Integration of CodeSynthesis and QIF

John Michaloski
7/13/2015 9:56:00 AM
QIFCodeSynthesisIntegration.docx

Abstract

This paper describes using CodeSynthesis "XSD" tool and Xerces C++ XML tools in basic Quality Information Framework (QIF) applications. The targeted platforms are Ubuntu and Linux as well as Microsoft Windows. The emphasis of the document is on the deployment of CodeSynthesis XSD tool to generate C++ code that in turn is used to handle various Quality Information Framework (QIF) XML applications. Using the QIF Schema, the method to use the CodeSynthesis XSD tool to generate C++ classes representing the given QIF will be shown. The use of the CodeSynthesis C++ generated code to read QIF XML file and produce QIF XML file from native C++ representations will be given. This QIF C++ code is available on the usnistgov github web site found at https://github.com/usnistgov.

Notation

ANSI	American National Standards Institute
DOM	Document Object Model
FAIR	First Article Inspection Report
GD&T	Geometric dimensioning and tolerancing
IMTS	International Manufacturing Technology Show
MBD	Model Based Design
MSVC	Microsoft Visual C
PDF	Portable Document Format
PMI	Product and Manufacturing Information
QIF	Quality Information Framework
SAX	Simple API for XML
SME	Society of Manufacturing Engineers
STD	Standard Library
STL	Standard Template Library
XML	Extensible Markup Language
XSD	XML Schema

Background

This report describes using CodeSynthesis "XSD" tool and Xerces C++ XML tools in basic Quality Information Framework (QIF) applications. The foundation of the software development is the CodeSynthesis "XSD" software tool (CodeSynthesis, 2015), which was used to generate C++ code for XML Tree Mapping. The primary purpose of this generated C++ code is to serialize and deserialize QIF XML. For our purposes, serialization is a process by which a program's C++ internal representation is

transformed into an XML serial data format. Likewise, deserialization is used to convert the XML, program's C++ internal representation. The CodeSynthesis "XSD" software tool runs on many flavors of Linux as well as Windows platforms. There are nuances to properly using XSD, including correct flags, setup and application that will be covered in this document. The emphasis of the document is on the deployment of CodeSynthesis XSD tool to generate C++ code that in turn is used to handle various Quality Information Framework (QIF) XML applications.

CodeSynthesis uses the Xerces C++ XML Schema (XSD) and XML tools (Mozilla, 2015) to generate code that will parse the QIF XML, as well as to verify compliance to the QIF XSD specification. CodeSynthesis is an open-source, freely distributable code generation licensing, cross-platform W3C XML Schema to C++ data binding compiler. Provided with the QIF Schema, it generates C++ classes that represent the given QIF inspection vocabulary as well as XML parsing and serialization code. Once parsed, you can access the data stored in XML using C++ types and functions that semantically correspond to your application domain rather than dealing with generic XML mechanisms. Typically, but not always, the C++ representation matched to the application domain (in this case inspection) provides an easier programming method than dealing directly with the XML.

This report covers installation of the CodeSynthesis XSD tool onto Windows and Linux machines. Using the CodeSynthesis XSD tool, the process in which to generate the C++ code will be presented. Next, compilation, linking and execution for Windows Visual C++ 2010 and Linux Eclipse CDT will be shown. Finally, major discrepancies between the two platforms and the programming steps to alleviate these problems will be presented. The basic application of the CodeSynthesis C++ generated code to read QIF XML file and produce QIF XML file from native C++ representations will be given. Underlying generated C++ code parses the QIF XML and serializes QIF XML. This QIF C++ code is available on the usnistgov github web site found at https://github.com/usnistgov.

Quality Information Framework (QIF)

The Quality Information Framework (QIF) is an ANSI standard sponsored by the Dimensional Metrology Standards Consortium (DMSC) that defines an integrated set of XML information models to enable the effective exchange of metrology data throughout the entire manufacturing process – from product design to inspection planning to execution to analysis and reporting. QIF handles feature-based dimensional metrology, quality measurement planning, first article inspection, and discrete quality measurement.

The Extensible Markup Language (XML) is a metalanguage, that is, a language used to create other XML languages. XML is typically used for defining a language to exchange data across the internet. With XML, the contextual meaning of the data can be represented to create a more application-specific language. The XML structure is a tree of elements containing branches of XML elements. An XML element is delimited by a start tag (ex: <ElementName>) and an end tag (</ElementName >) with CDATA being the non-markup character data (not XML tag related) between the tags. Elements can also have attributes, which are name-value pairs inside start tag (e.g.,: < PartDefinition id="partDefinition1">). The structure of XML

is fundamentally tree oriented (a graph without any cycles). In QIF, the cross linking of information is done using unique identifiers (as shown by the "id" attribute).

QIF is based on XML, and uses terminology and semantics from the inspection world to represent the various elements in the QIF specification. The QIF information models are contained in files written in the XML Schema Definitions (XSD). The QI XSD Version 2.0 models consists of six application schema files QIFRules, QIFResults, QIFPlans, QIFProduct, QIFStatistics and QIFMeasurmentResources bundled into a QIF Document. QIF also includes a library of XSD schema files containing information items used by all QIF applications (Auxiliary, Characteristics, Expressions, Features, GenericExpressions , Geometry , IntermediatesPMI , Primitives , PrimitivesPD , PrimitivesPMI , Statistics, Topology, Traceability, Units, and Visualization). Figure 1 shows a high level view perspective of the relationship among the different QIF information models. At the core of the QIF architecture is the reusable QIF library which contains definitions and components that are referenced by the application areas. Around the QIF library core, Figure 1 shows the six QIF application area information models, Model Based Design (MBD), QIF Plans, QIF Resources, QIF Rules, QIF Results, and QIF Statistics. The "QIF Execution" model is not shown and work for a future standardization effort that is now handled by the DMIS standard. The order of generation of QIF data generally proceeds clockwise around the diagram, beginning with QIF MBD and ending with QIF Statistics. Of note, users of the QIF information model are not required to implement the entire model.

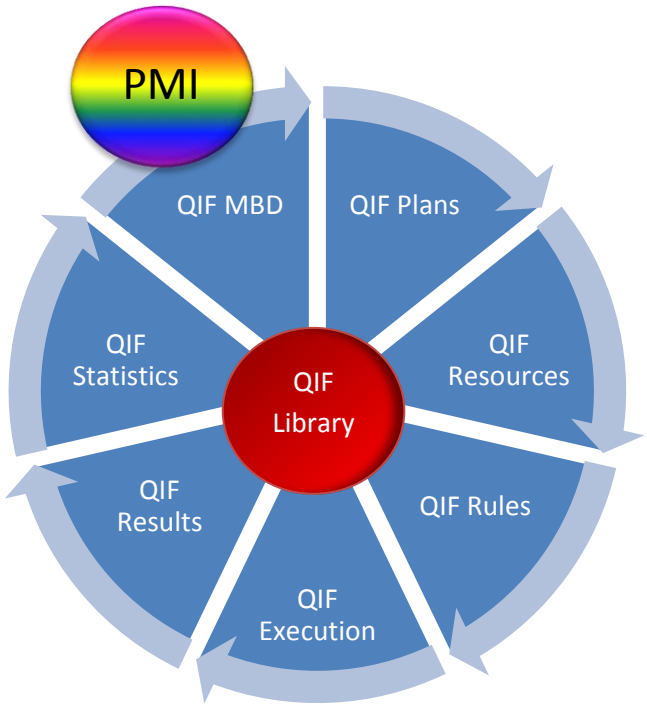


Figure 1 QIF 2 'Life Saver' Architecture

The QIF Library contains all the miscellaneous definitions required of the QIF applications. QIF Library includes the XSD modules: Auxiliary.xsd, Characteristics.xsd, Features.xsd, GenericExpressions.xsd ,

Geometry.xsd, IntermediatesPMI.xsd, Primitives.xsd, PrimitivesPD.xsd, PrimitivesPMI.xsd, Statistics.xsd, Topology.xsd, Traceability.xsd, Units.xsd, and Visualization.xsd. Front-end PMI information needs to be mapped into QIF Model Based Design (MBD).

CodeSynthesis

CodeSynthesis XSD is an open-source, cross-platform W3C XML Schema to C++ data binding compiler. Given as input an XML instance specification (XML Schema or XSD), it generates C++ classes that represent the given vocabulary as well as XML parsing and serialization code.

The CodeSynthesis web site codesynthesis.com/products/xsd/ gives a background on the XML functionality. Briefly, CodeSynthesis supports two XML Schema to C++ mappings: in-memory C++/Tree and stream-oriented C++/Parser. The C++/Tree mapping represents the information stored in XML documents as a tree-like, in-memory document object model (DOM). The C++/Parser is a SAX-like mapping which represents the data stored in XML as a hierarchy of vocabulary-specific parsing events.

This project chose to use a Tree/DOM representation, so an entire QIF XML file is read into corresponding C++ structures and then can be manipulated in either CodeSynthesis C++ Tree form, or Mozilla Xerces DOM form. CodeSynthesis provides the XSD tool to generate a C++ representation of XML schema (either string or file) into a DOM representation (called Tree in Code Synthesis). In CodeSynthesis, the XML can be parsed (read an XML file) or serialized (write an XML file) which are useful when parsing or generating XSD compliant XML.

Some benefits accrue from the code generation process. Compared to XML library APIs that rely exclusively on generic XML signatures, such as DOM and SAX, XML data binding allows access to the XML data in inspection domain vocabulary instead of generic elements, attributes, and text. Using domain specific C++ code, static typing helps catch errors at compile-time rather than at run-time. Overall, automatic code generation saves time and minimizes the effort needed to adapt your applications to changes in the document structure.

Mozilla Xerces

CodeSynthesis uses the open source Mozilla Xerces XML parser (xerces.apache.org) for the XML parsing. Xerces is a validating open source XML parser written in a portable subset of C++. Xerces provides an application the ability to read and write XML data. Xerces-C++ is faithful to the many XML recommendations and associated standards. Mozilla Xerces provides an open source XML DOM and SAX toolkit, originally released by IBM to Mozilla then to the Apache Software Foundation, with a liberal license agreement (http://www.ibm.com/developerworks/library/x-xercc/index.html). Xerces can be compiled shared library is provided for parsing, generating, manipulating, and validating XML documents. The Xerces parser provides high performance, modularity, and scalability. In fact, Xerces toolkit is included in the CodeSynthesis XSD to C++ distribution.

The Mozilla Xerces DOM form is useful for XPATH navigation of the QIF XML tree to find matches and either nodes or values of the matches. CodeSynthesis does not include XQilla in its distribution, so th the ability to perform the XPATH 2.0 queries on the XML tree requires additional downloading, compilation, etc. Helpful XPath Websites include:

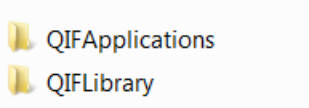
```
https://developer.mozilla.org/en-US/docs/XPathResult
http://www.wrox.com/WileyCDA/Section/XPath-Support-in-Browsers-Page-2.id-291861.html
```

Installation

Windows Installation

Go to the CodeSynthesis XSD Download URL (www.codesynthesis.com/products/xsd/download.xhtml) and select the xsd-4.0.msi installer. When you run this install it will determine if you have a 32 or 64 bit platform. Both 32 and 64 bit architectures are included in the distribution.

Copy the XSD files from the QIF distribution into a folder separated as in the distribution:



The XSD tool will use the include directive in the qif xsd files to file the files.

To generate C++ classes that can be serialized (generate XML) you must have the “generate-serialization” flag. To incorporate the XML inheritance hierarchy (including extensions, substitution groups, etc.), you must have the “generate-polymorphic “flag enabled before compiling all the QIF xsd files. Thus, for the QIFDocument.xsd file you would have a command line such as:

```
xsd.exe cxx-tree --generate-serialization --generate-polymorphic   QIFDocument.xsd
```

This assumes that the xsd.exe is in the environment variable path. Thus, a simple approach is to use the explicit path to the CodeSynthesis xsd program:

```
“C:\Program Files (x86)\CodeSynthesis XSD 4.0\bin\xsd.exe"
```

Of note, the code synthesis program is always installed in the 32-bit program files folder. Because of the blank spaces in the path, double quote the path when using the DOS cmd shell.
Next, create a batch file (GenCPPCode.bat) in both the application and the library directories. The batch file will contain:

```
"C:\Program Files\CodeSynthesis XSD 4.0\bin\xsd.exe" cxx-tree --generate-serialization --generate-polymorphic --generate-test-driver QIFDocument.xsd
"C:\Program Files\CodeSynthesis XSD 4.0\bin\xsd.exe" cxx-tree --generate-serialization --generate-polymorphic QIFMeasurementResources.xsd
"C:\Program Files\CodeSynthesis XSD 4.0\bin\xsd.exe" cxx-tree --generate-serialization --generate-polymorphic QIFPlan.xsd
"C:\Program Files\CodeSynthesis XSD 4.0\bin\xsd.exe" cxx-tree --generate-serialization --generate-polymorphic QIFProduct.xsd
"C:\Program Files\CodeSynthesis XSD 4.0\bin\xsd.exe" cxx-tree --generate-serialization --generate-polymorphic --generate-test-driver QIFResults.xsd
"C:\Program Files\CodeSynthesis XSD 4.0\bin\xsd.exe" cxx-tree --generate-serialization --generate-polymorphic QIFRules.xsd
"C:\Program Files\CodeSynthesis XSD 4.0\bin\xsd.exe" cxx-tree --generate-serialization --generate-polymorphic QIFStatistics.xsd
pause
```

Upon running this batch script, the xsd command will generate both *.cxx and *.hxx file for each xsd file listed in the batch file. Fortunately, all include paths to other QIF files are relative.

Next, you need to create a Visual Studio C++ console project and then copy the generated files *.cxx and *.hxx to an underlying folder (either QIFLibrary or QIFApplication) depending on the QIF XSD file origination folder. Below shows a MSVC (Microsoft Visual C) folder layout for the project/solution files.

```
├──MSVCQIF
│   ├──Win32
│   └──x64
├──QIFApplications
└──QIFLibrary
```

You will need to have these folders in your path and the simplest way is to create an “includes.txt” file that is included in your MSVC C++ command line:

```
-I.
-I"C:\Users\michalos\Documents\GitHub\QIF\src\MSVCQIF"
-I"C:\Users\michalos\Documents\GitHub\QIF\src\MSVCQIF\QIFApplications"
-I"C:\Users\michalos\Documents\GitHub\QIF\src\MSVCQIF\QIFLibrary"
-I"C:\Program Files (x86)\CodeSynthesis XSD 4.0\include"
```

Also while you are changing the compilation flags, CodeSynthesis generates a lot of template code (which does static checking) but can take up a lot of memory, which exceeded the Visual C++ memory allotment size. It tells you to add /bigobj flag, which was added to the C++ Command Line along with the include.txt directive.

```
@Include.txt /bigobj
```

To reach this Right click your XML project -> C++ -> Command Line and place the above in the “Additional Options” text box.

CodeSynthesis walks you thru the Visual Studio Project setting to enable the compilation and linking:

```
http://wiki.codesynthesis.com/Using_XSD_with_Microsoft_Visual_Studio
```

I did not bother adding the recompilation of the xsd file. It does explain the lookup order difference between 32 and 64 bit architectures. Basically, you need to add the CodeSynthesis directories for include, library and bin to your project.

Then you will need to add the CodeSynthesis lib file using a "#pragma comment" to the project (in the main file). The example below handles 32 and 64 bit platforms, and is using Visual C++ 2010 depending on the compilation flags (for either 32 or 64 bit debug or release builds).

```
#if defined(WIN64) && defined( _DEBUG)
#pragma message( "DEBUG x64" )
#pragma comment(lib, "C:\\Program Files (x86)\\CodeSynthesis XSD 4.0\\lib64\\vc-10.0\\xerces-c_3D.lib")

#elif !defined( _DEBUG) && defined(WIN64)
#pragma message( "RELEASE x64" )

#elif defined(_DEBUG) && defined(WIN32)
#pragma message( "DEBUG x32" )
pragma comment(lib, "C:\\Program Files (x86)\\CodeSynthesis XSD 4.0\\lib\\vc-10.0\\xerces-c_3D.lib")

#elif !defined( _DEBUG) && defined(WIN32)
#pragma message( "RELEASE x32" )
#endif
```

The preprocessor flag WIN64 is defined when 64 bit compiles are done, and WIN32 is defined when 32 bit compilations are performed. No other magic was done to get the CodeSynthesis generated C++ code to compile and link and generate an executable. Visual Studio handled the .cxx and the .hxx file extensions with no problems.

Linux Installation

The Web site http://codesynthesis.com/products/xsd/download.xhtml explains how to download and install the XSD tool.

The Eclipse compiler on Ubuntu 14.04 OS was used to test the CodeSynthesis generated C++ code. Again a full path to the CodeSynthesis XSD executable was done to alleviate any PATH environment settings, since again it was only done once. Note, Unix CodeSynthesis XSD executable does not have an "exe" MIME extension. The "--root-element QIFDocument" XSD option was added to prevent the error from occurring during the bash shell execution, since QIF has many global element definitions, but only QIFDocument is the root. Overall, create a bash script file (GenCPPCode.bash) was created in both the application and the library directories. The bash script file will contain

```
/home/michalos/xsd/bin/xsd cxx-tree --root-element QIFDocument --generate-serialization --generate-polymorphic
QIFDocument.xsd
/home/michalos/xsd/bin/xsd cxx-tree --root-element QIFDocument --generate-serialization --generate-polymorphic
QIFMeasurementResources.xsd
```

```
/home/michalos/xsd/bin/xsd cxx-tree --root-element QIFDocument --generate-serialization --generate-polymorphic QIFPlan.xsd
/home/michalos/xsd/bin/xsd cxx-tree --root-element QIFDocument --generate-serialization --generate-polymorphic QIFProduct.xsd
/home/michalos/xsd/bin/xsd cxx-tree --root-element QIFDocument --generate-serialization --generate-polymorphic QIFResults.xsd
/home/michalos/xsd/bin/xsd cxx-tree --root-element QIFDocument --generate-serialization --generate-polymorphic QIFRules.xsd
/home/michalos/xsd/bin/xsd cxx-tree --root-element QIFDocument --generate-serialization --generate-polymorphic QIFStatistics.xsd
```

A new Eclipse program was created, and the Eclipse has options for compilation and the GNU C++ compiler (GCC) was chosen. The generated QIF *.cxx and *.hxx file were copied into the a new matching folder, and then dragged into the Eclipse Workspace.

```
├──.settings
├──Debug
│  └──src
│       │  Qif.cpp
─────| QIFApplications
│      │     QIFDocument.cxx
│      │     QIFDocument.hxx
│      │     QIFMeasurementResources.cxx
│      │     QIFMeasurementResources.hxx
│      │     QIFPlan.cxx
│      │     QIFPlan.hxx
│      │     QIFProduct.cxx
│      │     QIFProduct.hxx
│      │     QIFResults.cxx
│      │     QIFResults.hxx
│      │     QIFRules.cxx
│      │     QIFRules.hxx
│      │     QIFStatistics.cxx
│      │     QIFStatistics.hxx
│     ├──QIFLibrary
│     ├──  ….
│     └──Xsd
├──src
│  ├──QIFApplications
│  └──QIFLibrary
```

The code to parse the QIF example file, is the same as for Windows.

```
// Parse QIF XML and generate FAIR reports
std::string filename=ExeDirectory() + "QIF_Results_Sample.xml";
std::auto_ptr<QIFDocumentType> qif (
            QIFDocument (filename,
            xml_schema::flags::dont_initialize|xml_schema::flags::dont_validate|xml_schema::flags::keep_dom)
            );

DOMElement* e = static_cast<DOMElement*> ((*qif)._node ());
```

Programming Differences MSVC and GCC

The transition from Windows Visual Studio C++ 2010 to Linux gcc on Eclipse uncovered some differences. This section will highlight some of the changes involved.

std::transform

First, some of the string utilities (MakeUpper, MakeLower) required a modification of the STD library transform function name scoping, from std::toupper to global naming ::toupper:

```
std::transform(s.begin(), s.end(), std::back_inserter(out), ::toupper);
```

That is, toupper is now defined in the global namespace, instead of the one defined in std namespace, which gcc could not find.

Handling boost intrusive pointer null equality

The boost intrusive pointer was used to handle smart pointer implementation (counts references and deletes the object when a "0" reference count occurs). Unfortunately, Visual C++ understood the comparison between a smart pointer and null, while gcc complained of operator overloading. To overcome this problem, the intrusive pointer used the reference method get() to retrieve the pointer, so it could be compared to NULL, as in:

```
IXmlNodePtr def;
if(def.get() == NULL)
```

Handling Null pointer signal

Xerces and CodeSynthesis rely on the use of pointers to navigate the XML tree. A null pointer has a value reserved for indicating that the pointer does not refer to a valid object. A null pointer can occur often in XML mappings to signify that an optional element branch is not instantiated. Since QIF XSD specifies numerous optional elements, which can be represented by a null pointer in C++, it is important to either test every XML element variable access to insure that it is valid (i.e., non-null) or develop some mechanism to simply the cascading of pointer variables.

Because of the multitude of pointers and the possibility of a NULL pointer exception, Window and Linux C++ code was developed to trap all signals/exceptions thrown when accessing NULL pointer. One easy way to do perform a safe pointer access is with a preprocessor macro, defined below as SAFEFETCH, which encloses all reference chains in a try block, and will catch any exception, and if an exception is thrown, the X variable will be assigned a default Z value.

```
#define SAFEFETCH(X,Y,Z) \
        try { X=Y; } catch(...) { X=Z;}

std::string version;
```

```
SAFEFETCH(version,  qif->Version()->ThisInstanceQPId(), "");
```

However, this assume that a zero pointer access will throw an exception, which is not handled in a standard by either the Microsoft Windows or Linux operating systems.

Windows – Structured Exception Handling

Windows and Visual C++ support structured exception handling (SEH), to catch common hardware and Operating System (OS) signals – such as divide by zero, access a null pointer, etc. This is different that a C++ standard exception. A C++ exception is a feature of the programming language C++. A structured exception is a different concept of the Windows operating system. Structured exceptions are provided by Windows, with support from the kernel. Structured exceptions are raised by Windows upon invalid operations such as access an invalid memory location, which happens when referencing a null pointer. If you don't handle structured exceptions your program will bomb with the dreaded unhandled exception popup dialog will appear on the screen.

Instead, to insure that all exceptions are caught as standard exception and not as Microsoft structured exceptions, you can catch all the structured exceptions, and then throw a C++ standard exception. To do this, you must first turn on the compiler setting to handle Microsoft structured exceptions:

```
// C++ Project ->Properties -> Compiler -> Code Generation -> Enable C++ Exceptions -> Yes with SEH Exceptions (/EHa)
```

Because Windows headers will cause a namespace collision with the Xerces DOM space, the structured exception mapping was included in a separate .cpp file to avoid these headaches. Microsoft Windows includes the XML parser and DOM without any namespace typing. For this reason, many of the Windows headers were not included whenever the Xerces or CodeSynthesis headers were included.

To map the structured exception, your first include the "Windows.h" and <exception> headers. Then, you define a static callback function with the `void trans_func( unsigned int u, EXCEPTION_POINTERS* pExp )` to handle the exception. Finally, this call back function is enabled to catch structured exception by using the `_set_se_translator` method. In the `trans_func` code it merely throws a standard exception, but the important item is that the exception is caught and dealt with.

```
#include "Windows.h"
#include <exception>
// C++ Project ->Properties -> Compiler -> Code Generation -> Enable C++ Exceptions -> Yes with SEH Exceptions (/EHa)
static void trans_func( unsigned int u, EXCEPTION_POINTERS* pExp )
{
	throw std::exception( "MSVC Exception caught in trans_func", pExp->ExceptionRecord->ExceptionCode);
}
struct CSetup
{
  CSetup::CSetup()
 {
	_set_se_translator( trans_func );   //Handles Win32 exceptions (C structured exceptions) as C++ typed exceptions
 }
};
```

```
CSetup setup;
```

The CSetup setup; just uses the static initialization time to setup the exception handling, and since this is a single threaded process, this only needs to be done once. If you had multi-threaded environment, you would need to handle structured exceptions in each thread.

Linux Null Pointer Signal

Next, SEH is not handled on Linux or by gcc. Instead, C++ code was added to handle signal that is mapped into a C++ exception. This functionality is needed to handle the (0)->x which signals an SIGENV interrupt in gcc. Code was added to map signal handling into C++ exception , so that (0)->x now causes a C++ std exception, which allows the safe fetch macro to operate properly.

In gcc this required the following steps:

1) -fnon-call-exceptions needed to be added to the gcc compile flags.

2) Then the C++ code was added to map the signal into a C++ exception. Note the clearing of the signal is important (sigprocmask(SIG_UNBLOCK, &x, NULL);) or the program terminates.

```
#include <signal.h>
static void segfault_sigaction(int signal, siginfo_t *si, void *arg)
{
    printf("Caught segfault at address %p\n", si->si_addr);
    sigset_t x;
    sigemptyset (&x);
    sigaddset(&x, SIGSEGV);
    sigprocmask(SIG_UNBLOCK, &x, NULL);

    throw std::exception();
}

void SetupSignalHandler() {
       struct sigaction sa;

       memset(&sa, 0, sizeof(sigaction));
       sigemptyset(&sa.sa_mask);
       sa.sa_sigaction = segfault_sigaction;
       sa.sa_flags = SA_SIGINFO;

       sigaction(SIGSEGV, &sa, NULL);

}
```

Simple Applications

Parsing

First step is to parse a QIF XML file. This entails specifying the file location and then parsing the XML into the corresponding C++ data structures. QIF provides examples, and the file "QIF_Results_Sample.xml" from the QIF/DMSC web site is used as an example.

```
XMLPlatformUtils::Initialize ();
std::string filename= ExeDirectory() + "\\QIF_Results_Sample.xml";
std::auto_ptr<QIFDocumentType> qif (
        QIFDocument (filename, xml_schema::flags::dont_initialize |
                xml_schema::flags::dont_validate |
                xml_schema::flags::keep_dom));
cerr << "Version: " << qif->Version()->ThisInstanceQPId() << endl
...
XMLPlatformUtils::Terminate ();
```

The code first initializes the Xerces library. Next, it uses a custom build method to retrieve the executable folder location, and in this folder is the QIF example QIF_Results_Sample.xml. There are flags that are passed into the parser: dont_initialize, dont_validate, and keep_dom. The dont_initialize flag is necessary since the command XMLPlatformUtils::Initialize (); was already done, so it would be redundant. The dont_validate flag tells the Xerces parse not to validate the XML with the embedded XSD (easier to get working). Finally, the flag keep_dom tells the CodeSynthesis core code to keep the Xerces generated DOM information, because later we want to use Xerces limited XPATH 1.0 facility, and this information is required. Otherwise the DOM nodes are discarded.

Assuming you created a console application, you will be able to see the output. The variable qif uses a pointer (i.e., ->) because it is defined as an auto_ptr which handles pointer destruction, otherwise you would have memory leaks. Optional elements can be tricky in that the () operator is overloaded and will fetch the pointer to the C++ element. Thus, the element "Version" has the () operator (and we know it's not NULL) to fetch the pointer, that can then get the optional element ThisInstanceQPId. The handling of the transformation between elements and strings can be dubious.

XML elements serve many purposes. XML elements can be empty as defined by:

```
<element></element>
or
<element/>
```

XML elements can be single or multiple instances:

```
<parent>
        <element>abc</element>
        <element>def</element>
        <element>ghi</element>
```

```
</parent>
```

In the QIF XSD, elements are declared with a name, type minOccurs and maxOccurs which are all part of the XSD standard. Below is a XSD definition for element "X" that is based on the XType with minOccurs=n where n can be "0" or "1" and maxOccurs=m where m can be "1", "2", … tor "unbounded".

```
<xs:element name="X" type="XType" minOccurs="n" maxOccurs="m">
```

Thus, the XSD definition determines whether a single element or an array of elements is possible by describing the minimum and maximum count. If the minimum count is zero, the element is optional. If the element count is unbounded, an array of elements is possible. If the minimum count is zero, and the maximum count is unbounded, then an optional array of elements is possible. And finally, the case where the minimum and maximum counts are both one, then a single mandatory element is required.

The code to handle element arrays is based on the STL vector. Thus, as shown below, the use of a STL vector iterator is common to transverse potentially multiple elements. Often the array may be empty, if the element is designated optional by the XSD. However, optional or not, an array of elements always has a STL vector ascribed to its C++ definition. Of note below, the Name element is optional, so it must have a pointer access (i.e., *) after its access in order to reference the underlying string type.

```
for (::xsd::cxx::tree::sequence< PartType>::iterator it=qif->Product()->PartSet()->Part().begin ();
            it != qif->Product()->PartSet()->Part().end (); ++it)
{
          std::string product_name = *((*it).Name());
}
```

Serialization

Serialization is a mechanism whereby you can archive a program's internal representation into some external format, in this case XML. The primary purpose of serialization in the CodeSynthesis is to enable the conversion of the C++ language runtime objects into XML files (although many think of serialization as a two-way process – both into and out). However, CodeSynthesis makes the distinction, "parsing" turns XML files into C++ internal representations, and "serialization" turns internal C++ representations into XML files. Serialization of C++ objects to XML facilitates persisting or transporting the state of such objects in an open, standards compliant and platform agnostic manner.

To serialize XML data, you must have the CodeSynthesis XSD flag --generate-serialization or serialization is not possible.

To generate and serialize XML data, you start by creating a Xerces root node and then assigning it to the QIFDocument. This entails the following coding steps:

```
DOMImplementation* impl =  DOMImplementationRegistry::getDOMImplementation(X("Core"));
DOMDocument* doc = impl->createDocument(
                        0,                    // root element namespace URI.
                        X("QIFDocument"),       // root element name
                        0);                   // document type object (DTD).
DOMElement* rootElem = doc->getDocumentElement();
QIFDocumentType qif(*rootElem);
```

Now, an element <QIFDocument> has been created in the QIFDocumentType. This element to type naming convention is similar to all classes in CodeSynthesis. Depending on whether the element has required subelements or all optional, the create child sequence is call the type constructor (with potentially required elements defined) and then you can assign optional elements values. The following code defines a Version element under the QIFDocument root node, and then fills the element ThisInstanceQPId with a QPIdType definition. CodeSynthesis relies heavily on using reference to variables (&var) so I am not sure what happens if the constructors declared on the heap go out of scope.

```
qif.Version(::xsd::qif2::VersionType()) ;
qif.Version()->ThisInstanceQPId(QPIdType("1.0"));
```

Many types are strings and CodeSynthesis handles the translation from C++ string into the Xerces XMLCh string:

```
qif.Header(QIFDocumentHeaderType());
qif.Header()->Scope("Results");
qif.Header()->Description()="Hello World";
```

In this case, a Header element is declared and the Scope and Description subelements string text fields are assigned values.

The handling of element arrays is handled differently. It uses logic based on the STL vector, so if you are familiar with STL or std::vector, it should be straightforward. First, a measurement results type is created, and then a measurement result is created (note the additional “s” for a vector, and no “s” for an instance – this is a common QIF programming convention). The measurement results type requires a definition for the InspectionStatus (using empty constructor) and for the attribute ID type which is assigned the integer 58. Then, the Measurement Result is pushed onto the measurements results vector via the push_back command.

```
using namespace ::xsd::qif2;
MeasurementsResultsType& results = qif.MeasurementsResults(MeasurementsResultsType());

MeasurementResultsType &m = MeasurementResultsType(
                                (const MeasurementResultsType::InspectionStatus_type&)
                                MeasurementResultsType::InspectionStatus_type(),
                                (const MeasurementResultsType::id_type&) 58);
results.MeasurementResults().push_back(m);
```

Now, the ability to map XML into C++ is addressed. First lets look at some code under the MeasurementResults element:

```
<EdgePointFeatureActual id="10">
        <FeatureItemId>9</FeatureItemId>
        <Location>2460.72 770.62 944.98</Location>
        <Normal>-0.735465884156759 -0.307902932144912 0.603560864882807</Normal>
</EdgePointFeatureActual>
```

This is the Edge Point Feature Actual which has been measured. In the C++ code we declare a EdgePointFeatureActualType with "id" attribute 10 and required FeatureItemId element 9.

```
EdgePointFeatureActualType &edge = EdgePointFeatureActualType(10,9);
```

Next, the location points are added using a STL vector to store the values, before assigned to EdgePointFeatureActual.

```
std::vector<double> pts;
pts.push_back(2460.72); pts.push_back(770.62); pts.push_back(944.98);
edge.Location() = ActualPointType(ListDoubleType(pts.begin(), pts.end()));
```

Then the normal vector is assigned values, again using a STL vector to create the array, and then assigning values to the Normal() element in EdgePointFeatureActual.

```
std::vector<double> norm;
norm.push_back(-0.735465884156759);norm.push_back(-0.307902932144912); norm.push_back(0.603560864882807);
edge.Normal() = ActualUnitVectorType(ListDoubleType(norm.begin(), norm.end()));
```

Next the EdgePointFeatureActual is pushed onto the FeatureActual vector. If you want the CodeSynthesis code to understand the inheritance hierarchy (that EdgePointFeatureActual is a supertype of FeatureActual, then you MUST use the XSD flag --generate-polymorphic or you will always get <FeatureActual> as your element name and lose the location and normal elements.

```
m.MeasuredFeatures()-> FeatureActuals().FeatureActual().push_back((EdgePointFeatureActualType&) edge);
```

It can be a little confusing when to use the pointer or when to use a reference, but fortunately, the compiler tells you when you are wrong.

Now, the serialization process with CodeSynthesis will be sketched. The underlying Xerces serialization needs a namespace, so we define a STL map of namespaces. The empty namespace (i.e., "") is given as the QIF2 xsd. Then the QIFDocument is serialized using std::cout (but could be filename), the XML tree

defined in qif, the namespace map, the encoding (i.e., UTF-8) and the serialization flags, again dont_initialize as this has already been done.

```
xml_schema::namespace_infomap map;

map[""].name = "http://qifstandards.org/xsd/qif2";
map[""].schema= "http://qifstandards.org/xsd/qif2 ../QIFApplications/QIFDocument.xsd";

QIFDocument (std::cout,
                        qif,
                        map,
                        "UTF-8",
                        xml_schema::flags::dont_initialize);
```

XPATH

The XML Path Language (XPath) defines a way how to select parts of XML documents. XPath is important since it used in many different XML technologies. XPath is used to navigate through elements and attributes in an XML document. XPath is a query language given as a W3C's standard. Both XQuery and XPointer are both built using XPath expressions. From the W3C documentation, "XPath models an XML document as a tree of nodes. There are different types of nodes, including element nodes, attribute nodes and text nodes. XPath defines a way to compute a string-value for each type of node. Some types of nodes also have names." XPATH is useful for specifying XML tree navigation paths, and even if the path is non-existent in the given XML tree, handling the tree traversal using a robust approach.

CodeSynthesis explains using XPATH here: http://codesynthesis.com/~boris/blog/2009/05/18/running-xpath-on-cxx-tree-object-model/, but the example did not work as described – it used the iterator paradigm, which wasn't supported in the Xerces distribution, and required an XPath add-on such as Xilla. Plus, the CodeSynthesis example code is geared toward using C++ structures with DOM. Our goal was to use XPATH to either get text values within an <element> or to return a list of DOM nodes that match a criteria and then parse the nodes further.

First and foremost, to make XPATH work in Code Synthesis, you must retain all the DOM elements parsed by Xerces. To do this you must pass in the flag "keep_dom ", which as its name implies keeps the DOM association in the resulting tree. This is imperative for using Xercess/XQilla XPATH, because it uses the DOM information for searching the XML tree for matching XPATH criteria. XQilla is included in the CodeSynthesis distribution, and is implemented in C++ on top of the Xerces-C++ library.

Example C++ code to use XPath is given below. In this case, we need to access the root DOM node using a static cast to store it into the variable "e". Then we declare a STL vector of string to store the values, which will be returned in order that they are found in the QIF XML tree. Finally the routine GetXpathResults is called to fetch the matching node values to the string:

```
"//MeasurementsResults/InspectionTraceability/ReportPreparer/Name"
```

starting with the root node. This code is later used to generate a field value for the First Article Inspection Report.

```
DOMElement* e = static_cast<DOMElement*> ((*qif)._node ());
std::vector<std::string> names;
names = GetXpathResults(e,  "//MeasurementsResults/InspectionTraceability/ReportPreparer/Name");
```

The routine GetXpathResults handles the messy XPATH setup and unwinding of the solution. GetXpathResults declares an empty STL vector of string. Then, the Xerces DOM document pointer is fetched using the getOwnerDocument method on the root element that is passed in. the namespace resolver is declared (and in the QIF case this is the empty (i.e., “”). Next, the XQuery is generated with the method doc->createExpression. Of importance here is the use of DOMXPathResult::ORDERED_NODE_SNAPSHOT_TYPE which tell the XQuery that the result will be a ordered snapshot of matching queries. The CodeSynthesis example uses OMXPathResult::ITERATOR_RESULT_TYPE which would cause an exception to be thrown, so the ordered snapshot was used instead. If there are no results, the empty string vector is returned. Otherwise, the snapshot result is navigated, and the value between each of the element tags (i.e., <element> value> </element>) in the result snapshot array is appended to the STL string vector. After completing the result traversal, the STL string vector is returned.

```
std::vector<std::string> GetXpathResults(DOMElement* root, std::string querystr)
{
	std::vector<std::string> values;

	DOMDocument* doc (root->getOwnerDocument ());

	// Obtain namespace resolver.
	xsd::cxx::xml::dom::auto_ptr<DOMXPathNSResolver> resolver (
		doc->createNSResolver (root));

	// Create XPath expression.
	xsd::cxx::xml::dom::auto_ptr<DOMXPathExpression> expr (
		doc->createExpression (
		xsd::cxx::xml::string (querystr.c_str()).c_str (),
		resolver.get ()));

	// Execute the query.
	xsd::cxx::xml::dom::auto_ptr<DOMXPathResult> r (
		expr->evaluate (
		root,
		DOMXPathResult::ORDERED_NODE_SNAPSHOT_TYPE ,
		 0));

	// If no query matches, then return empty vector
	if (!r.get() )
		return values;

	// Iterate over the result.
	for (int i=0; i < r->getSnapshotLength(); i++)
	{
		r->snapshotItem(i);
		DOMNode* n (r->getNodeValue ());
```

```
                const XMLCh * value = n->getTextContent          (                        );
                values.push_back(xsd::cxx::xml::transcode<char> (value));
        }
        return values;
}
```

Disclaimer

Commercial equipment and software, many of which are either registered or trademarked, are identified in order to adequately specify certain procedures. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

References

CodeSynthesis. (2015). Retrieved 6 17, 2015, from http://wiki.codesynthesis.com/XSD

Mozilla. (2015). *Xerces-C++Parser*. Retrieved 6 23, 2015, from http://xerces.apache.org/xerces-c/