLisd * XmlnsiWithPrefixIn)
{XmlnsiWithPrefix = XmlnsiWithPrefixIn;}

SchemaLocation * XmlHeaderForLine::getlocation()
{return location;}

void XmlHeaderForLine::setlocation(SchemaLocation * locationIn)
{location = locationIn;}

XmlToken * XmlHeaderForLine::getcolor()
{return color;}

void XmlHeaderForLine::setcolor(XmlToken * colorIn)
{color = colorIn;}
```

```
/* ************************************************************ */
```

Annex F C++ File lineParser.cc

This is the C++ code file lineParser.cc, generated by the xmlInstanceParserGenerator, as described in Section 5.4.

```
/*

The parser reads an input file and writes an output file.

The parser is called with one to six arguments as follows
parser <file name> [-n|N <places>] [-f|F <format>] [<times>]
For example (with all optional arguments): parser instances.xml -n 6 -F e 2
In that example
<file name> = instances.xml
[-n|N <places>]  = -n 6
[-f|F <format>]  = -F e
[<times>]        =  2

The [-n|N <places>] and [-f|F <format>] determine how xs:double,
xs:float, and xs:decimal are printed out. In the following text, DFD
means a number of one of those types, and DF means xs:double and
xs:float numbers. <places> is a non-negative integer. <format> is one
of f, e, or E and their meaning is as given for printf in the C++
standard (floating point, exponential with lower case e, exponential
with upper case E).

If -N is used, all DFDs are printed with <places> decimal places.

If -n is used, DFDs are printed out with the same number of decimal
places as they had when they were read in.

If neither -n nor -N is used, DFDs are printed out with the same
number of decimal places as they had when they were read in.

The -f and -F options have no effect on xs:decimal numbers (since
xs:decimal numbers may not use exponential notation).

If -F is used all DFs are printed with the given <format>.

If -f is used, DFs are printed out using exponential notation
(e or E) if they had exponential notation when they were read in,
 but otherwise are printed with the given <format>.

If neither -f nor -F is used, DFs are printed out using exponential
 notation (e or E) if they had exponential notation when they were
 read in, but otherwise are printed with the f format.

The <times> argument gives the number times the file should be parsed.
```

```
The argument exists for testing purposes and is not expected to be
useful otherwise.

If this is compiled with STRINGIN defined, it will read the input file
into a string and then parse it from the string. Otherwise it will
parse directly from the file.

If this is compiled with STRINGOUT defined, after the input is parsed,
it will print the parse tree to a string (with line endings but no
other extra whitespace) and then print the string to the output file.
Otherwise, it will pretty print the output file (with extra white space)
directly from the parse tree.

For STRINGOUT, the class file(s) must also be compiled with STRINGOUT
defined, and xmlSchemaInstanceStr.o (xmlSchemaInstance.cc compiled with
STRINGOUT defined) must be linked in.

For STRINGIN, the lex file must also be compiled with STRINGIN defined.

*/

#include <stdio.h>   // fprintf
#include <string.h>  // strlen
#include <stdlib.h>  // exit
#include "lineClasses.hh"
#if defined(STRINGIN) || defined(STRINGOUT)
#define MAX_SIZE 10000000
#endif

extern LineFile * LineTree;
extern FILE * yyin;
extern int yyparse();
extern void yylex_destroy();

#ifdef STRINGIN
extern char * yyStringInputPointer;
extern char * yyStringInputEnd;
#endif

int XmlSchemaInstanceBase::format;
int XmlSchemaInstanceBase::places;
int yyStartAnew;

void usageMessage(char * call) /* NO ARGUMENTS */
{
  fprintf(stderr,
     "Usage: %s <file name> [-n|N <places>] [-f|F <format>]
[<times>]\n",
          call);
  fprintf(stderr,
          "<places> and <times> are integers; format is f, e or E\n");
  fprintf(stderr, "Example 1: %s dFile.xml\n", call);
```

```
  fprintf(stderr, "Example 2: %s dFile.xml 2\n", call);
  fprintf(stderr, "Example 3: %s dFile.xml -n 5 \n", call);
  fprintf(stderr, "Example 4: %s dFile.xml -F e 2\n", call);
  fprintf(stderr, "Example 5: %s dFile.xml -n 6 -F E\n", call);
  exit(1);
}

void readArguments(/* ARGUMENTS                                          */
  int argc,        /* one more than the number of command arguments  */
  char * argv[],   /* array of executable name and command arguments */
  int * times)     /* number of times to parse                       */
{
  int places;

  if (argc % 2 == 1)
    {
      if (!sscanf(argv[argc - 1], "%d", times))
        {
          usageMessage(argv[0]);
        }
    }
  else
    *times = 1;
  if (argc < 2)
     {
      usageMessage(argv[0]);
    }
  else if (argc < 4)
    {
      XmlSchemaInstanceBase::format = 0;
      XmlSchemaInstanceBase::places = 6;
    }
  else if (argc < 6)
    {
      if (strcmp(argv[2], "-n") == 0)
        {
          XmlSchemaInstanceBase::format = 0;
          if (!sscanf(argv[3], "%d", &places))
            usageMessage(argv[0]);
          XmlSchemaInstanceBase::places = places;
        }
      else if (strcmp(argv[2], "-N") == 0)
        {
          XmlSchemaInstanceBase::format = 0;
          XmlSchemaInstanceBase::format = 0;
          if (!sscanf(argv[3], "%d", &places))
            usageMessage(argv[0]);
          XmlSchemaInstanceBase::places = -places;
        }
      else if (strcmp(argv[2], "-f") == 0)
        {
          XmlSchemaInstanceBase::places = 6;
```

```
          if (strlen(argv[3]) != 1)
            usageMessage(argv[0]);
          if (argv[3][0] == 'f')
            XmlSchemaInstanceBase::format = 0;
          else if (argv[3][0] == 'e')
            XmlSchemaInstanceBase::format = 1;
          else if (argv[3][0] == 'E')
            XmlSchemaInstanceBase::format = 2;
          else
            usageMessage(argv[0]);
        }
      else if (strcmp(argv[2], "-F") == 0)
        {
          XmlSchemaInstanceBase::places = 6;
          if (strlen(argv[3]) != 1)
            usageMessage(argv[0]);
          if (argv[3][0] == 'f')
            XmlSchemaInstanceBase::format = 3;
          else if (argv[3][0] == 'e')
            XmlSchemaInstanceBase::format = 4;
          else if (argv[3][0] == 'E')
            XmlSchemaInstanceBase::format = 5;
          else
            usageMessage(argv[0]);
        }
      else
        usageMessage(argv[0]);
    }
  else if (argc < 8)
    {
      if (strcmp(argv[2], "-n") == 0)
        {
          XmlSchemaInstanceBase::format = 0;
          if (!sscanf(argv[3], "%d", &places))
            usageMessage(argv[0]);
          XmlSchemaInstanceBase::places = places;
        }
      else if (strcmp(argv[2], "-N") == 0)
        {
          XmlSchemaInstanceBase::format = 0;
          if (!sscanf(argv[3], "%d", &places))
            usageMessage(argv[0]);
          XmlSchemaInstanceBase::places = -places;
        }
      else
        usageMessage(argv[0]);
      if (strcmp(argv[4], "-f") == 0)
        {
          if (strlen(argv[5]) != 1)
            usageMessage(argv[0]);
          if (argv[5][0] == 'f')
            XmlSchemaInstanceBase::format = 0;
```

```
          else if (argv[5][0] == 'e')
            XmlSchemaInstanceBase::format = 1;
          else if (argv[5][0] == 'E')
            XmlSchemaInstanceBase::format = 2;
          else
            usageMessage(argv[0]);
        }
      if (strcmp(argv[4], "-F") == 0)
        {
          if (strlen(argv[5]) != 1)
            usageMessage(argv[0]);
          if (argv[5][0] == 'f')
            XmlSchemaInstanceBase::format = 3;
          else if (argv[5][0] == 'e')
            XmlSchemaInstanceBase::format = 4;
          else if (argv[5][0] == 'E')
            XmlSchemaInstanceBase::format = 5;
          else
            usageMessage(argv[0]);
        }
    }
  else
    {
      usageMessage(argv[0]);
    }
}

int main(       /* ARGUMENTS                                          */
 int argc,      /* one more than the number of command arguments  */
 char * argv[]) /* array of executable name and command arguments */
{
  std::string outFileName;
  FILE * inFile;
  FILE * outFile;
  int times;
  int result;
  char buffer[10];
#if defined(STRINGIN) || defined(STRINGOUT)
  char * inString;
  int inStringSize;
  int n;
#endif
#ifdef STRINGOUT
  char * outString;
  size_t outStringSize;
  size_t k;
#endif

  yyStartAnew = 0;
  readArguments(argc, argv, &times);
#if defined(STRINGIN) || defined(STRINGOUT)
  for(inStringSize =10000; inStringSize <= MAX_SIZE; inStringSize *= 10)
```

```
    {
      inString = new char[inStringSize + 1];
      inFile = fopen(argv[1], "r");
      if (inFile == 0)
        {
          fprintf(stderr,"unable to open file %s for reading\n", argv[1]);
          exit(1);
        }
      for (n = 0; (((inString[n] = getc(inFile)) != EOF) &&
                   (n < inStringSize)); n++);
      fclose(inFile);
      if (n < inStringSize)
        break;
      else
        delete [] inString;
    }
  if (inStringSize > MAX_SIZE)
    {
      fprintf(stderr,
            "input file is too large (more than %d bytes), exiting\n",
              MAX_SIZE);
      return 1;
    }
  inString[n] = 0;
#endif
#ifdef STRINGOUT
  outStringSize = (size_t)(inStringSize * 2);
  outString = new char[inStringSize * 2];
#endif
  for (; times > 0; times--)
    {
#ifdef STRINGIN
      yyStringInputPointer = inString;
      yyStringInputEnd = (inString + n);
#else
      inFile = fopen(argv[1], "r");
      if (inFile == 0)
        {
          fprintf(stderr,"unable to open file %s for reading\n", argv[1]);
          exit(1);
        }
      yyin = inFile;
#endif
      yyStartAnew = 1;
      result = yyparse();
#ifndef STRINGIN
      fclose(inFile);
#endif
      if (result == 0)
        {
          outFileName = argv[1];
          sprintf(buffer, "%d", times);
```

```
          outFileName.append(buffer);
          outFile = fopen(outFileName.c_str(), "w");
#ifdef STRINGOUT
          k = 0;
          LineTree->printSelf(outString, &outStringSize, &k);
          outString[k] = 0;
          fprintf(outFile, "%s", outString);
#else
          LineTree->printSelf(outFile);
#endif
          fclose(outFile);
          delete LineTree;
        }
    }
#if defined(STRINGIN) || defined(STRINGOUT)
  delete [] inString;
#endif
#ifdef STRINGOUT
  delete [] outString;
#endif
  yylex_destroy();
  return 0;
}
```

Annex G YACC File line.y

This is the YACC file line.y, generated by the xmlInstanceParserGenerator, as described in Section 5.4. See section 17 regarding YACC files.

```
%{

#include <stdio.h>              // for stderr
#include <string.h>             // for strcat
#include <stdlib.h>             // for malloc, free
#include <map>                  // for map
#include "lineClasses.hh"

#define YYERROR_VERBOSE
#define YYDEBUG 1

LineFile * LineTree; // the parse tree

extern int yylex();
int yyReadData = 0;
int yyReadDataList = 0;
std::map<XmlSchemaInstanceBase *, XmlSchemaInstanceBase *> yyUnrefMap;

int yyerror(const char * s);
```

```
%}

%union {
  AttributePair *                    AttributePairVal;
  AttributePairLisd *                LiztAttributePairVal;
  XmlHeaderForLine *                 XmlHeaderForLineVal;
  XmlVersion *                       XmlVersionVal;
  int *                              iVal;
  char *                             sVal;
  LineFile *                         LineFileVal;

  LineType *                         LineTypeVal;
  PointType *                        PointTypeVal;
  VectorType *                       VectorTypeVal;
  XmlDecimal *                       XmlDecimalVal;
  XmlID *                            XmlIDVal;
}

%type <sVal>                         y_attributeName
%type <AttributePairVal>             y_AttributePair
%type <LiztAttributePairVal>         y_LiztAttributePair
%type <XmlHeaderForLineVal>          y_XmlHeaderForLine
%type <XmlVersionVal>                y_XmlVersion
%type <LineFileVal>                  y_LineFile
%type <XmlDecimalVal>                y_XmlDecimal
%type <XmlIDVal>                     y_XmlID

%type <LineTypeVal>                  y_LineType
%type <XmlIDVal>                     y_Name_XmlID
%type <PointTypeVal>                 y_PointType
%type <PointTypeVal>                 y_Point_PointType
%type <VectorTypeVal>                y_VectorType
%type <VectorTypeVal>                y_Vector_VectorType
%type <XmlDecimalVal>                y_X_XmlDecimal
%type <XmlDecimalVal>                y_Y_XmlDecimal

%token <iVal> BAD
%token <sVal> DATASTRING
%token <iVal> ENCODING
%token <iVal> ENDITEM
%token <iVal> ENDVERSION
%token <iVal> STANDALONE
%token <iVal> STARTVERSION
%token <sVal> TERMINALSTRING
%token <iVal> XMLNSPREFIX
%token <iVal> XMLNSTARGET
%token <iVal> XMLVERSION

%token <iVal> xmlnsATTR
%token <iVal> xmlnsColonATTR
%token <iVal> xsiSchemaLocationATTR
%token <iVal> xsiNoNameLocationATTR
```

```
%token <iVal> LineEND
%token <iVal> LineSTART
%token <iVal> NameEND
%token <iVal> NameSTART
%token <iVal> PointEND
%token <iVal> PointSTART
%token <iVal> VectorEND
%token <iVal> VectorSTART
%token <iVal> XEND
%token <iVal> XSTART
%token <iVal> YEND
%token <iVal> YSTART

%token <iVal> colorATTR

%token <iVal> PointTypeXSIDECL
%token <iVal> VectorTypeXSIDECL

%%

y_LineFile :
          y_XmlVersion LineSTART y_XmlHeaderForLine
          y_LineType LineEND
          {$$ = new LineFile($1, $3, $4);
           LineTree = $$;
           if ($3->getcolor())
             {
               $4->setcolor($3->getcolor());
               $3->setcolor(0);
             }
           if ($1) yyUnrefMap.erase($1);
           if ($3) yyUnrefMap.erase($3);
           if ($4) yyUnrefMap.erase($4);
           if (yyUnrefMap.size())
             {
               delete $$;
               return yyerror("bug: unreferenced memory exists");
             }
           if (XmlIDREF::idMissing())
             {
               delete $$;
               return yyerror("xs:ID missing for xs:IDREF");
             }
          }
        ;

y_XmlHeaderForLine:
          y_LiztAttributePair ENDITEM
          {$$ = new XmlHeaderForLine();
           yyUnrefMap[$$] = $$;
           yyUnrefMap.erase($1);
```

```
            if ($$->badAttributes($1))
              {
                delete $1;
                return yyerror("Bad header attributes");
              }
            delete $1;
           }
        ;

y_AttributePair :
          y_attributeName TERMINALSTRING
          {$$ = new AttributePair($1, $2);
           yyUnrefMap[$$] = $$;
           free($1);
           free($2);
          }
        ;

y_LiztAttributePair :
          y_AttributePair
          {$$ = new AttributePairLisd($1);
           yyUnrefMap[$$] = $$;
           yyUnrefMap.erase($1);
          }
        | y_LiztAttributePair y_AttributePair
          {$$ = $1;
           yyUnrefMap.erase($2);
           $$->push_back($2);
          }
        ;

y_XmlDecimal :
          DATASTRING
          {$$ = new XmlDecimal($1);
           yyUnrefMap[$$] = $$;
           free($1);
           if ($$->bad)
             {
              return yyerror("bad XmlDecimal");
             }
          }
        ;


y_XmlID :
          DATASTRING
          {$$ = new XmlID($1);
           yyUnrefMap[$$] = $$;
           free($1);
           if ($$->bad)
             {
              return yyerror("bad XmlID");
```

```
            }
          }
        ;

y_XmlVersion :
          STARTVERSION XMLVERSION TERMINALSTRING ENDVERSION
          {$$ = new XmlVersion();
           yyUnrefMap[$$] = $$;
           if (strcmp($3, "1.0"))
             {
              free($3);
              return yyerror("version number must be 1.0");
             }
           free($3);
          }
        | STARTVERSION XMLVERSION TERMINALSTRING
          ENCODING TERMINALSTRING ENDVERSION
          {$$ = new XmlVersion();
           yyUnrefMap[$$] = $$;
           if (strcmp($3, "1.0"))
             {
              free($3);
              free($5);
              return yyerror("version number must be 1.0");
             }
           else if ((strcmp($5, "UTF-8")) && (strcmp($5, "utf-8")))
             {
              free($3);
              free($5);
              return yyerror("encoding must be UTF-8 or utf-8");
             }
           else
             strncpy($$->encoding, $5, 6);
           free($3);
           free($5);
          }
        | STARTVERSION XMLVERSION TERMINALSTRING
          STANDALONE TERMINALSTRING ENDVERSION
          {$$ = new XmlVersion();
           yyUnrefMap[$$] = $$;
           if (strcmp($3, "1.0"))
             {
              free($3);
              free($5);
              return yyerror("version number must be 1.0");
             }
           else if ((strcmp($5, "yes")) && (strcmp($5, "no")))
             {
              free($3);
              free($5);
              return yyerror("standalone must be yes or no");
             }
```

```
            else
              strncpy($$->standalone, $5, 5);
            free($3);
            free($5);
           }
         | STARTVERSION XMLVERSION TERMINALSTRING ENCODING TERMINALSTRING
           STANDALONE TERMINALSTRING ENDVERSION
           {$$ = new XmlVersion();
            yyUnrefMap[$$] = $$;
            if (strcmp($3, "1.0"))
              {
               free($3);
               free($5);
               free($7);
               return yyerror("version number must be 1.0");
              }
            else if ((strcmp($5, "UTF-8")) && (strcmp($5, "utf-8")))
              {
               free($3);
               free($5);
               free($7);
               return yyerror("encoding must be UTF-8 or utf-8");
              }
            else if ((strcmp($7, "yes")) && (strcmp($7, "no")))
              {
               free($3);
               free($5);
               free($7);
               return yyerror("standalone must be yes or no");
              }
            else
              {
               strncpy($$->encoding, $5, 6);
               strncpy($$->standalone, $7, 5);
               free($3);
               free($5);
               free($7);
              }
           }
         ;

y_attributeName :
           xmlnsATTR  {$$ = strdup("xmlns");}
         | xmlnsColonATTR {$$ = strdup("xmlns:");}
         | xsiSchemaLocationATTR {$$ = strdup("xsi:schemaLocation");}
         | xsiNoNameLocationATTR {$$ = srtdup("xsi:noNamespaceSchemaLocation");}
         | colorATTR {$$ = strdup("color");}
         ;

y_LineType :
            y_Name_XmlID y_Point_PointType y_Vector_VectorType
           {$$ = new LineType($1, $2, $3, (const char *)0);
```

```
            yyUnrefMap[$$] = $$;
            if ($1) yyUnrefMap.erase($1);
            if ($2) yyUnrefMap.erase($2);
            if ($3) yyUnrefMap.erase($3);
           }
        ;


y_Name_XmlID :
          NameSTART ENDITEM {yyReadData = 1;} y_XmlID NameEND
          {$$ = $4;}
        ;

y_PointType :
          ENDITEM y_Name_XmlID y_X_XmlDecimal y_Y_XmlDecimal
          {$$ = new PointType($2, $3, $4, (const char *)0);
           yyUnrefMap[$$] = $$;
           if ($2) yyUnrefMap.erase($2);
           if ($3) yyUnrefMap.erase($3);
           if ($4) yyUnrefMap.erase($4);
          }
        ;

y_Point_PointType :
          PointSTART PointTypeXSIDECL y_PointType PointEND
          {$$ = $3;
           $$->printElement = "Point";
           $$->printTypp = "PointType";
          }
        | PointSTART y_PointType PointEND
          {$$ = $2;
           $$->printElement = "Point";
          }
        ;

y_VectorType :
          ENDITEM y_Name_XmlID y_X_XmlDecimal y_Y_XmlDecimal
          {$$ = new VectorType($2, $3, $4, (const char *)0);
           yyUnrefMap[$$] = $$;
           if ($2) yyUnrefMap.erase($2);
           if ($3) yyUnrefMap.erase($3);
           if ($4) yyUnrefMap.erase($4);
          }
        ;

y_Vector_VectorType :
          VectorSTART VectorTypeXSIDECL y_VectorType VectorEND
          {$$ = $3;
           $$->printElement = "Vector";
           $$->printTypp = "VectorType";
          }
        | VectorSTART y_VectorType VectorEND
```

```
          {$$ = $2;
           $$->printElement = "Vector";
          }
        ;

y_X_XmlDecimal :
          XSTART ENDITEM {yyReadData = 1;} y_XmlDecimal XEND
          {$$ = $4;}
        ;

y_Y_XmlDecimal :
          YSTART ENDITEM {yyReadData = 1;} y_XmlDecimal YEND
          {$$ = $4;}
        ;

%%

/* ************************************************************** */

/* yyerror

Returned Value: int (1)

Called By: yyparse

This:
1. deletes all entries in the yyUnrefMap.
2. clears the yyUnrefMap.
3. prints whatever string has been provided.
4. returns 1.

*/

int yyerror(      /* ARGUMENTS       */
 const char * s)  /* string to print */
{
  std::map<XmlSchemaInstanceBase *, XmlSchemaInstanceBase *>::iterator iter;

  if (strcmp(s, "bug: unreferenced memory exists"))
    { // get segmentation fault in for loop if unreferenced memory exists
      for (iter = yyUnrefMap.begin(); iter != yyUnrefMap.end(); iter++)
        {
          delete (iter->first);
        }
      yyUnrefMap.clear();
    }
  fflush(stdout);
  fprintf(stderr, "\n%s\n", s);
  return 1;
}

/* ************************************************************** */
```

Annex H **Lex File line.lex**

This is the Lex file line.lex, generated by the xmlInstanceParserGenerator, as described in Section 5.4.

To provide an option for echoing or not echoing the input while reading, the ECHO_IT and ECH macros are used and are controlled by the NO_ECHO compiler macro. Whenever the lexer reads anything, the ECH command is executed. If compiling is done with NO_ECHO defined, as shown in the Makefile of section 9.3.2, ECHO_IT is set to 0. If compiled with NO_ECHO not defined, ECHO_IT is set to 1. In the ECH macro, if ECHO_IT is set to 1, when ECH is executed, the Lex ECHO command runs and prints whatever has been read to stdout, which is normally the computer monitor. If ECHO_IT is not set to 1, ECH does nothing.

```
%{

/*

This ignores white space outside of meaningful strings of characters.

*/

#ifdef WIN32
#include <io.h>
#define strdup _strdup
#define fileno _fileno
#define isatty _isatty
#define YY_NO_UNISTD_H
#endif
#include <string.h>          // for strdup
#include "lineClasses.hh"
#include <lineYACC.hh>    // for tokens, yylval, etc.

#ifndef NO_ECHO
#define ECHO_IT 1
#else
#define ECHO_IT 0
#endif
#define ECH if (ECHO_IT) ECHO

extern int yyReadData;
extern int yyReadDataList;
extern int yyStartAnew;

#ifdef STRINGIN
char * yyStringInputPointer;
char * yyStringInputEnd;
```

```
#undef YY_INPUT
#define YY_INPUT(b, r, ms) (r = set_yyinput(b, ms))

int set_yyinput(char * buffer, int maxSize)
{
  int n;

  n = (maxSize < (yyStringInputEnd - yyStringInputPointer) ?
       maxSize : (yyStringInputEnd - yyStringInputPointer));
  if (n > 0)
    {
      memcpy(buffer, yyStringInputPointer, n);
      yyStringInputPointer += n;
    }
  return n;
}
#endif

%}

W [ \t\n\r]*
%x COMMENT
%x DATA
%x DATALIST
%x XMLVER

%%

  if (yyStartAnew)
    {
      yyrestart(yyin);
      yyStartAnew = 0;
      yyReadData = 0;
      yyReadDataList = 0;
    }
  else if (yyReadDataList)
    {
      BEGIN(DATALIST);
    }
  else if (yyReadData)
    {
      BEGIN(DATA);
      yyReadData = 0;
    }

{W}"<!--"            {ECH; BEGIN(COMMENT); /* delete comment start */}
<COMMENT>.           {ECH;  /* delete comment middle */ }
<COMMENT>\n          {ECH;  /* delete comment middle */ }
<COMMENT>"-->"       {ECH; BEGIN(INITIAL); /* delete comment end */ }

<XMLVER>"xml"[ \t]+"version"{W}"="    {ECH; return XMLVERSION;}
<XMLVER>"encoding"{W}"="              {ECH; return ENCODING;}
```

```
<XMLVER>"standalone"{W}"="            {ECH; return STANDALONE;}
<XMLVER>\"[^\"]+\"      {ECH;
                         int n;
                         for (n = 1; yytext[n] != '"'; n++);
                         yytext[n] = 0;
                         yylval.sVal = strdup(yytext + 1);
                         return TERMINALSTRING;
                        }
<XMLVER>"?>"                 {ECH; BEGIN(INITIAL); return ENDVERSION;}

<DATA>"<"               {BEGIN(INITIAL);
                         unput('<');
                         yylval.sVal = strdup("");
                         return DATASTRING;
                        }
<DATA>[^<]*             {ECH; BEGIN(INITIAL);
                         yylval.sVal = strdup(yytext);
                         return DATASTRING;
                        }

<DATALIST>[^<> \t\n\r]* {ECH;
                         yylval.sVal = strdup(yytext);
                         return DATASTRING;
                        }
<DATALIST>{W}           {ECH;}
<DATALIST>">"           {ECH; return ENDITEM;}
<DATALIST>"<"           {yyReadDataList = 0;
                         yyReadData = 0;
                         unput('<');
                         BEGIN(INITIAL);
                        }

"<?"                          {ECH; BEGIN(XMLVER); return
STARTVERSION;}

"</"{W}"Line"{W}">"           {ECH; return LineEND;}
"<"{W}"Line"                  {ECH; return LineSTART;}
"</"{W}"Name"{W}">"           {ECH; return NameEND;}
"<"{W}"Name"                  {ECH; return NameSTART;}
"</"{W}"Point"{W}">"          {ECH; return PointEND;}
"<"{W}"Point"                 {ECH; return PointSTART;}
"</"{W}"Vector"{W}">"         {ECH; return VectorEND;}
"<"{W}"Vector"                {ECH; return VectorSTART;}
"</"{W}"X"{W}">"              {ECH; return XEND;}
"<"{W}"X"                     {ECH; return XSTART;}
"</"{W}"Y"{W}">"              {ECH; return YEND;}
"<"{W}"Y"                     {ECH; return YSTART;}

{W}"color"{W}"="              {ECH; return colorATTR;}
{W}"xmlns"{W}"="              {ECH; return xmlnsATTR;}
{W}"xmlns:"                   {ECH; return xmlnsColonATTR;}
{W}"xsi:schemaLocation"{W}"=" {ECH; return xsiSchemaLocationATTR;}
```

```
{W}"xsi:noNamespaceSchemaLocation"{W}"=" {ECH; return
xsiNoNameLocationATTR;}

{W}"xsi:type"{W}"="{W}"\"PointType\""  {ECH; return PointTypeXSIDECL;
}
{W}"xsi:type"{W}"="{W}"\"VectorType\""  {ECH; return
VectorTypeXSIDECL; }

">"                              {ECH; return ENDITEM;}

[a-zA-Z]+{W}"="{W}\"[^\"]+\"   {ECH;
                                 int n;
                                 for (n = 0; yytext[n] != '"'; n++);
                                 for (n++;   yytext[n] != '"'; n++);
                                 yytext[n] = 0;
                                 yylval.sVal = strdup(yytext);
                                 return TERMINALSTRING;
                                }

\"[^\"]+\"                       {ECH;
                                 int n;
                                 for (n = 1; yytext[n] != '"'; n++);
                                 yytext[n] = 0;
                                 yylval.sVal = strdup(yytext + 1);
                                 return TERMINALSTRING;
                                }

{W}                              {ECH;}

.                                {ECH; return BAD;}

%%

int yywrap()
{
  return 1;
}
```

Annex I **XML Schema File lineNoAtt.xsd**

This `schema` file, lineNoAtt.xsd, was generated by the xmlSchemaAttributeConverter as described in Section 7.3. Note that the LineType has a “color” `element` and no “color” `attribute`.

```
<?xml version="1.0" encoding="UTF-8"?>

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

```
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">
  <xs:include schemaLocation="primitivesNoAtt.xsd"/>

  <xs:element name="Line"
    type="LineType">
    <xs:annotation>
      <xs:documentation>
        Root element
      </xs:documentation>
    </xs:annotation>
  </xs:element>

  <xs:complexType name="LineType">
    <xs:complexContent>
      <xs:extension base="BaseType">
        <xs:sequence>
          <xs:element name="Point"
            type="PointType"/>
          <xs:element name="Vector"
            type="VectorType"/>
          <xs:element name="color"
            type="xs:token"
            minOccurs="0"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

</xs:schema>
```

Annex J XML Instance File lineNoAtt1.xml

This XML instance file was written by hand.

```
<?xml version="1.0" encoding="UTF-8"?>
<Line
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="../schema/lineNoAtt.xsd">
  <Name>Line_1</Name>
  <Point>
    <Name>Point_1</Name>
    <X>0</X>
    <Y>0</Y>
  </Point>
  <Vector>
    <Name>Vector_1</Name>
    <X>0</X>
    <Y>1</Y>
  </Vector>
  <color>
    green
  </color>
</Line>
```

Bibliography

[1] P. Walmsley, Definitive XML Schema, Upper Saddle River, NJ, USA: Prentice Hall, 2002.

[2] "boost C++ Libraries," [Online]. Available: https://www.boost.org/. [Accessed 17 May 2020].

[3] D. Brown, J. Levine and T. Mason, lex & yacc, O'Reilly Media, 1992.

[4] C. Donnelly and R. Stallman, Bison, the YACC-compatible Parser Generator, http://dinosaur.compilertools.net/bison, 2006.

[5] World Wide Web Consortium, XML Schema Part 2: Datatypes Second Edition - W3C Recommendation 28 October 2004, World Wide Web Consortium, 2004.

[6] World Wide Web Consortium, XMLSchema Part 1: Structures Second Edition - W3C Recommendation 28 October 2004, World Wide Web Consortium, 2004.

[7] World Wide Web Consortium, XML Schema Part 0: Primer Second Edition - W3C Recommendation 28 October 2004, World Wide Web Consortium, 2004.

[8] World Wide Web Consortium, Extensible Markup Language (XML) 1.0 (Fifth Edition), World Wide Web Consortium, 2008.

[9] World Wide Web Consortium, XML Information Set (Second Edition), World Wide Web Consortium, 2004.

[10] "Valgrind Documentation," 9 October 2018. [Online]. Available: valgrind.org/docs/manual/manual.html. [Accessed 7 2 2019].

[11] ISO, ISO. 10303-11: 2004: Industrial automation systems and integration— Product data representation and exchange — Part 11 : Description method: The EXPRESS language reference manual, ISO, 2003.

[12] B. Stroustrup, C++ Programming Language, Addison-Wesley, 2000.

[13] M. Horridge, A Practical Guide To Building OWL Ontologies Using Prot´eg´e 4 and CO-ODE Tools, The University Of Manchester, 2011.

[14] World Wide Web Consortium, OWL 2 Web Ontology Language Primer (Second Edition) —

W3C Recommendation 11 December 2012, World Wide Web Consortium, 2012.

[15] World Wide Web Consortium, OWL 2 Web Ontology Language Structural Specification and Functional–Style Syntax (Second Edition) — W3C Recommendation 11 December 2012, World Wide Web Consortium, 2012.