Understanding the BizTalk Mapper

© Daniel Probert 2008
v1.0 Published 29th February 2008

The content in this article first appeared as a series of 13 posts on HTTP://WWW.BIZBERT.COM
The first post is available here:
HTTP://WWW.BIZBERT.COM/BIZBERT/2008/02/07/UNDERSTANDING+THE+BIZTALK+MAPPER+PART+1+INTRODUCTION.ASPX

Contents

Understanding the BizTalk Mapper: Part 1 - Introduction

In this section:

Introduction

The BizTalk Mapper is an integral part of the BTS toolkit and, depending on your inclination and experience, you will probably either love, hate it, or not care about it.

At its core, the BizTalk Mapper is simply a visual tool for specifying XSLT (eXtensible Stylesheet Language Transformations).
This XSLT is used to transform one stream of XML data into another stream of XML data. And since a "stream of XML data" = "message in BizTalk", this means transforming one message into another message.
[Aside: you might think that the Mapper allows you to specify multiple source or destination messages, but this is actually a trick – BizTalk creates a special schema with one part per input/output message – there's still only one input/output message]

How well the BizTalk Mapper performs its task depends on the complexity of the transformation you're attempting – and the skill/experience of the developer using the BizTalk Mapper!

BizTalk Mapper 101

The Mapper presents you with source and destination panes, displaying your source/destination schemas in a tree view.
You then link source elements/attributes to destination attributes/elements, optionally using functoids to perform an operation.
A large collection of default functoids are included, and you can write your own functoids if the operation you're performing isn't included.
One of the functoids (the Script functoid) allows you to directly type in XSLT/C#/VB.NET/JScript or specify a .NET assembly to call to perform an operation.

Additionally you can specify your own XSLT file to use if you have one (e.g. if you use Altova MapForce or Stylus Studio to maintain XSLT) – this will replace anything the Mapper would generate.

History

The BizTalk Mapper was originally unveiled with BizTalk Server 2000.
In its original incarnation, the Mapper was a separate executable, which worked hand-in-hand with the BizTalk Editor.

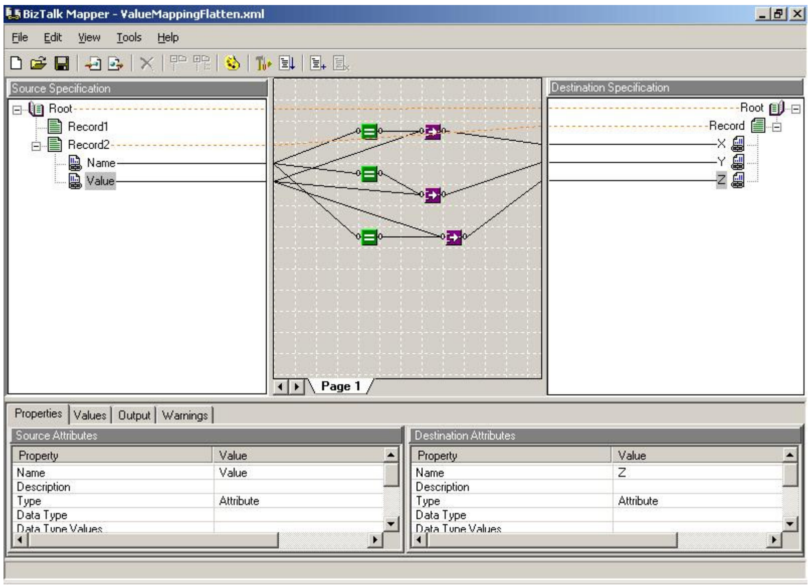
A developer would use the BizTalk Editor to create a specification (analogous to a schema in today's BizTalk), and then use the BizTalk Mapper to map from one specification to another.

A specification was an XDR (Xml-Data Reduced) representation of a document.
XDR specifications could represent DTDs, XML docs, EDI docs, flat files, etc. by noting the position of the elements within them.
(by comparison, BizTalk Server from v2004 onwards uses XSDs (Xml Schema Definition) to accomplish the same task).

BizTalk could convert from a source format to an Xml Document (and vice versa) by referencing the XDR specification.

The XSLT generated by the BizTalk Mapper would convert (map) one Xml Document into another Xml Document, and this Xml Document would then be converted into the destination format via the XDR.

The BizTalk Mapper from BTS 2000/2002 would look very familiar to current BizTalk developers – most of the functoids we use are all there, as is the familiar source/destination grid:



Source Attributes

Property	Value
Name	Value
Description	
Type	Attribute
Data Type	
Data Type Values	

Destination Attributes

Property	Value
Name	Z
Description	
Type	Attribute
Data Type	
Data Type Values	

Under the covers, however, the Mapper was predominantly VBScript based.
Most of the functoids used VBScript to perform the functions, and the script functoid only allowed you to type in VBScript (or JScript in BTS 2002).

One thing about the Mapper never changed though: it was used to generate XSLT, which in BTS 2000/2002 was consumed by the MSXML 3.0 parser to perform transforms.

BizTalk Mapper in BTS 2004 / 2006 / 2006R2

The Mapper present in BizTalk since 2004 still performs the same functions. In fact it looks very similar, and even has the same functoids (along with some new ones).
However there are some important differences:

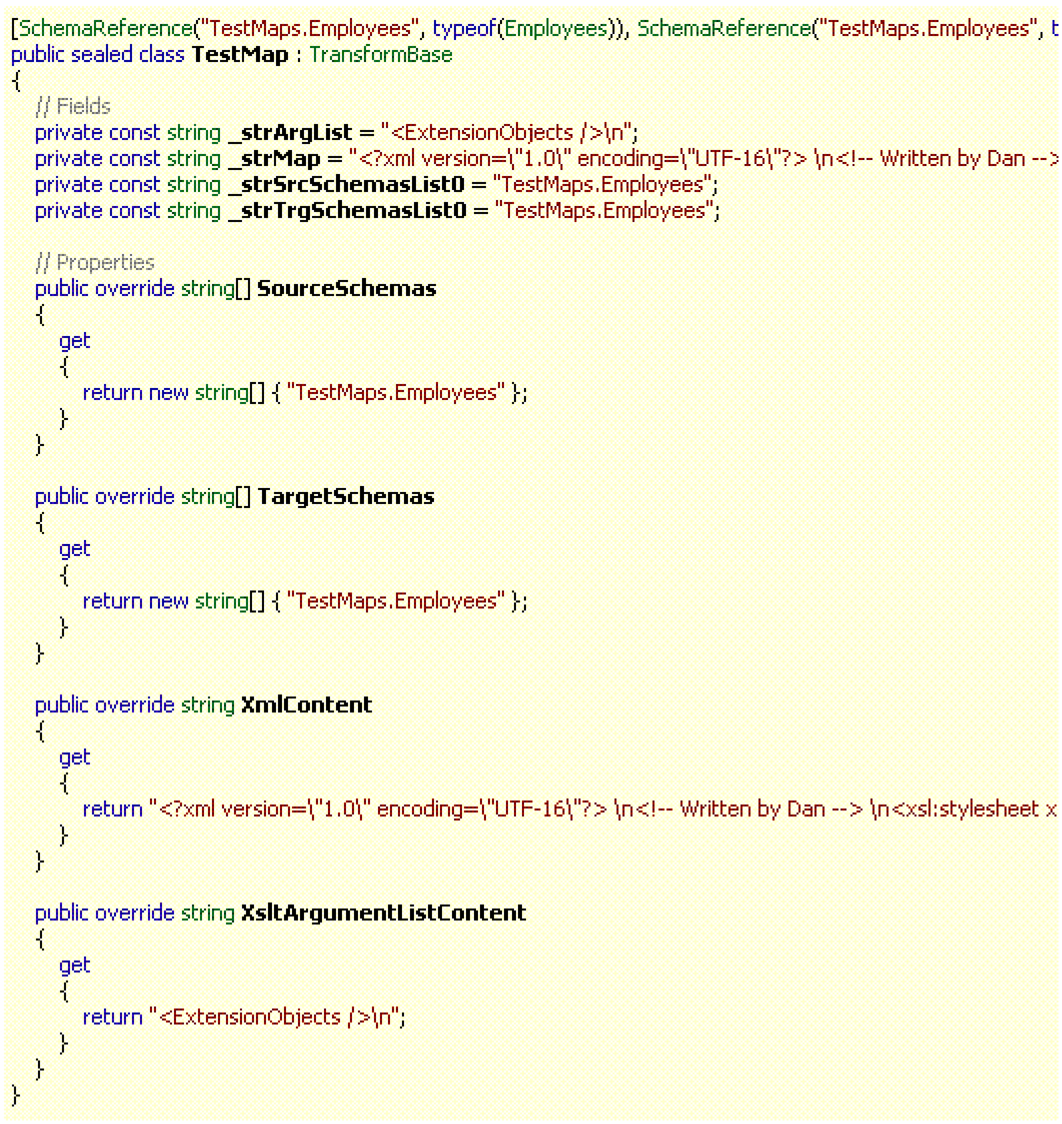
- The Mapper is now integrated into Visual Studio
- The default functoids that used to emit VBScript now emit inline C#
- The Scripting functoid allows you to specify inline XSLT/C#/VB/JScript or to call a method in a GACced assembly
- BizTalk uses XSDs to describe messages, so you select Schemas in the Mapper

What happens when a map is compiled

When the project containing your map is built, the compiler does the following:

1. Generates XSLT from your BTM file (you can see this XSLT by right-clicking your map and choosing "Validate Map" – in the output window you'll see a link to an .xsl file containing your XSLT)
2. Generates a class to represent your map which contains the XSLT as a string, and (optionally) the Extension Object data (also as a string). The class inherits from `Microsoft.XLANGs.BaseTypes.TransformBase`.

The generated class is what is used to perform the translation.
If you disassemble the class (here using .NET Reflector) you'll see something like this:



```
[SchemaReference("TestMaps.Employees", typeof(Employees)), SchemaReference("TestMaps.Employees", t
public sealed class TestMap : TransformBase
{
  // Fields
  private const string _strArgList = "<ExtensionObjects />\n";
  private const string _strMap = "<?xml version=\"1.0\" encoding=\"UTF-16\"?> \n<!-- Written by Dan -->
  private const string _strSrcSchemasList0 = "TestMaps.Employees";
  private const string _strTrgSchemasList0 = "TestMaps.Employees";

  // Properties
  public override string[] SourceSchemas
  {
    get
    {
      return new string[] { "TestMaps.Employees" };
    }
  }

  public override string[] TargetSchemas
  {
    get
    {
      return new string[] { "TestMaps.Employees" };
    }
  }

  public override string XmlContent
  {
    get
    {
      return "<?xml version=\"1.0\" encoding=\"UTF-16\"?> \n<!-- Written by Dan --> \n<xsl:stylesheet x
    }
  }

  public override string XsltArgumentListContent
  {
    get
    {
      return "<ExtensionObjects />\n";
    }
  }
}
```

What's interesting here is that the XSLT is stored as strings – and is repeated: once in a member variable at the top, and then again as a string literal in the `XmlContent` property (I would imagine that this is a bug in the compiler, and that the `XmlContent` property is supposed to return the value of the `_strMap` member variable – but who knows!).

What's important to note is that the XSLT is not *compiled* in any sense of the word.
That doesn't happen until you execute the map.

Important Note: If you use the *Custom XSL Path* property on a map file (to specify an external XSLT file) then the contents of that XSLT file are what appear in the above class – that is, there is no link to the file: instead the contents are copied into the class. In fact, once compiled you can't tell from looking at the class whether the XSLT was from an external file, or created by the BizTalk Mapper.

What happens when a map is executed

Ultimately, all current version of BizTalk use the .NET 1.x `XslTransform` class to perform transformations. (note that if you click on that link to go to the MS documentation there's a big warning saying:
NOTE: This API is now obsolete.
That's because the `XslTransform` class was replaced with the `XslCompiledTransform` class as of .NET 2.0. More on this later…)

The Transform property looks like this:

```
public XslTransform Transform
{
    get
    {
        StringReader input = new StringReader(this.XmlContent);
        XmlTextReader stylesheet = new XmlTextReader(input);
        XslTransform transform = new XslTransform();
        transform.Load(stylesheet, null, base.GetType().Assembly.Evidence);
        return transform;
    }
}
```

(I'm glossing over the fact that there is also a `ScalableTransform` property which returns a `Microsoft.BizTalk.ScalableTransformation.BTSXslTransform` object – which allows for messages to be streamed to/from disk if they're over a certain size - see here for more information).

When BizTalk calls the `Transform` property (for example, when executing a map inside of an orchestration, in which case the `XslTransform.Transform()` method is called by the `Microsoft.XLANGs.Core.Service.ApplyTransform()` method) the above code is executed to load your XSLT into an instance of an `XslTransform` class.

Note: The `XslTransform.Load()` method is called using the current Assembly's evidence – this causes the XSLT to be regarded as fully trusted, indicating that any inline scripts will be run with full permissions.

All that BizTalk has to do now is call `XslTransform.Transform()` to perform the transformation, passing in the source message and an `XsltArgumentList` parameter (containing instances of any classes/types used by the XSLT e.g. classes specified in external assemblies) and the `Transform()` method will return an output stream containing the transformed message.

XslTransform vs XslCompiledTransform

As mentioned above, all current flavours of BizTalk since BTS 2004 use the obsolete `XslTransform` class to perform transformations.
However, .NET 2.0 introduced the new `XslCompiledTransform` class for performing transformations

There are many differences between the two, but to my mind the two most important features introduced by the new class are:

- Full support for XSD includes and imports
- Proper compilation of XSLT to MSIL (which means better performance).

In a lot of cases, `XslCompiledTransform` will significantly outperform `XslTransform` for the same XSLT. The caveat is that because the XSLT is compiled to MSIL, the first time the transform is run there is a perf hit, but subsequent execution should be a lot faster.

For a detailed look at the perf differences between the two classes (plus comparisons with other XSLT processors) have a look at this post.

BizTalk doesn't support the `XslCompiledTransform` class at all. If you've ever tried to validate a real-world schema in BizTalk and had it fail because it doesn't load in the imports/includes (usually when you leave the *Validate TestMap Input/Output* option checked when using the *Test Map* functionality), then you're probably aware of this already…

At the moment, the only way to use the `XslCompiledTransform` class in BizTalk is to implement it yourself and call it from an orchestration.

XSLT v1.0 vs XSLT v2.0

I can't finish this post without a brief note about this: BizTalk (up to 2006 R2) only supports XSLT v1.0. XSLT v2.0 has a lot more functionality in it, but you can't use this in your maps as there is currently no support in .NET for XSLT v2.0 – and there is unlikely to ever be any (and see here).

At the moment if you want to use XSLT 2.0 in .NET (or BizTalk) you have to use an external XSLT processor – something such as SAXON (or here for the open-source version).
You would then have to roll your own class to use this, and use the class from within an orchestration – meaning you can only use XSLT 2.0 from within orchestrations, and not from receive/send ports.

XSLT 2.0 provides a much richer set of operations/functionality and would mean that there wouldn't be such a need to drop out to assemblies/inline code for more complex operations.

But I suspect that Microsoft is more likely to introduce either XQuery or LINQ to XML support in a future version of the BizTalk Mapper.

Understanding the BizTalk Mapper: Part 2 - Functoids Overview

This whole series of posts started because I wanted to show what XSLT was emitted when using the default functoids provided by Microsoft.
Specifically, I wanted to show the XSLT emitted by the Advanced Functoids. Understanding this XSLT can help in understanding how to use the functoids.
For some reason (as seems to happen with me) the post expanded into a whole series on the Mapper… every time I explain one thing, I seem to want to explain all the things that the first thing is based on… oops.

Anyway, suffice to say that the next 9 posts will cover the code emitted by all of the default functoids provided with BizTalk 2004 / 2006 / 2006 R2.

One thing to realise is that the majority of the default functoids emit inline C# code – which is odd as quite a lot of the functionality can be performed using pure XSLT.

So for each functoid I've shown:

1. Whether XSLT or C# is emitted
2. Whether an XSLT equivalent exists
3. The XSLT or C# emitted by the functoid
4. Where C# is emitted, the equivalent XSLT to achieve the same functionality (in both XSLT v1.0 and v2.0)

For the Advanced Functoids knowing what the functoids emit can be useful in understanding how to use them.

For the other functoids, knowing the equivalent XSLT to use is useful if you want a map which performs better as (generally speaking) using native XSLT will be faster than the equivalent C# for simple operations.

Some useful references when reading these sections are:
XSLT 1.0 Function reference
XSLT 2.0 Function reference
XSLT/XPath Operators

Understanding the BizTalk Mapper: Part 3 - String Functoids

The String Functoids are probably the most frequently used in maps (in my experience), mainly because they're the most familiar to a procedural programmer (i.e. a C# or VB programmer). However because they all emit inline C#, they perform the slowest so if you want your maps to run faster you're better off using the corresponding XSLT, or implementing the functionality you require in a separate assembly.

Functoids covered in this category:

String Functoids

Lowercase

Generates: C# **Has XSLT Equivalent**: in 2.0 only, can use *translate* in 1.0

Emitted Code:

```
public string StringLowerCase(string str)
{
    if (str == null)
    {
        return "";
    }
    return str.ToLower(System.Globalization.CultureInfo.InvariantCulture);
}
```

XSLT 1.0 Equivalent: translate(*string*, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ', 'abcdefghijklmnopqrstuvwxyz')

XSLT 2.0 Equivalent: lower-case(*string*)

Size

Generates: C# **Has XSLT Equivalent**: in 1.0 and 2.0

Emitted Code:

```
public int StringSize(string str)
{
    if (str == null)
    {
        return 0;
    }
    return str.Length;
}
```

XSLT 1.0 Equivalent: string-length(*string*)

XSLT 2.0 Equivalent: string-length(*string*)

String Concatenate

Generates: C# **Has XSLT Equivalent**: in 1.0 and 2.0

Emitted Code:

Note: there will be one overload per unique number of parameters.
Here we show an example with one input parameter, and three input parameters

```
public string StringConcat(string param0)
{
    return param0;
}

public string StringConcat(string param0, string param1, string param2)
{
    return param0 + param1 + param2;
}
```

XSLT 1.0 Equivalent: concat(*string*, *string*, ...)

XSLT 2.0 Equivalent: concat(*string*, *string*, ...)

String Extract

Generates: C# **Has XSLT Equivalent**: in 1.0 and 2.0

Emitted Code:

```
public string StringSubstring(string str, string left, string right)
{
    string retval = "";
    double dleft = 0;
    double dright = 0;
    if (str != null && IsNumeric(left, ref dleft) && IsNumeric(right, ref
dright))
    {
        int lt = (int)dleft;
        int rt = (int)dright;
        lt--; rt--;
        if (lt >= 0 && rt >= lt && lt < str.Length)
        {
            if (rt < str.Length)
            {
                retval = str.Substring(lt, rt - lt + 1);
            }
            else
            {
                retval = str.Substring(lt, str.Length - lt);
            }
        }
    }
    return retval;
}
```

XSLT 1.0 Equivalent: substring(*string*, *number*, *number*)

XSLT 2.0 Equivalent: substring(*string*, *number*, *number*)

Note: the *substring*() function takes a length as its last parameter, rather than the position used by the *String Extract* functoid. Additionally, there is an overload of *substring*() which takes two parameters, as well as additional *substring-before*() and *substring-after*() functions.

String Find

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public int StringFind(string str, string strFind)
{
    if (str == null || strFind == null || strFind == "")
    {
        return 0;
    }
    return (str.IndexOf(strFind) + 1);
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Note: The method is usually used as an input to the *String Extract* functoid – however, if using the XSLT *substring*() function then an index is not needed, so the fact that there is no XSLT equivalent should not cause any issues.

String Left

Generates: C# **Has XSLT Equivalent**: in 1.0 and 2.0

Emitted Code:

```
public string StringLeft(string str, string count)
{
    string retval = "";
    double d = 0;
    if (str != null && IsNumeric(count, ref d))
    {
        int i = (int)d;
        if (i > 0)
        {
            if (i <= str.Length)
            {
                retval = str.Substring(0, i);
            }
            else
            {
                retval = str;
            }
        }
    }
    return retval;
}
```

XSLT 1.0 Equivalent: substring(*string*, *number*)

XSLT 2.0 Equivalent: substring(*string*, *number*)

String Left Trim

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public string StringTrimLeft(string str)
{
    if (str == null)
    {
        return "";
    }
    return str.TrimStart(null);
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Note: the closest equivalent is *normalize-space*() which trims leading and trailing spaces, and also replaces any groups of spaces in a string with a single space.

String Right

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public string StringRight(string str, string count)
{
    string retval = "";
    double d = 0;
    if (str != null && IsNumeric(count, ref d))
    {
        int i = (int)d;
        if (i > 0)
        {
            if (i <= str.Length)
            {
                retval = str.Substring(str.Length - i);
            }
            else
            {
                retval = str;
            }
        }
    }
    return retval;
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)
Note: although there is no single XSLT function for this, the same result can be achieved through use of the *string-length*() and *substring*() functions.
e.g. substring(*string,* string-length(*string*) – *number*)

String Right Trim

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public string StringTrimRight(string str)
{
    if (str == null)
    {
        return "";
    }
    return str.TrimEnd(null);
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)
Note: the closest equivalent is *normalize-space*() which trims leading and trailing spaces, and also replaces any groups of spaces in a string with a single space.

Uppercase

Generates: C# **Has XSLT Equivalent**: in 2.0 Only, can use *translate* in 1.0

Emitted Code:

```
public string StringUpperCase(string str)
{
    if (str == null)
    {
        return "";
    }
    return str.ToUpper(System.Globalization.CultureInfo.InvariantCulture);
}
```

XSLT 1.0 Equivalent: translate(*string*, 'abcdefghijklmnopqrstuvwxyz', 'ABCDEFGHIJKLMNOPQRSTUVWXYZ')

XSLT 2.0 Equivalent: upper-case(*string*)

Common Code

(this is common code used by all the string functoids)

```
public bool IsNumeric(string val)
{
    if (val == null)
    {
        return false;
    }
    double d = 0;
    return Double.TryParse(val,
System.Globalization.NumberStyles.AllowThousands |
System.Globalization.NumberStyles.Float,
System.Globalization.CultureInfo.InvariantCulture, out d);
}

public bool IsNumeric(string val, ref double d)
{
    if (val == null)
    {
        return false;
    }
    return Double.TryParse(val,
System.Globalization.NumberStyles.AllowThousands |
System.Globalization.NumberStyles.Float,
System.Globalization.CultureInfo.InvariantCulture, out d);
```

```
        }
```

Understanding the BizTalk Mapper: Part 4 - Mathematical Functoids

Mathematics is not a strong point of XSLT. XSLT v1.0 has very poor mathematic support, whilst XSLT v2.0 has better support, but only by a small amount. Therefore most of the functoids in this category can only be implemented in C#.
So if you want to perform a complicated mathematical function (i.e. anything more than addition or subtraction!) you're better off using one of these functoids, or an external assembly.
Once again, inline C# isn't the fastest, but given a choice between a slow function and no function, you might not have a choice.

Functoids covered in this category:

Absolute Value	Modulo
Addition	Multiplication
Division	Round
Integer	Square Root
Maximum Value	Subtraction
Minimum Value	Common Code

Mathematical Functoids

Absolute Value

Generates: C# **Has XSLT Equivalent**: in 2.0 only

Emitted Code:

```
public string MathAbs(string val)
{
    string retval = "";
    double d = 0;
    if (IsNumeric(val, ref d))
    {
        double abs = Math.Abs(d);
        retval =
abs.ToString(System.Globalization.CultureInfo.InvariantCulture);
    }
    return retval;
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: abs(*number*)

Note: In XSLT 1.0, you can achieve the desired result through use of an XSLT template and the *xsl:when* statement (i.e. check if number is < 0).

Addition

Generates: C# **Has XSLT Equivalent**: in 1.0 and 2.0

Emitted Code:

Note: there will be one overload per unique number of parameters.
Here we show an example with two input parameters.

```
public string MathAdd(string param0, string param1)
{
    System.Collections.ArrayList listValues = new
System.Collections.ArrayList();
    listValues.Add(param0);
    listValues.Add(param1);
    double ret = 0;
    foreach (string obj in listValues)
    {
        double d = 0;
        if (IsNumeric(obj, ref d))
        {
            ret += d;
        }
        else
```

```
            {
                return "";
            }
        }
        return ret.ToString(System.Globalization.CultureInfo.InvariantCulture);
    }
```

XSLT 1.0 Equivalent: Use the "+" operator e.g. ((*number* + *number*) + *number*) or the sum(*number*, *number*, …) function

XSLT 2.0 Equivalent: Use the "+" operator e.g. ((*number* + *number*) + *number*) or the sum(*number*, *number*, …) function

Division

Generates: C# **Has XSLT Equivalent**: in 1.0 and 2.0

Emitted Code:

Note: there will be one overload per unique number of parameters.
Here we show an example with two input parameters.

```
public string MathDivide(string val1, string val2)
{
    string retval = "";
    double d1 = 0;
    double d2 = 0;
    if (IsNumeric(val1, ref d1) && IsNumeric(val2, ref d2))
    {
        if (d2 != 0)
        {
            double ret = d1 / d2;
            retval =
ret.ToString(System.Globalization.CultureInfo.InvariantCulture);
        }
    }
    return retval;
}
```

XSLT 1.0 Equivalent: Use the div operator e.g. *number* div *number*

XSLT 2.0 Equivalent: Use the div operator e.g. *number* div *number*

Integer

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public string MathInt(string val)
{
    string retval = "";
    double d = 0;
    if (IsNumeric(val, ref d))
    {
        try
        {
            int i = Convert.ToInt32(d,
System.Globalization.CultureInfo.InvariantCulture);
            if (i > d)
            {
                i = i - 1;
            }
            retval =
i.ToString(System.Globalization.CultureInfo.InvariantCulture);
        }
        catch (Exception)
        {
        }
    }
    return retval;
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Note: the *floor*() function will give an equivalent result e.g. floor(3.90) will give '3'.

Maximum Value

Generates: C# **Has XSLT Equivalent**: in 2.0 only

Emitted Code:

```
public string MathMax(string param0, string param1)
{
    double max = Double.NegativeInfinity;
    System.Collections.ArrayList listValues = new
System.Collections.ArrayList();
    listValues.Add(param0);
    listValues.Add(param1);
    foreach (string obj in listValues)
    {
        double d = 0;
        if (IsNumeric(obj, ref d))
        {
            max = (d >= max) ? d : max;
        }
        else
        {
            return "";
        }
    }
    if (Double.NegativeInfinity == max)
    {
        return "";
    }
    else
    {
        return
max.ToString(System.Globalization.CultureInfo.InvariantCulture);
    }
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: Use the max(*number*, *number*, …) function

Note: in XSLT 1.0 you could write a fairly complicated XPath statement to achieve this

Minimum Value

Generates: C# **Has XSLT Equivalent**: in 2.0 only

Emitted Code:

```
public string MathMin(string param0, string param1)
{
    double min = Double.PositiveInfinity;
    System.Collections.ArrayList listValues = new
System.Collections.ArrayList();
    listValues.Add(param0);
    listValues.Add(param1);
    foreach (string obj in listValues)
    {
        double d = 0;
        if (IsNumeric(obj, ref d))
        {
            min = (d < min) ? d : min;
        }
        else
        {
            return "";
        }
    }
    if (Double.PositiveInfinity == min)
    {
        return "";
    }
    else
    {
        return
min.ToString(System.Globalization.CultureInfo.InvariantCulture);
    }
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: Use the min(*number*, *number*, ...) function
Note: in XSLT 1.0 you could write a fairly complicated XPath statement to achieve this

Modulo

Generates: C# **Has XSLT Equivalent**: in 1.0 and 2.0

Emitted Code:

```
public string MathMod(string val, string denominator)
{
    string retval = "";
    double v = 0;
    double d = 0;
    if (IsNumeric(val, ref v) && IsNumeric(denominator, ref d))
    {
        if (d != 0)
        {
            retval = Convert.ToString(v % d,
System.Globalization.CultureInfo.InvariantCulture);
        }
    }
    return retval;
}
```

XSLT 1.0 Equivalent: Use the "mod" operator e.g. *number* mod *number*

XSLT 2.0 Equivalent: Use the "mod" operator e.g. *number* mod *number*

Multiplication

Generates: C# **Has XSLT Equivalent**: in 1.0 and 2.0

Emitted Code:

Note: there will be one overload per unique number of parameters.
Here we show an example with two input parameters.

```
public string MathMultiply(string param0, string param1)
{
    System.Collections.ArrayList listValues = new
System.Collections.ArrayList();
    listValues.Add(param0);
    listValues.Add(param1);
    double ret = 1;
    bool first = true;
    foreach (string obj in listValues)
    {
        double d = 0;
        if (IsNumeric(obj, ref d))
        {
            if (first)
            {
                first = false;
                ret = d;
            }
            else
            {
                ret *= d;
            }
        }
        else
        {
            return "";
        }
    }
    return ret.ToString(System.Globalization.CultureInfo.InvariantCulture);
}
```

XSLT 1.0 Equivalent: Use the "*" operator e.g. ((*number* * *number*) * *number*)

XSLT 2.0 Equivalent: Use the "*" operator e.g. ((*number* * *number*) * *number*)

Round

Generates: C# **Has XSLT Equivalent**: in 1.0 and 2.0

Emitted Code:

```
public string MathRound(string val)
{
    return MathRound(val, "0");
}

public string MathRound(string val, string decimals)
{
    string retval = "";
    double v = 0;
    double db = 0;
    if (IsNumeric(val, ref v) && IsNumeric(decimals, ref db))
    {
        try
        {
            int d = (int)db;
            double ret = Math.Round(v, d);
            retval =
ret.ToString(System.Globalization.CultureInfo.InvariantCulture);
        }
        catch (Exception)
        {
        }
    }
    return retval;
}
```

XSLT 1.0 Equivalent: round(*number*)

XSLT 2.0 Equivalent: round(*number*)

Note: the XSLT *round()* function does not take a second parameter – all number are rounded to 0 decimal places. You can achieve rounding to x dp's through use of the format-number(*number*, *string*) function:

e.g. format-number(3.55555, '#.00') – but be aware that rounding in this way might not always give the result you're looking for e.g. 9.2850 rounds to 9.28 using *format-number*() (although this can be coded around in XSLT).

Square Root

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public string MathSqrt(string val)
{
    string retval = "";
    double d = 0;
    if (IsNumeric(val, ref d))
    {
        if (d >= 0)
        {
                double ret = Math.Sqrt(d);
                retval =
ret.ToString(System.Globalization.CultureInfo.InvariantCulture);
            }
        }
    return retval;
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)
Note: A square-root can be calculated using an XSLT template – see here (http://bobcopeland.com/srcs/root_xsl.txt) for an example.

Subtraction

Generates: C# **Has XSLT Equivalent**: in 1.0 and 2.0

Emitted Code:
Note: there will be one overload per unique number of parameters.
Here we show an example with two input parameters.

```
public string MathSubtract(string param0, string param1)
{
    System.Collections.ArrayList listValues = new
System.Collections.ArrayList();
    listValues.Add(param0);
    listValues.Add(param1);
    double ret = 0;
    bool first = true;
    foreach (string obj in listValues)
    {
        if (first)
        {
            first = false;
            double d = 0;
            if (IsNumeric(obj, ref d))
            {
                ret = d;
            }
            else
            {
                return "";
            }
        }
        else
        {
            double d = 0;
            if (IsNumeric(obj, ref d))
            {
                ret -= d;
            }
            else
            {
                return "";
            }
        }
    }
    return ret.ToString(System.Globalization.CultureInfo.InvariantCulture);
}
```

XSLT 1.0 Equivalent: Use the "-" operator e.g. ((*number* – *number*) – *number*)

XSLT 2.0 Equivalent: Use the "-" operator e.g. ((*number* – *number*) – *number*)

Common Code

(this is common code used by all the mathematical functoids)

```
public bool IsNumeric(string val)
```

```
{
    if (val == null)
    {
        return false;
    }
    double d = 0;
    return Double.TryParse(val,
System.Globalization.NumberStyles.AllowThousands |
System.Globalization.NumberStyles.Float,
System.Globalization.CultureInfo.InvariantCulture, out d);
}

public bool IsNumeric(string val, ref double d)
{
    if (val == null)
    {
        return false;
    }
    return Double.TryParse(val,
System.Globalization.NumberStyles.AllowThousands |
System.Globalization.NumberStyles.Float,
System.Globalization.CultureInfo.InvariantCulture, out d);
}
```

Understanding the BizTalk Mapper: Part 5 - Logical Functoids

Whenever I've looked at the XSLT generated by a map I've always been confused by the amount of inline C# generated by these functoids.
After the String Functoids I'd say that these are the next most widely used and yet all but one of them has an XSLT v1.0 equivalent!
The code emitted for "Logical Equal" always makes me laugh – 12 lines of C# code can be replaced by... (wait for it)… one "=" symbol!

Functoids covered in this category:

Logical Functoids

Equal

Generates: C# **Has XSLT Equivalent**: in 1.0 and 2.0

Emitted Code:

```
public bool LogicalEq(string val1, string val2)
{
    bool ret = false;
    double d1 = 0;
    double d2 = 0;
    if (IsNumeric(val1, ref d1) && IsNumeric(val2, ref d2))
    {
        ret = d1 == d2;
    }
    else
    {
        ret = String.Compare(val1, val2, StringComparison.Ordinal) == 0;
    }
    return ret;
}
```

XSLT 1.0 Equivalent: Use the "=" operator e.g. *value* = *value*

XSLT 2.0 Equivalent: Use the "=" operator e.g. *value* = *value*

Note: Both the C# version and the XSLT version will perform a case-sensitive comparison.

Greater Than

Generates: C# **Has XSLT Equivalent**: in 1.0 and 2.0

Emitted Code:

```
public bool LogicalGt(string val1, string val2)
{
    bool ret = false;
    double d1 = 0;
    double d2 = 0;
    if (IsNumeric(val1, ref d1) && IsNumeric(val2, ref d2))
    {
        ret = d1 > d2;
    }
    else
    {
        ret = String.Compare(val1, val2, StringComparison.Ordinal) > 0;
    }
    return ret;
}
```

XSLT 1.0 Equivalent: Use the ">" operator e.g. *value* > *value*

XSLT 2.0 Equivalent: Use the ">" operator e.g. *value* > *value*

Greater Than or Equal To

Generates: C# **Has XSLT Equivalent**: in 1.0 and 2.0

Emitted Code:

```
public bool LogicalGte(string val1, string val2)
{
    bool ret = false;
    double d1 = 0;
    double d2 = 0;
    if (IsNumeric(val1, ref d1) && IsNumeric(val2, ref d2))
    {
        ret = d1 >= d2;
    }
    else
    {
        ret = String.Compare(val1, val2, StringComparison.Ordinal) >= 0;
    }
    return ret;
}
```

XSLT 1.0 Equivalent: Use the ">=" operator e.g. *value* >= *value*

XSLT 2.0 Equivalent: Use the ">=" operator e.g. *value* >= *value*

IsNil

Generates: XSLT **Has XSLT Equivalent**: N/A

Emitted Code:

string(*node*/@xsi:nil) = 'true'

Less Than

Generates: C# **Has XSLT Equivalent**: in 1.0 and 2.0

Emitted Code:

```
public bool LogicalLt(string val1, string val2)
{
    bool ret = false;
    double d1 = 0;
    double d2 = 0;
    if (IsNumeric(val1, ref d1) && IsNumeric(val2, ref d2))
    {
        ret = d1 < d2;
    }
    else
    {
        ret = String.Compare(val1, val2, StringComparison.Ordinal) < 0;
    }
    return ret;
}
```

XSLT 1.0 Equivalent: Use the "<" operator e.g. *value* < *value*

XSLT 2.0 Equivalent: Use the "<" operator e.g. *value* < *value*

Less Than or Equal To

Generates: C# **Has XSLT Equivalent**: in 1.0 and 2.0

Emitted Code:

```
public bool LogicalLte(string val1, string val2)
{
    bool ret = false;
    double d1 = 0;
    double d2 = 0;
    if (IsNumeric(val1, ref d1) && IsNumeric(val2, ref d2))
    {
        ret = d1 <= d2;
    }
    else
    {
        ret = String.Compare(val1, val2, StringComparison.Ordinal) <= 0;
    }
    return ret;
}
```

XSLT 1.0 Equivalent: Use the "<=" operator e.g. *value* <= *value*

XSLT 2.0 Equivalent: Use the "<=" operator e.g. *value* <= *value*

Logical AND

Generates: C# **Has XSLT Equivalent**: in 1.0 and 2.0

Emitted Code:

```
public bool LogicalAnd(string param0, string param1)
{
    return ValToBool(param0) && ValToBool(param1);
    return false;
}
```

XSLT 1.0 Equivalent: Use the "and" operator e.g. *expression* and *expression*

XSLT 2.0 Equivalent: Use the "and" operator e.g. *expression* and *expression*

Logical Date

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public bool LogicalIsDate(string val)
{
    return IsDate(val);
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Note: an XPath expression could be created using the dateTime function, but would require parsing out separate date/time portions. There are other implementations available on the internet, but none of them are trivial.
XPath 1.0 has no native DateTime functionality built in.

Logical Existence

Generates: C# **Has XSLT Equivalent**: in 1.0 and 2.0

Emitted Code:

```
public bool LogicalExistence(bool val)
{
    return val;
}
```

Note: The above method may seem a bit strange, but it makes sense if you understand that you test for existence of a node in XSLT by using the *boolean()* function, and passing a node (or any sort of value).
The XSLT generated by the *Logical Existence* functoid wraps the *boolean()* function round the value passed to the *LogicalExistence* method like this: **userC#:LogicalExistence(boolean(@name))** (where @name was an attribute that we connected to the *Logical Existence* functoid).

XSLT 1.0 Equivalent: boolean(*node*)

XSLT 2.0 Equivalent: boolean(*node*)

Logical NOT

Generates: C# **Has XSLT Equivalent**: in 1.0 and 2.0

Emitted Code:

```
public bool LogicalNot(string val)
{
    return !ValToBool(val);
}
```

XSLT 1.0 Equivalent: not(*expression*)

XSLT 2.0 Equivalent: not(*expression*)

Logical Numeric

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public bool LogicalIsNumeric(string val)
{
    return IsNumeric(val);
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Note: You can achieve this through use of a statement such as: string(number(*value*)) != 'NaN'
Or in XPath 2.0 using a statement such as: *value* castable as xs:decimal

Logical OR

Generates: C# **Has XSLT Equivalent**: in 1.0 and 2.0

Emitted Code:

Note: there will be one overload per unique number of parameters.
Here we show an example with two input parameters.

```
public bool LogicalOr(string param0, string param1)
{
    return ValToBool(param0) || ValToBool(param1);
    return false;
}
```

XSLT 1.0 Equivalent: Use the "or" operator e.g. ((*expression* or *expression*) or *expression*)

XSLT 2.0 Equivalent: Use the "or" operator e.g. ((*expression* or *expression*) or *expression*)

Logical String

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public bool LogicalIsString(string val)
{
    return (val != null && val != "");
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Note: the same thing can be achieved with an XPath expression.

Not Equal

Generates: C# **Has XSLT Equivalent**: in 1.0 and 2.0

Emitted Code:

```
public bool LogicalNe(string val1, string val2)
{
    bool ret = false;
    double d1 = 0;
    double d2 = 0;
    if (IsNumeric(val1, ref d1) && IsNumeric(val2, ref d2))
    {
        ret = d1 != d2;
    }
    else
    {
        ret = String.Compare(val1, val2, StringComparison.Ordinal) != 0;
    }
    return ret;
}
```

XSLT 1.0 Equivalent: Use the "!=" operator e.g. (*value* != *value*)

XSLT 2.0 Equivalent: Use the "!=" operator e.g. (*value* != *value*)

Common Code
(this is common code used by all the logical functoids)

```
public bool IsNumeric(string val)
{
    if (val == null)
    {
        return false;
    }
    double d = 0;
    return Double.TryParse(val, 
System.Globalization.NumberStyles.AllowThousands | 
System.Globalization.NumberStyles.Float, 
System.Globalization.CultureInfo.InvariantCulture, out d);
}

public bool IsNumeric(string val, ref double d)
{
    if (val == null)
    {
        return false;
    }
    return Double.TryParse(val, 
System.Globalization.NumberStyles.AllowThousands | 
System.Globalization.NumberStyles.Float, 
System.Globalization.CultureInfo.InvariantCulture, out d);
}

public bool IsDate(string val)
{
    bool retval = true;
    try
    {
        DateTime dt = Convert.ToDateTime(val, 
System.Globalization.CultureInfo.InvariantCulture);
    }
    catch (Exception)
    {
        retval = false;
    }
    return retval;
}

public bool ValToBool(string val)
{
    if (val != null)
    {
        if (string.Compare(val, bool.TrueString, 
StringComparison.OrdinalIgnoreCase) == 0)
        {
            return true;
        }
        if (string.Compare(val, bool.FalseString, 
StringComparison.OrdinalIgnoreCase) == 0)
        {
            return false;
        }
        val = val.Trim();
        if (string.Compare(val, bool.TrueString, 
StringComparison.OrdinalIgnoreCase) == 0)
        {
            return true;
        }
        if (string.Compare(val, bool.FalseString,
StringComparison.OrdinalIgnoreCase) == 0)
        {
            return false;
        }
        double d = 0;
        if (IsNumeric(val, ref d))
        {
            return (d > 0);
        }
    }
    return false;
}
```

Understanding the BizTalk Mapper: Part 6 - Date/Time Functoids

XSLT v1.0 has no support for Date/Time values, whilst XSLT v2.0 has full support.
Therefore it's not surprising that your only option is to use C#'s rich support for Date/Time values.
And this is why all of the functoids in this category emit inline C#.

Functoids covered in this category:

Date/Time Functoids

Note: XSLT 1.0 has no built-in Date/Time functions, whereas XSLT 2.0/XPath 2.0 does.

Add Days

Generates: C# **Has XSLT Equivalent**: in 2.0 only

Emitted Code:

```
public string DateAddDays(string date, string days)
{
    string retval = "";
    double db = 0;
    if (IsDate(date) && IsNumeric(days, ref db))
    {
        DateTime dt = DateTime.Parse(date);
        int d = (int)db;
        dt = dt.AddDays(d);
        retval = dt.ToString("yyyy-MM-dd",
System.Globalization.CultureInfo.InvariantCulture);
    }
    return retval;
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: xs:dateTime('2007-12-12') + xdt:dayTimeDuration('PxD')
Where x (in dayTimeDuration()) is number of days to add e.g. 5 days would be 'P5D'

Date

Generates: C# **Has XSLT Equivalent**: in 2.0 only

Emitted Code:

```
public string DateCurrentDate()
{
    DateTime dt = DateTime.Now;
    return dt.ToString("yyyy-MM-dd",
System.Globalization.CultureInfo.InvariantCulture);
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: fn:current-date()

Date and Time

Generates: C# **Has XSLT Equivalent**: in 2.0 only

Emitted Code:

```
public string DateCurrentDateTime()
{
    DateTime dt = DateTime.Now;
    string curdate = dt.ToString("yyyy-MM-dd",
System.Globalization.CultureInfo.InvariantCulture);
    string curtime = dt.ToString("T",
System.Globalization.CultureInfo.InvariantCulture);
    string retval = curdate + "T" + curtime;
    return retval;
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: fn:current-dateTime()

Time

Generates: C# **Has XSLT Equivalent**: in 2.0 only

Emitted Code:

```
public string DateCurrentTime()
{
    DateTime dt = DateTime.Now;
    return dt.ToString("T",
System.Globalization.CultureInfo.InvariantCulture);
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: fn:current-date()

Common Code

(this is common code used by all the date/time functoids)

```
public bool IsNumeric(string val)
{
    if (val == null)
    {
        return false;
    }
    double d = 0;
    return Double.TryParse(val,
System.Globalization.NumberStyles.AllowThousands |
System.Globalization.NumberStyles.Float,
System.Globalization.CultureInfo.InvariantCulture, out d);
}

public bool IsNumeric(string val, ref double d)
{
    if (val == null)
    {
        return false;
    }
    return Double.TryParse(val,
System.Globalization.NumberStyles.AllowThousands |
System.Globalization.NumberStyles.Float,
System.Globalization.CultureInfo.InvariantCulture, out d);
}

public bool IsDate(string val)
{
    bool retval = true;
    try
    {
        DateTime dt = Convert.ToDateTime(val,
System.Globalization.CultureInfo.InvariantCulture);
    }
    catch (Exception)
    {
        retval = false;
    }
    return retval;
}
```

Understanding the BizTalk Mapper: Part 7 - Conversion Functoids

Surprisingly, neither XSLT v1.0 nor XSLT v2.0 have any built-in conversion support (well, not for the scenarios represented in this category anyway).
It is possible to download XSLT libraries which can do this sort of conversion (as mentioned in the notes below each functoid), but the XSLT is not pretty, and I'm not convinced about performance.
So C# is generally your only option here.

Functoids covered in this category:

ASCII to Character	Octal
Character to ASCII	Common Code
Hexadecimal	

Conversion Functoids

ASCII to Character

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public string ConvertChr(string val)
{
    string retval = "";
    double d = 0;
    if (IsNumeric(val, ref d))
    {
        int v = (int)d;
        if (v >= 1 && v <= 127)
        {
            char c = (char)v;
            retval =
c.ToString(System.Globalization.CultureInfo.InvariantCulture);
        }
    }
    return retval;
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Note: There are some inelegant hacks you can use (e.g. listing all the ASCII chars in a variable and selecting by index, or using the translate() function with a list of possible values) but there is no built-in support for this conversion.

Character to ASCII

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public string ConvertAsc(string val)
{
    if (val == null || val == "")
    {
        return "";
    }
    else
    {
        char c = val[0];
        int x = c;
        return x.ToString(System.Globalization.CultureInfo.InvariantCulture);
    }
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Note: There are some inelegant hacks you can use (e.g. listing all the ASCII chars in a variable and selecting by index, or using the translate() function with a list of possible values) but there is no built-in support for this conversion.

Hexadecimal

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public string ConvertHex(string val)
{
    string retval = "";
    double d = 0;
    if (IsNumeric(val, ref d))
    {
        int v = (int)d;
        retval = Convert.ToString(v,
16).ToUpper(System.Globalization.CultureInfo.InvariantCulture);
    }
    return retval;
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Note: There are some inelegant templates you can use to perform this conversion, but they are all quite long and involve string manipulation.

Octal

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public string ConvertOct(string val)
{
    string retval = "";
    double d = 0;
    if (IsNumeric(val, ref d))
    {
        int v = (int)d;
        retval = Convert.ToString(v, 8);
    }
    return retval;
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Note: There are some inelegant templates you can use to perform this conversion, but they are all quite long and involve string manipulation.

Common Code

(this is common code used by all the conversion functoids)

```
public bool IsNumeric(string val)
{
    if (val == null)
    {
        return false;
    }
    double d = 0;
    return Double.TryParse(val,
System.Globalization.NumberStyles.AllowThousands |
System.Globalization.NumberStyles.Float,
System.Globalization.CultureInfo.InvariantCulture, out d);
}

public bool IsNumeric(string val, ref double d)
{
    if (val == null)
    {
        return false;
    }
    return Double.TryParse(val,
System.Globalization.NumberStyles.AllowThousands |
System.Globalization.NumberStyles.Float,
System.Globalization.CultureInfo.InvariantCulture, out d);
}
```

Understanding the BizTalk Mapper: Part 8 - Scientific Functoids

Yet another category which has no direct support in XSLT v1.0 or XSLT v2.0!
However, given the strong support for scientific functions in .NET, it's easy to call out to .NET classes, which is exactly what every single one of the functoids in this category does.

Having said that: have you ever used one of these functoids in a map? Care to share a real world example?
I'd be interested to find out how often they are used.

Functoids covered in this category:

10^n	Natural Logarithm
Arc Tangent	Sine
Base-Specified Logarithm	Tangent
Common Logarithm	X^Y
Cosine	Common Code
Natural Exponential Function	

Scientific Functoids

XSLT has no intrinsic trigonometric functions, although you can download open source libraries such as FXSL (http://fxsl.sourceforge.net/articles/xslCalculator/The%20FXSL%20Calculator.html)

10^n

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public string SciExp10(string val)
{
    string retval = "";
    double dval = 0;
    if (IsNumeric(val, ref dval))
    {
        double ret = Math.Pow(10.0, dval);
        retval =
ret.ToString(System.Globalization.CultureInfo.InvariantCulture);
    }
    return retval;
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Arc Tangent

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public string SciArcTan(string val)
{
    string retval = "";
    double dval = 0;
    if (IsNumeric(val, ref dval))
    {
        double ret = Math.Atan(dval);
        retval =
ret.ToString(System.Globalization.CultureInfo.InvariantCulture);
    }
    return retval;
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Base-Specified Logarithm

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public string SciLogn(string val1, string val2)
{
    string retval = "";
    double dval1 = 0;
    double dval2 = 0;
    if (IsNumeric(val1, ref dval1) && IsNumeric(val2, ref dval2))
    {
        if (dval1 > 0 && dval2 > 0)
        {
            double denom = Math.Log(dval2);
            if (denom != 0)
            {
                double ret = Math.Log(dval1) / denom;
                retval =
ret.ToString(System.Globalization.CultureInfo.InvariantCulture);
            }
        }
    }
    return retval;
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Common Logarithm

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public string SciLog10(string val)
{
    string retval = "";
    double dval = 0;
    if (IsNumeric(val, ref dval))
    {
        if (dval > 0)
        {
            double ret = Math.Log10(dval);
            retval =
ret.ToString(System.Globalization.CultureInfo.InvariantCulture);
        }
    }
    return retval;
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Cosine

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public string SciCos(string val)
{
    string retval = "";
    double dval = 0;
    if (IsNumeric(val, ref dval))
    {
        double ret = Math.Cos(dval);
        retval =
ret.ToString(System.Globalization.CultureInfo.InvariantCulture);
    }
    return retval;
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Natural Exponential Function

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public string SciExp(string val)
{
    string retval = "";
    double dval = 0;
    if (IsNumeric(val, ref dval))
    {
        double ret = Math.Exp(dval);
        retval =
ret.ToString(System.Globalization.CultureInfo.InvariantCulture);
    }
    return retval;
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Natural Logarithm

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public string SciLog(string val)
{
    string retval = "";
    double dval = 0;
    if (IsNumeric(val, ref dval))
    {
        if (dval > 0)
        {
            double ret = Math.Log(dval);
            retval =
ret.ToString(System.Globalization.CultureInfo.InvariantCulture);
        }
    }
    return retval;
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Sine

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public string SciSin(string val)
{
    string retval = "";
    double dval = 0;
    if (IsNumeric(val, ref dval))
    {
        double ret = Math.Sin(dval);
        retval =
ret.ToString(System.Globalization.CultureInfo.InvariantCulture);
    }
    return retval;
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Tangent

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public string SciTan(string val)
{
    string retval = "";
    double dval = 0;
    if (IsNumeric(val, ref dval))
    {
        double ret = Math.Tan(dval);
        retval = 
ret.ToString(System.Globalization.CultureInfo.InvariantCulture);
    }
    return retval;
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

X^Y

Generates: C# **Has XSLT Equivalent**: No

Emitted Code:

```
public string SciPow(string val1, string val2)
{
    string retval = "";
    double dval1 = 0;
    double dval2 = 0;
    if (IsNumeric(val1, ref dval1) && IsNumeric(val2, ref dval2))
    {
        double ret = Math.Pow(dval1, dval2);
        retval = 
ret.ToString(System.Globalization.CultureInfo.InvariantCulture);
    }
    return retval;
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Common Code

(this is common code used by all the scientific functoids)

```
public bool IsNumeric(string val)
{
    if (val == null)
    {
        return false;
    }
    double d = 0;
    return Double.TryParse(val, 
System.Globalization.NumberStyles.AllowThousands | 
System.Globalization.NumberStyles.Float, 
System.Globalization.CultureInfo.InvariantCulture, out d);
}

public bool IsNumeric(string val, ref double d)
{
    if (val == null)
    {
        return false;
    }
    return Double.TryParse(val, 
System.Globalization.NumberStyles.AllowThousands | 
System.Globalization.NumberStyles.Float,
System.Globalization.CultureInfo.InvariantCulture, out d);
}
```

Understanding the BizTalk Mapper: Part 9 - Cumulative Functoids

Of the functoids in this category, only *Cumulative Sum* has a counterpart in XSLT v1.0 – all the others can be performed in XSLT v2.0, but not XSLT v1.0.

Cumulative Functoids

Note: All of the cumulative functoids work in the same way.
They each emit three calls to each of the three C# methods.
For example, the *Cumulative Concatenate* functoid emits the following:

```
<xsl:variable name="var:v1" select="userC#:InitCumulativeConcat(0)" />
<xsl:for-each select="/ns0:Employees/Employee">
  <xsl:variable name="var:v2" select="userC#:AddToCumulativeConcat(0,string(@name),"1000")" />
</xsl:for-each>
<xsl:variable name="var:v3" select="userC#:GetCumulativeConcat(0)" />
<xsl:attribute name="name">
  <xsl:value-of select="$var:v3" />
</xsl:attribute>
```

There's an initialise call, followed by a loop over a given element/attribute (which a call to the Add method) and then a final call to get the cumulative result.

The first parameter to each of the methods (0 in the above example) is the index of this type of functoid in the map e.g. the second occurrence of the *Cumulative Concatenate* functoid would use 1 etc.

Cumulative Average

Generates: C# **Has XSLT Equivalent**: in 2.0 only

Emitted Code:

```
public string InitCumulativeAvg(int index)
{
    if (index >= 0)
    {
        if (index >= myCumulativeAvgArray.Count)
        {
            int i = myCumulativeAvgArray.Count;
            for (; i <= index; i++)
            {
                myCumulativeAvgArray.Add("");
                myCumulativeAvgCountArray.Add(0);
            }
        }
        else
        {
            myCumulativeAvgArray[index] = "";
            myCumulativeAvgCountArray[index] = 0;
        }
    }
    return "";
}

private System.Collections.ArrayList myCumulativeAvgArray = new
System.Collections.ArrayList();
private System.Collections.ArrayList myCumulativeAvgCountArray = new
System.Collections.ArrayList();

public string AddToCumulativeAvg(int index, string val, string notused)
{
    if (index < 0 || index >= myCumulativeAvgArray.Count)
    {
        return "";
    }
    double d = 0;
    if (IsNumeric(val, ref d))
    {
        if (myCumulativeAvgArray[index] == "")
        {
            myCumulativeAvgArray[index] = d;
        }
        else
        {
            myCumulativeAvgArray[index] =
(double)(myCumulativeAvgArray[index]) + d;
        }
    }
    myCumulativeAvgCountArray[index] = (int)(myCumulativeAvgCountArray[index])
+ 1;
    return (myCumulativeAvgArray[index] is double) ?
((double)myCumulativeAvgArray[index]).ToString(System.Globalization.CultureInf
o.InvariantCulture) : "";
}

public string GetCumulativeAvg(int index)
{
    if (index < 0 || index >= myCumulativeAvgArray.Count)
    {
        return "";
    }
    if ((int)(myCumulativeAvgCountArray[index]) == 0 ||
myCumulativeAvgArray[index] == "")
    {
        return "";
    }
    else
    {
        double numer = (double)(myCumulativeAvgArray[index]);
        double denom =
Convert.ToDouble((int)(myCumulativeAvgCountArray[index]),
System.Globalization.CultureInfo.InvariantCulture);
        double d = numer / denom;
        return d.ToString(System.Globalization.CultureInfo.InvariantCulture);
    }
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: avg(node-set) or avg(*value*, *value*, *value*, …) e.g. `avg(/*[local-name()='number' and namespace-uri()=''])`

Note: In XSLT 1.0 you can use the Sum() and Count() functions along with the div operator to calculate an average:

```
sum(/*[local-name()='number' and namespace-uri()='']) div count(/*[local-
name()='number' and namespace-uri()=''])
```

Cumulative Concatenate

Generates: C# **Has XSLT Equivalent**: in 2.0 only

Emitted Code:

```
public string InitCumulativeConcat(int index)
{
    if (index >= 0)
    {
        if (index >= myCumulativeConcatArray.Count)
        {
            int i = myCumulativeConcatArray.Count;
            for (; i <= index; i++)
            {
                myCumulativeConcatArray.Add("");
            }
        }
        else
        {
            myCumulativeConcatArray[index] = "";
        }
    }
    return "";
}

public System.Collections.ArrayList myCumulativeConcatArray = new
System.Collections.ArrayList();

public string AddToCumulativeConcat(int index, string val, string notused)
{
    if (index < 0 || index >= myCumulativeConcatArray.Count)
    {
        return "";
    }
    myCumulativeConcatArray[index] = (string)(myCumulativeConcatArray[index])
+ val;
    return myCumulativeConcatArray[index].ToString();
}

public string GetCumulativeConcat(int index)
{
    if (index < 0 || index >= myCumulativeConcatArray.Count)
    {
        return "";
    }
    return myCumulativeConcatArray[index].ToString();
}
```

XSLT 1.0 Equivalent: concat(*string, string, string, …*)

XSLT 2.0 Equivalent: concat(*string, string, string, …*)

Cumulative Maximum

Generates: C# **Has XSLT Equivalent**: in 2.0 only

Emitted Code:

```
public string InitCumulativeMax(int index)
{
    if (index >= 0)
    {
        if (index >= myCumulativeMaxArray.Count)
        {
            int i = myCumulativeMaxArray.Count;
            for (; i <= index; i++)
            {
                myCumulativeMaxArray.Add("");
            }
        }
        else
        {
            myCumulativeMaxArray[index] = "";
        }
    }
    return "";
}
```

```
public System.Collections.ArrayList myCumulativeMaxArray = new
System.Collections.ArrayList();

public string AddToCumulativeMax(int index, string val, string notused)
{
    if (index < 0 || index >= myCumulativeMaxArray.Count)
    {
        return "";
    }
    double d = 0;
    if (IsNumeric(val, ref d))
    {
        if (myCumulativeMaxArray[index] is string || (d >
(double)(myCumulativeMaxArray[index])))
        {
            myCumulativeMaxArray[index] = d;
        }
    }
    return (myCumulativeMaxArray[index] is double) ?
((double)myCumulativeMaxArray[index]).ToString(System.Globalization.CultureInf
o.InvariantCulture) : "";
}

public string GetCumulativeMax(int index)
{
    if (index < 0 || index >= myCumulativeMaxArray.Count)
    {
        return "";
    }
    return (myCumulativeMaxArray[index] is double) ?
((double)myCumulativeMaxArray[index]).ToString(System.Globalization.CultureInf
o.InvariantCulture) : "";
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: max(node-set) or max(*value*, *value*, *value*, …)

Note: in XSLT 1.0 you can achieve the same result through use of XPath:
e.g. `Number[not(parent::Numbers/Number &gt; .)]` where <Numbers> and <Number> are the parent and child nodes respectively.

Cumulative Minimum

Generates: C# **Has XSLT Equivalent**: in 2.0 only

Emitted Code:

```
public string InitCumulativeMin(int index)
{
    if (index >= 0)
    {
        if (index >= myCumulativeMinArray.Count)
        {
            int i = myCumulativeMinArray.Count;
            for (; i <= index; i++)
            {
                myCumulativeMinArray.Add("");
            }
        }
        else
        {
            myCumulativeMinArray[index] = "";
        }
    }
    return "";
}
```

```
public System.Collections.ArrayList myCumulativeMinArray = new
System.Collections.ArrayList();

public string AddToCumulativeMin(int index, string val, string notused)
{
    if (index < 0 || index >= myCumulativeMinArray.Count)
    {
        return "";
    }
    double d = 0;
    if (IsNumeric(val, ref d))
    {
        if (myCumulativeMinArray[index] is string || (d <
(double)(myCumulativeMinArray[index])))
        {
            myCumulativeMinArray[index] = d;
        }
    }
    return (myCumulativeMinArray[index] is double) ?
((double)myCumulativeMinArray[index]).ToString(System.Globalization.CultureInf
o.InvariantCulture) : "";
}

public string GetCumulativeMin(int index)
{
    if (index < 0 || index >= myCumulativeMinArray.Count)
    {
        return "";
    }
    return (myCumulativeMinArray[index] is double) ?
((double)myCumulativeMinArray[index]).ToString(System.Globalization.CultureInf
o.InvariantCulture) : "";
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: min(node-set) or min(*value*, *value*, *value*)

Note: in XSLT 1.0 you can achieve the same result through use of XPath:
e.g. `Number[not(parent::Numbers/Number &lt; .)]` where <Numbers> and <Number> are the parent and child nodes respectively.

Cumulative Sum

Generates: C# **Has XSLT Equivalent**: in 1.0 and 2.0

Emitted Code:

```
public string InitCumulativeSum(int index)
{
    if (index >= 0)
    {
        if (index >= myCumulativeSumArray.Count)
        {
            int i = myCumulativeSumArray.Count;
            for (; i <= index; i++)
            {
                myCumulativeSumArray.Add("");
            }
        }
        else
        {
            myCumulativeSumArray[index] = "";
        }
    }
    return "";
}
```

```
public System.Collections.ArrayList myCumulativeSumArray = new
System.Collections.ArrayList();

public string AddToCumulativeSum(int index, string val, string notused)
{
    if (index < 0 || index >= myCumulativeSumArray.Count)
    {
        return "";
    }
    double d = 0;
    if (IsNumeric(val, ref d))
    {
        if (myCumulativeSumArray[index] == "")
        {
            myCumulativeSumArray[index] = d;
        }
        else
        {
            myCumulativeSumArray[index] =
(double)(myCumulativeSumArray[index]) + d;
        }
    }
    return (myCumulativeSumArray[index] is double) ?
((double)myCumulativeSumArray[index]).ToString(System.Globalization.CultureInf
o.InvariantCulture) : "";
}

public string GetCumulativeSum(int index)
{
    if (index < 0 || index >= myCumulativeSumArray.Count)
    {
        return "";
    }
    return (myCumulativeSumArray[index] is double) ?
((double)myCumulativeSumArray[index]).ToString(System.Globalization.CultureInf
o.InvariantCulture) : "";
}
```

XSLT 1.0 Equivalent: sum(node-set) or sum(*value*, *value*, *value*, …)

XSLT 2.0 Equivalent: sum(node-set) or sum(*value*, *value*, *value*, …)

Common Code
(this is common code used by all the cumulative functoids)

```
public bool IsNumeric(string val)
{
    if (val == null)
    {
        return false;
    }
    double d = 0;
    return Double.TryParse(val,
System.Globalization.NumberStyles.AllowThousands |
System.Globalization.NumberStyles.Float,
System.Globalization.CultureInfo.InvariantCulture, out d);
}

public bool IsNumeric(string val, ref double d)
{
    if (val == null)
    {
        return false;
    }
    return Double.TryParse(val,
System.Globalization.NumberStyles.AllowThousands |
System.Globalization.NumberStyles.Float,
System.Globalization.CultureInfo.InvariantCulture, out d);
}
```

Understanding the BizTalk Mapper: Part 10 - Database Functoids

This category contains both Database and Cross Referencing Functoids – but they all connect to a database to retrieve/update data.
Unlike all other default functoids, these functoids all call classes/methods in external assemblies – no inline C# is emitted at all. Because of this, this is the only category that emits an *ExtensionObjects* file listing the strong names of the external assemblies used.

Note: in this category I show some of the source code from the external assemblies as well.

Functoids covered in this category:

Database Lookup	Get Common Value
Error Return	Remove Application ID
Format Message	Set Common ID
Get Application ID	Value Extractor
Get Application Value	Common Code
Get Common ID	

Database Functoids

Note: using any of the Database Functoids will cause the following to be emitted as an Extension Object xml file. This is used to tell the XSLT compiler in which assembly it can find the database/cross-referencing lookup methods. All of these functoids use classes in one of two external assemblies. For the database functoids the source code of the class/method is shown.

```
<ExtensionObject
Namespace="http://schemas.microsoft.com/BizTalk/2003/ScriptNS0"
AssemblyName="Microsoft.BizTalk.BaseFunctoids, Version=3.0.1.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35"
ClassName="Microsoft.BizTalk.BaseFunctoids.FunctoidScripts" />
<ExtensionObject
Namespace="http://schemas.microsoft.com/BizTalk/2003/ScriptNS1"
AssemblyName="Microsoft.BizTalk.CrossReferencing, Version=3.0.1.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35"
ClassName="Microsoft.BizTalk.CrossReferencing.CrossReferencing" />
```

Database Lookup

Generates: XSLT calling a DLL **Has XSLT Equivalent**: No

Emitted Code:

```
<xsl:variable name="var:v1" select="ScriptNS0:DBLookup(0 , string(@name) ,
string(@id) , "" , "")" />

public string DBLookup(int index, string value, string connectionString,
string table, string column)
{
    DBFunctoidHelper helper = null;
    bool flag = false;
    InitDBFunctoidHelperList();
    if (!myDBFunctoidHelperList.Contains(index))
    {
        helper = new DBFunctoidHelper();
        myDBFunctoidHelperList.Add(index, helper);
    }
    else
    {
        helper = (DBFunctoidHelper) myDBFunctoidHelperList[index];
    }
    try
    {
        if (((helper.ConnectionString == null) || ((helper.ConnectionString !=
null) && (string.Compare(helper.ConnectionString, connectionString,
StringComparison.Ordinal) != 0))) || (helper.Connection.State !=
ConnectionState.Open))
        {
            flag = true;
            helper.MapValues.Clear();
            helper.Error = "";
            if (helper.Connection.State == ConnectionState.Open)
            {
                helper.Connection.Close();
            }
            helper.ConnectionString = connectionString;
            helper.Connection.ConnectionString = connectionString;
            helper.Connection.Open();
        }
        if ((flag || (string.Compare(helper.Table, table,
StringComparison.Ordinal) != 0)) || (((string.Compare(helper.Column, column,
StringComparison.OrdinalIgnoreCase) != 0) || (string.Compare(helper.Value,
value, StringComparison.Ordinal) != 0)) || ((helper.Error != null) &&
(helper.Error.Length > 0))))
        {
            helper.Table = table;
            helper.Column = column;
            helper.Value = value;
            helper.MapValues.Clear();
            helper.Error = "";
            OleDbCommand command = new OleDbCommand("SELECT * FROM " + table +
" WHERE " + column + "= ?", helper.Connection);
            OleDbParameter parameter = new OleDbParameter();
            parameter.Value = value;
            command.Parameters.Add(parameter);
            IDataReader reader = command.ExecuteReader();
            if (reader.Read())
            {
                for (int i = 0; i < reader.FieldCount; i++)
                {
                    string str =
reader.GetName(i).ToLower(CultureInfo.InvariantCulture);
                    object obj2 = reader.GetValue(i);
                    helper.MapValues[str] = obj2;
                }
            }
            reader.Close();
        }
    }
    catch (OleDbException exception)
    {
        if (exception.Errors.Count > 0)
        {
            helper.Error = exception.Errors[0].Message;
        }
    }
    catch (Exception exception2)
    {
        helper.Error = exception2.Message;
    }
    finally
    {
        if (helper.Connection.State == ConnectionState.Open)
        {
            helper.Connection.Close();
        }
    }
    return index.ToString(CultureInfo.InvariantCulture);
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Error Return

Generates: XSLT calling a DLL **Has XSLT Equivalent**: No

Emitted Code:

```
<xsl:variable name="var:v2" select="ScriptNS0:DBErrorExtract(string($var:v1))" />

public string DBErrorExtract(int index)
{
    string error = "";
    InitDBFunctoidHelperList();
    try
    {
        if (myDBFunctoidHelperList.Contains(index))
        {
            DBFunctoidHelper helper = (DBFunctoidHelper)
myDBFunctoidHelperList[index];
            if (helper != null)
            {
                error = helper.Error;
            }
        }
    }
    catch (Exception)
    {
    }
    if (error == null)
    {
        error = "";
    }
    return error;
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Format Message

Generates: XSLT calling a DLL **Has XSLT Equivalent**: No

Emitted Code:

```
<xsl:variable name="var:v9" select="ScriptNS1:FormatMessage("" , "")" />
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Get Application ID

Generates: XSLT calling a DLL **Has XSLT Equivalent**: No

Emitted Code:

```
<xsl:variable name="var:v5" select="ScriptNS1:GetAppID("" , "" , "")" />
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Get Application Value

Generates: XSLT calling a DLL **Has XSLT Equivalent**: No

Emitted Code:

```
<xsl:variable name="var:v4" select="ScriptNS1:GetAppValue("" , "" , "")" />
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Get Common ID

Generates: XSLT calling a DLL **Has XSLT Equivalent**: No

Emitted Code:
`<xsl:variable name="var:v8" select="ScriptNS1:GetCommonID("" , "" , "")" />`

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Get Common Value

Generates: XSLT calling a DLL **Has XSLT Equivalent**: No

Emitted Code:
`<xsl:variable name="var:v3" select="ScriptNS1:GetCommonValue("" , "" , "")" />`

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Remove Application ID

Generates: XSLT calling a DLL **Has XSLT Equivalent**: No

Emitted Code:
`<xsl:variable name="var:v6" select="ScriptNS1:RemoveAppID("" , "" , "")" />`

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Set Common ID

Generates: XSLT calling a DLL **Has XSLT Equivalent**: No

Emitted Code:
`<xsl:variable name="var:v7" select="ScriptNS1:SetCommonID("" , "" , "")" />`

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Value Extractor

Generates: XSLT calling a DLL **Has XSLT Equivalent**: No

Emitted Code:

```
xsl:variable name="var:v2" select="ScriptNS0:DBValueExtract(string($var:v1) ,
string(Field/text()))" />

public string DBValueExtract(int index, string columnName)
{
    string str = "";
    InitDBFunctoidHelperList();
    if (myDBFunctoidHelperList.Contains(index))
    {
        DBFunctoidHelper helper = (DBFunctoidHelper)
myDBFunctoidHelperList[index];
        columnName = columnName.ToLower(CultureInfo.InvariantCulture);
        object obj2 = helper.MapValues[columnName];
        if (obj2 != null)
        {
            str = obj2.ToString();
        }
    }
    return str;
}
```

XSLT 1.0 Equivalent: (none)

XSLT 2.0 Equivalent: (none)

Common Code

(this is common code used by all the database functoids)

```
<xsl:variable name="var:v3" select="ScriptNS0:DBLookupShutdown()" />

public string DBLookupShutdown()
{
    return string.Empty;
```

```
}

private class DBFunctoidHelper
{
    private string column;
    private OleDbConnection conn = new OleDbConnection();
    private string connectionString;
    private string error;
    private Hashtable mapValues = new Hashtable();
    private string table;
    private string value;

    public string Column
    {
        get
        {
            return this.column;
        }
        set
        {
            this.column = value;
        }
    }

    public OleDbConnection Connection
    {
        get
        {
            return this.conn;
        }
    }

    public string ConnectionString
    {
        get
        {
            return this.connectionString;
        }
        set
        {
            this.connectionString = value;
        }
    }

    public string Error
    {
        get
        {
            return this.error;
        }
        set
        {
            this.error = value;
        }
    }

    public Hashtable MapValues
    {
        get
        {
            return this.mapValues;
        }
    }
```

```
        public string Table
        {
            get
            {
                return this.table;
            }
            set
            {
                this.table = value;
            }
        }

        public string Value
        {
            get
            {
                return this.value;
            }
            set
            {
                this.value = value;
            }
        }
    }
```

Understanding the BizTalk Mapper: Part 11 - Advanced Functoids

Interestingly, all of the advanced functoids emit XSLT. No C# in sight at all.
The reason for this is that the functoids in this category all perform operations best suited to trees of data i.e. XML.
The only way to do this in C# would be to load the data into a DOM (i.e. *XmlDocument*) or *XmlReader*, or treat the XML as string data and search for tokens.

Note: this category was the one that actually started this series – I felt that if you knew the XSLT emitted by these functoids it would help understand when to use them, and what you can achieve with them.

Functoids covered in this category:

Assert	Record Count
Index	Scripting
Iteration	Table Looping
Looping	Table Extractor
Mass Copy	Value Mapping
Nil Value	Value Mapping (Flattening)

Advanced Functoids

Assert

Generates: XSLT **Has XSLT Equivalent**: N/A

Emitted Code:

```
<xsl:if test="(value-test)='false'">
      <xsl:message terminate="yes">
        <xsl:value-of select="(value-error)" />
      </xsl:message>
</xsl:if>
<xsl:if test="(value-test)='true'">
      <xsl:variable name="var:v1" select="(value-out)" />
      <xsl:value-of select="$var:v1" />
</xsl:if>
```

Note: *(value-test)* is the element/attribute/value tested for a true/false value.
(value-error) is the element/attribute/value used as the error text.
(value-out) is the element/attribute/value emitted if there is no error.

Index

Generates: XSLT **Has XSLT Equivalent**: N/A

Emitted Code:

```
<xsl:variable name="var:v1"
select="./(node)[index][index][...]/(element)/(attribute)" />
```

for example:

```
<xsl:variable name="var:v1" select="./Employee[2]/@id" />
```

Note: in the above code, *Employee* is the repeating node, 2 is the index we want to use, and @id is an attribute of the Employee element we wish to retrieve.
This functoid uses standard XSLT indexing to build up an XSLT select query.
The sample code shown above is a very simple example.

Iteration

Generates: XSLT **Has XSLT Equivalent**: N/A

Emitted Code:

```
<xsl:variable name="var:v1" select="position()" />
```

Note: the functoid places the above code under the relevant element we're getting the iteration (index) of.
The variable is then used to emit the index at the appropriate place.

Looping

Generates: XSLT **Has XSLT Equivalent**: N/A

Emitted Code:

```
<xsl:for-each select="(node)">
</xsl:for-each>
```

Note: where (node) is whatever input node is being looped-over.
The Looping functoid is unique in that if you link from a repeating node in the source tree, then the Mapper will implicitly insert a looping functoid (although no functoid shape will be visible on the map).
Understanding how to use the looping functoid is a vital part of creating advanced maps (e.g. concatenating multiple messages into one, or flattening loops).
If you have more than one input element into the loop, then a for-each loop is created over each input element, and one output element is created for each input element. This is a common way of concatenating (aggregating) two separate messages.

Mass Copy

Generates: XSLT **Has XSLT Equivalent**: N/A

Emitted Code:

```
<xsl:copy-of select="./@*" />
<xsl:copy-of select="./*" />
```

Note: the functoid places the above code under the appropriate destination element, and the "." indicates the current element being processed from the source message.
As an example:

```
<ns0:Employees>
  <xsl:for-each select="Employee">
    <Employee>
      <xsl:copy-of select="./@*" />
      <xsl:copy-of select="./*" />
    </Employee>
  </xsl:for-each>
</ns0:Employees>
```

Nil Value

Generates: XSLT **Has XSLT Equivalent**: N/A

Emitted Code:

```
<xsl:attribute name="xsi:nil">
   <xsl:value-of select="'true'" />
</xsl:attribute>
```

Note: the functoid places the above code under the appropriate element in the destination message.
If a parameter is supplied to the functoid, then the above code is wrapped in an <xsl:if> node.
For example:

```
<xsl:if test="@name='true'">
<Field>
  <xsl:attribute name="xsi:nil">
    <xsl:value-of select="'true'" />
  </xsl:attribute>
</Field>
</xsl:if>
```

Record Count

Generates: XSLT **Has XSLT Equivalent**: N/A

Emitted Code:

```
<xsl:variable name="var:v1" select="count(node-set)" />
```

Note: (node-set) is an element/record which repeats.
The variable is then used in a value-of statement.
For example:

```
<xsl:variable name="var:v1" select="count(/ns0:Employees/Employee)" />
<ns0:Employees>
  <Employee>
    <xsl:attribute name="id">
      <xsl:value-of select="$var:v1" />
    </xsl:attribute>
  </Employee>
</ns0:Employees>
```

Scripting

Generates: XSLT/Inline Script **Has XSLT Equivalent**: N/A

Emitted Code:

External Assembly

Emits XSLT to call the appropriate method, along with an Extension Object file containing the strong name of the assembly to use.

For example (this example passes in two constant parameters to the method):

```
<xsl:variable name="var:v1" select="ScriptNS0:AddComponent('1', '2')" />
<xsl:value-of select="$var:v1" />

<ExtensionObjects>
  <ExtensionObject
Namespace="http://schemas.microsoft.com/BizTalk/2003/ScriptNS0"
AssemblyName="Microsoft.BizTalk.DefaultPipelines, Version=3.0.1.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35"
ClassName="Microsoft.BizTalk.DefaultPipelines.PassThruReceive" />
</ExtensionObjects>
```

Inline C#
Inline JScript .NET
Inline Visual Basic .NET

Emits a call to the method, and wraps the actual code in a CDATA block.

For example:

```
<xsl:variable name="var:v1" select="userC#:MyConcat'1', '2')" />
<xsl:value-of select="$var:v1" />

  <msxsl:script language="C#" implements-prefix="userC#"><![CDATA[
///*Uncomment the following code for a sample Inline C# function
//that concatenates two inputs. Change the number of parameters of
//this function to be equal to the number of inputs connected to this
functoid.*/

public string MyConcat(string param1, string param2)
{
      return param1+param2;
}
]]>
  </msxsl:script>
```

Inline XSLT

Emits the XSLT as entered at the appropriate place in the destination tree.

Inline XSLT Call Template

Emits the template and a call to the template.

For example:

```
<xsl:call-template name="MyXsltConcatTemplate">
      <xsl:with-param name="param1" select="'1'" />
      <xsl:with-param name="param2" select="'2'" />
</xsl:call-template>

<xsl:template name="MyXsltConcatTemplate">
      <xsl:param name="param1" />
      <xsl:param name="param2" />
      <xsl:element name="field">
         <xsl:value-of select="$param1" />
         <xsl:value-of select="$param2" />
      </xsl:element>
</xsl:template>
```

Table Looping

Generates: XSLT **Has XSLT Equivalent**: N/A

Emitted Code:

(auto generated)

Note: the *Table Looping* functoid (along with the *Table Extractor* functoid) builds custom XSLT in the destination XSLT. This custom XSLT will comprise a number of *for-each* loops (for the repeating element specified as the first parameter to the *Table Looping* functoid), and then a number of structures, one per row configured in the functoid.
If the Gated property is set, then each structure is surrounded by an *if* statement.

This is an example of the output generated by use of the *Table Looping* | *Table Extractor* functoids.
In this sample, we're filtering the input list, so that the output only contains *Employee* nodes with a department value of 'Managers'.
In this example, the *Table Looping* functoid has the source *Employee* element as input, with number of columns set to 5, followed by a *Logical Equal* functoid (true if @department equals 'managers', followed by a single constants (Management) and then three attributes (@name, @id, @department).

The sample XML used as input was:

```
<ns0:Employees xmlns:ns0="http://TestMaps.Employees">
  <Employee name="Mary Briggs" id="1" department="Managers" />
  <Employee name="Sam Gamgee" id="5" department="Staff" />
  <Employee name="Frodo Smith" id="20" department="Staff" />
</ns0:Employees>
```

The looping functoid was then setup to generate one record (row) per input element, with the Gated property set on the first column the *Logical Equal* functoid).
Four Table Extractor functoids were setup, with each one connecting to one of the four attributes on the output message (@type, @name, @id, @department).
This is the XSLT generated by these functoids:

```
<xsl:template match="/s0:Employees">
<ns0:Employees>
  <xsl:for-each select="Employee">
    <xsl:variable name="var:v1"
select="userCSharp:LogicalEq(string(@department), 'Managers')" />
    <xsl:if test="$var:v1">
      <xsl:variable name="var:v2" select="'Management'" />
      <xsl:variable name="var:v3" select="@name" />
      <xsl:variable name="var:v4" select="@id" />
      <xsl:variable name="var:v5" select="@department" />
      <Employee>
        <xsl:attribute name="type">
          <xsl:value-of select="$var:v2" />
        </xsl:attribute>
        <xsl:attribute name="name">
          <xsl:value-of select="$var:v3" />
        </xsl:attribute>
        <xsl:attribute name="id">
          <xsl:value-of select="$var:v4" />
        </xsl:attribute>
        <xsl:attribute name="department">
          <xsl:value-of select="$var:v5" />
        </xsl:attribute>
      </Employee>
    </xsl:if>
  </xsl:for-each>
</ns0:Employees>
</xsl:template>
```

Table Extractor

Generates: XSLT **Has XSLT Equivalent**: N/A

Emitted Code:

(see the *Table Looping* functoid above)

Value Mapping

Generates: XSLT **Has XSLT Equivalent**: N/A

Emitted Code:

```
<xsl:for-each select="(source element)">
<(destination element parent)>
  <xsl:if test="(node-test)='true'">
    <xsl:variable name="var:v1" select="@id" />
    <(destination element)>
      <xsl:value-of select="$var:v1" />
    </(destination element)>
  </xsl:if>
</(destination element parent)>
</xsl:for-each>
```

Note: if the *Value Mapping* functoid is used with a repeating source element, then it will place a *for-each* element outside of the parent destination element.
If the source element is not repeating, then the functoid emits a *value-of* statement, with an optional if statement (if an element/attribute is used as the first parameter rather than a constant).

For example, if the input message was:

```
<ns0:Employees xmlns:ns0="http://TestMaps.Employees">
  <Employee name="Mary Briggs" id="1" department="Managers" />
  <Employee name="Sam Gamgee" id="5" department="Staff" />
  <Employee name="Frodo Smith" id="20" department="Staff" />
</ns0:Employees>
```

and the source/destination schemas were the schema used for the above message, and the *Value Mapping* functoid linked the id attribute to the Field element, then the following XSLT would be generated:

```
<ns0:Employees>
  <xsl:for-each select="Employee">
    <xsl:variable name="var:v1" select="@id" />
    <Employee>
      <Field>
        <xsl:value-of select="$var:v1" />
      </Field>
    </Employee>
  </xsl:for-each>
</ns0:Employees>
```

which would generate the following output:

```
<ns0:Employees xmlns:ns0="http://TestMaps.Employees">
      <Employee>
               <Field>1</Field>
      </Employee>
      <Employee>
               <Field>5</Field>
      </Employee>
      <Employee>
               <Field>20</Field>
      </Employee>
</ns0:Employees>
```

Value Mapping (Flattening)

Generates: XSLT **Has XSLT Equivalent**: N/A

Emitted Code:

```
<(destination element parent)>
<xsl:for-each select="(source element)">
  <xsl:if test="(node-test)='true'">
```

```
    <xsl:variable name="var:v1" select="(source-value)" />
    <(destination element)>
      <xsl:value-of select="$var:v1" />
    </(destination element)>
  </xsl:if>
</xsl:for-each>
</(destination element parent)>
```

Note: the *Value Mapping (Flattening)* functoid constructs a for-each loop around the source element, and outputs one instance of the destination element per instance of the source element.
For example, if the input message was:

```
<ns0:Employees xmlns:ns0="http://TestMaps.Employees">
  <Employee name="name_0" id="1" department="department_2" />
  <Employee name="name_0" id="5" department="department_2" />
  <Employee name="name_0" id="20" department="department_2" />
</ns0:Employees>
```

and the source/destination schemas were the schema used for the above message, and the *Value Mapping (Flattening)* functoid linked the id attribute to the Field element, then the following XSLT would be generated:

```
<ns0:Employees>
  <Employee>
    <xsl:for-each select="Employee">
      <xsl:variable name="var:v1" select="@id" />
      <Field>
        <xsl:value-of select="$var:v1" />
      </Field>
    </xsl:for-each>
  </Employee>
</ns0:Employees>
```

which would generate the following output:

```
<ns0:Employees xmlns:ns0="http://TestMaps.Employees">
      <Employee>
            <Field>1</Field>
            <Field>5</Field>
            <Field>20</Field>
      </Employee>
</ns0:Employees>
```

Understanding the BizTalk Mapper: Part 12 - Performance and Maintainability

In this section:

Any large BizTalk project will likely have had the inevitable conversations about performance and maintainability: will it be fast/sustainable enough, and will the tech support team (or whoever looks after the code once the developers have finished) be able to maintain it?

In this post I want to look at the performance of the Mapper, and also look at how maintainable maps are generally.
In order to do this, I want look at the different options you have for executing XSLT with the Mapper, and compare this to the most common non-Mapper mechanism for performing transformation: using serializable classes.

In BizTalk, your options for transformations using the Mapper are:

- Built-in functoids
- Inline XSLT script
- Inline C# / VB.NET / JScript.NET
- Classes/methods in external assemblies
- Custom functoids
- External XSLT file

Normally you'd use a combination of these in a complex BizTalk solution.

So how do you decide which to use?
Which gives you the best performance?
Which option(s) are the easiest to maintain?

The answer is: it depends!
In this section I'm going to try and give you some hard data you can use to try and answer these questions. In the next post, I'll try and answer the questions.

Performance

Performance is a very subjective subject. You could spend weeks getting your maps to execute in under 10ms, but if your performance requirement is "anything less than 100ms" then why bother?
What's more important is that performance is "good enough" and is sustainable.
Sustainable performance is the key - its one thing to be able to perform a single transformation in less than 10ms, but what about performing 40 simultaneous transformations continuously for over an hour? And what about the memory footprint?
When we talk about performance, we also need to look at memory/resource usage: If you have a very fast system that for each transformation uses 1MB memory, then at a certain point under sustained load your memory usage will start to affect performance as memory is paged out to disk (assuming that garbage collection can't keep up with the number of transformations you're performing).

In order to find out the performance and memory footprints of the various options, I put together some tests.
I ran two suites of tests:

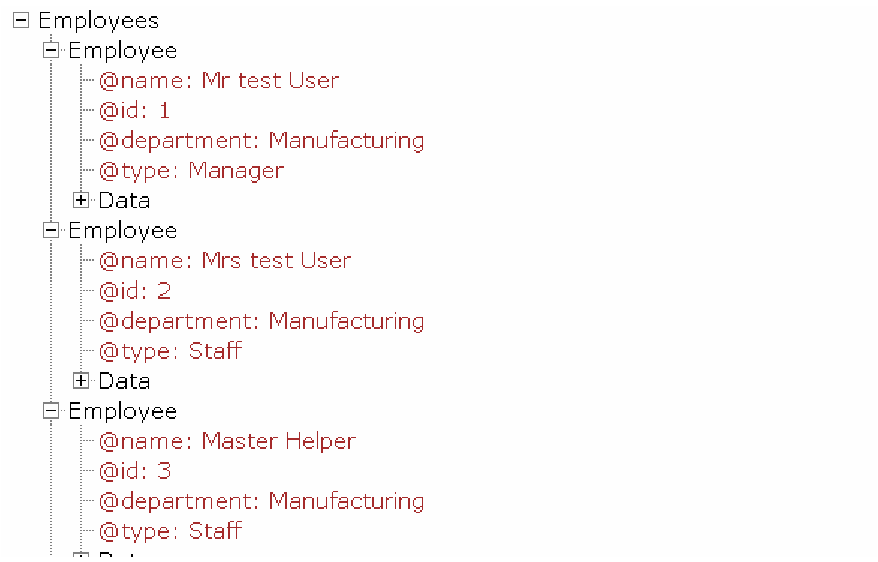
- Using a stand-alone test harness with the `XslTransform` class to perform transformation in isolation (this gives the comparative performance)
- Using a BizTalk application to measure memory usage (this gives the comparative memory usage)

Summary of Tests

I simplified the six options above into four separate XSLT tests, and then added two tests involving serializable classes for comparison:

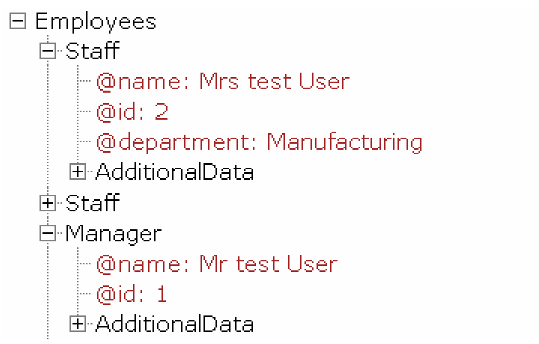
1. **Standard Map**
   A map using default built-in functoids, with a mix of functoids that emit XSLT and functoids that emit C#
2. **Map using external XSLT**
   A map which used an external XSLT file consisting of pure XSLT i.e. no external assemblies or inline script
3. **Map using inline C#**
   A map using inline C# code in a Script functoid
4. **Map using external assembly**
   A map using an external assembly via a Script functoid
5. **Transformation using Classes**
   De-serializing the input message into a class, transforming to a new class, and serializing the class back into the output message
6. **Transformation using classes and serializers**
   Same as 5. above but using the sgen.exe tool to pre-generate a serialization class

My test scenario was: Transforming a message containing 20 employee records into a new message which contained separate Manager and Staff records:
i.e. this:



```
⊟ Employees
  ⊟ Employee
     @name: Mr test User
     @id: 1
     @department: Manufacturing
     @type: Manager
    ⊞ Data
  ⊟ Employee
     @name: Mrs test User
     @id: 2
     @department: Manufacturing
     @type: Staff
    ⊞ Data
  ⊟ Employee
     @name: Master Helper
     @id: 3
     @department: Manufacturing
     @type: Staff
```

becomes this:



Each of the maps performed the transformation in the same way, as much as was possible.

Testing performance in isolation (non-BizTalk)

For the XSLT tests, I tested using both the `XslTransform` class (used by BizTalk) and the newer `XslCompiledTransform` class (for comparison).
For both of these class tests I used static and non-static instances of the classes:

- In the static test a transform class was created once and then re-used for each iteration
- In the non-static test a new transform class was created for each iteration

I ran each test twice: once with 20 iterations, and once with 500 iterations.

I also measured the amount of memory in use before and after each test - from this I calculated a rough "memory used" for each test (this is before garbage collection kicked in).

For each test I show:

1. **Total**
   This is the total time (in ms) that the test took to run
2. **Average**
   This is the average time for each iteration
3. **Average without first**
   This is the average time without the first iteration
4. **Average setup**
   This is the average time it took to create the transform object in each iteration (will be 0 for the static tests as creation of transform is performed once and not included in the timings)
5. **Average transform**
   This is the average time it took to perform the transformation
6. **Memory Used**
   This is the amount of memory used to perform the entire test

Performance Test Results

The results of the 500 iteration tests are:

	Standard Map	**Map with External Assembly**	**Map with Inline Script**	**Map with External XSLT**	**Class Transform**	**Class Transform with Serializer**
Iterations	500	500	500	500	500	500
XslTransform - NonStatic						
Total (ms)	82402	78815	79507	1622	1749	728
Average (ms)	164	157	159	3	3	1
Average without first (ms)	164	157	158	3	2	1
Average setup (ms)	151	151	152	1	1	0
Average transform (ms)	12	5	5	1	1	1
Memory Used	504MB	452MB	456MB	62MB	90MB	86MB
XslTransform - Static						
Total (ms)	4392	2234	1564	720	769	735

Average (ms)	8	4	3	1	1	1
Average without first (ms)	8	4	3	1	1	1
Average setup (ms)	0	0	0	0	0	0
Average transform (ms)	8	4	3	1	1	1
Memory Used	64MB	40MB	40MB	30MB	88MB	88MB
XslCompiledTransform - NonStatic						
Total (ms)	101214	88181	89608	16325		
Average (ms)	202	176	179	32		
Average without first (ms)	202	176	179	32		
Average setup (ms)	156	145	147	3		
Average transform (ms)	45	30	30	28		
Memory Used	192MB	141MB	145MB	82MB		
XslCompiledTransform - Static						
Total (ms)	95	131	70	55		
Average (ms)	0	0	0	0		
Average without first (ms)	0	0	0	0		
Average setup (ms)	0	0	0	0		
Average transform (ms)	0	0	0	0		
Memory Used	14MB	15MB	13MB	13MB		

(the lowest result on each row is highlighted in green)

And for comparison, the results from the 20 iteration tests:

	Standard Map	**Map with External Assembly**	**Map with Inline Script**	**Map with External XSLT**	**Class Transform**	**Class Transform with Serializer**
Iterations	20	20	20	20	20	20
XslTransform - NonStatic						
Total (ms)	3820	3931	3718	78	1213	82
Average (ms)	191	196	185	3	60	4
Average without first (ms)	171	194	184	3	4	1
Average setup (ms)	175	189	178	1	56	1
Average transform (ms)	15	7	6	1	4	2
Memory Used	20MB	18MB	17MB	2MB	4MB	3MB
XslTransform - Static						
Total (ms)	169	100	66	29	122	83
Average (ms)	8	5	3	1	6	4
Average without first (ms)	8	4	3	1	2	1
Average setup (ms)	0	0	0	0	2	1
Average transform (ms)	8	4	3	1	3	2
Memory Used	4MB	1MB	1MB	1MB	3MB	3MB
XslCompiledTransform - NonStatic						
Total (ms)	4628	3758	5094	1190		
Average (ms)	231	187	254	59		
Average without first (ms)	224	183	254	50		
Average setup (ms)	180	154	211	11		
Average transform (ms)	50	32	42	48		
Memory Used	8MB	6MB	6MB	3MB		
XslCompiledTransform - Static						
Total (ms)	46	39	67	81		
Average (ms)	2	1	3	4		
Average without first (ms)	0	0	1	1		
Average setup (ms)	0	0	0	0		
Average transform (ms)	2	1	3	4		
Memory Used	642KB	647KB	583KB	542KB		

(the lowest result on each row is highlighted in green)

Measuring Memory Usage in BizTalk

Although the *Memory Used* amount from the performance tests was useful, I wanted to know exactly how much memory BizTalk used for performing transformations – and what objects were in memory. In order to measure this I used SciTech Software's .NET Memory Profiler.
This tool attaches to the BizTalk service (BTSNTSvc.exe) and can create a snapshot of all the object instances currently in use, including how many there are and how much memory they're using.

I created a BizTalk application which contained a separate map for each of the tests above, and created orchestrations to execute the maps (and to call the C# code to perform the transform using the classes).

I performed a memory snapshot before and after running the orchestrations, and restarted the BizTalk service between each test.

I ran each test twice: once with a single message, and once with 50 messages.

For each test I show:

1. **Byte[] Instances**
   This is the count of Byte arrays in use (the relevance of Byte arrays is explained below)
2. **Byte[] Instances Size (MB)**
   The is the total size of all Byte arrays (i.e. the size of all the data they contain)
3. **Total Instances**
   The is the count of all .NET objects in use
4. **Total Instances Size(MB)**
   This is the total size of all .NET objects in use

Note that any sizes measured are *after* garbage collection has occurred i.e. these are objects which are still classed as being in-use.

BizTalk Memory Test Results

The results I measured were:
Single Message:

Test - 1 Iteration	Byte[] Instances	Byte[] Instances Size (MB)	Total Instances	Total Instances Size (MB)
Standard	5,652	0.63	17,277	1.78
External XSLT	5,752	0.59	19,038	2.11
Inline Script	5,763	0.63	20,522	2.23
Referenced Assembly	5,757	1.25	19,897	2.42
Class	30	0.03	7,476	0.74

(the lowest result in each column is highlighted in green)

50 Messages:

Test - 50 Iterations	Byte[] Instances	Byte[] Instances Size (MB)	Total Instances	Total Instances Size (MB)
Standard	146	1.63	18,354	3.24
External XSLT	126	0.80	14,359	1.87
Inline Script	5,780	1.89	22,590	3.13
Referenced Assembly	5,757	1.36	20,100	3.00
Class	146	0.85	14,202	1.81

(the lowest result in each column is highlighted in green)

Byte Arrays

I can let Microsoft explain this in their own words (this is taken from a knowledge base article here):

The System.Policy.Security.Evidence object is often used in transforms and can consume a lot of memory. Whenever a map contains a scripting functoid that uses inline C# (or any other inline language), the assembly is created in memory. The System.Policy.Security.Evidence object uses the *object of the actual calling assembly. This situation creates a rooted object that is not deleted until the BizTalk service is restarted.*

Most of the default BizTalk functoids are implemented as inline script. These items can cause System.Byte[] objects to collect in memory. To minimize memory consumption, we recommend that you put any map that uses these functoids into a small assembly. Then, reference that assembly. Use the chart below to determine which functoids use inline script and which functoids do not use inline script.

In the second column, "Yes" means that this functoid is implemented as inline script, and that it will cause System.Byte[] objects to collect in memory. "No" means that this functoid is not implemented as inline script, and that it will not cause System.Byte[] objects to collect in memory.

Functoids	**Inline script?**
All String Functoids	Yes
All Mathematical Functoids	Yes
All Logical Functoids except IsNil	Yes
Logical IsNil Functoid	No
All Date/Time Functoids	Yes
All Conversion Functoids	Yes
All Scientific Functoids	Yes
All Cumulative Functoids	Yes
All Database Functoids	No
All Advanced Functoids (apart from Script functoids using Inline C#/VB/Jscript)	No

Basically what they're saying is: whenever you use the default functoids, or inline code (i.e. C# or any other .NET language) the assembly containing the map is used as evidence to the `XslTransform` class to ensure that it's safe to use scripts. And this assembly is loaded into the appropriate AppDomain and kept there. In fact, it's loaded in as a byte array (byte[]).
Assemblies created by compiling inline script in an XSLT are temporary assemblies and are loaded into the appropriate AppDomain - they will remain in memory until the AppDomain is unloaded i.e. the BizTalk Host Instance is restarted.

So if you keep all your maps, orchestrations, schemas, etc in one big assembly, then that assembly will stay loaded in memory until the BizTalk Host Instance that loaded it is restarted.
Solution? Keep your maps (especially maps using inline C#) in a separate project/assembly - and try and keep that assembly as small as possible!

Analysing the performance results

The results shouldn't really come as any surprise.
What they say in a nutshell is: pure XSLT is much faster than XSLT which uses inline script or referenced assemblies. How much faster? Well, my tests show an average 5500% speed increase over using the default functoids (i.e. 55 times faster)!
Additionally, using pure XSLT uses 1/8 the memory.
Of course, your mileage will vary.

What's interesting though is how fast using serializable classes is i.e. de-serializing a message into a class, performing operations to create a new class, and then serializing the new class into a message.
When used with a pre-generated serialization assembly, this mechanism chases closely behind using pure XSLT (and actually beat it in one of the tests).

Maintainability

One of BizTalk's trump cards is the BizTalk Mapper: you can create maps which can be easily maintained – what's more, because the Mapper is a visual tool you can see at a glance how the mapping works.
At least, that's the theory.
If you have a relatively simple map which uses no script functoids, and has less then, say, 50 connections then Yes, I'd say this was true: it's easy to see what the map does, and probably easy to maintain it.

But if you have a map with 1000 connections, or with a whole smattering of Script functoids (or a bird's nest of Logical functoids) then No, I don't think it's easy to see what the map does or to maintain it.

At what point do you have to admit defeat with a Map and say that it's got as bit too complex? or that the next developer to come along will have problems maintaining it?
In that case, would you be better off using external XSLT or serializable classes?

External XSLT

One of the main complaints I hear about using external XSLT with the Mapper is that it's difficult to maintain. This can be true – if your editing tool is notepad! But there are great tools around for maintaining XSLT – have a look at Altova's MapForce for an example of one.
The other complaint is that XSLT is difficult, or hard to learn. Well, so is C# if you've never used it before. Go buy a book on it, or do a course!

Truth be told, if you work for a company which uses XSLT for other projects, then you're more likely to find support for using it as external XSLT in BizTalk.
Some companies have teams in IT which do nothing but create XSLT.

My point here is that although there's a myth that external XSLT is hard to maintain, it's a lot easier to maintain than a complicated map. And if you use the right tool, you can get a graphical view of what it does as well.

Serializable Classes

In my experience, it's very very common for developers to create a whole slew of transform and utility classes for handling transformations.
Sometimes this is because it's the best way to do it.
Other times it's because they simply didn't know how to achieve something in the Mapper.
One of the best features of BizTalk (the ability to call out to C# classes/methods from an orchestration or map) can also be its worst: Just because you *can* create C# code, doesn't mean that you *should*.

Maintaining poorly written C# code is a nightmare.
So if you're using serializable classes to perform transformations make sure they're well written and well documented – but most importantly: understand *why* you're using them over XSLT.

Why is it so difficult to edit code in the Script functoid?

Ever wondered why you can't resize the Script functoid code window? Or do a Ctrl-A to select all the code in it? It turns out there is actually a reason for it…

Scott Woodgate (former Product Manager for BizTalk) had this to say…
The article is correct when it points out we discourage the use of .NET objects directly inside the map. While this is possible, we encourage good developer design which is encapsulating code in external assemblies. This turns out to be much better because you have a single assembly with code shared across multiple maps that can be versioned once and managed more easily

Unfortunately, this restriction also means that it's difficult to use inline XSLT, which is a shame.

Documentation

You can't get away from the fact that code that is easy to maintain is either self-documenting, or is accompanied by excellent documentation.
Regardless if you use maps, external XSLT, or serializable classes you should really document your transforms: explain what they do, how they do it, why they do it – and most importantly, give some context: explain why you chose to do it that particular way.
A developer who has to maintain your code in 2 years time might not have your background of development and political issues to understand why you made your choices.

The next post is going to look at the performance/maintainability of the different transformation options and attempt to help you to decide when to use which option.

Understanding the BizTalk Mapper: Part 13 - Is the Mapper the best choice for Transformation in BizTalk?

In this section:

Transformation Choices

When performing transformations in BizTalk, you have four choices (that I can think of):

1. Using the BizTalk Mapper
2. Using a custom XSLT file with the BizTalk Mapper
3. Using a separate transformation engine (called from code)
4. Performing transformations in code

Each of these offers their own benefits depending on your requirements.
Normally your choice will depend on 3 factors:

- Performance
- Complexity
- Maintainability

Generally you will get one (or two) of these, at the cost of the third.
For simple transformations, you can get all three with the Mapper using the built-in functoids.

In order to decide if the BizTalk Mapper is the best choice, we have to look at these alternatives and compare them.

BizTalk Mapper

The BizTalk Mapper is the de-facto standard for transformations in BizTalk.
With this option I'm including maps created via the Mapper visual designer – I'm not including maps which reference external XSLT.
Maps created with the designer include those which use:

- No functoids, or only functoids which emit XSLT
- Default functoids
- Custom functoids / Script functoids with referenced assemblies
- Script functoids with inline C#/VB/JScript
- Script functoids with inline XSLT

If you're creating a map which is relatively simple (e.g. has less than 40 or so links, and less than 20 functoids) then you could argue that the map is easy to follow and maintain.
If your map has no functoids, or only uses functoids which emit XSLT, then your map will likely perform very well.
If you're only processing a small number of messages (e.g. 20 an hour) then performance won't be that important to you.
However, the BizTalk Mapper isn't really suited where you have hundreds of links/functoids, or in situations where you need the best performance possible (e.g. thousands of messages per hour).
If you're not sure if the Mapper will give you the performance you need, you're better off doing a quick proof-of-concept with a realistic map and seeing what performance you can get.

Pros:

- Can use it with Receive/Send Ports
- Easiest to start using

Cons:
- Easy to create complex/slow maps
- Difficult to do advanced mapping
- Uses the .NET 1.x `XslTransform` class
- Standard functoids use inline C# code

Custom XSLT with the BizTalk Mapper

Here we're talking about using the Custom XSL Path property of a BizTalk Map and specifying an external XSLT file. The contents of this XSLT file will be compiled into the assembly containing the map.
Note: bear in mind that you can't use `<xsl:include>` or `<xsl:import>` tags in your external XSLT – the external XSLT file needs to be complete. For more information on this, see here.

The advantage of external XSLT is that it is very fast – usually faster than using functoids in a map (depending on the functoids and the XSLT obviously). In you refer to the speed tests in my previous post, you'll see that in my tests, external XSLT was 55 time faster than using inline scripts/default functoids.
If you're au fait with XSLT and your project has the ability to support external XSLT then this is the best option to use in my mind. If you use a good external XSLT editor (e.g. Altova's MapForce) then the issues of maintenance and being able to visualise the map should disappear.

However, using XSLT can be tricky: because the Mapper uses XSLT v1.0 you're limited in certain areas (such as string handling, date handling, and mathematical functions). Additionally, using inherited types can be tricky if you've never used them before in XSLT.
In these cases, you might be better performing your transformations in code.

Pros:
- Can use it with Receive/Send Ports
- Transforms are *fast*
- XSLT is the best choice for XML structures

Cons:
- Need good understanding of XSLT
- Can be difficult to maintain if not using a good XSLT editor
- Uses the .NET 1.x `XslTransform` class

External Transform Engine

Because of BizTalk's ability to execute external code from an orchestration, you can perform transformations via an external engine.
For example you might choose to use the `XslCompiledTransform` class to increase the performance or your existing maps, or use an XSLT v2.0 engine such as SAXON.
In both these cases, you could even use your existing BTM maps if you wanted (by accessing the XSLT from the compiled map class).

Note: When using `XslCompiledTransform`, using inline C# (or whatever language) in a map will normally be faster than using an external assembly i.e. using ExtensionObjects.
The reason for this is that with inline script (i.e. using the `<msxsl:script>` tag) the compiler can generate direct calls to the compiled script assembly from the transform assembly, whereas with external assemblies, reflection needs to be used at runtime to invoke a call. See here for more information on this.

Using an external transformation engine makes sense if you must use XSLT v2.0 (or are sharing XSLT v2.0 maps with other projects), or if an external engine gives you a required feature.

However you can only use these engines from an orchestration (i.e. can't use them with receive/send ports), and any performance related caching of classes will be up to you to e.g. caching one instance of a compiled transform for an orchestration's AppDomain.

However you can get substantial performance increases if you use the right engine and caching strategy.

Plus if your company has an existing maintenance strategy for a given engine then maintenance of these maps might be a no-brainer.

Pros:
- Transformation can be very fast (if correct engine is chosen)
- Can share engine/transforms between different technologies
- Can use company-standard transforms if present/required

Cons:
- Can't use it with Receive/Send Ports, only with Orchestrations
- Can be difficult to maintain if not well understood
- Caching is left up to developer
- Performance is relevant to the skill of the developer and engine

Transformation in code

By this I mean transformation involving classes and serialization.
If you've read the previous post (Understanding the BizTalk Mapper: Part 12 - Performance and Maintainability) then you'll know that this sort of transformation can be blazingly fast.
However this needs to be taken with a caveat: transformation in code will only be as good as the programmer writing the code.
If you write C# or VB.NET code which performs badly, then your transformations will perform badly. XSLT doesn't suffer from this problem: because the language is designed to execute rapidly over XML structures: it's more difficult to write poorly performing XSLT code than C#/VB.NET code.

However, using C#/VB.NET can be a god-send in certain situations.
One example is where you're using inheritance i.e. base elements/types in an XSD (if any of the elements/attributes in an XML instance have an `xsi:type` attribute, then you're using inheritance).
In these cases, using source code and inheritance can reduce the amount of code you need to write as you work with the base class without worrying about what derived class was actually passed to you.

But remember: don't use C#/VB.NET code to perform transformation simply because you don't understand XSLT. For a lot of situations it can be significantly faster to create XSLT code to perform a transformation than the C#/VB.NET equivalent – and to another XSLT developer it will be easier to understand and maintain.

Pros:
- Transforms can be *fast*
- Can be easier to develop/maintain transforms if developer knows C# better than XSLT

Cons:
- Can't use it with Receive/Send Ports, only with Orchestrations
- Can be difficult to maintain if not well documented
- Performance is relevant to the skill of the developer

Which one should you use?

Well without knowing your requirements that's a bit like saying "which airline should I fly with?"
Tell me your requirements, and I'll tell you the best airline!
[the answer's always Virgin Atlantic or Air New Zealand, by the way... ;-)]

However (in my own biased opinion) I would suggest that a pure XSLT way is best, followed by the BizTalk Mapper using default functoids, followed (closely) by doing transformations in code using serializable classes.

If your performance/maintenance requirements are minimal then it won't really matter.
If you must have the *best* performance, then use pure XSLT or transformations in code.

One thing to bear in mind is this: The BizTalk Mapper creates XSLT.

XSLT is a functional language which uses templates and pattern matching and as such is very different from a procedural/imperative language, such as Java, C#, C++, or VB.

It's quite common for non-XSLT programmers to ask how to create a for/while loop, or how to do an if/case statement in XSLT.

If you're asking those questions, you should to go do a basic course on XSLT (or read a book) as it works in a very different way.

Properly written XSLT is elegant and fairly simple to follow.

OK, so that's the end of this series of posts.
I hope it's been useful to you in some way.
I'll gladly welcome any feedback on the series.

Agree or disagree with what I've written? Please let me know.

Have I made any mistakes, or misrepresented a point of view? I'll happily make corrections if necessary.

Let me know what you think by leaving comments, or emailing me at bizbertATprobertsolutions.com (replacing the AT with an @).

Thanks for taking the time to stop by and read.

Appendix A: Useful Links

This is a collection of links to articles and blog posts that you may find useful.

- Calling .NET assemblies using Script Functoid use *XSLT Extension Objects*
  http://msdn2.microsoft.com/en-us/library/tf741884.aspx

- Scripts in XSLT files will cause ASP.NET to run out of memory
  https://connect.microsoft.com/VisualStudio/feedback/ViewFeedback.aspx?FeedbackID=97311

- PRB: Cannot unload assemblies that you create and load by using script in XSLT
  http://support.microsoft.com/kb/316775

- XSLT scripting (msxsl:script) in .NET - pure fast evil
  http://www.tkachenko.com/blog/archives/000620.html

- XSLT Security Considerations
  http://technet.microsoft.com/en-us/library/wk7yxab1(VS.80).aspx

- XPath operators
  http://www.w3schools.com/xpath/xpath_operators.asp

- Microsoft XML Team's WebLog: Introducing XslCompiledTransform
  http://blogs.msdn.com/xmlteam/articles/Introducing_XslCompiledTransform.aspx

- Microsoft XPath/XSLT Extensions
  http://msdn2.microsoft.com/en-us/library/ms256453(VS.80).aspx