Architectures in an XML World

Joshua Lubell
National Institute of Standards and Technology
email lubell@nist.gov

ABSTRACT

XML (Extensible Markup Language) developers have at their disposal a variety of tools for achieving schema reuse. An often-overlooked reuse method is the specification of architectures for creating and processing data. Experience with APEX, an architecture processing tool implemented using XSLT (Extensible Style Language Transformations), demonstrates that architectures can fulfill a role not well served by alternative approaches to reuse.

KEYWORDS

architecture; architectures; architectural forms; XSLT; APEX; XML; architecture software; schema reuse; reuse

INTRODUCTION

Developers of markup languages have long recognized the importance of reuse. Since the early days of SGML (Standard Generalized Markup Language) [SGML], authors of DTDs (Document Type Definitions) have used parameter entities to help make markup declarations more reusable. Newer approaches to reuse run the gamut from the relatively simple concept of namespaces [Names] to more sophisticated methods such as the facilities available in the W3C's (World Wide Web Consortium's) XML Schema [XSchema] specification. As a result, XML developers today have at their disposal a variety of tools for achieving reuse.

An often-overlooked reuse method is the specification of *architectures* [AFDR] [Kimber] [Megginson] for creating and processing data. Architectures, alternatively referred to as "architectural forms" or "inheritable information architectures," have been around since the mid-1990s. Although the architecture mechanism's invention predates the standardization of XML, architectures are still being used today — most notably in the ISO Topic Maps standard [TM] and in the W3C's XML Linking specification [XLink].

In this paper, I briefly describe the architecture mechanism. Next, I discuss APEX (Architectural Processor Employing XSLT), a tool implemented using XSLT (Extensible Style Language Transformations) [XSLT] for processing architectures. I conclude by discussing how architectures compare with some alternative reuse techniques.

ABOUT ARCHITECTURES

Within the context of markup languages, an architecture is a collection of rules for creating and processing a class of documents. Architectures allow applications to:

- Extend XML vocabularies without breaking existing applications.
- Create architecture-specific document views, retaining only relevant markup and character data while hiding all other content.
- Promote data sharing between user communities with inconsistent terminologies by enabling the substitution of identifier names and by allowing simple document transformations.

Architectural Forms and Architecture Support Attributes

Unlike a grammar, which defines every aspect of representation and processing for a class of documents, an architecture need not specify a complete document type. Instead, an architecture defines rules known as *architectural forms* that application designers can apply in defining their XML vocabularies.

An XML document using an architecture contains special attributes, called *architecture support attributes*, describing how its elements, attributes, and data correspond to their counterparts in the architecture (which are governed by the architecture's architectural forms). Because architecture support attribute values are usually invariant for all documents throughout an XML vocabulary, these attributes can be given default values. Hence, it is easy to hide a document's use of architectures from architecture-unaware software tools as well as from humans viewing or editing the data.

Architectural Processing

Software tools for processing architectures are called *architecture engines*. An architecture engine may be specific to a particular architecture, or it may be generic (able to process any architecture).

An architecture engine draws upon the following sources of information to process an XML document:

- Instructions specifying the architecture being processed and which attributes in the document are the architecture support attributes.
- The document's architecture support attribute values.
- Syntax rules for the architecture itself. For a generic architecture engine, these rules might be in the form of a DTD, XML schema, or some other formalism for specifying syntax rules. For an architecture-specific engine, these rules could be hard-coded into the engine itself. Some syntax rules can even be implied using support attributes alone, making it possible in some cases for a generic architecture engine to process an architecture without reading the architecture's DTD or schema.

Example of Basic Architecture Processing

Consider a simple architecture called **inv** for inventory processing. Suppose that software exists for processing data structured according to **inv**'s syntax. Assume that the following markup declarations define **inv**'s syntax. Although I use DTD syntax to define **inv**, I could have instead used some other syntax such as a non-DTD XML schema language.

```
<!ELEMENT  item        (name, price, quantity)
                                                  >
<!ATTLIST  item
             id        ID
                                          #REQUIRED >
<!ELEMENT  name        (#PCDATA)
                                                  >
<!ELEMENT  price       (#PCDATA)
                                                  >
<!ELEMENT  quantity    (#PCDATA)
                                                  >
```

Now suppose I want to create some XML data consisting of reproductions of works of art that I have, with each work of art having a unique identifier, title, artist, price, and quan-

tity of reproductions on hand. Assume I have three copies of a painting, "Leapin' Lizards," painted by "El Gecko," with each copy selling for $15. This data can be represented as:

```
<art id="a1">
  <title>Leapin' Lizards</title>
  <artist>El Gecko</artist>
  <price>15</price>
  <quantity>3</quantity>
</art>
```

The following table shows the correspondence between the elements in my data and **inv**'s architectural elements:

element	corresponding architectural element
art	item
title	name
artist	[no corresponding element]
price	price
quantity	quantity

In order to process my data using the software that already exists for inventory processing, I add a *form attribute* to my data. The form attribute is an architecture support attribute whose purpose is to provide the architecture engine with the information in the table above. My form attribute has the same name, **inv**, as the architecture name. With the form attribute added, the data for "Leapin' Lizards" looks like this:

```
<art id="a1" inv="item">
  <title inv="name">Leapin' Lizards</title>
  <artist>El Gecko</artist>
  <price inv="price">15</price>
  <quantity inv="quantity">3</quantity>
</art>
```

Although architecture support attributes add complexity to the data, hiding the complexity is easy. Because the form attribute values for the <art>, <title>, <price>, and <quantity> elements are the same for all works of art, these attribute values can be specified as defaults. Thus, the form attributes can be hidden from any architecture-unaware software tool. For example, suppose I had the following DTD with system identifier "art.dtd":

```
<!ELEMENT  art         (title, artist, price, quantity)
                                                      >
<!ATTLIST  art
             inv       NMTOKEN                    #FIXED
                                                    "item"
             id        ID
                                                 #REQUIRED >
<!ELEMENT  title       (#PCDATA)
                                                      >
<!ATTLIST  title
             inv       NMTOKEN                    #FIXED
                                                    "name" >
<!ELEMENT  artist      (#PCDATA)
                                                      >
<!ELEMENT  price       (#PCDATA)
                                                      >
<!ATTLIST  price
             inv       NMTOKEN                   #FIXED
                                                   "price" >
<!ELEMENT  quantity    (#PCDATA)
                                                      >
<!ATTLIST  quantity
             inv       NMTOKEN                 #FIXED
                                                "quantity" >
```

Then I could specify the "Leapin' Lizards" data as:

```
<!DOCTYPE art SYSTEM "art.dtd">
<art id="a1">
  <title>Leapin' Lizards</title>
  <artist>El Gecko</artist>
  <price>15</price>
  <quantity>3</quantity>
</art>
```

Now suppose I tell an architecture engine to process my data using the **inv** architecture. The architecture engine should produce as output the following *architectural document* containing only the markup and data defined by **inv**:

```
<item id="a1">
  <name>Leapin' Lizards</name>
  <price>15</price>
  <quantity>3</quantity>
</item>
```

The architecture engine replaces each element from my data with its corresponding architectural element. The <artist> element is not processed because nothing in the architecture corresponds to it. If the architecture engine were a validating architecture engine, then it could also determine whether my data is valid with respect to **inv**'s DTD or schema.

The preceding example showed only the most rudimentary capabilities of architectures. Other possibilities include, but are not limited to:

- Renaming attributes;
- Selectively ignoring markup and/or content during architecture processing;
- Specifying and processing a document using multiple architectures.

ARCHITECTURES AND APEX

APEX is a non-validating generic architecture engine written in XSLT. The APEX XSLT stylesheet is available as part of the XSLToolbox [XSLToolbox], a collection of XSLT stylesheets available from NIST. APEX implements a simple but useful subset of the AFDR (Architectural Form Definition Requirements) specified in Annex A.3 of ISO/IEC 10744:1997. APEX behaves similarly to David Megginson's XAF package for Java[1] [XAF] and differs from the AFDR in the same ways as XAF. Unlike other architecture engines, which use XML processing instruction syntax to specify architecture usage and control information, APEX obtains this information through XSLT stylesheet parameters.[2] Thus input to APEX consists of an XML document plus stylesheet parameters for identifying the document's architecture support attributes and for controlling architectural processing. APEX produces as output an architectural document conforming to the architecture specified by the stylesheet parameters and the input document's architecture support attributes.

The following picture shows how APEX can be deployed to enable my artwork data from the example in section Example of Basic Architecture Processing to be processed using software supporting the **inv** inventory architecture. APEX's input is:

- The artwork data augmented with architecture support attributes. The architecture support attributes may either be explicitly specified in the data, or they may be specified as defaults in a DTD or schema.
- Stylesheet parameters directing APEX to process the data using **inv**.

APEX's output is data that can be fed to an inventory processing application. The inventory processing application need not be capable of processing artwork data. All it needs to know about are **inv**'s syntax rules. As the picture shows, **inv** is the "glue" that holds everything together. The inventory architecture describes the inventory processing application's information requirements. The inventory architecture also influences the artwork data in that the data has to be derivable from **inv** using the data's architecture support attributes.

1. Java is identified in order to adequately specify David Megginson's XAF software. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that Java is necessarily the best programming language available for the purpose.
2. If a processing instruction were used to supply this information, then APEX would need to parse the processing instruction's string value. Since XSLT processors do not parse this string value, APEX would have to be augmented with (non-XSLT) programming language code. Passing architecture usage and control information through stylesheet parameters is therefore a more sensible approach.

Using APEX

Since XSLT provides no standard syntax for specifying stylesheet parameters, APEX's architecture usage syntax is XSLT processor-dependent. To understand how this affects the use of APEX, assume that APEX uses a fictitious XSLT processor called **xslt** whose command line syntax is as follows:

```
xslt xml-document stylesheet [|parameter=value ...]
```

To process my artwork data using the **inv** architecture, I use two parameters: **name** for the name of the architecture being processed (**inv**), and **auto** to specify how elements in the data are associated with elements in the architecture. I specify "nArcAuto" as the value for **auto**. This tells APEX not to automatically associate elements, i.e. not to process an element unless the element has a form attribute. The resulting XSLT processor invocation is:

```
xslt art.xml apex.xsl name=inv auto=nArcAuto
```

A more complete discussion of APEX's usage and behavior is available as part of the documentation in the XSLToolbox distribution.

Customizing APEX

I mentioned back in section Architectural Processing that it is possible to perform some architectural processing using architecture support attributes alone and without any a priori syntactic knowledge of the architecture itself. In these cases, a generic architecture engine can process a document with respect to an architecture in the absence of any formal specification of the architecture's syntax rules. APEX, which does not use such syntax rules, is such an architecture engine. Although this limits APEX's capabilities somewhat, APEX's ease of customization helps to mitigate this limitation. In fact, APEX's lack of dependence on any particular representation method for syntax rules (such as DTDs) can be viewed as an *advantage* because architectures processed by APEX are free to specify their syntax rules using any schema language they want.

Because APEX is written in XSLT instead of in a programming language, APEX's functionality is easy to extend using XSLT's <xsl:import> or <xsl:include> elements. Thus adding transformation capabilities to APEX that are supported by XSLT but not by the Architectural Form Definition Requirements is simple. For example, suppose I want to create an **inv** architectural view of my artwork data that retains the content of the <artist> element such that "Leapin' Lizards" appears as follows:

```
<item id="a1">
  <name>Leapin' Lizards, artist: El Gecko</name>
  <price>15</price>
  <quantity>3</quantity>
</item>
```

Augmenting APEX with a template rule that adds the

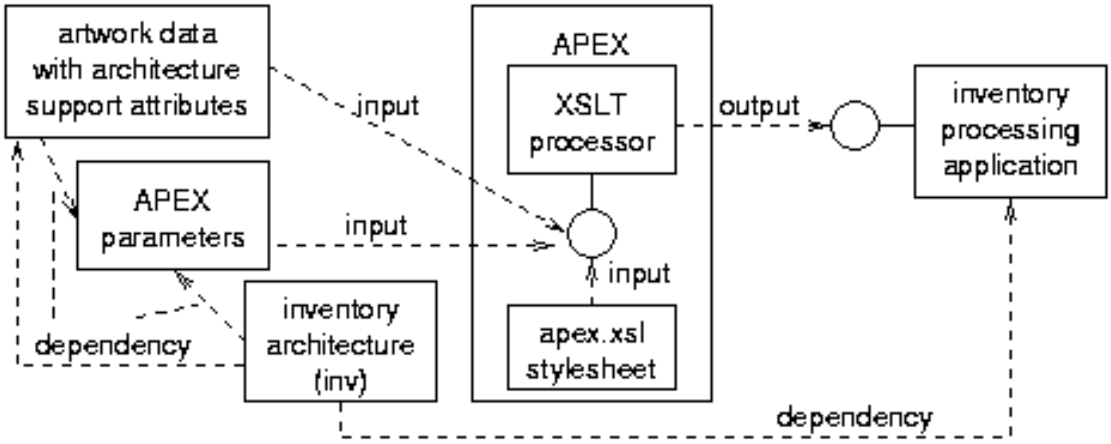

Figure 1

<artist> element's content to the <title> element's content can do this. The following XSLT stylesheet accomplishes the customization:

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/
Transform">
  <xsl:include href="apex.xsl"/>
  <xsl:template match="art/title">
    <xsl:element name="name">
      <xsl:value-of select="."/>
      <xsl:text>, artist: </xsl:text>
      <xsl:value-of select="../artist"/>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

ARCHITECTURES VERSUS OTHER REUSE METHODS

Architectures are among the oldest of several DTD/schema reuse methods available to XML developers. This raises the question of whether newer methods for achieving reusability make architectures obsolete? To answer this question, consider two contemporary XML technologies: W3C XML Schema and XSLT.

W3C XML Schema has several features designed to promote reusability. The <any> element enables schemas to allow embedded XML belonging to a foreign namespace. Type derivation by extension or restriction enables reuse through inheritance. Substitution groups permit elements to be substituted for other elements, allowing elements to be used interchangeably. As W3C XML Schema practitioners gain more experience [Costello], they might discover that these features can duplicate the benefits of architectures. However, even if using architectures were to add no value to W3C XML Schemas, architectures would still be worthwhile for applications not using W3C XML Schema. Because architecture processing is attribute-driven rather than schema-driven, architectures are compatible with any XML application, regardless of the schema language used.

XSLT, a language for transforming one XML document into another XML document, lets developers specify conversions between different XML vocabularies. XSLT is also handy for solving the common systems integration problem where XML documents almost but not quite conform to a given vocabulary. Although XSLT has considerable power, stylesheets that perform non-trivial transformation can be quite complex, and writing XSLT is often time-consuming. Thus, having to write an XSLT transform every time two systems need to talk to one another is a less than satisfying way to achieve interoperability.

Although architectures, like XSLT, can be used for transformation, the architecture mechanism also allows for validation and architecture-specific processing (although APEX does not support these capabilities). Further, XSLT transforms are specified differently than architectural mappings. In XSLT, mappings are specified algorithmically. With architectures, however, a developer need only formally state the conformance requirement. Also, an XSLT stylesheet (unlike an architecture's support attributes) is completely separate from the schema and data, making it potentially difficult to keep an XSLT stylesheet in sync with the vocabulary it is supposed to transform.

As the implementation of APEX in XSLT demonstrates, architectures and XSLT are complementary. Although the transformations architectures allow are more limited than those possible with XSLT, there is no guarantee in the general case that an XSLT transformation result is valid with respect to an intended vocabulary. Also, the verbosity and complexity of XSLT syntax makes it impractical to write an XSLT transform that could have been specified more succinctly using architecture support attributes. When used together though,

architectures and XSLT allow developers to have the best of both worlds.

I wish to thank Simon Frechette, Don Libes, Sandy Ressler, and the Extreme Markup Languages peer reviewers for their helpful feedback and suggestions for improving an earlier draft of this paper. I am also grateful to NIST's Systems Integration for Manufacturing Applications program (http://www.nist.gov/sima) and Advanced Technology Program (http://atp.nist.gov) for funding this work.

BIBLIOGRAPHY

[**SGML**] ISO 8879:1986. *Information processing—Text and office systems—Standard Generalized Markup Language (SGML)*.

[**Names**] World Wide Web Consortium. *Namespaces in XML*. W3C Recommendation 14 January 1999. See http://www.w3.org/TR/REC-xml-names.

[**XSchema**] World Wide Web Consortium. *XML Schema Part 1: Structures*. W3C Recommendation 2 May 2001. See http://www.w3.org/TR/xmlschema-1.

[**AFDR**] ISO/IEC 10744:1997. *Information processing—Time-based Structuring Language (HyTime)—2d edition*. Annex A.3 Architectural Form Definition Requirements (AFDR). See http://www.ornl.gov/sgml/wg8/docs/n1920/.

[**Kimber**] W. Eliot Kimber. *A Tutorial Introduction to SGML Architectures*. ISOGEN International Corp. See http://www.isogen.com/papers/archintro.html.

[**Megginson**] David Megginson. *Structuring XML Documents*. Prentice Hall. 1998. Chapters 9–11.

[**TM**] ISO/IEC 13250:2000. *Topic Maps: Information Technology—Description and Markup Languages*. See http://www.y12.doe.gov/sgml/sc34/document/0129.pdf.

[**XLink**] World Wide Web Consortium. *XML Linking Language (XLink) Version 1.0*. W3C Proposed Recommendation 20 December 2000. See http://www.w3.org/TR/xlink/.

[**XSLT**] World Wide Web Consortium. *XSL Transformations (XSLT) Version 1.0*. W3C Recommendation 16 November 1999. See http://www.w3.org/TR/xslt.

[**XSLToolbox**] Joshua Lubell. XSLToolbox package. See http://www.nist.gov/xsltoolbox.

[**XAF**] David Megginson. XAF package for Java. See http://megginson.com/XAF/.

[**Costello**] Roger Costello, ed. *XML Schemas: Best Practices (A Collectively Developed Set of Schema Design Guidelines)*. See http://www.xfront.com.

BIOGRAPHY

Josh Lubell is a computer scientist in NIST's Manufacturing Systems Integration Division, where he design and implement software systems for product data exchange applications. His technical interests include markup languages, database technology, and Internet computing. His previous experience includes artificial intelligence systems design and prototyping, software development for the building materials industry, and knowledge engineering. He has an M.S. in computer science from the University of Maryland at College Park, where he performed graduate research in diagnostic problem solving, and a B.S. in mathematics from Binghamton University.