Model-Driven Programming Using XSLT

WRITTEN BY SOUMEN SARKAR

AN APPROACH TO RAPID DEVELOPMENT OF DOMAIN-SPECIFIC PROGRAM GENERATORS

Model-driven programming is a software development paradigm that strives to bring out the abstract model manipulation that we're trying to achieve through a body of programming language code. This approach focuses first on *what* is being achieved in a computing system and then on *how* it's being achieved. It's the responsibility of a software subsystem to translate the *what* to *how*.

Typically, the purpose of the software subsystem is to generate a concrete implementation from declarative models. This could be viewed as an extension of MVC (Model-View-Controller) architecture by incorporating a generator component (i.e., MVCG).

Adopting a generative approach in software development is a goal cherished by many application developers. Why write repetitive code when a single source of specification can generate the needed variations? Generic programming partially addresses this by allowing us to write template-based code in which the variability is expressed in the template parameter. For example, in C++ Standard Template Library (STL), the concepts of collections, iterators, and algorithms operating over iterators are all implemented in terms of parametric types T_i. When we use STL with a concrete type such as a C++ class, the code is generated by a C++ compiler/linker. In software development projects it's desirable to replace repetitive coding with some form of automation, and many times we see a need to achieve this. Custom program generators could replace the volume of manually written code by generating code from domain-specific models. Less manually written code is a good thing.

Plenty of software products around us apply the concept of model-driven development. Consider, for example, a professional HTML editor. The user interface assists the user in developing the Web site through a look-and-

feel visual approach. The editor will generate HTML, JavaScript, JSP/ASP, JDBC programs, and other software artifacts based on the model of the Web site that the user has built through the highly intuitive visual interface. However, these products are built to perform specific tasks that may not be of use in general software development. My point is that there's a dearth of tools to facilitate the rapid translation of domain-specific models into desired programming language constructs.

Compiler-compilers (CC) are tools that help in the building of program generators. The input specification to CC is typically language grammar. Based on the input syntax, the CC generates a parsing/lexing framework, which then builds a custom program generator. You can build a C compiler by using a CC. In a CC-based code generation approach, the focus on code generation logic is lost due to the distraction of building various infrastructures, such as an abstract syntax tree (AST), syntax and semantic validation, and so on. A rapid approach to program generator development needs tools supporting scripting capabilities with reference to parsing and syntax tree building/visiting. We don't want to spend time and resources in program generator acquisition. Since code generation invariably involves tree walking, the entire focus should be on visiting the syntax tree of the model and generating the output syntax tree.

One view of XML is that it's a metalanguage used to define other domain-specific languages. XML tags, attributes, and namespaces could be used to capture models of application domain concepts. XML model syntactic validity can be checked by tools that enforce XML schema. In-memory representation of XML documents is standardized with the DOM (Document Object Model).

With XSLT you have scripted access to the XML DOM. The XSLT processor parses the XML document to construct the DOM, and XSLT scripts manipulate the DOM input to construct the DOM output. It's possible to construct plaintext source code files from the DOM output. The plaintext output forms one to many source code files.

These facts make the source code generation task extremely easy since the tasks of parsing, validating, and generating code can all be performed with freely available tools in the manner of rapid scripting-based development. Development of a visual modeling interface may require significant effort, but it's not a must-have. Plaintext editors can be used to capture domain-specific models in XML.

In a source code generation scheme a central concern is how to integrate generated code with manually written code. Typically, a framework is generated that is then customized manually. Regeneration caused by model change shouldn't wipe out past customizations. In other words, there should be a clear physical separation of custom code and generated code.

I've found that the inheritance and encapsulation mechanisms of object-oriented languages are a neat way to address this separation concern. More specifically, manually written code could inherit or use generated base classes or interfaces. In the other direction, generated code could also inherit or use handwritten classes. A generated code module/package is kept separate from manually developed modules/packages. In procedural languages the same effect could be achieved by effectively mimicking OO languages. It isn't good practice to mix generated text with handwritten text physically since this approach is brittle from an automation standpoint. Code generation examples in this article are in Java.

Though we're focusing on source code generation in this article, many kinds of documents can be generated out of XML models. For example, from the XML object relational model of domain entities you can generate a PDF file capturing the model in pictorial form. The notable point is that plaintext that can be used for various purposes is generated, source code being one of them.

This article documents experience gained in the application of a model-driven programming ideal in software development projects. To achieve the ideal in a given application domain, we must provide three things: (1) a modeling language specific to that domain, (2) a visual tool for building models in that language, and (3) automatic code generation from models in that language to the appropriate implementation code. Software projects would like to have a model-driven program generator without investing significantly in the program generator acquisition. This article demonstrates an approach to rapid development of domain-specific program generators by using XML as the domain modeling language and XML Stylesheet Language Transformations (XSLT) as a syntax-tree scripting mechanism.

The benefits of domain model–based software development are well known; they include a higher level of development abstraction, solid consistency, resiliency to behavior changes, rapid application prototyping, and increased focus on application logic development.

In my opinion, complexities in the current approach to source code generation have impeded widespread adoption of the generative approach in software development projects. For a reasonably sophisticated source code generation scheme, the learning curve to achieve XML/XSLT-based source code generation is appreciably less compared to the benefits obtained. This article shows how freely available XSLT processors supporting the W3C standard can be used as a custom program generator in many software development projects.

Our example is drawn from a real-world software project (refer to "Application in a Software Development Project" section) in which 60% of the source code is generated. The purpose of the project was to develop a management system for a large network of telecommunication equipment. The project used source code generation extensively in the areas of EJB, SNMP-based network management, Java helper utilities, and the SQL schema for relational databases. Though the project didn't call for it, other kinds of documents such as SVG pictures or PDF documents could have been generated from the XML models the project developed.

The remainder of this article discusses the following four topics:

1. Source code generation by XML/XSLT compared to the CC approach
2. A simple example showing the details of source code generation step by step
3. An application in a software development project
4. Things to be aware of

While the XML/XSLT-based code generation approach is easy enough for it to be considered in many projects, it isn't a general approach. There are some limitations. The third item mentioned above addresses this and puts the applicability of the current XML transformation approach into proper perspective.

Code Generation via XML/XSLT vs CC-Based Approaches

Code generation processing generally consists of a number of distinct phases, as shown in Figure 1.

Figure 2 shows the building blocks in CC (like Yacc, JavaCC) based program generators. The input to the CC is a grammar specification tagged with target language code (like Java/C++) for semantic actions and values. Typically, complementary tools have to be used in conjunction for tree building and semantic analysis. A lot of glue code has to be written to integrate output generated by a CC to produce the program generator. If the grammar allows things to be referenced before the declaration is encountered, syntactic checks must be replaced with semantic checks; thus a CC's value as a development tool is reduced.

The logic for generating output from the in-memory representation of input is some sort of tree walk processing. The output may also be produced in tree form in memory before being written in source code in one or more files. The output has to conform to the target language grammar; otherwise target language compilation will fail. The target language compilation errors will generally provide feedback to correct source code generation in the iterative development process.

Examination of the CC-based source code generation process reveals a number of things not central to the objective:

- The need for an internal abstract syntax tree
- The need for glue code and data structure
- The need to build the program generator first

The central processing step of code generation logic is tree walking. The key to rapid development of a source code generator is to have some kind of scripting capability for tree walking. Everything else is there to support these steps. With this background we enter the XML/XSLT-based code generation scheme. Code generation using XML/XSLT requires the following abilities on the part of the program generator developer:

1. The ability to define the domain specifications in XML, which involves the determination of XML elements, XML element attribute names, values, and the nesting structure
2. The ability to visualize the tree that will result from the input XML documents containing domain specifications
3. The ability to use XSLT and XPATH facilities to browse through the XML tree and generate output text that gives rise to one or more source code files

Please note that the XML syntax for the domain modeling language is definitely inferior to a natural language syntax that could be supported in the CC-based approach. However, in a software development project this may not be an issue. On the other hand, the gains derived from adopting XML are many. With the input language in XML, parsing and in-memory representation aren't a development concern since this part is supported by the XSLT processor.

The developer writes code generation logic in terms of the XSLT and XPATH language facilities (see Figure 3). XPATH is a language that specifies how to denote the expression accessing the in-memory tree representation. XSLT is the specification for transforming the input tree parts selected by XPATH. The result is that through XSLT/XPATH scripts the developer is constructing an output tree. When serialized to plaintext this output tree gives rise to one or more source code files. The compile correctness of the generated output files provides feedback to correct the source code generation scheme in an iterative fashion. Tree-oriented assertion statements may validate the input.

It's worthwhile to note that in an XML/XSLT-based approach:

- Building a syntax tree isn't a concern.
- The program generator logic is nothing but a collection of XSLT/XPATH scripts that are interpreted by an XSLT processor.

Scripting means rapid development. Having this capability to browse a syntax tree means rapid development of program generators. Thus it becomes extremely easy to change code generation logic or to accommodate a structural change in the input language. Table 1 summarizes the benefits obtained.

Steps to Generate Source Code

This section shows the steps involved in XML/XSLT-based source code generation with a simple contrived example. The purpose is to highlight the steps while omitting irrelevant details. Let's consider Java code generation. Java doesn't have an enumeration type. A project using Java may use the following XML specification to generate Java utility classes having enumeration functionality:

```
<EnumDef name = 'DayOfWeek'>
    <choices>
        <choice name = 'SUNDAY'
          value = '0' />
        <choice name = 'MONDAY'
          value = '1' >
        ……
        <choice name = 'SATURDAY'
          value = '6' />
    </choices>
</EnumDef>
```

This specification is intuitive enough to convey the fact that DayOfWeek is an enumerated type with seven distinct values. It also shows that the process of defining a domain-specific language in XML consists of defining the markup elements and element attributes. Where domain language elements clash with preexisting XML elements, the XML namespace facility should be used to distinguish the domain language elements. The set of XML namespace, elements, and tag attributes defines the XML-based domain language.

The specification has the following constraints arising out of the fact that the target language is Java. These constraints are not expressible formally with XML syntax:

- The <EnumDef> element attribute name, <choice> element attribute name, can have values only according to Java language specifications for variable names.

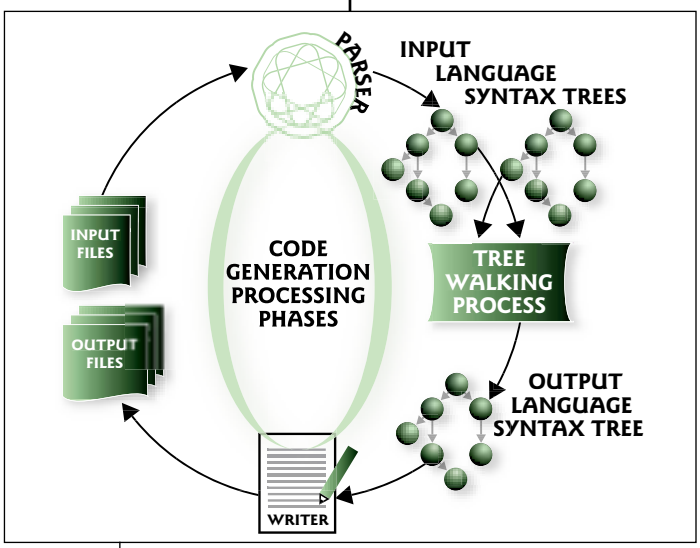

FIGURE 1 Code generation processing phases

	COMPILER-COMPILER BASED	XML / XSLT BASED
Pros	• Input language can have intuitive and elegant syntax • Logic for code generation is written in powerful languages like C++ or Java	• Easy to take care of changes in code generation specification • Scripted, thus aiding rapid development • Little effort spent in support infrastructure. Focus mostly in code generation logic
Cons	• Changes in grammar specification cause widespread changes • Not scripted, i.e., program generator executable needs to be built first. Not a rapid approach • Appreciable effort spent in building support infrastructure used by actual code generation (tree walking) logic • Depending on language structure, syntactic validation may not be in proportion, i.e., not much help from parser generator • Steeper learning curve	• Input language has to be well-formed XML, whose syntax is considered ugly by some. Further validation performed by tree-oriented assertions • XSLT is a specialized programming language having functional and logic flavors. Not an imperative language like C++/Java

TABLE 1 Comparison of code generation approaches

- The <choice> element attribute value can only have distinct integral values. At least one <choice> element has to be nested within the <choices> element.

Once the input language document is defined in XML, XSLT scripts can be written to process documents conforming to the input language and generate output documents in various forms. Processors supporting the XSLT language standard are used to translate XML documents into other XML or text documents. A number of free XSLT processors are available; this study used the SAXON XSLT processor developed by Michael Kay.

XSLT is used for document transformation in, for example, a multiple-line publishing scenario including HTML for Web clients, WML (Wireless Markup Language) for wireless clients, and PDF (Portable Document Format) for print documentation. The concern about one document content, yet multiple presentation formats, is neatly addressed by using XSLT transformation in the Web-based information architecture. In my opinion, however, the preceding use of XSLT has received the most attention in Web document publishing scenarios, and not for its utility of custom code/document generation in application software development projects. This article emphasizes that XSLT offers an easy approach to code generation.

Please note that this strategy of code generation is perfectly in accordance with the processing model in Figure 1. Compared to the traditional approach of code generation using a CC, it has many features aiding the faster development of code generators, as follows:

- ***Parser:*** XML document parsing is implemented in the XSLT processor. With a custom parser you need to implement the parser or understand how to use a program generator such as Lex/Yacc to generate the parsing framework.
- ***Tree:*** The XSLT processor constructs the tree and provides access to it according to the XPATH specification. The user doesn't have to construct the tree at all. With the custom parser the user needs to populate the tree as the parsing progresses.
- ***Tree processing:*** The XSLT processor provides access to the tree through the XPATH expression as well as many programmatic constructs and functions to perform tree processing. In other words, XSLT users write XSLT scripts to perform operations on the tree. On the other hand, this part needs to be implemented programmatically by code generator writers following other approaches. XSLT programming for code generation programming is at a high level – namely, at the level of tree abstraction. Note that with XSLT, even though the code generation programming is at the tree level abstraction, the programmer never needs to worry about tree data structure implementation details.
- ***Writer:*** The XSLT processor implements this part too.

To write a code generator, the developer writes XSLT scripts using XSLT and XPATH facilities. Another benefit of XSLT-based code generation is the flexibility that's allowed in the change of input document grammar: additional elements and attributes can be introduced in the input language specification without affecting code generation scripts. Contrast this to a custom parser–driven approach in which major change propagation occurs throughout the code generation system. These are complexities in the application development of custom code generation not present in the use of XSLT.

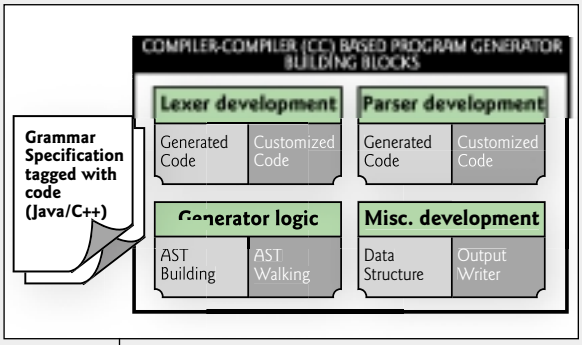

FIGURE 2 CC-based code generation

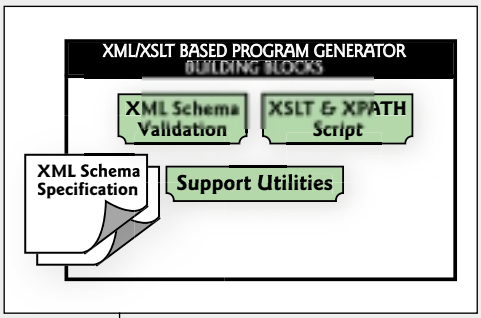

FIGURE 3 XML/XSLT-based code generation

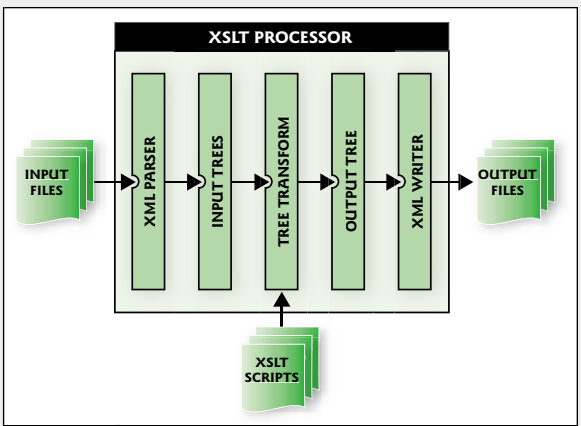

FIGURE 4 XSLT-based document generation process

Figure 4 shows the process of program generation using XSLT. Generally, XSLT processors produce one output file, although an external utility could break the single output file into multiple files. For example, in the case of source code the output file can delimit the beginning and end of each file with markers not likely to occur as part of the source code. Furthermore, the location of each file could be indicated as part of the generation. The external utility then processes the single output file to produce multiple files in multiple locations. Similarly, input to the XSLT processor should preferably be one XML file including other XML files. There are techniques to achieve XML include, and W3C is working on a standardized XML include mechanism.

Details, Details, Details

The detail lies in the use of XSLT and XPATH to implement the desired transformation from the XML source document to the destination document. XPATH is a separate W3C standard that's intimately related to XSLT. XSLT defines a transformation language that operates on the tree representation of XML documents. XPATH defines the syntax of a language that's used within XSLT scripts to address parts of the XML document tree being transformed. XPATH is the "tree addressing" language. An XSLT processor implements XSLT and XPATH specifications and, optionally, some extension functions to make the life of an XSLT scriptwriter easier.

Let's demonstrate the Java enumeration utility class code generation. We first need to decide how we'd like to specify the code generation (Ability 1). We decided that a particular enumeration specification would look like this:

```
<EnumDef name = 'name of the enum'>
<choices>
<choice name = 'name of the choice' value = 'integer value'/>
</choices>
</EnumDef>
```

The specification has the following statements, not expressible formally with XML syntax:

- The <EnumDef> element attribute name, <choice> element attribute name, can have values only according to Java language specifications for variable names. This is because the target language is Java.
- The <choice> element attribute value can only have distinct integral values. There has to be at least one <choice> element nested within the <choices> element.

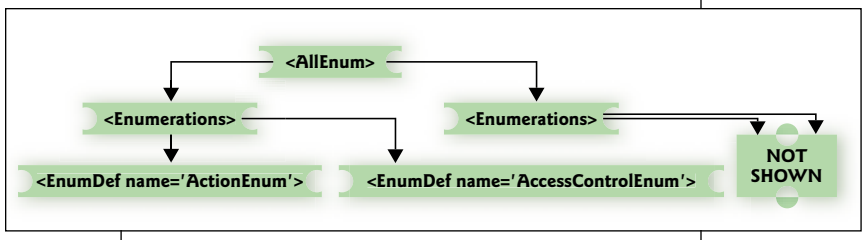

FIGURE 5 Part 1 of the tree structure

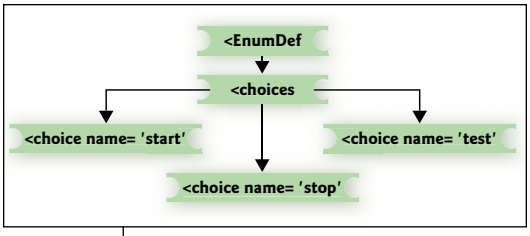

FIGURE 6 Part 2 of the tree structure

With this specification in mind, we start defining XML documents containing Java enumeration code generation specifications. The document in Listing 1 was authored by project software developers, whereas the one in Listing 2 was generated by a utility program from some data source.

One technique of XML inclusion (using an external parseable entity reference) is used to present only one XML input file at XSLT processing time. The XML file in effect has all the enumeration definitions. The XML file showing the technique for XML include is shown in Listing 3.

The second step is to visualize the internal tree structure that will be created inside the XSLT processor. The tree structure is depicted in two parts. The first part shows the overall tree structure (see Figure 5); the second, the tree structure rooted at the first <EnumDef> element (see Figure 6).

For the sake of accuracy, it should be emphasized here that the actual tree structure is much more detailed than what is depicted here. For example, attributes are nodes, text values are nodes, and so on. However, Figures 5 and 6 are sufficient to continue our present discussion.

The third step is to use XSLT and XPATH facilities to process this tree to produce the output files. The XSLT script processes the AllEnums.xml file and produces one output file named enum_codegen.snp that's processed (snipped) by a utility program (file snipper) to produce four Java files named AccessEnum.java, SeverityEnum.java, ActionEnum-.java, and AccessControlEnum.java. For the sake of simplicity, each Java class has only one method, whose purpose is to return the string value of the enum, given the integral value. After setting the classpath environment variable to contain saxon6.4.3.jar, the following command will execute the XSLT script to generate code for the enumerations:

```
java com.icl.saxon.StyleSheet –o gen_enum.snp AllEnums.xml
enum_codegen.xslt
```

What this command is saying is "execute the XSLT processor com.icl.saxon.StyleSheet to produce output file gen_enum.snp from input file AllEnums.xml by using XSLT script file enum_codegen.xslt."

Listing 4 shows the XSLT script; the output it produces is shown in Listing 5. The output is shortened for space considerations.

The code generation script enum_codegen.xslt, version 1, can be explained as follows:

1. The first four lines are declarations, which aren't very relevant from a code generation point of view.
2. The line <xsl:template match = '/'> instructs the XSLT processor to find the root node in the tree and apply the rules contained in the template body. The body works as follows:
   —The xsl:for-each loop selects <EnumDef> child nodes within the root node or from its descendants. Four <EnumDef> children are found within the node hierarchy, starting from the root node. Thus the xsl:for-each loop will execute four times. Please refer to Figure 5.
   —For each execution of the xsl:for-each loop, all non-XSL text is copied to the output. That means the first line within the xsl:for-each loop copies //@@@BEGIN_FILE to the output. The XSLT processor then processes the instruction <xsl:value-of select='@name'/>.java, which outputs the value of the name attribute of the current <Enumdef> child followed by the .java extension. Similarly, you can analyze what happens for the other lines within the xsl:for-each loop.

This step illustrates why it's important to visualize the input tree. At every instruction we're using the XSLT or XPATH facility to navigate the input tree and select content from it to mix with our Java bits and pieces to produce the output Java files. Without a clear idea of the input tree, it wouldn't be possible to use XSLT effectively to generate the desired code.

I hope this explains the code generation development process with XML/XSLT. We're now in a position to show the XSLT script fragment that completes getEnumValueAsString() (see Listings 6 and 7).

Application in a Software Development Project

The purpose of the project was to develop a network management system (NMS). Network management systems are like typical IT applications except that they have to manage the information of a communication network. As a result, a major portion of the NMS goes into providing network device access through some communication protocol. Another core part of the NMS is the object infrastructure, which captures the information of the managed network based on an object and relational model. The project benefited heavily from the use of a generative approach in these core areas. Although we didn't undertake to do so, we could have generated other kinds of documents from the project's XML data.

Network Access Code Generation

Simple Network Management Protocol is a UDP-based network management protocol applied extensively in the data communication industry. Data exchanged over SNMP is defined in a management information base (MIB) with a language called Abstract Syntax Notation One (ASN.1). ASN.1 defines protocol data units (PDU) at the abstract level of data type semantics. ASN.1 doesn't care how the PDU is digitally encoded; this aspect is taken care of by the transfer syntax specification. The name *abstract syntax notation* derives from this fact. A commercial ASN.1 MIB compiler was used to "dump" the MIB in text form. The MIB dump was then processed by a Java utility program to convert the ASCII MIB dump to XML. Essentially, the XML form has all the information that's available in the ASN.1-based management information base.

Once the SNMP MIB was available in XML form, XSLT processing was applied to generate object-oriented Java APIs on a very high level (i.e., far removed from the drudgery of programming according to low-level SNMP APIs). The high-level SNMP APIs were used without difficulty by all project software developers. This approach can be compared with CORBA. In CORBA, programs communicate with each other over TCP protocol without knowing anything about TCP network programming. Before CORBA, distributed application programming used to be done by expert TCP/IP programmers. Now, with CORBA, TCP/IP-based code is generated from a contract language called IDL that specifies what messages and parameters are exchanged between communicating programs. Thus distributed application programming today no longer requires TCP/IP experts. Likewise, SNMP code generation in my project – device-control code – no longer required SNMP experts.

There's another benefit of generating high-level APIs for network access. Two types of implementation were generated. One provides network access by way of SNMP. The other simulates it by storing/retrieving data from local files. Application code using the high-level API remains unaffected when one implementation is switched with another. Code generation for file-based SNMP simulation allowed the project to proceed without waiting for the actual device to be ready. This is a tremendous advantage since network management development can proceed in parallel and can be tested with a large, simulated network.

Server-Side Code Generation

An object-based server-side infrastructure was used in this project. Application servers complying with the Enterprise JavaBean standard provide concurrency, transaction, security, persistence, and naming services for the objects. A server-side system development consists of implementing the information model by using EJBs and providing interfaces for remote access to the information, which satisfies graphical user interface use cases. The project specified the information model along with a number of relationships in XML. A fictitious object showing object and relational model capability is shown in Listing 8. The model specifies the following:

- The Employee object has attributes like salary, job title, join date. It will inherit other attributes from the Person object.
- The Employee object can't exist without a containing Division object.
- If you have access to the Employee object, you can get access to all its subordinates, which are objects of class Employee.
- If you have access to the Employee object, you can get access to the supervisor, which is an object of class Employee.

The foregoing specifications were implemented by autogeneration. Similarly, a relational database schema in SQL was generated to implement the persistence of object and relational model information. XML deployment descriptors were generated for runtime deployment of EJBs. In addition, semantics were mixed in by the use of application classes inheriting generated classes, and application-specific behavior was coded in derived classes.

Object and relational model code generation, coupled with middleware services provided by the EJB framework, created a powerful paradigm of server-side infrastructure development for the project. The project had a tremendous lead by being able to build further on this sophisticated server-side infrastructure rather than spend time on building the infrastructure itself. The project focused totally on building application logic and delivering functionality.

Things to Be Aware Of

I experienced some limitations in starting from XML and applying XSLT in real application software projects.

- The input language is XML, whose syntax may be abhorrent to some. XML should be considered an underlying data representation language. Tools such as editors, both textual and visual, provide high-level browsing and editing capabilities and should be used while working with XML. If XML is generated by some tools (like SNMP MIB browsers), this concern doesn't arise.
- XSLT is a purely functional language, which means no side effects. For example, to implement a for-loop of fixed count you need to implement a tail-recursive template with a suitable termination condition. Application programmers have to make many more "habit adjustments" before becoming comfortable with XSLT programming. Moreover, XSLT has a "logic programming flavor," a paradigm that some application developers may not be familiar with. XSLT syntax is quite verbose, which is bothersome to say the least. XML escape mechanisms have to be used to generate symbols like "<" (less than), ">" (greater than), and others that are heavily used in source code. Some XSLT workbenches are available to mitigate this inconvenience.
- Unlike the IDEs we have for C++/Java development that have graphical debugging capabilities, no such IDE is available yet for XSLT. The debugging technique we used was to print part of the tree in the output document itself, i.e., by using <xsl:value-of ..>. We then inspected the printed tree to understand how a desired selection/transformation could be created using XSLT/XPATH.
- The XSLT processor is quite permissive with reference to mistakes/omissions made in inputting the document. For example, missing elements or attributes don't cause the XSLT processor to fail. They cause missing output! Consequently, for source code generation, there will be compiler errors somewhere down the line. There are a number of solutions to this problem of input document valida-

LISTING 1 AuthoredEnums.xml

```
<Enumerations>
<EnumDef name = 'AccessEnum'>
<choices>
<choice name = 'read_only' value = '1'/>
<choice name = 'read_write' value = '2'/>
</choices>
</EnumDef>
<EnumDef name = 'SeverityEnum'>
<choices>
<choice name = 'critical' value = '1'/>
<choice name = 'major' value = '2'/>
<choice name = 'minor' value = '3'/>
<choice name = 'warning' value = '4'/>
</choices>
</EnumDef>
</Enumerations>
```

LISTING 2 GeneratedEnums.xml

```
<Enumerations>
<EnumDef name = "ActionEnum">
<choices>
<choice name = "start" value = "1"/>
<choice name = "stop" value = "2"/>
<choice name = "test" value = "3"/>
</choices>
</EnumDef>
<EnumDef name = "AccessControlEnum">
<choices>
<choice name = "permit" value = "1"/>
<choice name = "deny" value = "2"/>
</choices>
</EnumDef>
</Enumerations>
```

LISTING 3 AllEnums.xml

```
<?xml version = "1.0"?>
<!DOCTYPE AllEnums[
<!ENTITY include_authored_enums SYSTEM "AuthoredEnums.xml">
<!ENTITY include_generated_enums SYSTEM
"GeneratedEnums.xml">
]>
<AllEnums>
&include_authored_enums;
&include_generated_enums;
</AllEnums>
```

LISTING 4 Enum_codegen.xslt(version 1)

```
<?xml version='1.0'?>
<xsl:stylesheet version='1.0'
xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
<xsl:output omit-xml-declaration='yes'/>

<xsl:template match = '/'>
<xsl:for-each select='//EnumDef''>
//@@@BEGIN_FILE <xsl:value-of select='@name'/>.java
//@@@LOCATION common.gencode.enums
//************************************
//*********** Generated code.  ***********
//************************************
public class <xsl:value-of select='@name'/>
{
    public static String getEnumValueAsString
     (int enumValue)
    {
    }
}
```

tion. For example, you can use Schematron, an XSLT-based assertion facility for XML document validation. XML schema–enabled parsers may be able to do some validation based on the schema description of the document.

Regardless of these limitations, it can be argued that XSLT is being used to support application software development rather than used directly in application software development. Consequently, project members' exposure to XSLT will be minimal once the correct code generation has been figured out. For most code generation purposes, an XSLT-based solution is feasible. I believe that application software developers need to become more familiar with this XML-based document transformation approach so they too can achieve the benefits set forth in this article.

Summary/Conclusion

This article demonstrated how XML-based document transformation technology can benefit application software development projects. Here are some software metrics that show how my project benefited:

- Out of approximately 2,480 Java files in the project, 1,520 were generated. In other words, 61% of the project Java code was generated. SQL and XML files were generated too.
- SNMP coding was extremely easy. For example, it took only one line of Java code to display the RFC1213 ipRouteTable:

```
System.out.println(new
IpRouteTableUtils().getIpRouteTable(deviceInfo));
```

IpRouteTableUtils is generated from RFC1213 ASN.1 MIB. The method getIpRouteTable returns the table object, which is displayed in the System.out.println call. The deviceInfo object has the device IP address and the password for read-only SNMP access (also known as read community name).

With the standardization of XML transformation by XSLT/XPATH, and with the increasing availability of good literature on the subject, it's my firm belief that code generation by XML transformation using XSLT will give application software development projects tremendous leverage in attaining increased productivity without sacrificing quality. One developer's expertise in XSLT and code generation could dramatically lessen the burden on other software developers by providing semantically rich, high-level APIs through code generation. With XSLT 2.0 features the task of code generation will become easier.

References

- Cleveland, J.C. (2001). *Program Generators with XML and Java.* Prentice-Hall.
- Kay, M. (2001). *XSLT Programmer's Reference*, 2nd edition. Wrox.
- *SAXON XSLT Processor:* http://saxon.sourceforge.net
- *GSLgen:* www.imatix.com/html/gslgen/index.htm
- *Fxt, generator of XML document transformers:* www.informatik.uni trier.de/~aberlea/Fxt/
- *The Schematron: An XML Structure Validation Language Using Patterns in Trees:* www.ascc net/xml/resource/schematron/schematron.html
- *XML in 10 points:* www.w3.org/XML/1999/XML-in-10-points
- *Extensible Markup Language (XML) 1.0:* www.w3.org/TR/REC-xml
- *XSL Transformation (XSLT) Version 1.0:* www.w3.org/TR/xslt
- *XML Path Language (XPATH) Version 1.0:* www.w3.org/TR/xpath

AUTHOR BIO

Soumen Sarkar, currently lead software engineer at Atoga Systems, Inc., specializes in building large-scale distributed object systems for telecommunications network management. His interests include programming languages, CORBA and EJB, XML and related technologies, communication protocols, and software architectures. He has more than 10 years of experience in building object-oriented software.

SARKAR_SOUMEN@YAHOO.COM

XML-J ADVERTISER INDEX

ADVERTISER	URL	PHONE	PAGE
ADOS Co., Ltd.	http://www.a-dos.com	81-3-5475-1551	59
Altova	www.xmlspy.com		60
Borland			11
engindata Research	www.engindata.com	201-802-3082	24, 25
HitSoftware	www.hitsw.com/xmlrdb	408-345-4001	15
IBM	www.ibm/com/websphere/winning		2, 3
JDJ Store	www.jdjstore.com	888-303-JAVA	55
Macromedia	www.macromedia.com/go/cfmxad	888-939-2545	9
Sonic Software	www.sonicsoftware.com/websj		6
Sprint	http://developer.sprintpcs.com		4
SYS-CON Events	www.sys-con.com	201-802-3069	27, 28
SYS-CON Media	www.sys-con.com/suboffer.cfm	888-303-5282	48
SYS-CON Media	www.sys-con.com	888-303-5282	47
WebLogic Developer's Journal	www.weblogicdevelopersjournal.com	888-303-5282	51
WebServices Edge World Tour	www.sys-con.com	201-802-3069	34, 35
WebSphere Developer's Journal	www.sys-con.com/websphere	888-303-5282	48
XML for Financial Services	www.worldrg.com/fw251	800-647-7600	55

General Conditions: The Publisher reserves the right to refuse any advertising not meeting the standards that are set to protect the high editorial quality of ***XML-Journal***. All advertising is subject to approval by the Publisher. The Publisher assumes no liability for any costs or damages incurred if for any reason the Publisher fails to publish an advertisement. In no event shall the Publisher be liable for any costs or damages in excess of the cost of the advertisement as a result of a mistake in the advertisement or for any other reason. The Advertiser is fully responsible for all financial liability and terms of the contract executed by the agents or agencies who are acting on behalf of the Advertiser. Conditions set in this document (except the rates) are subject to change by the Publisher without notice. No conditions other than those set forth in this "General Conditions Document" shall be binding upon the Publisher. Advertisers (and their agencies) are fully responsible for the content of their advertisements printed in ***XML-Journal***. Advertisements are to be printed at the discretion of the Publisher. This discretion includes the positioning of the advertisement, except for "preferred positions" described in the rate table. Cancellations and changes to advertisements must be made in writing before the closing date. "Publisher" in this "General Conditions Document" refers to SYS-CON Publications, Inc.

```
//@@@END_FILE AccessEnum.java
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

LISTING 5 gen_enum.snp

```
 //@@@BEGIN_FILE ActionEnum.java
//@@@LOCATION common.gencode.enums
//************************************
//*********** Generated code.  ***********
//************************************
public class ActionEnum
{
    public static String getEnumValueAsString
     (int enumValue)
    {
    }
}
//@@@END_FILE ActionEnum.java
…….
Repetition of above pattern for SeverityEnum.java,
AccessEnum.java, AccessControlEnum.java
…….
```

LISTING 6 XSLT code fragment

```
Public static String getEnumValueAsString(int enumValue)
{
    switch(enumValue)
    {<xsl:for-each select='choices/choice'>
        case <xsl:value-of select='@value'/>:
        return "<xsl:value-of select='@name'/>;
    </xsl:for-each>

        default:
        return null;
    }
}
```

LISTING 7 Corresponding generated code fragment

```
7 public static String getEnumValueAsString(int enumValue)
{
    switch(enumValue)
    {
        case 1:
        return "start";
        case 2:
        return "stop";
        case 3:
        return "test";

        default:
        return null;
    }
}
```

LISTING 8 Example of object and relational modeling in XML

```
<managed-object class-name = "Employee">
<base-object class-name = "Person"/>
<containing-object class-name = "Division"/>
< one-to-many-relation  role-name = "my-subordinates"
class-name = "Employee"/>
<one-to-one-relation  role-name = "my-boss" class-name =
"Employee"/>
<attribute name = "salary" type = "float"/>
<attribute name = "job-title" type = "string"/>
<attribute name = "join-date" type = "java.util.Date"/>
```

Download the Code
www.sys-con.com/xml