Unit Testing in XSLT 2.0

Norman **Walsh**

23 Sep 2005

Abstract

One of the tenets of modern software design is that early and frequent testing is a key contributor to successful application development.

Unit testing frameworks, tools designed to ease the development and execution of unit tests, exist for many programming languages. This paper discusses how unit testing can be applied to the development of stylesheets and describes a testing framework for XSLT 2.0 unit tests.

Table of Contents

One of the tenets of modern software design is that early and frequent testing is a key contributor to successful application development.

Unit testing frameworks, tools designed to ease the development and execution of unit tests, exist for many programming languages. This paper discusses how unit testing can be applied to the development of stylesheets and describes a testing framework for XSLT 2.0 unit tests.

1. Why do you test?

The obvious answer is, because you need to make sure that your code is functioning correctly. The techniques described in this paper were developed in conjunction with the development of XSLT 2.0 stylesheets for DocBook.

The XSLT 1.0 Stylesheets for DocBook consist of more than 3,500 templates spread over more than 150 files. Many of these temples have dynamic behavior influenced by the setting of any one of almost 500 parameters.

It's likely that the XSLT 2.0 stylesheets for DocBook will be roughly the same size. Two goals of this rewrite, in addition to taking advantage of XSLT 2.0 features, are better API documentation and more robust testing.

2. How can you test?

One strategy for testing a stylesheet is to run the entire stylesheet over a suite of test documents and determine (either mechanically or visually) if the results are correct. This is often called “integration testing” because it tests the entire application as a whole.

During the development of the XSLT 1.0 stylesheets for DocBook, integration testing was used almost exclusively. This lead to the creation of a large, useful set of test documents. Integration testing has several deficits:

1. Coverage is spotty: not all functions and templates can be tested easily. Processing an element, for example an `orderedlist`, tests the set of templates and functions necessary to process that element. However, the set of templates and functions can vary depending on the context in which the element occurs and on the values of several parameters. Establishing a test suite that exercises all possible code paths requires considerably more effort than simply creating test documents.

2. Comparison is difficult. After you've made changes to a stylesheet, running the test suite is only the first step. The new results need to be compared against some previous baseline results to determine if the changes made have (a) had the desired effect and (b) not introduced any undesirable changes.

   A manual or visual comparison is time-consuming, tedious, and error prone. The only really effective method for performing this comparison is with an automated tool. Unfortunately, such a tool has to know which changes are significant and which are not. It's not uncommon for changes to introduce insignificant differences in *every* test: different line breaks, attribute orders, new or reordered metadata elements, etc. In practice, writing such a tool has proved to be too difficult.

3. Coverage is too coarse. Assuming that you have overcome the comparison hurdle, discovering that you've introduced a bug in, say, the way appendixes are numbered, only tells you that you have a bug. It doesn't provide very many clues as to where that bug might be. It probably has something to do with the code you changed, but it might not. What if other members of the team have also checked in changes recently? Then you're stuck with a “start from scratch” debugging exercise. Ideally, testing would tell you not only that something is wrong, but give you some very detailed information about where it might have gone wrong.

Integration testing is a necessary and important form of testing, but the goal of unit testing is to evaluate the stylesheet at a finer level of granularity. Such an evaluation should be able to test all of the code paths, provide for efficient, automated comparison against results that are known to be correct, and identify where errors arise at the granularity of individual functions or templates.

3. What can you test?

An XSLT 1.0 stylesheet consists of a collection of templates. Many are match templates that are evaluated by the processor based on the “best match” for the context node, others are named templates evaluated by the stylesheet writer through explicit calls.

XSLT 2.0 adds an important new kind of structure, the function definition. Functions are similar to named templates in many respects, but they can be called directly from within XPath expressions.

In unit testing for XSLT 2.0, we'd like to be able to evaluate the correctness of individual:

- Named templates
- Functions
- Match templates

In principle, this is straight-forward for named templates and functions (match templates are considerably more difficult; we'll come back to them later). Consider the following trivial function:

```
<xsl:function name="f:filename-basename" as="xs:string">
  <xsl:param name="filename" as="xs:string"/>

  <xsl:value-of select="tokenize($filename,'/')[last()]"/>
</xsl:function>
```

It's easy to see that the following expressions test this function:

```
<xsl:if test="f:filename-basename('/path/to/my/file.ext')
                                  = 'file.ext'">PASS</xsl:if>

<xsl:if test="f:filename-basename('http://path/spec/to/here')
                                  = 'here'">PASS</xsl:if>

<xsl:if test="f:filename-basename('noslashes')
                                  = 'noslashes')">PASS</xsl:if>
```

Simple though they appear, these expressions are woefully inadequate for any real testing framework:

1. They indicate success but not failure. The conditions could be reversed, so that failure is tested, but that would only indicate that some tests failed, nothing in the messages identifies which test is being run.
2. There's too much repetitive boilerplate. We want to encourage programmers to construct test cases for all the templates they write. One of the key factors that we need to address is usability: it should be easy to write the tests.
3. Functions that expect string parameters and return string results are the easiest to test. In practice, we're going to have to deal with functions and templates that expect a variety of inputs and return a variety of results. We need to be able to handle not only strings, but atomic values of any type and nodes. In addition, some functions expect to be called within a specific context which we will have to construct in the test harness.

Our challenges are:

1. Make the tests easy to write.

2. Make the results easy to read.
3. Support atomic values and nodes as input and output.
4. Support tests that require a specific context (e.g., an `emphasis` element inside a `title`).

4. Structure of the harness

Experience suggests that keeping the documentation and tests as close to the actual code as possible increases the likelihood that the documentation and tests will be updated when the code is modified. Literate programming systems take this to the extreme, making the code a byproduct of the documentation. The approach outlined here does not attempt to go that far.

The ability to place foreign markup containing, for example, documentation and tests, directly in a stylesheet gives us a practical intermediate ground.

A better testing framework is one goal of the XSLT 2.0 rewrite of the DocBook stylesheets. Another goal is better reference documentation for the stylesheets themselves. For every stylesheet artifact, we want documentation and, where appropriate, a set of tests. We introduce these additional features directly in the stylesheet:

```
<xsl:stylesheet>
…

<doc:function name="f:function-name">
   …documentation…
</doc:function>

<u:unittests name="f:function-name">
   …tests…
</u:unittests>

<xsl:function name="f:function-name">
   …code…
</xsl:function>

…
<xsl:stylesheet>
```

Similar structures are used to document templates, modes, top-level parameters, etc.

If we were designing a framework without any constraints, we would likely nest some of these structures together and use content models to enforce requirements, such as: every function must have a documentation section and a unit test section.

However, we don't have complete freedom, XSLT imposes some constraints. The easiest way to achieve our goals within those constraints is to make our new elements “top-level”, that is, peer to `xsl:function`. We will take advantage of the fact that XSLT is written in XML, and is therefore amenable to XSLT processing, to enforce any additional requirements we want to impose.

The documentation sections are themselves written in an extension of DocBook that includes specific sections for documenting the purpose, description, parameters, and return values of templates and functions. This paper doesn't discuss the documentation in any detail, the following sections will describe how the unit tests are entered into the stylesheet and how they are subsequently processed by the framework.

5. The unittests

Unit tests are identified with the `unittests` element in the `http://nwalsh.com/xsl/unittests#` namespace. Throughout this paper, we assume that `u:` is bound to the unit tests namespace.

The `u:unittests` wrapper has two functions: it holds a collection of related tests and it allows the author to specify the value of global parameters (that is, top-level `xsl:param` elements).

All `u:param` elements that are direct children of the `u:unittests` element will be transformed into top-level parameters in the stylesheet that performs the test.

6. The tests

A `u:unittests` element contains one or more `u:test` elements. In the simple case, each `u:test` specifies the function or template parameters and the expected result. For example, the *f:filename-basename* tests described above can be expressed in the following `u:unittests`:

```
<u:unittests function="f:filename-basename">
  <u:test>
    <u:param>/path/to/my/file.ext</u:param>
    <u:result>'file.ext'</u:result>
  </u:test>
  <u:test>
    <u:param>http://path/spec/to/here</u:param>
    <u:result>'here'</u:result>
  </u:test>
  <u:test>
    <u:param>noslashes</u:param>
    <u:result>'noslashes'</u:result>
  </u:test>
</u:unittests>
```

More complex tests can define variables and establish a context.

7. Test parameters

Parameters are specified with the `u:param` element. The `u:param` is a semantic clone of `xsl:param`:

1. Content can be specified with either a `select` attribute or as content.
2. The `as` attribute can be used to specify the type of the parameter.
3. When parameters are passed to a named template or match template, they must have a `name` attribute; when passed to a function, they must not.

8. Test results

Each test declares the correct result with `u:result`. Results are always specified in the element content. Literal strings must be quoted.

9. Using variables

Variables are specified with the `u:variable` element which, like `u:param`, is a semantic clone of `xsl:param`:

1. Variables must have a `name` attribute.
2. Content can be specified with either a `select` attribute or as content.
3. The `as` attribute can be used to specify the type of the parameter.

Variables are necessary when you want to select part of a structure and pass it as an argument to a function or named template. Suppose, for example, that you want to test the behavior of a function when processing a paragraph inside a structure like this one:

```
<db:book>
  <db:title>Some Title</db:title>
  <db:chapter>
    <db:title>Some Chapter Title</db:title>
    <db:para>My para.</db:para>
  </db:chapter>
</db:book>
```

Because there's no way direct way to simultaneously pass that structure and select a node inside it, we can create a variable, `$mydoc` and then use that variable in our `u:param`.

The final `u:test` looks like this:

```
<u:test>
  <u:variable name="mydoc">
    <db:book>
      <db:title>Some Title</db:title>
      <db:chapter>
        <db:title>Some Chapter Title</db:title>
        <db:para>My para.</db:para>
      </db:chapter>
    </db:book>
  </u:variable>
  <u:param select="$mydoc//db:para[1]"/>
  <u:result>'R.1.2.2'</u:result>
</u:test>
```

10. Establishing context

The techniques described so far allow you to construct parameters and pass them explicitly to named templates and functions. These techniques are not sufficient to test named templates which expect to process the context node.

In order to test templates which operate on the context node, you must be able to establish it. The `u:context` element is used for this purpose.

When a `u:test` has a `u:context`, the node in that `u:context` will be made the context node before the template is evaluated. (Establishing a context node before evaluating a function is unnecessary as functions do not inherit the context.)

The following example shows how a named template (in this case, *inline-charseq*) can be evaluated in the context of a particular `varname`.

```
<u:test>
  <u:context as="element()">
    <db:varname xml:id="varfoo">someVarName</db:varname>
  </u:context>
  <u:result>
    <span xmlns="http://www.w3.org/1999/xhtml"
          class="varname" id="varfoo">someVarName</span>
  </u:result>
</u:test>
```

10.1. How does context work?

Transforming the variables and parameters of unit tests into XSLT that can be evaluated to perform the test proceeds in the natural way: `u:variable` elements are transformed into `xsl:variable` elements, `u:param` elements are transformed into `xsl:param` elements, and the named template or function is evaluated. To test functions, temporary variables are used to hold each of the parameters so that they can be passed in XPath syntax.

Context is more elusive. The technique employed by the framework is in three parts:

1. The context is stored in a variable with an automatically generated, unique name.
2. Instead of calling the named template directly, `xsl:apply-templates` is performed on that variable in an automatically generated, unique mode.
3. Finally, a top-level template is generated which matches `node()` in the appropriate mode. The call to the named template is placed inside that template.

The practical effect of this three-step process is to change the context node to the specified context and call the named template in that context.

11. Testing match patterns

Establishing a context is a necessary step in testing a match pattern, but it is not a sufficient step. Simply calling `apply-templates` can not guarantee that the correct template will be evaluated because the stylesheet processor will select the template based on the match pattern, priority, mode, and input precedence.

To overcome this obstacle, the testing framework takes an additional step for match patterns: it creates a copy of the template that should be tested and gives it an automatically generated, unique name. It can then call the template directly using that name.

This approach allows a large number of match templates to be tested. However, it does have a number of drawbacks:

- The framework selects the template to test based on the match pattern, mode, and priority. When those three parameters are insufficient to locate a unique template in the stylesheet (for example, because of overrides due to import precedence), the framework cannot perform the test. It simply reports “indeterminate results”.
- An exact, textual match is made against the match pattern, mode, and priority. This is more restrictive than the real processor which would see no distinction between `priority="1"` and `priority="01"`.

- It is possible to force a template to be applied in a context that it can never match in the real stylesheet. For example, you could establish an `emphasis` element as the context node and then force the evaluation of a template that has `match="filename"` in the real stylesheet.

If these restrictions are encountered in practice, recall that templates can have both a match pattern and a name. The stylesheet author can work around these restrictions simply by giving the template in question a name.

12. Running the tests

Running the tests and constructing the report is a mostly-automated process. In order to allow tests to specify different top-level stylesheet parameters, it is often necessary to transform the collection of unit tests into several stylesheets. After the stylesheets have been written, each must be run. Each stylesheet produces a fragment of XHTML that describes the test results. These fragments are then assembled to produce the final report.

12.1. Writing the tests

Transform the stylesheet that contains the tests with the `writetests.xsl` stylesheet. The results of this transformation will be a set of stylesheet documents named `test0.xsl`, `test1.xsl`, etc. and a manifest.

The `writetests.xsl` stylesheet has a top-level parameter, `basename`, that can be set to prepend a prefix on the test filenames.

Note that the `writetests.xsl` stylesheet will expand `xsl:include` and `xsl:import` statements to find all of the available tests.

12.2. Running the tests

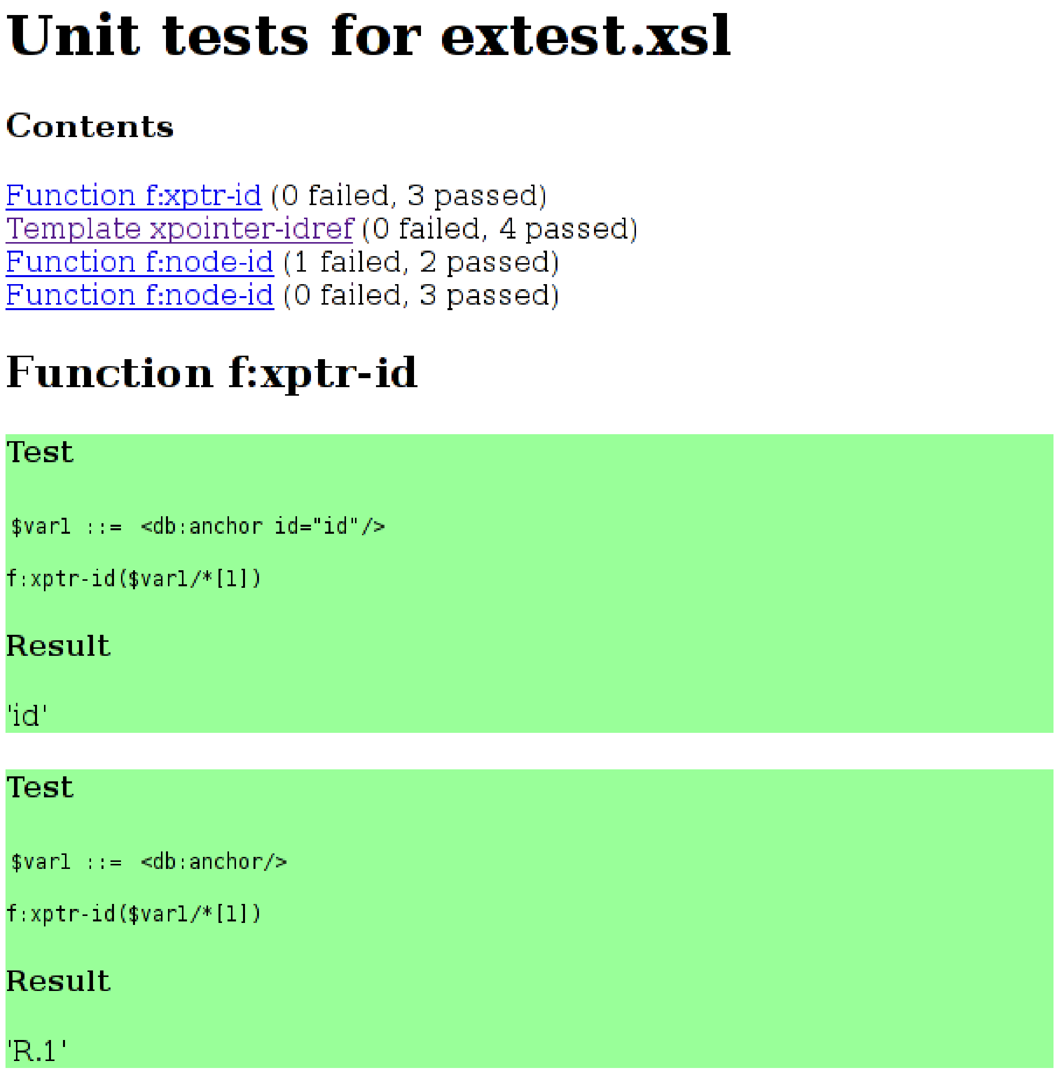
Run each `test`*`n`*`.xsl` stylesheet. The input to the stylesheet is entirely irrelevant, so you can transform any document with the stylesheet to run it.

The results of this transformation will be a new HTML fragment that enumerates all of the tests and identifies those which pass or fail.

The results of `test0.xsl` must be stored in `test0.xml`, `test1.xsl` in `test1.xml`, etc.

12.3. Assembling the results

Process the manifest document with `assembletests.xsl` to assemble the results. This will construct a single HTML document that contains the test results. An example of such a report is shown in Figure 1, “Unit Testing Report”.



Unit tests for extest.xsl

Contents

Function f:xptr-id (0 failed, 3 passed)
Template xpointer-idref (0 failed, 4 passed)
Function f:node-id (1 failed, 2 passed)
Function f:node-id (0 failed, 3 passed)

Function f:xptr-id

Test

```
$var1 ::= <db:anchor id="id"/>
f:xptr-id($var1/*[1])
```

Result

'id'

Test

```
$var1 ::= <db:anchor/>
f:xptr-id($var1/*[1])
```

Result

'R.1'

Figure 1. Unit Testing Report

The stylesheet tested to produce this report is shown in Appendix A, *Example Stylesheet: extest.xsl*.

Fix bugs. Re-run tests. Repeat until done.

A. Example Stylesheet: extest.xsl

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:db="http://docbook.org/ns/docbook"
                xmlns:doc="http://nwalsh.com/xsl/documentation/1.0"
                xmlns:f="http://docbook.org/xslt/ns/extension"
                xmlns:fp="http://docbook.org/xslt/ns/extension/private"
                xmlns:l="http://docbook.sourceforge.net/xmlns/l10n/1.0"
                xmlns:m="http://docbook.org/xslt/ns/mode"
                xmlns:u="http://nwalsh.com/xsl/unittests#"
                xmlns:xs="http://www.w3.org/2001/XMLSchema"
                exclude-result-prefixes="db doc f fp l m u xs"
                version="2.0">

<u:unittests function="f:node-id">
  <u:param name="persistent.generated.ids" select="0"/>

  <u:test>
    <u:param><db:anchor id='id'/></u:param>
    <u:result>'id'</u:result>
  </u:test>

  <u:test>
    <u:param><db:anchor/></u:param>
    <u:result>'generated id; (failure expected)'</u:result>
  </u:test>

  <u:test>
    <u:param select="//db:para[1]">
      <db:book>
  <db:title>Some Title</db:title>
  <db:chapter>
    <db:title>Some Chapter Title</db:title>
    <db:para xml:id='mypara'>My para.</db:para>
  </db:chapter>
      </db:book>
    </u:param>
    <u:result>'mypara'</u:result>
  </u:test>
</u:unittests>

<u:unittests function="f:node-id">
  <u:param name="persistent.generated.ids" select="1"/>

  <u:test>
    <u:param><db:anchor id='id'/></u:param>
    <u:result>'id'</u:result>
  </u:test>

  <u:test>
    <u:param><db:anchor/></u:param>
    <u:result>'R.1'</u:result>
  </u:test>

  <u:test>
    <u:param select="//db:para[1]">
      <db:book>
  <db:title>Some Title</db:title>
  <db:chapter>
    <db:title>Some Chapter Title</db:title>
    <db:para>My para.</db:para>
  </db:chapter>
```

```
      </db:book>
    </u:param>
    <u:result>'R.1.2.2'</u:result>
  </u:test>
</u:unittests>

<xsl:function name="f:node-id" as="xs:string">
  <xsl:param name="node" as="node()"/>

  <xsl:choose>
    <xsl:when test="$node/@xml:id">
      <xsl:value-of select="$node/@xml:id"/>
    </xsl:when>
    <xsl:when test="$persistent.generated.ids != 0">
      <xsl:variable name="xpid" select="f:xptr-id($node)"/>
      <xsl:choose>
  <xsl:when test="$xpid = '' or $node/key('id', $xpid)">
    <xsl:value-of select="generate-id($node)"/>
  </xsl:when>
  <xsl:otherwise>
    <xsl:value-of select="$xpid"/>
  </xsl:otherwise>
      </xsl:choose>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="generate-id($node)"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:function>

<u:unittests function="f:xptr-id">
  <u:test>
    <u:param><db:anchor id='id'/></u:param>
    <u:result>'id'</u:result>
  </u:test>

  <u:test>
    <u:param><db:anchor/></u:param>
    <u:result>'R.1'</u:result>
  </u:test>

  <u:test>
    <u:param select="//db:para[1]">
      <db:book>
  <db:title>Some Title</db:title>
  <db:chapter>
    <db:title>Some Chapter Title</db:title>
    <db:para>My para.</db:para>
  </db:chapter>
      </db:book>
    </u:param>
    <u:result>'R.1.2.2'</u:result>
  </u:test>
```

```
</u:unittests>

<xsl:function name="f:xptr-id" as="xs:string">
  <xsl:param name="node" as="element()"/>

  <xsl:choose>
    <xsl:when test="$node/@xml:id">
      <xsl:value-of select="$node/@xml:id"/>
    </xsl:when>
    <xsl:when test="$node/@id">
      <xsl:value-of select="$node/@id"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of>
  <xsl:choose>
    <xsl:when test="not($node/parent::*)">R.</xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="concat(f:xptr-id($node/parent::*), '.')"/>
    </xsl:otherwise>
  </xsl:choose>
  <xsl:value-of select="count($node/preceding-sibling::*)+1"/>
      </xsl:value-of>
    </xsl:otherwise>
  </xsl:choose>
</xsl:function>

<u:unittests template="xpointer-idref">
  <u:test>
    <u:param name="xpointer">#xpointer(id("foo"))</u:param>
    <u:result>'foo'</u:result>
  </u:test>

  <u:test>
    <u:param name="xpointer">#xpointer(id('bar'))</u:param>
    <u:result>'bar'</u:result>
  </u:test>

  <u:test>
    <u:param name="xpointer">#baz</u:param>
    <u:result>'baz'</u:result>
  </u:test>

  <u:test>
    <u:param name="xpointer">http://www.example.org/some/document</u:param>
    <u:result>''</u:result>
  </u:test>
</u:unittests>

<xsl:template name="xpointer-idref">
  <xsl:param name="xpointer" select="'http://...'"/>
  <xsl:choose>
    <xsl:when test="starts-with($xpointer, '#xpointer(id(')">
      <xsl:variable name="rest"
```

```
        select="substring-after($xpointer, '#xpointer(id(')"/>
      <xsl:variable name="quote" select="substring($rest, 1, 1)"/>
      <xsl:value-of select=
                      "substring-before(substring-after($xpointer, $quote),
                           $quote)"/>
    </xsl:when>
    <xsl:when test="starts-with($xpointer, '#')">
      <xsl:value-of select="substring-after($xpointer, '#')"/>
    </xsl:when>
    <!-- otherwise it's a pointer to some other document -->
  </xsl:choose>
</xsl:template>

</xsl:stylesheet>
```

Biography

Norman **Walsh**
Sun Microsystems, Inc. [http://www.sun.com]
Belchertown
Massachusetts
United States of America
http://norman.walsh.name/

Norman Walsh is an XML Standards Architect in the Java and Web Services group at Sun Microsystems, Inc.

Norm is an active participant in a number of standards efforts worldwide, including the XML Core and XSL Working Groups of the World Wide Web Consortium where he is also an elected member of the Technical Architecture Group, the OASIS RELAX NG Committee, the OASIS Entity Resolution Committee for which he is the editor, and the OASIS DocBook Technical Committee, which he chairs.

He is the principal author of *DocBook: The Definitive Guide*, published by O'Reilly & Associates.