Extreme Markup Languages 2004®

Montréal, Québec
August 2-6, 2004

Automated Quality Assurance for Heuristic-Based XML Creation Systems

Bruce Rosenblum
Inera Incorporated

Irina Golfman
Inera Incorporated

Abstract

Large volumes of XML document instances are created by conversion systems that rely on heuristic-based tagging. Quality assurance is typically conducted on individual document instances, but often there is no systematic regression testing of the conversion system applications used to tag the instances, especially when the patterns used for heuristic-based tagging are changed or updated.

Because XML conversion systems have a deterministic output for a given input document, automated quality assurance can be used to ensure the stability of markup creation by a conversion application over a large volume of documents in a conversion system. The automated test suite can also be used to validate general improvements to processing heuristics when introducing broad-based changes to the overall system

This paper examines the issues that are encountered when developing conversion systems, especially those systems that rely on heuristic-based tagging, and the quality problems that arise when updates are made to the heuristics when adequate regression testing is not completed. It includes information for the development and deployment of automated quality assurance systems that revalidate the quality of conversion systems used to automatically create XML tags.

A Conference of IDEAlliance
Extreme Markup Languages® 2004

Automated Quality Assurance for Heuristic-Based XML Creation Systems

Table of Contents

A Conference of IDEAlliance
Extreme Markup Languages® 2004

Automated Quality Assurance for Heuristic-Based XML Creation Systems

Bruce Rosenblum and Irina Golfman

§ Introduction

Markup creation has long since moved past the phase where all that counted was a document that parsed. As XML has proliferated, projects have become increasingly sophisticated, and the quality requirements for those projects have become more rigorous. It’snot enough that a file parses against a DTD or Schema—it's critical that the data between the tags is semantically correct according to the tag set.

Markup technologies have not kept pace with the demands of quality assurance. Data types in W3C XML Schema have mitigated some problems, but they are of limited value for many situations, especially for text-based documents with mixed content models.

Greater levels of quality assurance are necessary to produce high-quality, richly structured XML. In this paper, we examine how Quality Assurance can be automated for heuristic-based XML creation systems. This automated quality assurance can ensure the stability of markup creation over a large volume of documents in a system that is enhanced over time to provide higher-quality semantic results.

§ Markup Creation

XML and SGML document instances are created using a variety of methods. Common among these methods are hand-markup, use of native editors, and content conversion.

Hand Markup

In the early days of SGML, before adequate authoring tools were developed, a few brave souls created SGML by hand. Often using editors like VI and EMACS, they worked carefully to author their content and apply tags by hand. Sometimes the tagging was done on the fly; other times, authors went back to tag their content after typing it.

Native Markup Authoring

SGML never would have been used by more than a tiny handful of authors if hand-tagging remained the only way to create markup. Eventually, a variety of SGML- and XML-native authoring tools were developed. With these tools, authors can create content and easily apply markup as they write. Often these tools are enhanced with template-driven overlays that guide the author through the process of creating a valid document.

Native markup authoring tools have worked well in specialized environments such as aerospace, where the entire document life cycle (creation, publication, revision, repurposing) is under the control of a single entity. However, these tools have been less successful in environments where:

- Content is authored by freelance writers using non-XML tools
- Tagged documents are not required for the entire document lifecycle

Content Conversion

Others have written about the business reasons for why organizations choose not to use XML-native authoring tools [Bide 2000]; [Rosenblum 2002]. However, for the purposes of this paper, we will accept as the reality that a significant portion of content is tagged by content conversion rather than native markup authoring. There are two common tactics when documents are marked up after authoring: manual re-keying and software-based conversion.

Manual Re-Keying

Manual re-keying of the content requires complete re-creation of the electronic file from a print copy. It is often done by organizations working "offshore" in relatively low-wage countries, most commonly in India. The re-keyed text is often double- or triple-keyed (meaning the same content is typed by two or three operators) and then automatically compared. Any discrepancies in the content or tagging are resolved by an additional operator.

Software-Based Conversion

Software-based conversion uses the electronic file provided by the author or publishing organization and converts the file to XML or SGML with a conversion application. The source document for such conversions is often a Microsoft Word file or a file that was used for typesetting in a system like Quark XPress. It is these software-based conversions that are the main focus of this paper.

§ Software-Based Markup Conversion

Software-based conversions are often very specialized systems that have been custom-developed for a specific vertical markup segment (e.g., financial publishing, scholarly publishing). They are sometimes developed for the specific requirements of a single DTD [Smith 1997].

Style-Based Markup Conversion

In their simplest form, these systems convert the available formatting information to XML markup. For example, Rich Geimer's RTF to XML Omnimark converter converts Microsoft Word documents in RTF format to XML [Geimer 2000]; [Microsoft Technical Support 2001]. The primary characteristics of this converter are:

- It converts existing Word paragraph styles to tags of the same name
- It converts existing Word character styles to tags of the same name
- It converts special characters in symbol font or Unicode to Unicode entities
- It converts Word tables to CALS tables

Converters like RTF2XML are valuable starting points; however, for DTDs that require additional granularity, they do not offer sufficient markup. To gain additional granularity in the final XML file, there are two options. The first option is to add markup by hand (either by adding character styles in the pre-conversion file or by hand-tagging the post-conversion file). The second option is to add markup based on heuristic-based pattern recognition.

Pattern-Based Markup Conversion

Pattern-based markup conversion uses software-based rules to add granularity to documents. It moves one step past style-based markup creation, because it relies on more than the correct manual application of paragraph and character styles. Pattern-based markup creation relies on regular expressions to examine the text for specific patterns and add markup based on exact matches of those patterns.

For example, the Extreme Markup DTD includes the elements acronym and expansion. The expected markup for the text "Standard Generalized Markup Language (SGML)" is:

```
<acronym.grp><acronym>SGML</acronym><expansion>Standard Generalized
Markup Language</expansion></acronym.grp>
```

If the editorial style guide for Extreme papers (should such a guide be created and published) stated that acronyms should appear spelled out in title case followed by the acronym in parentheses (as illustrated above), then a regular expression can be developed that will search for such cases and tag them accordingly.

With pattern-based markup, it is no longer necessary for an author, editor, or production person to mark up by hand granular content that follows a specific editorial style. Instead, markup is added automatically by the software.

Often, the quality of pattern-based markup creation can be improved by limiting regular expressions to a specific scope of the document. For example, it might be wise to limit a regular expression that searches for US or Canadian postal codes to text with an "Address" paragraph style, so as to avoid inadvertent markup of a similarly patterned device part number in the body of the document.

Pattern-based markup creation works for relatively simple situations. However, XML markup is often required in more complex situations where simple regular expressions are insufficient for accurate markup of content. For more complex situations, heuristic-based markup is required.

Heuristic-Based Markup

Heuristic-based markup also uses software-based pattern recognition to add granularity to documents as an alternative to hand-markup or simple regular expressions. However, the distinction we make is that

heuristic-based markup uses the most appropriate solution found after running the text through multiple patterns where each pattern uses alternative models. The result of each pattern is then used in successive stages of processing, so that by the end hundreds of patterns may have been tried, and the software automatically selects the best result from all of the attempted patterns. Heuristic-based markup is best used when the material to be tagged is complex and irregular and a high level of granularity in the resulting markup is required.

Heuristic-Based Examples

A good example of such irregular content is the list of references at the end of scholarly papers. Consider the following reference:

1. Viggeswarapu M, Boden SD, Liu Y. et al. Adenoviral delivery of LIM mineralization protein-1 induces new-bone formation in vitro and in vivo. *J Bone Joint Surg Am* 2001; 83-A(3): 364–76.

and the required XML markup:

```
<ref id="R1"><citation><person-group><name><surname>Viggeswarapu</
surname><given-names>M</given-names></name><name><surname>Boden</
surname><given-names>SD</given-names></name><name><surname>Liu</
surname><given-names>Y</given-names></name><etal>et al.</etal></person-
group><year>2001</year><article-title>Adenoviral delivery of LIM
mineralization protein-1 induces new-bone formation in vitro and in vivo.
</article-title><source>J Bone Joint Surg Am</source><volume>83-A</
volume><issue>3</issue><fpage>364</fpage><lpage>76</lpage></citation></
ref>
```

Using regular expressions, an application might demarcate the major parts of the reference by applying these heuristics:

- Author list ends with a period
- Journal name starts and ends with italic
- Year is followed by a semi-colon
- Volume and issue are followed by a colon
- Pages are at the end of the reference

Using such pattern-based rules, we can easily break down this and other references that follow the same pattern.

However, if the editorial style of the journal requires the text "in vitro" and "in vivo" to be italicized, the reference will look like this:

1. Viggeswarapu M, Boden SD, Liu Y. et al. Adenoviral delivery of LIM mineralization protein-1 induces new-bone formation *in vitro* and *in vivo*. *J Bone Joint Surg Am* 2001; 83-A(3): 364–76.

and the resulting XML markup based on the heuristic of starting the journal name when italic begins will be:

```
<ref id="R1"><citation><person-group><name><surname>Viggeswarapu</
surname><given-names>M</given-names></name><name><surname>Boden</
surname><given-names>SD</given-names></name><name><surname>Liu</
surname><given-names>Y</given-names></name><etal>et al.</etal></person-
group><year>2001</year><article-title>Adenoviral delivery of LIM
mineralization protein-1 induces new-bone formation</article-
title><source><italic>in vitro</italic> and <italic>in vivo</italic>. J
Bone Joint Surg Am</source><volume>83-A</volume><issue>3</
issue><fpage>364</fpage><lpage>76</lpage></citation></ref>
```

Needless to say, while valid according to the parser, the markup of `<source>` as:

```
<source><italic>in vitro</italic> and <italic>in vivo</italic>. J Bone
Joint Surg Am</source>
```

is far from correct.

In the specific discipline where this markup is applied, the misapplication of the `<article-title>` and `<source>` tags causes more than incorrect rendering. The incorrect markup also causes this reference to fail when attempts are made to link through online databases such as Pubmed or CrossRef, because both of these systems rely on an accurate journal name to complete a reference lookup. Because click-through linking of references is one of the more compelling reasons to create XML for scholarly research articles, incorrect markup undermines the one of the rationales for creating XML in the first place.

At this point, some readers may already be considering alternative patterns to solve this problem. One pattern that may spring to mind is to search for the first period after the end of the author list rather than the first instance of italic text. Such a change works quite handily for the example above, but it will fail for other references such as:

2. Baltzer AWA, Lattermann C, Whalen JD, Braunstein S, Robbins PD, Evans CH. A gene therapy approach to accelerating bone healing. Evaluation of gene expression in a New Zealand white rabbit model. *Knee Surg Sports Traumatol Arthrosc* 1999; 7: 197–202.

where the article title has two sentences. Use of the "period" pattern results in the following incorrect markup:

```
<ref id="R2"><citation><person-group><name><surname>Baltzer</
surname><given-names>AWA</given-names></name><name><surname>Lattermann</
surname><given-names>C</given-names></name><name><surname>Whalen</
surname><given-names>JD</given-names></name><name><surname>Braunstein</
surname><given-names>S</given-names></name><name><surname>Robbins</
surname><given-names>PD</given-names></name><name><surname>Evans</
surname><given-names>CH</given-names></name></person-group><year>1999</
year><article-title>A gene therapy approach to accelerating bone healing.
</article-title><source>Evaluation of gene expression in a New Zealand
white rabbit model. Knee Surg Sports Traumatol Arthrosc</
source><volume>7</volume><issue></issue><fpage>197</fpage><lpage>202</
lpage></citation></ref>
```

Problem Scope

Anyone who has worked to develop markup systems that use patterns or heuristic markup as illustrated above has come to appreciate the difficulty of finding patterns that are simple enough to solve the problem but complex enough to cover a wide range of possible cases.

What makes this problem even more difficult is that in almost all cases, the resulting XML file was deemed valid by the parser, even though the content between the tags was almost entirely useless!

No system that applies markup automatically can work perfectly in all cases. Mark Gross commented at SGML '96:

> A high volume conversion is never easy. Nor does it have to be done one page at a time by an SGML expert. The surest approach to achieving the lowest possible cost (while maintaining quality) is to find the best balance between automated conversion and manual review.
>
> If you plan carefully and consider the issues we've discussed, you can avoid most of the potential problems. And if you're using software, preventative measures can be programmed in, saving costly, repetitive clean-up tasks with a single tweak. [Gross 1996]

An automated conversion based on style, patterns, or heuristics can be used to achieve high-volume conversion. However, the "single tweak" can cause devastating damage rather then incremental improvement if there is no system to validate the change with a significant volume of content. Quality assurance is an essential part of an XML or SGML delivery; it is also a vital part of conversion systems delivery. "The costs for not doing QA, or doing it as a patch later, are much more frightening" [W3C 2004].

§ Quality Assurance

The first step in the development and deployment of automated conversion systems is to include a quality assurance pass on all tagged materials. The quality assurance pass must ensure more than the DTD-level validity of the file!

Parsers provide a first-order check on the quality of markup by ensuring that the markup is valid against the DTD. Back in the last century, this was often good enough. Kathryn Henniss of Highwire Press commented recently, "In 1999, we were happy just to receive files that parsed" (personal communication). However, as markup technologies have matured, we have learned that what's between the tags is more important than the syntax of the tags themselves.

Even with modern document definition meta-languages such as W3C XML Schema, parsers often cannot provide adequate validation of what's between the tags. Quality assurance processes can be enhanced with automated tools to highlight potential problems [Piez 2003].

However, automated tools are not a replacement for manual proofing. Manual proofing is still required to backstop the automated tools and to ensure that errors do not slip through between the tags.

XML Proofing

In the examples above, an automated quality assurance check might warn in the following situations:

- A journal name in <source> includes italic text
- A journal name in <source> includes words that start with lower case letters

Neither of these checks is absolutely perfect. Some journal names might include italic, and others certainly do include words that start with lower case letters, but the *probability* that either of these conditions is valid is sufficiently low that a warning generated by a post-parse tool can immediately highlight some cases where heuristic-based processing has failed.

In addition, these automated checks can be enhanced by use of a false-color proof [Piez 2003] during the manual check to rapidly search for patterns that look incorrect.

Feedback and Improvements

In a static system, the developers complete the code and deploy the system. The users of the system fix parsing errors as well as errors logged by automated proofing tools. The users conclude a file conversion with a manual proof of the file for any remaining errors.

This system works adequately, but not efficiently. Any system that includes quality assurance can only improve if the results of the checks are analyzed and used to enhance the quality of the overall system.

Let's take another look at our reference example from above. In the first case, the pattern that was employed worked for the example reference. The programmer who developed the pattern happily went home at the end of the day after testing this reference and perhaps a few others, content in the knowledge that his pattern would create the desired markup.

When the problem illustrated in the second example is shown to the programmer sometime (days, weeks, months?) later, he may well attempt to fix the problem by changing the pattern to use a period rather than italic markup. After testing this change on a few references, the programmer may commit the change to the source control system (if he's fortunate enough to be maintaining his code under source control) and then move back to the more pressing problems of the day.

This approach to problem resolution and system improvement is a recipe for disaster in pattern- and heuristic-based systems! The only way to ensure the quality and stability of the application is through rigorous regression testing.

Automated Regression Testing

It's a well-known truism that programmers cannot test their own code. Even the best programmers need a person or process to back them up and provide quality assurance. However, in many organizations, quality assurance efforts are limited (or even nonexistent) due to lack of time and/or resources. Quality assurance incurs extra cost, and many organizations are unwilling or unable to make that investment.

Traditionally, when quality assurance is done, it is conducted by passing the application "over the wall" to a quality assurance team. The team runs the application through its paces, reports any problems to be

fixed, and then regression checks the fixes. However, this process is labor-intensive and can be very expensive in a development project, such as heuristic-based XML conversion systems, where continual improvement is required.

XML conversion systems have an important feature that is not true of many other applications: For a specific input, there is a deterministic output. In other words, for a given input to the system, there should be only one output that best meets the desired output requirements.

Automated regression testing can take advantage of this feature by permitting continual and cost-effective retesting of specific inputs to ensure that the output requirements are still being met. We call the application of this procedure the Automated Quality Assurance Test Suite.

§ Automated Quality Assurance Test Suite

An automated quality assurance test suite provides automatic regression testing and quality validation for markup conversion systems at a very low cost. It can ensure the integrity of the system when it is deployed, and later updated and re-released for large-volume automatic conversion of documents.

The test suite should be run daily, *even if only one line of code has been changed in the application.* Because the test suite is automated and the only manual intervention is the analysis of the results, this approach provides a constant and cost-effective solution to large-scale quality assurance.

Building the Test Suite

When development starts on a new conversion, we are always thrilled at the first valid document instance that we generate. The very next thing we do is start a new set of test suite documents and add this file that parses.

Setting up the test suite requires two parts. The first part is collecting data files on which to run the test process. The second part is building a batch process that can run the application in "test" mode on a collection of data files.

Organizing Data Files

Adding the file to the test suite is relatively simple. The original file, before the conversion has been run, is added to a directory called "Input," and the final XML file, after the conversion has been run and the file has been proofed, is added to a directory called "Good." These "good" files are the master files against which the output of automated conversions will be compared. To ensure the integrity of these files, they are added to a source control system.

Each time the test suite is run, the "Input", "Output" and "Good" directories are deleted from the test system. The directories are recreated and clean copies the files in the "Input" and "Good" directories are retrieved from the source control system.

The Test Harness

Markup conversion processes are often written in languages such as Perl, Omnimark, or XSLT that can be called from the DOS or Unix command line. Automating the conversion of test documents in a command line environment is relatively easy. A batch file or shell script (the test harness) is written to enumerate all of the files in the "Input" directory and call the conversion process on each of them. The results of the conversion are placed in a directory called "Output."

If the conversion processes cannot be run from the command line, it may be necessary to add code to the application being tested that will allow batch processing of files for test purposes.

When all of the files have been run, the test harness executes a post-test script. This script compares each file in the "Output" directory with the copy that is in the "Good" directory. Any differences are reported in a log file. Figure 1 shows the test suite log after changing the heuristic for identifying the journal title from italic text to a period.

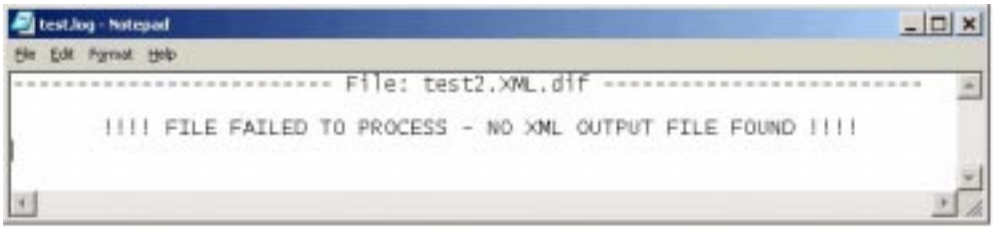
In addition, the test suite checks to ensure that each file in the "Input" directory has a corresponding file in the "Output" directory, because the authors have found cases when a change to the code may have caused a code crash during conversion, so that one or more files failed to convert. Without this additional check (Figure 2), complete failures of file conversion will not be noted.

The entire test suite is controlled by a single script. This script deletes all files from previous test suite runs, recreates the necessary directory structure, gets all of the latest code and data directly from the

Figure 1: Figure 1

Example of the log file from the test suite. This report shows the "Good" XML version above the three dashes, and the "Output" version below the three dashes. The report illustrates how changing the heuristic to identify the journal title has improved the result for the first reference but has adversely affected the result for the second reference. These reports can alert developers to problems with newly developed heuristics.

Figure 2: Figure 2



This log file illustrates the importance of checking that each file completed processing. Failure to check for the existence of an "Output" file corresponding to each "Input" file can mask fatal bugs that are introduced into a heuristic-based conversion process.

source control system, runs the test suite, and generates the log report. This script is designed to ensure a consistent test environment and the integrity of the test results.

Test Suite Expansion

As the project proceeds, more files are added to the test suite. Eventually, there may be hundreds or even thousands of files in the test suite.

However, there is an important shift in the nature of the files added after the initial phases of development. Files added early in development are carefully proofed to ensure that all markup is completely correct. Files added during later stages of development may only be proofed for one element related to a specific change in the code.

For example, if the rules to find the start of the journal name are changed based on a single test case that failed (as in the example above), the entire input file, rather than only the specific reference, may be added to the test suite to revalidate this one change that has been made in the application code. *Only the specific reference may have been proofed, but the entire file is added to the test suite.* It is not necessary

to proof the remainder of the file because the only part that is being explicitly revalidated on each run is the reference that has been proofed.

Test Suite Evaluation

Adding an unproofed file may seem illogical at first, but this use of the test suite provides one of the greatest benefits. When a file is added to provide a test case for a single element of the code, it may not be efficient (or necessary) to proof all other elements of the file. The only element of importance is the one in the file for which regression testing need be performed.

However, over time, as the code, its patterns, and its heuristics evolve, each of these new test files provide additional cases on which the processing is run. In some situations, certain infrequent (or pathological) patterns may appear in these files that provide additional validation of current code or of future improvements.

Let'sthink about this from the practical perspective for a moment, by once again considering our reference examples above. When the pattern recognition was changed to use a period rather than italic text to identify the journal title, a new problem appeared in the test suite as a result of the changed pattern. This new problem is illustrated in the second reference where the article title has multiple periods.

If a single reference exists among the hundreds of references in the test suite in which the article title has two periods, the markup of the resulting XML will differ from the markup where the italic pattern was used, and the difference will be reported in the test suite log.

Identifying these problems, often in content that had not previously been manually proofed, is one of the greatest strengths of the test suite. In this manner, large volumes of content can be automatically revalidated to ensure that no changes are made to the application that cause lower-quality results.

Test Suite Improvements

The most important use of the test suite is to ensure the overall stability of the application. The test suite is used to guarantee that code that has been proven to work correctly is not altered in such a way that one or more instances fail to produce correct results.

However, the nature of heuristic-based systems is such that general improvements to the heuristics can provide broad-based improvements to the overall system. The inclusion of otherwise unproofed material in the test suite can also aid in validating this type of system improvement.

To illustrate this point, let'stake another look at our reference examples. When italic text was used to identify the journal title, the code worked in cases that had been proofed. When a problem was identified in a new case and the programmer updated the application to use a different pattern (in our example, a period), he used the test suite to check that the change more generally improved the problem he was trying to fix. Because the test suite included the following unproofed reference:

1. Viggeswarapu M, Boden SD, Liu Y. et al. Adenoviral delivery of LIM mineralization protein-1 induces new-bone formation *in vitro* and *in vivo*. *J Bone Joint Surg Am* 2001; 83-A(3): 364–76.

the change to a period caused a difference in the file created in the Output directory. When the test suite comparison was run, this discrepancy was noted in the log file. Manual evaluation of each difference can determine if it is a new problem resulting from the change that requires further attention by the developer (as it is in the example), or if it is an improvement to the heuristic-based system.

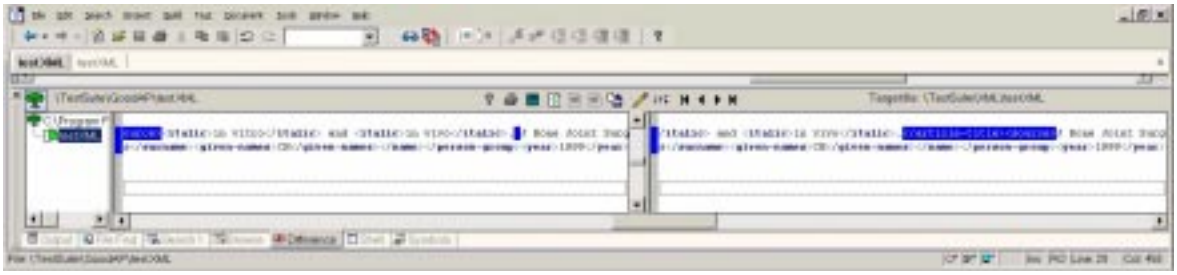
If the change is determined to be an improvement in the overall system, the better version of the XML file replaces the existing one in the "Good" directory, and this improved version is checked into the source control system. In this manner, the test suite is used to identify and "lock in" improvements to the overall system so that they are not lost during future code updates.

As shown here, the test suite is used not just to maintain stability—it also provides a platform for validating the quality of general improvements to the entire conversion system.

Differencing Techniques

Rapid evaluation of the test suite is a key component to using it effectively as a development aid. By enhancing the view of differences reported in the log file, review of the test suite can be completed faster, allowing more time to be spent on development.

Figure 3: Figure 3



Differencing of files is sometimes more readable in a color-coded differencing window than in the output from GNU Diff (Figure 11). This image shows the same two files being compared in Borland's CodeWright.

Initially, the difference log was the immediate output from GNU Diff. However, when lines of XML content exceeded the width of the screen, the log files could be difficult to read. As an improvement over the initial log, a three-step difference was created:

- The Output and Good files are compared
- If a difference is found, the Output and Good files are reformatted into temporary copies with line breaks approximately every 80 characters. Care is taken to break lines only at white space, and not inside a tag except between attributes
- The reformatted file copies are compared. Figure 1 illustrates the difference after reformatting to 80-character lines

In some cases where the lines are very long or the change is very subtle, it can still be difficult to review the log results. In such cases, a text editor with color-coded differencing is used to compare the files. Figure 3 shows the screen used in a text editor that features comparisons of two directories of files at once.

Additional Test Suite Uses

Automated test suites can be used to maintain quality in applications besides markup conversion. For example, a test suite can be used to check the results of deterministic search engine results (i.e., those results that should remain constant given a fixed set of input data).

In one test case, the authors run a daily test of the CrossRef DOI lookup database. The test uses a fixed set of input data. In this situation, both the input and output data are non-XML streams. But because they are both text files, the streams lend themselves to the same kind of automatic comparison after running the test application.

When testing CrossRef, the results of the lookup should not change with each daily test, even when CrossRef fuzzy matching is turned on. However, one morning several queries were found to have changed. After CrossRef was contacted, the source of the change was found to be incorrect redeposit of several thousand records by a publisher in which the given names and surnames of the authors had been inadvertently swapped. Without automated testing, deposit of incorrect data may have gone undetected for many months, and searches conducted by end users may have yielded incorrect results.

§ Conclusions

The stability of XML conversion system applications should be maintained by regularly conducted regression testing. Especially when tags are applied by heuristic applications, retesting of any changes to the system should be completed to ensure that new heuristics do not adversely affect existing cases.

A large volume of document instances provides the basis for broad and robust system coverage testing. However, manual re-conversion of a large number of documents on a frequent basis to revalidate the system is time-consuming and expensive.

The use of an automated test suite permits fast and expansive coverage of a large number of test cases in a relatively cost-effective manner to ensure the overall stability of the conversion application. The automated test suite also provides a platform for validating the quality of general improvements to the entire conversion system.

Acknowledgements

We are grateful to Igor Kleshchevich for insisting that we build automated quality assurance systems and use them daily. The authors would like to thank Igor Kleshchevich and Matthew Schreiner for their valuable contributions in heuristic-based processing, and Casey Bell and Liz Blake for finding the bugs that the automated systems did not.

Bibliography

[Bide 2000] Bide, M. 2000. Standards for Electronic Publishing. **http://www.kb.nl/coop/nedlib/results/e-publishingstandards.pdf**. Accessed April 15, 2004.

[Geimer 2000] Geimer, R. 2000. RTF2XML converter. **http://www.xmeta.com/omlette/rtf2xml/**. Accessed April 15, 2004.

[Gross 1996] Gross, M., and Zirilli, D. 1996. The Real Costs of Conversions to SGML: Myth vs. Reality. Proceedings of the SGML '96 Conference. Graphic Communications Association, Alexandria, VA.

[Microsoft Technical Support 2001] Microsoft Technical Support. 2001. Rich Text Format (RTF) Specification Version 1.7. **http://support.microsoft.com/default.aspx?scid=http://support.microsoft.com:80/support/kb/articles/q86/9/99.asp&NoWebContent=1**. Accessed April 15, 2004

[Piez 2003] Piez, W. 2003. XSLT for Quality Checking in a Publication Workflow. **http://www.idealliance.org/papers/dx_xml03/papers/04-04-02/04-04-02.pdf**. Accessed April 15, 2004.

[Rosenblum 2002] Rosenblum, B.D., and Golfman, I. 2002. A Decade of DTDs and SGML in Scholarly Publishing. **../../2002/Rosenblum01/EML2002Rosenblum01.html**. Accessed April 15, 2004.

[Smith 1997] Smith, N.E. 1997. Practical Guide to SGML Filters. Wordware Publishing, Inc. Plano, Texas.

[W3C 2004] W3C. 2004. Quality Assurance Activity Statement. **http://www.w3.org/QA/Activity**. Accessed April 15, 2004

The Authors

Bruce Rosenblum
Inera Incorporated
815 Washington Street, Suite 3,
Newton
MA
02460
USA
tel: (617) 969-3053
fax: (617) 969-4911
bruce@inera.com
http://www.inera.com

Bruce Rosenblum, CEO of Inera, has spent more than twenty years designing and implementing electronic publishing solutions. He consults on the application of XML in publishing and the design of electronic production workflows. As a system designer, he co-authored the new NLM journal and archive DTDs and the CrossRef Metadata Deposit Schema. At Inera, Mr. Rosenblum leads the design and development of *eXtyles*, Inera's integrated suite of editorial and XML tools for Microsoft Word, which automates editorial and XML production processes and is used in the production of more than 300 journals worldwide. Prior to joining Inera, Mr. Rosenblum was Vice President at Turning Point Software, where he led the design and development of software products for companies such as Microsoft, Word Perfect, and Houghton Mifflin.

Irina Golfman
Inera Incorporated
815 Washington Street, Suite 3,
Newton
MA
02460
USA
tel: (617) 969-3053
fax: (617) 969-4911
irina@inera.com
http://www.inera.com

Irina Golfman founded Inera Incorporated in 1992 to provide SGML-related consulting services. Under her guidance, Inera has successfully completed SGML- and XML-based projects for clients in the printing, publishing, manufacturing, computer, and financial services industries. Prior to founding Inera, Ms Golfman was Director of Product Development for Kurzweil Computer Products, a division of Xerox. In that role, she managed the design and development of OCR/ICR products for Macintosh, Windows, and Unix. Before joining Xerox, Ms. Golfman held positions in software development at Wang Laboratories, Linkware, Codex Corporation, and Prime Computer.

Extreme Markup Languages 2004®
Montréal, Québec, August 2-6, 2004
This paper was formatted from XML source via XSL
by Mulberry Technologies, Inc.