XML SCHEMA INFERENCE WITH XSLT

By

SCOTT MCCOLLUM BUNTIN

A THESIS PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2001
ACKNOWLEDGMENTS

I would like to express my sincere gratitude to the chairman of my committee Dr. Wilson for his guidance, support and encouragement. Thanks also go to my committee members Dr. Bermudez and Dr. Hammer. I greatly appreciate my family, including my wife, my parents and my sister, for always being there for me.
# TABLE OF CONTENTS

ACKNOWLEDGMENTS................................................................................................................................. ii

LIST OF FIGURES............................................................................................................................................. vi

ABSTRACT ........................................................................................................................................................... vii

CHAPTERS

1 INTRODUCTION........................................................................................................................................... 1

2 XML AND XSLT ........................................................................................................................................... 6

   XML............................................................................................................................................................. 6
      Document Object Model........................................................................................................................... 6
      XML Syntax.............................................................................................................................................. 9
      XML Namespaces.................................................................................................................................. 10

   XSLT............................................................................................................................................................. 11
      Parsers .................................................................................................................................................... 12
      XSLT Processing Model .......................................................................................................................... 12
      Context .................................................................................................................................................. 15
      Templates............................................................................................................................................... 17
      Variables............................................................................................................................................... 18
      Data Structures................................................................................................................................... 19
      Expressions............................................................................................................................................ 20
      Patterns.................................................................................................................................................. 24

3 DTDS AND XML SCHEMAS...................................................................................................................... 25

   Document Type Definition ....................................................................................................................... 25
      Element Type Declarations .................................................................................................................... 26
      Attribute List Declaration.................................................................................................................... 27
      Entity Declarations............................................................................................................................... 28
      Notation Declarations............................................................................................................................ 30

   XML Schema ............................................................................................................................................... 30
      Advantages of XML Schemas................................................................................................................ 30
      Validation.............................................................................................................................................. 32
      XML Schema Specification Methodology............................................................................................. 32
      Abstract Data Model ............................................................................................................................. 33
      Primary Components............................................................................................................................. 35
      Secondary Components.......................................................................................................................... 46
D  POSSIBLE CUSTOMIZED SCHEMA DOCUMENT OF SCHEMA
   IN APPENDIX C ................................................................................................... 114
E  THE SCHEMAENGINE ....................................................................................... 116
BIBLIOGRAPHY ....................................................................................................... 132
BIOGRAPHICAL SKETCH ....................................................................................... 135
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Document Object Model (DOM)</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>StyleSheet Structure</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>Scope of Variables in XSLT Programs</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>XML Schema Type Definition Hierarchy</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>Overview of XSLTEngine Structure</td>
<td>59</td>
</tr>
<tr>
<td>6</td>
<td>Append-Copy</td>
<td>86</td>
</tr>
<tr>
<td>7</td>
<td>Replace-Copy</td>
<td>89</td>
</tr>
<tr>
<td>8</td>
<td>Add-Attributes</td>
<td>92</td>
</tr>
<tr>
<td>9</td>
<td>XML Schema Inference</td>
<td>95</td>
</tr>
</tbody>
</table>
XML SCHEMA INFERENCE WITH XSLT

By

Scott McCollum Buntin
August, 2001

Chairman: Dr. Joseph Wilson
Major Department: Computer and Information Science and Engineering

With the increasing popularity of the eXtensible Markup Language (XML) as a data representation language, the need arises to reconcile disparate collections of data represented in XML. Document Type Definitions (DTDs) are the original mechanism for describing XML documents and determining if a given XML document conforms to a standard. Recently (2000-10-24) the XML Schema model has been proposed to replace DTDs. Schemas have several advantages over DTDs, most notably that they are themselves XML documents, allowing them to be manipulated with the same tools that are used to manipulate XML documents.

While some tools exist for inferring DTDs and XML Schemas from an arbitrary XML document, these tools are not written in XSLT, which is the language created specifically for processing and transforming XML documents. A schema inference engine, written in XSLT, that directly infers a schema from an XML document, is a
useful tool in its own right, but also serves as a demonstration of the use of XSLT as a programming language to perform highly computational tasks. In this project, XSLT is used as a general-purpose graph transformation language to transform and refine an XML Schema (possibly empty) into another XML Schema based on an arbitrary XML document input. To accomplish this goal, a new XSLT interpreter, called the XSLTEngine, was created. This interpreter implements features of the latest XSLT specification (XSLT 1.1) and also introduces some extension instructions based on the protocol outlined in the XSLT specification.
CHAPTER 1
INTRODUCTION

The need for defining a uniform model for the description of data, particularly as applied to information on the World Wide Web, has been recognized for some time. The World Wide Web Consortium (W3C) is a collection of industry and academic specialists that has proposed XML [1-3] for answering this need. The W3C XML Working Group activity statement is as follows [4]:

Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML (ISO8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web. XML

- Enables internationalized media-independent electronic publishing
- Allows industries to define platform-independent protocols for the exchange of data, especially the data of electronic commerce
- Delivers information to user agents in a form that allows automatic processing after receipt
- Makes it easier to develop software to handle specialized information distributed over the Web
- Makes it easy for people to process data using inexpensive software
- Allows people to display information the way they want it, under style sheet control
- Makes it easier to provide metadata, or data about information, that will help people find information and help information producers and consumers find each other

XML is a generalized markup language that provides metadata, or information about information. XML does not in itself, provide mechanisms for interacting with the
data that an XML document contains. In order to realize all the aspirations of the W3C, tools for interpreting and manipulating XML data must be produced. Luckily, the W3C has developed a programming language specifically designed for this purpose, eXtensible Stylesheet Language Transformation (XSLT) [5-9] XSLT, or some equivalent, is vital for most of the goals listed above. This topic is explored further below.

The XML data model is very general, a necessary prerequisite to insure its widespread acceptance. However, the general model must be constrained further in such a way that a certain business, application, etc. will be assured that information is presented to it in a predictable and useful manner. The concept of a schema [2], which has been around for many years, is a way of accomplishing this goal of constraining data. Schemas are familiar to programmers as grammars. Schemas are necessary to fully exploit the promise of XML. They help reconcile disparate collections of data, enable the efficient storage of data, help people formulate queries in databases, etc. [10] A schema can also serve the purpose of explaining an XML document by providing an abbreviated description of the document’s structure.

Schemas, in the general sense, can be used to define sets of all possible valid documents in given domains. In XML, a document is valid if it conforms to the rules in its schema. The original document description language adopted for XML is known as Document Type Definition (DTD) [1]. The DTD standard is based upon regular expressions, which are a powerful mechanism for expressing patterns but are limited in various ways. Some of the limitations of DTDs as listed in Anderson et al. [2] are

- They are difficult to write and understand
- Programmatic processing of their metadata is difficult
- They are not extensible
- They do not provide support for namespaces
• There is no support for datatypes
• There is no support for inheritance


While XML 1.0 supplies a mechanism, the Document Type Definition (DTD) for declaring constraints on the use of markup, automated processing of XML documents requires more rigorous and comprehensive facilities in this area. . . . The XML Schema Working Group is addressing means for defining the structure, content and semantics of XML documents.

The proposed XML Schema standard produces document descriptions having greater specificity than DTDs and has the important advantage that XML Schema documents are themselves XML compliant. Since an XML Schema document is also an XML document, a schema can be automatically modified by XSLT scripts, and thus is more maintainable than a DTD. Additionally, it becomes easier for a single application to process schemas and XML documents concurrently.

XML was developed with many goals in mind, but one of the key desires motivating its formation was its possible application to the World Wide Web [12, 13]. HTML documents form a subset of XML documents and there is an effort to migrate the web from HTML to XML [13]. Thus, XML applications should be able to work with web pages. A well-known and popular WWW application is the search engine. One problem with current search engines is that they often return too many web pages that are not actually related to the inquiry. With the integration of schemas into searches, a search could return web pages that closely conform to the schema, and further, only return that part of the web page that is of interest to the user.

One of the tools needed to exploit XML is an integrated utility that allows for inferring schemas given a document [10, 14-16] and refining that schema given
additional valid documents. Also, given a schema, one can automatically generate an XSLT document that can extract information conforming only to the schema from an arbitrary XML document. This ability would be useful in extracting only the desired information from an XML document.

XSLT is a new language that is still undergoing changes. There is still some debate on whether XSLT has the appropriate design to be an effective XML processing language. In fact, many alternative XML Schema languages have been proposed [17]. The process of formally evaluating XSLT [7, 9, 18] is progressing however, and improvements are still being made to the language [6]. Some, however, have stated that the XSLT model is not suitable for semantic translations at all [19], or that XSLT can not be used as a general-purpose tree transformation language. Reasons given for XSLT’s weakness include: it has insufficient constraints on the data it processes, it is difficult to perform computations, etc.[19]. This thesis refutes that claim by creating an application that is both a general-purpose tree transformation application and performs semantic translations.

In this thesis, two objectives will be addressed:

- Construction of an algorithm, written in XSLT, for inferring and refining schemas given as input an XML document and a (possibly empty) schema
- Implementing and extending the XSLT programming language, with a processor called the XSLTEngine, for this purpose

The remainder of the thesis consists of the following. Chapter 2 is a review of XML, and XSLT. Chapter 3 is a review of schemas, including Document Type Definitions and the XML Schema language. Chapter 4 is a description of the
XSLTEngine. Chapter 5 is a description of the SchemaEngine. Chapter 6 is a discussion.

The notation used in this thesis is as follows: classes in object-oriented models (DOM, Perl, XML Schema) are written in bold faced, fixed-width font (Node). Document elements and methods and attributes of classes are written in fixed width font <xsl:for-each>. 
The Extensible Markup Language (XML) and Extensible Stylesheet Language Transformation (XSLT) are both members of the XML family of technologies. XML is a way of marking up, or providing information about, data. XSLT is used to transform an XML document into another XML document.

XML

XML was derived from older markup languages. First was the Generalized Markup Language (GML), which was then refined to Standard Generalized Markup Language (SGML), which was adopted as an international data storage and exchange standard in 1986 by the International Organization for Standardization (ISO). SGML has been widely adopted by government agencies, large manufacturing firms, and publishers of technical information. However, the complexity and high cost of implementation have limited the number of businesses and individuals capable of using this technology [2]. With the spread of the Internet, interest was renewed in a simplified generalized markup language, and thus XML was born.

Document Object Model

XML is a graph-structured language. Mostly XML documents will be tree structured, but there are some instances when references are allowed, turning the tree into
a graph. An XML document is a plain text document that follows the rules for XML. Applications that use XML will generally convert the text document into some internal graph or tree representation. Given the tree-like structure of XML documents, it is natural for applications to represent an XML document in an object-oriented way, though this is not required. To facilitate the representation of XML documents, the W3C has provided the Document Object Model (DOM), which defines a standard model for the objects and their properties that should be used in representing an XML document. The implementation of the DOM is not specified, allowing the model to be platform and language independent. A DOM tree is composed of a collection of nodes with predefined types and relationships, allowing for the representation of hierarchical information. The object classes are listed below:

- **Node** – the superclass of all other objects
- **NodeList** – an ordered set of nodes
- **NamedNodeMap** – an unordered set of nodes
- **Document** – represents the entire document. Is the top node in an XML DOM tree.
- **DocumentFragment** – a lightweight representation of a DOM tree
- **DocumentType** – associates an XML document to a schema
- **Element** – part of a document bounded by start and end tags
- **Text** – sequence of characters in PCDATA part of doc. Text nodes are always leaves.
- **CharacterData** – superclass of **Text** and **Comment**
- **CDATASection** – used to escape blocks of text containing characters that would otherwise be regarded as markup
- **Attribute** – includes name and value of attribute written in start tag
- **Comment** – represents comment in source doc, ignored by processor
- **ProcessingInstruction** – pass information to the application
- **Namespace** – represents namespace declaration
- **Notation** – processor directives related to entities
- **Entity** – an unparsed external thing
- **EntityReference** – reference to entity, denoted with preceding ampersand (&amp)
• **DomImplementation** – may be used to create implementation specific **Documents**

A diagram of the DOM can be found in figure 1.

![Diagram of Document Object Model (DOM)](image)

**Figure 1: Document Object Model (DOM)**

The root of all XML DOM structures is an object of type **Document**. This **Document** object represents the whole tree. The primary objects used in creating the remaining structure of the DOM tree are **Elements** and **Attributes**. There must be one element which is the only child of the top-level node, and which has as descendants all other elements in the DOM tree. Each element has a single parent, an ordered set of children (null in case of leaves), an unordered set (possibly empty) of **Attributes** and possibly a preceding and/or following sibling. **Attributes** are always associated with an **Element**, and do not have parents or children. **CharacterData** always has a parent and consists of non-markup string data.
**XML Syntax**

In an XML text document, there are 2 primary types of markup used to create the DOM tree: elements and attributes. The syntax for these 2 will be discussed. For a more complete discussion, see the book Professional XML [2].

**Elements** are created with the use of tags. There are three types of tags: start-tags, end-tags and empty-element tags. Tags must contain an *element type name*, which is a string literal consisting of valid XML name characters (NMTOKEN). A tag is delimited by angle brackets. The first string within a tag denotes an *element type name*. An end tag has a slash as the first character after the leading less than symbol (<*/element type name*>). A start tag has no leading or trailing slashes (<element type name>), and an empty-element tag has a slash as the last character preceding the trailing greater than symbol (<element type name/>). There are 2 ways to declare an *Element*. For the first way, the element declaration must consist of a start tag followed by an end tag, both with the same *element type name*, with possible intervening markup denoting the children of that element. The other way is to use one empty-element tag, which can have no children.

**Attributes** can be declared within element declarations, either in the start tag or an empty-element tag. To declare an *Attribute*, in the tag, following the *element type name*, a valid XML name is written followed by an equals sign, then a quoted string. More than one attribute can be declared for an element (<myElement attribute1="value1" attribute2="value2">). In this case, the *Element* named myElement will have 2 *Attributes* associated with it, the first with name attribute1 and value value1, and similarly for the second attribute. XML processors are not required to impose an
order on attributes (the DOM specifies that attributes are stored in a NamedNodeMap),
so no order is imposed.

An XML document is well-formed if these rules are followed. That is, if there is
a corresponding end-tag following each start tag, and there is one Element which
contains the entire document. An XML document is valid if it conforms to the rules in its
schema (DTD, XML Schema, etc.). Valid documents will be discussed in the XML
Schema section.

XML Namespaces

XML Namespaces [20] provide a mechanism for associating specified nodes in a
DOM tree with explicit or implicit namespaces. A namespace is designated by a Uniform
Resource Indicator (URI). URIs are based on the familiar Uniform Resource Locator
(URL) protocol, which is used for providing global namespaces on the World Wide Web.
All URLs are valid URIs, syntactically, but URIs extend URLs in the sense that a URI
may reference a logical structure within a document, instead of just a file on a computer.
Namespaces can be declared with attribute-like syntax, but a namespace is not an
attribute. Namespace is a separate data type. A namespace may be declared like this:

```xml
<xsl:stylesheet xmlns:xsl=
    "http://www.w3.org/1999/XSL/Transform ">
    <xsl:element name="x"/>
</xsl:stylesheet>
```

In the preceding example, the string xmlns indicates that what follows is a
namespace declaration. The string xsl is the prefix that will be associated with this
namespace and the quoted string after the equals sign is the URI which indicates the
expanded namespace name. The scope of the namespace is all descendants of the element that contains the declaration. Any element or attribute that has the xsl prefix is part of this namespace. Default namespaces can also be declared by omitting the prefix in the namespace declaration.

There are certain namespace prefixes that are reserved by the W3C. For example, the prefix xsl is reserved by the W3C and refers to the namespace for the eXtensible Stylesheet Language.

An example XML document can be found in Appendix A.

**XSLT**

XSLT is a powerful language that operates on the underlying tree structure of an XML document (indicating the need for a DOM parser, to be discussed shortly) to produce a new document which is a transformation of the original. The specifications for XSLT can be found at [http://www.w3.org/XSLT](http://www.w3.org/XSLT). The first specification for XSLT became a recommendation (stable document) in 1999. This version is known as XSLT 1.0. There is now a working draft of the next version of XSLT that is known as XSLT 1.1, that proposes some important modifications to the language, but the basic processing model remains the same for both versions. In the following discussion, characteristics represented as belonging to XSLT apply to both XSLT 1.0 and XSLT 1.1, with the exception of the **DocumentFragment** object, which only appears in XSLT 1.0. The term **DocumentFragment** will often be used to apply to the equivalent data structure in XSLT 1.1. Characteristics that apply to only one of the specifications will be explicitly noted.
Parsers

There are two types of XML parsers available, a Document Object Model (DOM) parser and Simple API for XML (SAX) parser. The SAX parser is an event-based parser, meaning that as the parser scans an XML document certain actions are triggered when an event is encountered. Events include the discovery of a start tag, end tag or text element. A SAX parser generally is responsible for signaling events. Whatever application is using this parser defines what action is to be taken on discovery of an event. The primary advantage of a SAX parser is that it uses less memory than a DOM parser.

A DOM parser operates by constructing a physical representation of the tree structure inherent in an XML document. The nodes of the tree are objects that conform to the Document Object Model as defined by the W3C [1]. A DOM parser requires more memory than a SAX parser, but the increasing availability of cheap memory makes this less of a concern except when dealing with exceptionally large documents. The advantage of the DOM parser is that, once the tree is constructed, an application can freely navigate the document without reparsing it. For an XSLT interpreter to run, some representation of the XML documents must be created at some point, either in the parser or in the application itself. All available XSLT implementations maintain the whole document tree, in some fashion. This is a basic requirement of the proposed application and, thus a DOM parser is used for the XSLTEngine.

XSLT Processing Model

There are two inputs to an XSLT interpreter:

- The XML document to be transformed
- The XSLT stylesheet or program.
There is one output from an XSLT interpreter, the ResultTree. The product of running an XSLT script against an XML document is a new tree (ResultTree), which is a transformation of the original tree according to the rules in the XSLT script. At the completion of execution, the ResultTree can be written to a file.

The fundamental components of an XSLT stylesheet are instructions, also known as XSLT elements. These entities follow the syntax of XML elements, but are distinguished by a unique prefix (xsl). There are a number of instructions that are described in the XSLT specifications, and these instructions must be implemented in the construction of a fully compliant XSLT processor. The prefix xsl is reserved by the W3C to reference the XSLT namespace (http://www.w3.org/1999/XSL/Transform for XSLT 1.0). The namespace may have variations, such as the year, which indicate details about the version of XSLT that is being implemented. If a processor implements extension instructions, they must have a prefix other than xsl and belong to a namespace other than the one above, but they may operate in a similar fashion to predefined instructions.

XSLT instructions provide directives to the XSLT processor on program behavior, but the implementation of those instructions is left to the processor. Elements that do not have the xsl prefix (or a prefix associated with a declared extension instruction namespace) must conform to the XML data model. These elements are treated as literal elements and may be copied directly to the ResultTree.

XSLT is a declarative language, meaning that instead of providing the steps a computer should take to produce a given result, an XSLT program provides the result that is wished for and the XSLT processor determines which steps to execute [8]. The
declarations take the form of templates, defined in the stylesheet. XSLT then transforms XML documents according to the rules defined in the templates of the XSLT stylesheet. In addition to being a declarative language, there are also some elements of functional programming present in XSLT. Namely, the inability to update variables and the passing of expressions as parameters to instructions and named templates.

Once a variable has been assigned a value in XSLT, the value cannot be changed (updated). When a variable has gone out of scope, a new variable with the same name can be created, and this variable will be assigned a new value. Also, expressions are passed as parameters to instructions via attribute syntax and may be passed to named templates as parameters. Templates, are not themselves functions, however, and cannot be passed to other templates as parameters.

As stated earlier, there are 2 inputs to an XSLT interpreter. The XML source document and the XSLT stylesheet. These are typically plain text files conforming to XML and XSLT standards respectively. XSLT documents are also XML documents, but there are some rules that apply only to XSLT documents. Namely, the root node may have any children that are allowed by an element, not just a single child node, as for XML documents. Assuming the input is given in the form of text files, both documents are parsed so that an internal DOM representation is created (see figure 2). An empty tree (consisting of only a root node) is then created for the result tree. As the processing of the stylesheet progresses, the result tree is created.
Figure 2: StyleSheet Structure

Context

An important factor in understanding how XSLT interpreters work is to understand context. The interpreter must be aware at all times of what context it is in. There are three main components to context:

- **Current Source Node List** – The Node or Nodes in the source tree that is/are currently being examined. If there is more than one Node in the Current Source Node List, then each will be processed in order.

- **Current Style Node** – The Node in the style sheet that is currently being processed

- **Current Result Node** – The Node in the Result Tree (or DocumentFragment) that is currently being written to
The interpreter must keep all three trees in memory, along with pointers to the context position in all three trees, at all times. The current source node may be used to determine which template (to be discussed) to select for processing and also serves as input in the evaluation of expressions. The current style node is used to determine what actions will be taken, and the current result node determines where the results of processing will be written. All of these context nodes will change throughout the execution of the program. There is a default mode for order of execution for each of these contexts. In the case of the current source node and current style node, the trees are traversed in pre-order (document order in plain text) fashion. Nodes are similarly written (appended) to the result tree in pre-order fashion.

There are ways to force a context switch for all components of the context. A context switch can be accomplished via a given XSLT instruction (element). These instructions are elements found in the XSLT style sheet, and which match a predefined instruction element in the XSLT lexicon. The instruction must be in the XSLT namespace. These instructions will be discussed in more detail later. The <xsl:apply-templates> XSLT instruction can optionally be called on the source nodes that are returned by an expression. This will typically be a NodeList. If so, the current source node will successively be each node in the NodeList (Current Source Node List) that is returned by the expression. If no source nodes are specifically chosen by an <xsl:apply-templates> instruction, the Current Source Node List becomes the children of the current source node.
There is no goto statement in XSLT, but the current style node can be switched with the use of the `<xsl:call-templates>` instruction, which forces the current style node to be the corresponding named template.

The current result node can be switched by use of the `<xsl:variable>` instruction. If a variable is used to construct a `DocumentFragment`, the current result node will be the root node of the `DocumentFragment` (note: this data type has been removed from XSLT 1.1, but the same concept remains. Kay uses the term “temporary tree” [8] to describe the equivalent data structure in XSLT 1.1, but I will retain the term `DocumentFragment`).

**Templates**

An XSLT program is composed of templates, which can be of 2 types: named templates and un-named templates. The behaviors of these 2 types of templates are different. Un-named templates must have a `match` attribute, which determines the nodes to which that template applies. If the XSLT interpreter is executing an `<xsl:apply-templates>` instruction, the nodes in the current source node list are checked to see if they match the pattern of any un-named template. If there is more than one template that matches the current source node, the template with the highest priority is selected.

Every un-named template has a priority. The priority can be explicitly set via a `priority` attribute, or if it is not explicitly set, the XSLT interpreter assigns a priority based on the specificity of the pattern in that templates `match` attribute.

Named templates are never applied by default. They must be called with the use of the XSLT instruction `<xsl:call-template>`. This instruction allows for the
passing of parameters. Once a template has been called and completed its execution, the current style node becomes the next style node in the style sheet in document order.

Variables

The scoping of variables in XSLT is molded by the tree-like nature of XSLT. Since a variable declaration may have children, it is important that those children not be able to reference the variable declaration itself. Otherwise, one could have a circular definition where a variable references itself. Therefore a variable is in scope for all following elements or descendants of following elements (Figure 3). XSLT is statically scoped in that the scope of a variable can be determined prior to execution, by examining the tree structure of the stylesheet document.

![Figure 3: Scope of Variables in XSLT Programs](image)

Variables cannot be updated in XSLT. This may cause difficulty for some used to imperative programming styles. This difficulty can be overcome by use of the...
<xsl:param> instruction defined in XSLT. This instruction allows one to pass a parameter to a named template, and that parameter can be changed on recursive calls to the template. Thus, due to this recursive variable updating and the pre-order tree traversal nature of processing, XSLT is highly recursive.

Data Structures

The conventional wisdom on XSLT used to state that the language does not provide for the convenient construction of complex data structures [21]. After all, XSLT was originally intended to be primarily a presentation language, not to perform highly computational tasks such as computing missile trajectories. However, XSLT is still in the development stage and is constantly evolving. The paucity of data structures has been recognized as a problem by the w3c and some improvements have been proposed in the XSLT 1.1 working draft [22].

In a computational program, where data needs to be manipulated in some way to produce a computed result, representing that data in XSLT 1.0 can be cumbersome. The data types that are allowed in XSLT 1.0 are: strings, numbers, nodes, node-sets and tree fragments. Node-sets cannot be generated de novo, but only as the result of an expression that returns existing nodes, so usually one is limited to strings, numbers and booleans to describe user-defined objects.

The obvious choice for user defined data structures is the DocumentFragment, and this is the data type that is altered in XSLT 1.1. DocumentFragments can be built explicitly within a variable declaration. They cannot be modified after construction, but they can be passed as parameters to named templates that create new DocumentFragments based on their content. The DocumentFragment data type
is dispensed with in XSLT 1.1, but the same rules for “temporary tree” construction still apply [8] (The term DocumentFragment will be used to represent the equivalent data structure in XSLT 1.1). For this style of using DocumentFragments as data structures to work, it must be possible to run path expressions on DocumentFragments to select certain nodes contained in the DocumentFragment. This capability is not present in Xalan (version 2), Saxon (version 6.2.2), MXSML3 or infoteria.

Expressions

A fundamental concept of XSLT is expressions. The specifications for the expressions used in the XSLT language are defined separately from the XSLT specifications. In fact, XPath [22] (the expression specification) is defined as a separate language, which can therefore be used outside of the XSLT domain. The rules for XPath expressions are the same whether they are used in XSLT or in other contexts.

XPath expressions are used to return one of 5 datatypes: boolean, string, number, node or node-sets. The datatype returned by an expression will depend on the nature of the expression. There are 2 fundamental types of XPath expression, which may be combined within a single expression. The value of an expression may be calculated without reference to the source tree (e.g. 1+1). Also, the value of an expression may be equal to or dependent on a subset of the nodes in a source tree, where the nodes satisfy the conditions placed by the rules in the expression. These node-returning expressions are path expressions. Nodes may not be created via an expression, they may only be selected from a pre-existing set of nodes. If these types of operations (computational
expressions and path expressions) are combined, automatic type conversions occur, where possible.

Path expressions consist of a series of node-tests and optionally additional path expressions as related to the context node(s) returned by preceding path expressions (steps) in the complete expression. The context of an arbitrary path expression is determined by the current source node and, in the case where variables are part of the expression, the position of the variable declaration in the style sheet. The current node is the Node that is currently being processed, while the current node list is the set of all Nodes that will be processed before the current context is exited. The current node is always part of the current node list. A node-test determines if the current node satisfies some criteria, for example, if its name is X or if it has an attribute named Y. Path expressions are ways of selecting certain Nodes that satisfy some criteria. An expression may be divided into a series of steps, each separated by a slash or double slash. A node test is performed at each step.

AxisSpecifier. Path expressions start at the current source node and move along a specified axis at each step, determining if the nodes along that axis satisfy an arbitrary node test. These axes include parent, child, ancestor, descendant and sibling. The nodes along that axis which satisfy the node test (which may be a test for the name of the node, type of the node, etc.) are either returned as the result of the expression or are used in the next step of the expression. Steps in an expression are separated either by a slash (“/”) or double slash (“//--”). A slash at the beginning of an expression represents the root node while double-slash at the beginning of an expression represents the descendants of the
root node. There is a rather complicated syntax to determine the semantics of these steps. For a more complete description, see Kay [8].

The nodes that will be either returned or passed to the next step of an expression are, in part, determined by the axis along which the step will operate. These axes include: child, attribute, next-sibling and parent. There are more axes, but they can generally be reproduced by a combination of the former. The child axis consists of all of the immediate children of the current node. The attribute axis consists of all of the attributes of the current node and so on. The axis specifier can either be supplied explicitly or implicitly in an expression. There are also shorthand notations for commonly used axes. For example, an @ signifies the attribute axis and the dot sign (.) signifies the self (current source node) axis. As an example, the expression "/child::employee/@salary" would return all attributes with name salary of the children, with name employee, of the root node. All expressions of this type return an instance of a DOM object called NodeList, which is an augmented array of nodes.

Predicate. An interesting aspect of XPath expressions is the possible use of predicates. Predicates are enclosed in braces ([Expression]) and consist of another expression. A predicate serves as a test of nodes contained in the NodeList returned by a step in an expression. A predicate can be a number, in which case all nodes in the NodeList which correspond to that position in the NodeList are returned. A predicate can also be any other type of expression, in which case the nodes in the NodeList which satisfy the conditions of that expression are returned.

An example of a path expression that uses a predicate is: "/company[1]/@revenue". This expression will return a NodeList consisting of the
Attribute named revenue associated with an Element named company, if the element is first member of the NodeList returned by the “/company” step. The “/company” step returns all of the children Elements of the root node with name company.

Computational expressions. XPath expressions support a limited number of mathematical operations, including standard operations such as multiply, divide, modulus, etc., but do not include more complex operations such as exponentiation and logarithms. Mathematical operations, such as addition and subtraction, produce standard results (1+1=2), with possible type conversions for non-number arguments. The comparison operators (=, !=, >, <, >=, <=) need a little extra explanation however.

Comparisons always return a boolean value of true or false. Comparisons between numbers operate in the standard way. For example, 1 < 2 is true while 1 = 2 is false. The complications arise when comparisons involve node sets. The rules for comparisons involving node-sets are rather complex, with different behaviors indicated for comparisons between different data types (e.g boolean and node-set, string and node-set). For a full description of these rules, see Kay [8], but as an example a description of an equality comparison between a string and a node-set will be given. This operation will become important in the description of the SchemaEngine.

The expression ” //myElement/@myAttribute = ‘stringName’ “ will produce an equality comparison between a node-set, represented by the class NodeList, and a string. The left-hand side of the equation searches the source tree and produces a NodeList consisting of all the Attributes with name myAttribute associated with Elements named myElement. The right-hand side of the equation is a single
quoted string, which is evaluated to a literal that is of type string. The equation will return true if there is a Node in the NodeList whose string value, determined as if by a call to the string function, is lexically equal to the string value, also determined as if by a call to the string function, of any of the nodes contained in the NodeList. The string value of an Attribute is the string value of its associated value property.

For the above expression, and the XML document:

```xml
<doc>
  <myElement myAttribute="stringName"/>
  <myElement myAttribute="otherName"/>
</doc>
```

The resulting NodeList will contain only the Attribute for the first Element child of doc.

Patterns

Patterns are a subset of expressions. Patterns must return a NodeList, while expressions may return other datatypes. Patterns are basically used to determine if a given node (generally the current source node) fits a pattern. This technique is used to determine which template to execute given a current context node. A template definition must have either a pattern or a name associated with it, but not both. Given a collection of templates with a pattern in a match attribute, and a context node, the template or set of templates whose pattern matches the context node is/are determined.

The node N matches a pattern if and only if,
There is a node A that is an ancestor-or-self of N,
Such that evaluating P as an expression
With A as the context node
Returns a node-set that contains N.

[8]
CHAPTER 3
DTDS AND XML SCHEMAS

One of the great powers of XML is that it is extensible. The language is designed so that a particular industry, academic discipline or even individual can create its own vocabulary with XML. Thus, XML documents that conform to a given vocabulary will have a predefined structure that is understandable and has a well known meaning.

The word schema can be used in a general way to describe a set of rules to which a valid document must conform. Schemas are metadata, or data about data. There are 2 types of schemas commonly used with XML documents: 1) the Document Type Definition (DTD) [4], which is well established and 2) the XML Schema language [11], which is a recent proposal. The term XML Schema refers to a particular type of schema meant to replace DTDs, and should not be confused with the general use of the term schema, which can be any method of vocabulary definition.

Document Type Definition

A DTD is generally contained in a plain text file. The method of associating a DTD with an XML document may vary. There can be a reference within an XML document to the DTD that should be used for its validation, or in some implementations, the DTD can be specified on the command line. The DTD can referenced from within an XML document with <!DOCTYPE DocumentDefinition> where
DocumentDefinition is either an external file, or an inline DTD.

In the DTD itself, there must be a document type declaration of the DTD

<DOCTYPE name [ markup]/>

where name is the name of the DTD and markup is the content of the DTD. Allowable content of an XML document is defined with the use of 4 basic markup declarations. They are:

- element type declarations
- attribute list declarations
- entity declarations
- notation declarations

**Element Type Declarations**

Element type declarations specify the element structure to which an XML document being validated must conform. An element declaration specifies the name of a particular element to validate and the model to which that element’s children must conform. An element declaration may look something like this:

```xml
<!ELEMENT nodename content>
```

The `<!>` enclosing tags state that this is a DTD declaration. The first word ELEMENT states that this is an element declaration. The next word is the node name of the element and the last word states what kind of content (children) this element may contain. There are 4 types of content that an element can have: element, any, mixed or empty. If the content type is empty, the element must have no children. A specification of any, means, as stated above, any type of child in any order is allowed. A declaration with element content means only element children are allowed. If the content model is mixed, both character data and element children are allowed. If an element declaration specifies element content, a regular expression language, similar to EBNF [23] is used.
The order of children must be the same as the order that element names appear in the
element content portion of the declaration. Cardinality is specified by regular
expressions. For a more specific element declaration one might write:

```xml
<!ELEMENT foo (a, b?, (c+ | d*))>
```

Here, there is a regular expression that states that the foo element contains an
element child named a, followed by 0 or 1 b elements, which is followed by either 1 or
more c elements or 0 or more d elements.

An element declaration that has a content model of MIXED indicates that
elements of this name may have both character data children and element children. In
this case, the type of children may be constrained, but not their order or number of
appearances.

**Attribute List Declaration**

Attributes may only appear in conjunction with elements in XML documents.
Attribute list declarations can constrain these attributes in the following ways [1]:

- to define the set of attributes pertaining to a given element type
- to establish type constraints for these elements
- to provide default values for attributes

Attribute list declarations are enclosed in angle brackets and begin with
!ATTLIST. Each attribute list declaration contains the name of the element to which it
applies, followed by a list containing attribute names, types and a possible default
declaration. There are 3 possible types for attributes: string type, tokenized type and
enumerated type. String types are indicated with CDATA. This type indicates that the
element may have any character data children. There are a variety of options for
tokenized types, all indicating attributes with values conforming to some pre-defined
XML information item. For example, NMTOCKENS refers to a valid XML name. The
third type is an enumeration of notations or XML names. Here is an example of an
attribute list declaration:

`<!ATTLIST myElement attributeName CDATA #REQUIRED>`

This states that the element myElement must have an attribute called
attributeName with type CDATA.

**Entity Declarations**

An XML document may contain one or more storage units, some of which may
not be XML compliant. These chunks of content are referred to as entities. An entity
may be a block of regular text, a document type definition or the name of an external file
containing text or binary data [10]. The `Document` element that is the root of an XML
document is also an entity.

Entities may be parsed or unparsed, general or parameter, internal or external. A
parsed entity is well-formed XML content, while an unparsed entity may not even be
text.

**General entity declaration.** General entity declarations associate parsed text with a name.
The parsed text is referred to as replacement text and can be used in an XML document
to replace references to its name. General entities are declared with the word ENTITY,
and may be followed by the replacement text in quotes. An internal entity declaration
looks like this:
<!--ENTITY title "A New XSLT interpreter"-->  

If the replacement text is given in an external file, a system identifier (SYSTEM or PUBLIC) follows the name of the entity. For an external entity located at a web URL, the declaration might look like this:

<!--ENTITY title SYSTEM " http://www.cise.ufl.edu " -->

An external general entity declaration may have a public identifier in addition to a system identifier. If this is the case, an XML processor may first use the public identifier to try to construct an alternative URI reference. If it is unable to do so, the system identifier must be used. For example:

<!--ENTITY title PUBLIC
 "~/server/homes/myName/thesis/title"
 "http://www.ufl.edu/~myName/thesis/title" -->

In this case, the XML processor would first try to find the external entity at “~/server/homes/myName/thesis/title”. If this is not successful, the URI “http://www.ufl.edu/~myName/thesis/title” is used. An external general entity declaration may also have an NDataDeclaration. If this is present, the external object is an unparsed entity, otherwise it is a parsed entity. For example:

<!--ENTITY title SYSTEM
 "http://www.ufl.edu/~myName/thesis/title.gif"
 NDATA gif -->

This declaration may be used to reference an image file, for example.

After an entity is declared, it can be referenced later in the XML document with an ampersand followed by the name, like this: &title.
Parameter entity declaration. Parameter entities are only used within DTDs, not in validated XML documents. They are declared with the ENTITY keyword, a percent sign, a name, and the replacement value. The only significant difference in syntax between general and parameter entity declarations is the use of the percent sign instead of an ampersand in parameter entity declarations. For example, an internal parameter entity declaration looks like this:

```xml
<!ENTITY % thesisChapters "introduction CDATA #REQUIRED discussion CDATA #REQUIRED"/>
```

An external parameter entity declaration looks like this:

```xml
<!ENTITY % thesisChapters SYSTEM "http://www.ufl.edu/myName/chapters"/>
```

Notation Declarations

A notation declaration provides a name for a notation, which is used to specify the format of unparsed entities, the format of elements which bear a notation attribute, or the application to which a processing instruction is addressed. A notation declaration also identifies helper applications to process unparsed entities. For example,

```xml
<!NOTATION pdf SYSTEM "Adobe Acrobat"/>
```

XML Schema

Advantages of XML Schemas

The XML Schema language has many advantages over the older DTD model for constraining XML documents. Some limitations of DTDs were listed above and are reproduced here for convenience:
• They are difficult to write and understand
• Programmatic processing of their metadata is difficult
• They are not extensible
• They do not provide support for namespaces
• There is no support for datatypes
• There is no support for inheritance

The XML Schema addresses all of these concerns.

XML Schemas conform to XML syntax, so there is no necessity to learn a new representation format to write and understand them. Also, with the SchemaEngine and other schema inference programs, writing XML Schemas can be automated. XML Schemas use XML specifications, making programmatic processing of schemas possible using the same tools (e.g. XSLT) that are used to manipulate other XML documents. Type definitions in XML Schema are extensible through the use of extensions and/or restrictions of other type definitions. Namespaces are supported through the methodology inherent in XML. The association of XML Schemas with XML documents can be much more sophisticated than that of DTDs. Namespaces can be declared in the top-level schema element, and individual type definitions can be associated with different namespaces (through target namespaces), with each namespace possibly corresponding to a different XML Schema document. XML Schema allows for very robust typing of data. XML Schema has a vast lexicon of pre-defined data types, as well as the capability to define new data types. XML Schema follows an object-oriented model and includes inheritance capabilities.

Additionally, XML Schema documents can express more information about a document than a DTD. For example, an XML schema has more flexibility in constraining the cardinality of the children of an element. It can specify that there must
be between 3 and 4 children of a certain type for a specified element, as opposed to “one or more” (+) as a DTD might specify.

Validation

Schema validity assessment consists of two processes:

- **Local-validity determination** – determining whether specific element or attribute information items conform to the rules specified by a corresponding component in an associated XML Schema document

- **Synthesizing outcome** – determining if the logical structure of an XML document conforms to the specifications in an XML Schema document and augmenting the outcome in accordance to rules in the schema document

The term valid, with its variations, refers to item one above. The term assessment refers to the overall process undertaken by a fully conforming XML Schema validating application, including local-schema validation, assessment of all schema descendants and infoset augmentation. Augmentation may include the synthesis or altering of information that appears in the outcome of validation.

XML Schema Spefication Methodology

The specification for the XML Schema language [11, 24, 25] has 2 levels of description. The most basic level of the description occurs at the abstract data model level. XML Schema is defined in an object-oriented manner, involving definitions of classes that have a rigorous inheritance model. The objects in XML Schema are referred to as components, and are described in terms of the properties each component possesses,
the relationships between components and how a component constrains the content of information items within an XML document being validated.

The next level of XML Schema specification provides for the XML representation of schema components. This description provides details on how schema components should be represented in a plain text document, so that all properties and relationships inherent in the abstract data model of XML Schema are adequately represented. The details of how applications aware of XML Schema are implemented are not defined, but in order to be fully compliant, these applications must be able to meet the standards set out at both the XML representation and abstract data model levels.

Abstract Data Model.

A schema written in the XML Schema language consists of components. An XML Schema component is defined by a collection of properties. A property’s range, or the values it may have, provides that property’s definition. The values of a property in a simple type definition (to be discussed) may be further constrained by facets. An XML Schema document can be considered a labeled directed graph. Components may reference other components, converting what would otherwise be a tree structure into a graph. In a schema document, the root represents the entire schema, every other vertex is a component or literal, and every labeled edge is a property.

Properties may be optional, in which case they need not be present in a component. Properties that are not optional may have a value of absent if they are not present, which is not the same as a non-existent optional property. A non-existent
optional property has no value, while a property that is not optional, but is not defined, is assumed to have a value, which is equal to \texttt{absent}.

Before discussing the full range of components, 2 categories should be introduced: definitions and declarations.

**Declarations.** Declarations are associated by (qualified) names to information items being validated. There is a direct correspondence between a declaration and an information item (element, attribute, etc.) in an XML document being validated.

**Definitions.** Definitions define internal schema components that can be used in other schema components. There is not necessarily a direct correspondence between a definition and an information item in an XML document being validated.

There are 13 specific kinds of components, falling into three groups: primary, secondary and helper. The primary components are as follows:

- Simple Type Definitions
- Complex Type Definitions
- Attribute declarations
- Element declarations

The secondary components, which must have name values, are:

- Attribute group definitions
- Identity-constraint definitions
- Model group definitions
- Notation declarations

The “helper” components provide small parts of other components, they are not independent of their context:

- Annotations
- Model groups
- Particles
- Wildcards
- Attribute uses
These components are discussed in the following sections. The same format is used for every component type except simple type definitions. Simple types are specified in their own document *XML Schema Part 2: Datatypes* [24], and defined in a unique way. Therefore, the section on simple type definitions provides a brief overview of the syntax used to describe simple types and a text description of their specifications.

All other components are described in a way meant to encapsulate their description in the W3C specification document *XML Schema Part 1: Structures* [25]. The properties of these components are listed, with descriptions of the way they would be represented in an actual XML Schema document and a description of their semantics. Also, the term actual value is used in relation to attributes to signify the value of an attribute, as opposed to its name. The term information item refers to an entity that is contained in an XML document and that conforms to XML 1.0 specifications. The term does not distinguish between the plain text representation of that entity and its DOM representation.

**Primary Components**

Primary components provide the basic structure of an XML Schema document. All primary components may potentially be top-level elements (children of the `schema` root). Primary components may be type definitions or declarations.

There are 2 basic kinds of type definition components: simple and complex. Every type definition, except for a unique ur-type definition (named `anyType`), is either a restriction or extension of another type definition. These relationships form a tree, with
the ur-type definition as the single root [25]. A diagram of this tree, known as the Type Definition Hierarchy, can be found in Figure 4.

![Diagram of Type Definition Hierarchy]

Figure 4: XML Schema Type Definition Hierarchy

A type definition is said to be a restriction if it has a one-to-one correspondence of declarations and facets with another type definition, with each declaration or facet restricting the corresponding declaration or facet. Members of type A, whose definition is a restriction of type B, are also members of type B [25]. In other words, a type definition that restricts another type definition makes all declarations or facets more specific. Both complex types and simple types, pre-defined or user-defined, can be restricted or extended.

A type definition that allows for new content, in addition to that allowed by another specified type, is said to be an extension.
A type definition used as the basis for an extension or restriction is said to be the base type definition.

There are 2 types of primary declaration components: element declarations and attribute declarations. These are named components that must correspond to an actual information item (attribute or element) in an XML document being validated.

**ComplexType definitions.** A complex type definition constrains element information items that have children and/or attributes. A complex type definition is represented by an XML element, with name `complexType`. Properties of complex type definitions are represented as attributes and/or children of the `complexType` element. The purpose of the complex type definition is to characterize and constrain the possible characteristics of its corresponding element information item in a document being validated. To accomplish this goal, a complex type definition has the following properties and XML representations:

- **Name** – Optional. Represented by the actual value of the name attribute. If there is no name attribute, the complex type definition is anonymous and must appear as part of the content model of another complex type definition. If the name attribute is present, its value must be unique. Named complex type definitions are not required to correspond directly to element information items being validated, instead they may be used only for internal schema purposes.

- **Target namespace** – Represented by the value of the targetNamespace attribute of the schema ancestor of the component. If no such attribute is present, the target namespace property has value absent. Otherwise, its
value must be a valid XML Namespace that identifies the namespace to which this element belongs. By default, the namespace of components in an XML Schema document will be that reserved by the W3C for XML Schemas

- **Base type definition** – Represented by the actual value of the `base` attribute. This attribute only occurs in conjunction with a derivation method, otherwise it will not appear.

- **Derivation method** – Represented by a child element of the content model with name `extension` or `restriction`. Determines the type of derivation that pertains to children type definitions.

- **Final** – Represented by the list value of the attribute `final`, if this is attribute is present. If the attribute `final` is not present in the complex type definition, the value of this property may be represented by the list value of the attribute `finalDefault` of the `schema` ancestor of the complex type definition. If neither of these attributes is present, the value of the property is absent. If the set that is determined in this way contains the string `restriction`, no further complex type definitions can be constructed which are restrictions of this definition. Similarly for the string `extension`

- **Prohibited substitutions** – Represented as above, except with attribute values of `block` and `blockDefault`. May disallow restriction or extension definitions in the content model.

- **Abstract** – Represented by the boolean value of the `abstract` attribute. If true, this type definition must not be used for validating element information
items, but may be used as a base definition for extensions or restrictions within the XML Schema document. This value defaults to false.

- **Attribute uses** – Represented as attributes in attribute declarations (see Attribute Uses)

- **Attribute wildcard** – Optional. If an `anyAttribute` element is present, the value of the attribute wildcard property will be set according to a set of rules, otherwise the value is `absent`. This mechanism provides more flexible ways of describing attribute uses (see Wildcards)

- **Content type** – Represented by a simple type definition (see simple type definition), a content model (see Content Model) or may be empty. Describes the allowable children for validated element information items

- **Annotations** – Represented by 0 or 1 elements with node name `annotation`. Used to provide comments for humans to read.

All components except the ur-type definition are a restriction or extension of another type definition. A complex type definition however can only be a restriction or extension in one of the following ways:

- The complex type restricts a base type definition
- The complex type extends a simple or complex base type definition
- The complex type restricts the primitive type definition `anyComplexType` 

**SimpleType definitions.** A simple type definition corresponds to an element that has no children or attributes. A simple type definition is a set of constraints on strings, applicable to the value of an attribute information item or an element information item.
The value of an element information is different from its name. It is the value of non-element (CharacterData) children of the element. Simple type definitions also have information about the values that they encode.

Simple type definitions may be represented by an XML element with name simpleType, or by a reference to a built-in simple type. Built-in simple types have predefined names and are located within the XML Schema namespace. For example, the XML Schema element:

```xml
<xsd:element name="myElement" type="xsd:string"/>
```

references the pre-defined simple type string via the attribute type.

Simple type definitions may be of two types: primitive (built-in) or user-defined. The XML Schema language has a number of built-in types, including date, positiveInteger, byte, etc. Each simple type definition is a restriction of some other base type definition or definitions. It may be a restriction of the built in anySimpleType, or a list of other items that are themselves restricted by other simple types.

Data types in an XML document have value spaces. This is particularly important in the case of simple type definitions. A value space is the set of values for a given data type. For example, an element with character data children has in its value space the string values of those children. Equivalently, an attribute has in its value space, its value (the value of name="Bob" is the string Bob). Simple type definitions serve to constrain a data types value space. This is accomplished with facets.

Data types can be constrained by facets. There are 2 types of facets: fundamental and non-fundamental. Fundamental facets serve to constrain the data type, while non-
fundamental facets constrain the values of a data type. For example, an element declaration can have a fundamental facet of decimal:

```xml
<xsd:element name="myElement" type="xsd:decimal"/>
```

A simple type definition can also have non-fundamental facets to constrain the above value:

```xml
<xsd:simpleType name="myConstrainedElement">
  <xsd:restriction base="xsd:decimal">
    <xsd:totalDigits value="4"/>
  </xsd:restriction>
</xsd:simpleType>
```

This definition states that myConstrainedElement must be of type decimal and have no more than 4 digits. There are a large number of built-in simple types (fundamental facets), which are analogous to the primitive types (int, string, double, etc.) of a language like C. There are also a large number of constraining facets (non-fundamental facets).

**Element declarations.** Element declarations are named components that are used to validate element information items of the same name in an XML document. An element declaration is associated with a type definition to accomplish this task. Element declarations may also be used to specify default or fixed values for validated elements and constrain reference relationships among validated elements. These tasks are accomplished with the following properties.

Note: if the `ref` attribute is present, the element declaration becomes a particle.
• **Name** - Represented by the actual value of the `name` attribute. The value of the `name` property in an element declaration must match the names of the elements being validated.

• **Target namespace** – The rules for representing the target namespace property depend on three attributes: `targetNamespace`, `formDefault` and `form`. The first two may be present as an attribute of the `schema` ancestor of the element declaration, while the third may be present as an attribute of the element declaration. The actual value of the `targetNamespace` attribute of the `schema` root will represent the property’s value, unless modified by the `formDefault` or `form` attributes. These attributes may have actual value of qualified or unqualified. The `form` attribute has higher precedence. If either `form` or `formDefault` has value qualified, the target namespace property value is the actual value of `targetNamespace`. Otherwise, (value is unqualified) the target namespace property has value absent. If the target namespace property is defined, elements with the corresponding namespace may be validated. Otherwise, elements with no namespace may be validated by this declaration.

• **Type definition** – Represented by the actual value of the `type` attribute, an anonymous simple or complex type definition, or given default value. The `type` attribute can be used if the type of the element declaration can be resolved to a built-in simple type. If the element declaration has a `complexType` or `simpleType` child, the type definition is that child. If
these conditions are not met, and there is a substitutionGroup attribute, the type definition is that resolved to by the actual value resolved to by the element declaration in the substitutionGroup. If none of these conditions are met, the value of the type definition property is the ur-type definition anyType.

- **Scope** – Optional. Not given explicit representation in XML Schema document, instead the scope of an element is determined by the position of the declaration within the document. The value of the scope property is global if the parent of an element declaration is the schema root element, or if the element declaration contains a ref attribute. Global elements are available throughout the XML Schema document. If there is a complexType ancestor for an element declaration, and that declaration does not contain a ref attribute, the value of the scope property is that complexType. The scope of element declarations within a model group definition is determined when those element declarations are referenced from within a complex type definition. Scope values are used to determine where element declarations can be referenced from within XML Schema documents.

- **Value constraint** – Optional. Represented by the value of either a fixed or default attribute. These attributes are mutually exclusive. If the default attribute is present, and a corresponding element being validated is empty, the post-schema validation value of the validated element becomes the value of the attribute. If the fixed attribute is present, the corresponding
validated element must be empty, in which case the it acts as default, or else the value of the validated element must be the same as the value of the fixed attribute.

- **Nillable** – Represented by the attribute nillable. Its value must be a boolean., and defaults to false if the attribute is not present. If the value is true, elements with attribute nil that have a value of true may be valid even if they contain no content, despite an indication of content in the XML Schema document.

- **Identity constraint definitions** – Represented by a set of children of the element declaration. Constrains uniqueness and reference relationships on elements and attributes (see identity constraint definitions)

- **Substitution group affiliation** – Optional. Represented by the actual value of the attribute substitutionGroup. The value of the attribute must be the name of a top-level element declaration. The element declaration is a member of any group to which the associated element declaration is a member.

- **Disallowed substitutions** – Represented by the actual value of a block attribute, if present. If not present, represented by the value of a blockDefault attribute in the root schema element. If the root has no such attribute, the value defaults to the empty set. The value of this property is a set consisting of restriction, extension and/or substitution. The value #all is equivalent to a set containing all 3 members. This property may be used to prevent the element declaration from additionally validating elements that are extensions and/or restrictions of the element
declaration. It also may prevent the element declaration from validating elements that are in the substitution group headed by the element declaration.

- **Substitution group exclusions** – Represented by the actual value of attributes `final` and `finalDefault`, with the same rules as for disallowed substitutions, except that possible set values are `extension` and `restriction`. This value may be used to disqualify element declarations from being used as the substitution group affiliation.

- **Abstract** - Represented by the actual value of the attribute `abstract`. The value must be a boolean, and defaults to false if the attribute is not present. Same rules as for complex type definitions.

- **Annotation** – same as for complex type definitions

**Attribute declarations.** Attribute declarations are named components that are used to validate attribute information items of the same name in an XML document. An attribute declaration is represented by an XML element with name `attribute`. An attribute declaration is associated with a type definition to accomplish this task. These tasks are accomplished with the following properties.

- **Name** - Represented by the actual value of the `name` attribute. The value of the `name` property in an attribute declaration must match the names of the attributes being validated.

- **Target namespace** – same as for element declarations, except that the target namespace applies to attributes.
• **Type definition** – Represented by the value of a `type` attribute, if present. Otherwise, represented by a simple type definition child, if present. Otherwise, the ur-type definition **anyType**. If a type definition is supplied, it must resolve to a simple type.

• **Scope** – Not represented explicitly. If the attribute declaration is a child of the `schema` element, the value of the scope property is global. If the attribute declaration has as an ancestor a complex type definition, the scope is that definition. If the attribute declaration is part of an attribute group definition, the value of the scope property is **absent**.

• **Value constraint** – Represented by an attribute of `fixed` or `default` or the absence thereof. These attributes are mutually exclusive. If neither attribute is present, the value of the value constraint property is **absent**. If the `fixed` attribute is present,

• **Annotation** – same as for complex type definitions

**Secondary Components**

Secondary components may be used to provide a definition for groups of information items (attribute and model group definitions) that are not directly validated, but referenced within an XML Schema document. Secondary components may also be used for constraining programmatic aspects of XML documents (notation declarations and identity-constraint definitions).

**Attribute group definition.** Attribute group definitions name a collection of attribute declarations that can be referenced within complex type definitions. Attribute group
definitions are trees represented by a root `attributeGroup` node. Attribute group definitions are not directly validated, but internally referenced within an XML Schema document to simplify schema construction.

The properties of attribute group definitions are as follows:

- **Name** – Represented by actual value of `name` attribute. Provides name by which attribute group definition may be referenced.

- **Target namespace** – Represented by the actual value of the `targetNamespace` attribute of the parent `schema` element.

- **Attribute uses** – Represented by the collection of attribute uses associated with the attribute declaration children of the `attributeGroup` root. The value of the attribute uses property is the union of the attribute uses of all attribute declaration children. This property makes up the content of an attribute group definition.

- **Attribute wildcard** – same as for complex type definitions

- **Annotation** – same for complex type definitions

Identity-constraint definitions. Identity constraint definitions provide for uniqueness and reference constraints with regard to the contents of multiple element and attributes. An identity-constraint definition is a tree represented by a root element with a name corresponding to its identity constraint category. Identity-constraint definitions are used in conjunction with element or attribute declarations to constrain the corresponding declarations in regard to XML 1.0’s capabilities for `ID/IDREF`, `key` and also to constrain elements for uniqueness and location within a document.

Identity constraints have the following properties:
- **Name** – Represented by actual value of `name` attribute. The name property must uniquely identify an identity constraint definition within the XML Schema document.

- **Target namespace** – Represented by the actual value of the `targetNamespace` attribute of the `schema` ancestor of the identity-constraint definition. This value is used in conjunction with the name property to uniquely name an identity-constraint definition.

- **Identity-constraint category** – Represented by the name of the root element that comprises the identity constraint definition. Must be one of `key`, `keyref` or `unique`. If it is `unique`, the content identified by the selector property must be unique, as well as the result of evaluating the `fields` property. If it is `key`, the selector property must be unique as well as having such fields. If it is `keyref`, there must be a correspondence between the selector tuples.

- **Selector** – Represented by the actual value of the `xpath` attribute of the `selector` element that is a child of the identity-constraint definition root. This value must be a restricted XPath expression. The selector property is used to constrain the appearance of nodes in an XML document relevant to an element being declared, so that these nodes satisfy the conditions set forth in the XPath expression.

- **Fields** – Represented by the actual value of the `xpath` attribute `field` element that is a child of the identity-constraint definition root. This value
must be a list of restricted XPath expressions related to each of the nodes returned by the expressions in the selector property. The expressions in the field property must resolve to a single node whose content or value (which must be of simple type) is used in the constraint. A list of expressions may be specified.

- **Referenced key** – Represented by the actual value of the `refer` attribute. Only allowed in `keyref` identity-constraint definitions. This property is used to assert a correspondence between the content specified by `selector`, of the tuples resulting from the evaluation of `fields`, and those of the referenced key.

- **Annotation** – same as above

**Model group definitions.** Model group definitions associate a model group and possible annotations to a name. A model group definition is represented by an XML element with `name group`. The main purpose of model group definitions is to provide a mechanism for complex type definitions to reference model groups, so that a frequently occurring model group need not be reproduced many times.

Model group definitions have the following properties:

- **Name** – Represented by the actual value of a `name` attribute. The name uniquely identifies a model group definition.

- **Target namespace** – Represented by the actual value of the `targetNamespace` attribute of the parent `schema` element.
• **Model group** – a model group which is a child of the `group` element (see model group).

• **Annotation** – same as above

**Notation declaration.** Notation declarations reconstruct XML 1.0 notation declarations. Notation declarations are represented by a `notation` element. These elements serve the same purpose as notations in DTDs. The XML Schema notation declaration component is simply meant to represent all of the information in an equivalent XML 1.0 notation.

The notation declaration has the following properties:

• **Name** – Represented by the value of a `name` attribute. Specifies the name of the notation.

• **Target namespace** – Represented by the actual value of the `targetNamespace` attribute of the parent `schema` element.

• **System identifier** – Represented by the actual value of the `system` attribute.

• **Public identifier** – Represented by the actual value of the `public` attribute.

**Helper Components.**

Helper components are used in conjunction with other components to provide utilities that provide more information about a component, or ease the use of those components within an XML Schema document.

**Particles.** Particles contribute to the definition of content models. Particles serve as wrappers for the three types of XML Schema elements that allow `minoccurs` and
maxoccurs attributes: element declarations that are not children of the schema root, model groups and wildcards.

The properties of particles are as follows:

- **Minoccurs** – Represented by the actual value of a minoccurs attribute. Its value must be a non-negative integer. The value of this property indicates the minimum number of times that an information may appear in a certain context. The default value for this property in the absence of a minoccurs attribute is 0.

- **Maxoccurs** - Represented by the actual value of a maxoccurs attribute. Its value must be a non-negative integer or unbounded. The value of this property indicates the maximum number of times that an information may appear in a certain context. The default value for this property in the absence of a maxoccurs attribute is 1.

- **Term** – Represented as the associated element declaration, model group or wildcard. A term may be a model group, which may contain particles, so this is a circular definition.

Attribute uses. Attribute uses serve a similar purpose as particles, except they pertain to attribute declarations. Attributes uses serve as wrappers around attribute declarations that occur within a complex type definition. Attribute uses constrain occurrence and defaulting behavior of its associated attribute declaration. The representation of attribute uses is an attribute use associated with an attribute declaration, or the values of attributes fixed and default as specified in attribute declarations. The use attribute may have
actual value of optional, prohibited or required. If the value is optional, the attribute may or may not appear in a validated element. If the value is prohibited, the attribute is not allowed to appear in the validated element. If the value is required, the attribute must appear in the validated element.

Attribute uses have the following properties:

- **Required** – Represented by actual value of use attribute, which must be required. If the actual value of use is required, this attribute must be present in validated elements.

- **Attribute declaration** – Represented as above (see attribute declaration). Attribute uses appear as attributes of an attribute declaration. The value of the attribute declaration property in an attribute use is the attribute declaration to which it is attached.

- **Value constraint** – corresponds to the fixed and default attributes of an attribute declarations as specified in attribute declarations.

**Model groups.** Model groups are used to constrain the children of element information items. Model groups may be defined explicitly with model group definitions (see model group definitions) or anonymously as subtrees (with root all, choice or sequence) of complex types.

Model Groups have the following properties:

- **Compositor** – Represented as an element with name all, choice or sequence. The compositor also represents the entire model group. The type of compositor provides information about how its children should be
interpreted. If the compositor is sequence, the compositor’s children should be interpreted as representing a sequential list, with the particles appearing as children of an element being validated must appear in the same sequence as found in the model group. If the compositor is choice, the compositor’s children represent a disjunctive list, where exactly one of the compositor’s children appears as a child of the validated element. If the compositor is all, the children represent a conjunctive list, where either zero or one of all of the compositor’s children appears as children of a validated element.

- **Particles** – The representation for particles is discussed in its own section. In general, the particles in a model group are element declarations or other model groups, so model groups has a circular definition. The particles constrain the types and order of children of an element being validated.

- **Annotations** – same as above

**Wildcards.** Wildcards allow for validation of elements and attributes dependent on their namespace name, but independent of their local names. The representation of a wildcard is either an any or anyAttribute element. Wildcards may occur as the term property of a particle.

The properties of wildcards are:

- **Namespace constraint** – Represented by the actual value of the namespace attribute. In the absence of a namespace attribute, the value defaults to
any. The allowable actual values of the namespace attribute are ##any, ##other, ##targetNamespace, ##local, or a space de-limited list of Namespace names. The particle associated with a wildcard will be used to validate information items that correspond to those indicated by one of these values.

- **Process contents** – Represented by the actual value of the processContents attribute. The possible values are strict, lax, or skip. In the absence of a processContents attribute, the property’s value defaults to strict. This property provides processor directives as to what degree of rigor should be used in validating information items described by the wildcard.

- **Annotation** – same as above

Annotation components. Annotations provide information to human readers of an XML Schema document, but do not affect the semantics of the schema. They are equivalent to comments in other languages (Java, Perl, C, etc.). Annotations are represented as annotation, appinfo or documentation elements within a schema document.

Annotations have the following properties:

- **Application information** – Represented as a sequence of element children of the root annotation node, if the root is of type appinfo.

- **User information** - Represented as a sequence of element children of the root annotation node, if the root is of type documentation
• **Attributes** - Represented as a sequence of attribute information item children of the root annotation node, if the root is of type `annotation`. These attributes are those allowed as wildcards, and are associated with the root `annotation` node, or for the enclosing item within which the annotation component is located.

An example of an XML Schema can be found in Appendix C.
CHAPTER 4
THE XSLTENGINE

There are many XSLT interpreters available, so why build a new one? In the course of constructing an XSLT program to infer XML Schema documents, it became apparent that current implementations were insufficient, and in fact there might be a necessity to expand the XSLT language itself. There are open source XSLT interpreters that make it possible to create extension functions, and the possibility of modifying an existing implementation was considered. However, as it would be necessary to make alterations in the processing model of XSLT, as well as possibly creating new extension functions, it was decided that creating an XSLT interpreter from the bottom up was the best option.

Perl

The XSLTEngine is written in Perl, a language that is most commonly used on Unix machines, but has recently become completely portable to other operating systems, and has wide acceptance in writing server-side web applications. The program is written in an object-oriented style, which is supported by Perl in a way different from the classical object-oriented languages such as Java and C++. Objects are constructed using Perl packages. An instance object of a class is created through the use of an arbitrarily
named constructor method (customarily named new) which uses the Perl command bless to associate a reference to a built in data structure representing the object. The Perl hash data structure is typically used as the underlying object representation. Perl supports much of the standard functionality expected in object-oriented languages (such as inheritance, encapsulation, etc) while avoiding the restrictions inherent in other languages. Thus, Perl gives you enough rope to hang yourself if you’re not careful, and enough freedom to express yourself fully if you are careful.

Overview of XSLTEngine

The XSLTEngine is divided into 7 sections, 6 of which reside in their own directories and correspond to a subset of the XML/XSLT specifications:

1) **Document Object Model (DOM)** – a collection of classes that specify the types of allowable nodes in an XML document. The DOM is used in constructing XML trees (corresponding to the source document, stylesheet and result tree) and providing low-level methods for navigating and manipulating these trees.

2) **Parser** – the parser is used to read in an arbitrary document and produce an equivalent DOM tree

3) **Expression** – the expression classes fully implement the XPath expression language. These expressions really perform two functions: they can either be used to evaluate traditional expressions, such as mathematical functions and string comparisons, or navigate a DOM tree and return a result based on the composition of that tree
4) **Function** – there are a number of functions defined in the official XSLT specifications that perform various operations, such as returning the name of the current node or providing string operations. Their implementations are located in this directory.

5) **Pattern** – Patterns are used primarily in choosing templates but also may be used in selected other ways.

6) **Instructions** – instructions are identified in the stylesheet with the prefix `xsl`, as in `<xsl:apply-templates/>`. These are predefined commands that direct the processors actions.

7) **XSL** – The XSL portion of the program provides the `XSLT` class, which controls the execution of the program. Also, various other utility classes are contained here.

An overview of the XSLTEngine’s structure can be found in Figure 5:
The XSLT Engine Implementation

There are some points that need to be clarified for understanding the following discussion. First is the use of the term attribute. This term has different meanings depending on the domain in which it is being applied. In object-oriented programming, data associated with a class is called an attribute. Therefore an attribute of, say, the class Template, refers to an internally stored variable of that class. In the XML domain, nodes of type Element may have associated nodes of type Attribute. Therefore given the XML line `<employee branch="engineering"/>`, a parser would create an Element object with node name employee and associate to it an Attribute object which has a node name of branch and string value engineering. The concepts are similar, but have subtle distinctions.
Many times in the XSLTEngine a class is created which has the same name as an entity in the XML/XSLT domain. When a word is bold-faced in a fixed width font (ApplyTemplates), the thing being referenced is the class being used in the XSLTEngine. If the word is not capitalized, and in fixed-width font (xsl) the thing being referenced is the DOM node corresponding to an XML object. If the word(s) are surrounded by less than and greater than symbols, and is in fixed-width font, the thing being referenced is an XSLT instruction. For instance, <xsl:apply-templates select="1"/> would refer to an XSLT instruction Element object which has the node name apply-templates, the namespace prefix xsl and an associated Attribute object with node name select and node value 1. In this domain, there might be a reference to the attribute of the XML node apply-templates, which would be select.

If a term, such as node, is used without being bold-face, capitalized or fixed-width font, then the term is referring to the general concept, rather than a specific instance of, for example, a Node.

Also, the XSLTEngine is constantly dealing with three DOM trees: the source tree, the style tree and the result tree. When a reference is made to a style node, that means a DOM node which is found in the style tree. A source node is a DOM node residing in the source tree and a result node is a DOM node that resides in the result tree.
Initiation and Preprocessing

The class \texttt{Main} is used to initiate the program. It initially takes arguments from the command line specifying the XML source document and the XSL stylesheet. An instance of the \texttt{Parser} class is instantiated for each of these documents. A \texttt{Parser} can only parse one document. Each \texttt{Parser} then parses its document and returns the root node of the resulting DOM tree. These trees are then sent to a newly created, unique instance of an \texttt{XSLT} object, which will serve as the wrapper class for the whole program execution. The trees are stored in \texttt{XSLT} attributes as references to the tree root nodes.

The \texttt{XSLT} object initiates some preprocessing by creating a \texttt{TemplateList} object and instructing it to preprocess the candidate templates, initializing local and global variable tables (\texttt{LocalVarTable} and \texttt{GlobalVarTable}) and a stack (\texttt{LVTStack}) for local variable tables. These are stored as attributes of the \texttt{XSLT} object. The \texttt{XSLT} object then identifies, evaluates and stores global variables and parameters.

The nodes that are the immediate children of the stylesheet node, which is the only child of the root of the stylesheet in a well-formed XML stylesheet, are candidates for templates, parameters and variables. The preprocessing of the templates proceeds by first creating an object of type \texttt{Template} class, then template nodes (\texttt{<xsl:template>}) are identified and for each template a new object of type \texttt{Template} is created. The new \texttt{Template} object is associated with the style node from which it is derived. The attributes of the template style node are then analyzed. A template must have either a \texttt{name} attribute, for use with the instruction \texttt{<xsl:call-template>}, or a \texttt{match} attribute for use with the instruction \texttt{<xsl:template>}.\texttt{match}.\texttt{matches}(\texttt{node}())
<xsl:apply-templates>. If neither attribute is present, it is an error. Depending on which attribute is present in the style node, the internal attributes of the Template object are set. If there is for instance, no match attribute in the style node, the corresponding Template attribute remains undefined. A template may also have a mode, which is sometimes used in determining which template should be selected to process a node-set. If there is a mode attribute in the template style node, then a corresponding attribute is set in the Template object. All templates also have an associated priority, which are used to select the appropriate template if there is more than one template that matches a node. If the template has a match attribute, the value of the priority is determined by analyzing the specificity of the expression associated with the match attribute. If the template instead has a name attribute, the priority is set to a default value. As part of the template preprocessing, the priority of each template is determined and a corresponding Template attribute is set to that value.

A template may have associated parameters for use on being called by an <xsl:call-template> instruction. The value of a parameter can not be determined until run time, so the parameter values cannot be evaluated during the preprocessing stage, but in order to make the runtime evaluation of these parameters easier, some information about them is stored during preprocessing. All <xsl:param> style nodes must be the first children of the template style node. If the param style nodes have any preceding siblings, it is an error. So, first of all, this condition is checked. If this condition is satisfied, a ParamShell object is created for each <xsl:param> node. The ParamShell is associated with its corresponding style node, and then added to a ParamTable, which stores all of the ParamShells in an array. The ParamTable
is then stored in the Template object. All of these Template attributes, except for the ParamTable, are used in retrieving the correct template during stylesheet processing. The ParamTable is used for setting the correct value for the parameters when a template is called with parameters.

Once all of the attributes of the Template object have been calculated, that Template is stored in an array attribute of the TemplateList object. This TemplateList object is then stored in the XSLT object. Only one TemplateList is created for a stylesheet, and only one XSLT is instantiated for a transformation. The XSLT object serves as a type of wrapper or driver for the entire program.

Environments for Variable Evaluation

The scoping of variables in XSLT, as stated above, is somewhat different from the standard imperative model. Templates, in some respects, are similar to functions or methods in other languages, but there are significant differences. A template does provide an environment for variables, in the sense that a variable declared in one template cannot be referenced from within another template. However, scoping rules are dependent on the tree-like structure of XSLT programs. A variable reference is in scope if the node containing the reference is in a following sibling node of the variable declaration, or a descendant of one of these nodes. Therefore, two variables with the same name may be declared within a template, while both declarations provide non-overlapping scopes. Also, nested templates are not allowed, and so one template environment can not be contained within another template environment. Therefore, the environment model of variable evaluation as described in Abelson and Sussman [26] is
not entirely appropriate. However, a modification of the environment model of evaluation is used. This was made because, due to the recursive nature of XSLT, there may be a recursion stack of templates being executed, and it is necessary to hide variables from one template from other templates.

A stack (LVTStack) of environment frames, or variable tables, (LocalVarTable) is maintained within a unique object instantiation of the XSLT class for the declaration and evaluation of variables. Whenever a new template begins its execution, a new variable table is pushed onto the stack. When a variable declaration is encountered in that template, it is added to the variable table on top of the stack. When a variable reference is encountered, it looks for the variable in the variable table on top of the stack. When a template has finished its execution, all variables within the stack frame corresponding to that template have gone out of scope, so the variable table is popped off the stack and discarded.

Processing Model for the XSLTEngine

Once this preprocessing is done, all within the constructor of the class, the method execute is called. Then control is passed from Main to XSLT. The execute method proceeds by searching for a template which matches the root node of the source tree, and then executing that template. The driving function, processCurrentStyleNode for the engine is defined in XSLT. This method is used for processing template bodies, which are defined in the stylesheet. These template bodies may be either the nodes of a tree which has as its root an <xsl:template> node (a template), or the nodes of a tree which has as its root one of a collection of other
**XSLInstructions** (such as `<xsl:attribute>`). In the second case, there is a template body, but no template. The method `processCurrentStyleNode` requires three input parameters:

1) **Current Source Node** – the node which is currently being processed in the internal DOM representation of the XML source document

2) **Current Style Node** - the node which is currently being processed in the internal DOM representation of the XSLT style sheet document

3) **Current Result Node** - the node which is currently being processed in the internal DOM representation of the result document

These three parameters are required in most XSLTEngine operations, demonstrating the need for the processor to constantly be aware of its position in each of the three trees necessary for a transformation. These three values together generally make up the context of an operation, and will heretofore be referred to as such.

The `processCurrentStyleNode` method does a preorder tree traversal (which in the XSLT language corresponds to a document order traversal) of a subtree of the stylesheet tree, starting at the root of that subtree. The root of the subtree to be traversed is given as the current style node parameter. If the current style node is the root of the entire stylesheet, the entire stylesheet will be traversed. The `processCurrentStyleNode` method takes appropriate actions depending on the node it is currently visiting. There are 2 possible types of action the method may take: one type of action is initiated on encountering an instruction and the other is initiated on encountering a non-instruction.
If the node is in the XSLTransformation domain, generally signified by a prefix of xsl (as in `<xsl:name>`), a new **XSLInstruction** is created, with reference to the current context, and then executed. The **XSLInstruction** may update the result tree by appending children to the current result node.

**XSLInstructions** are responsible for processing all of their children, and therefore, the `processCurrentStyleNode` method skips the children of **XSLInstructions**. This is important, because if an **XSLInstruction** processes its own children, and then the `processCurrentStyleNode` method also processes them on return from the recursive call, these children will be processed twice or more, which is not correct. It should be noted here that **XSLInstructions** may be nested, and that **XSLInstructions** also frequently use the `processCurrentStyleNode` method. Each nested instruction is responsible for its own children, so that the problem of double processing is avoided.

If the `processCurrentStyleNode` method encounters a non-instruction node in the stylesheet, this node is cloned and appended to the current result node, while the current result node is set to the newly appended node. By the use of recursive calls, the structure of the non-instruction stylesheet tree is recreated in the result tree. It is important to clone the nodes in the stylesheet, or the structure of the stylesheet will be altered by the processing.

**Expressions**

The XSLTEngine implements expressions by following closely the specifications outlined by the World Wide Web Consortium. There is a class defined for each
production, with the class having the same name as the production. For example, the class designed to implement path expressions is named **PathExpression**. Every class in the expression hierarchy is a subtype of the class **Node**. The class **Node** as defined in the DOM was not specifically meant to serve this function. However, as the evaluation of an expression consists of the construction of a tree, and the class **Node** has many useful functions, this choice was made. The construction of the expression tree does not strictly follow the rules of the construction of a DOM tree, in that the children, and attribute nodes of the expression tree are not set in the same way. This option should not be ruled out for future builds of the XSLTEngine though.

The different classes in the expression hierarchy do not inherit from a common **Expression** superclass, as there are no operations common to all subtypes, but the expression classes are all coded as if they were implementing a common interface. As Perl does not enforce interface rules, this design decision is enforced only by the consistent application of the principles of an interface. If this program were ported to a language that does enforce interfaces, there would have to be at least 2 separate interfaces, as path expressions and mathematical expressions take different parameters for their respective methods.

Each class in the expression hierarchy implements 3 methods: **initialize**, **parse** and **evaluate**. The evaluation of any expression begins by creating an object of type **Expr**. The three interface methods are called in succession on the **Expr** object, causing a cascading effect in which each of the three methods are called in succession on the expression subclasses. The **initialize** method is used simply to pass the appropriate string expression (such as “/child::employee”) to the class that is responsible
for evaluating that expression or sub-expression (sub-set of the original expression). Immediately after an object is initialized, it is parsed. The parse method takes advantage of the pattern matching capacity of Perl. The string expression is compared to various regular expressions to determine the nature of the expression. Substrings of the incoming string expression are isolated and used to instantiate, initialize and parse the appropriate objects that are subtypes if the Expr class. Once these objects have been parsed, they are stored as attributes of the calling object, thereby creating a tree. Then, when the original Expr object has its evaluate method called, each child object in the tree optionally performs some processing and calls the evaluate method of its children. The current context is passed through the evaluate method to all children of the expression. The context passed through the evaluate method lacks the result node, but includes a reference to the XSLT wrapper object. This version of context is passed because expressions are not allowed to update the result tree, but variable references need to access the variable tables stored in the XSLT object. Leaves of the tree simply return a value, and all other nodes return the value of the evaluate method of their children, after some optional processing. The value of the expression is the value returned by the original Expr object’s evaluate method.

The evaluate interfaces of the mathematical expression classes and the path expression classes is different in the types of parameters that are passed. Whereas mathematical expressions will return a value such as a string, number or boolean, path expressions will return a NodeList. Therefore, when it becomes obvious that the only possible value that can be returned by an expression (generally in the LocationPath class) a NodeList is created and passed by reference to all children of that object.
Then each child in the expression tree can simply modify that `NodeList`, instead of creating a new `NodeList` in each class and returning it.

Patterns

Patterns are used to determine if a particular node or set of nodes satisfies an expression. As such, a pattern is the subset of expressions that return a `NodeList`, and a pattern is satisfied if the current node is in the `NodeList` that results from evaluating that expression. The definition would imply that in order to evaluate a pattern in a given context, one would need to evaluate the expression and then determine if the current node is contained in the `NodeList` resulting from the expression. The evaluation of patterns can be optimized in ways that will be discussed shortly.

The syntax of patterns is a subset of expressions, and therefore the design of pattern evaluation is similar to that used for expressions. Once again, there is a class designed for each production rule in the pattern specification. The same interface is used, with 3 methods: `initialize`, `parse` and `evaluate`. The same cascading style is used where each object initializes, parses and evaluates its children. The real difference comes in the way the `evaluate` method operates. Whereas an expression may return a number, string, node, node list or tree fragment, a pattern may return only true or false. Also, whereas an expression is evaluated left to right, a pattern is evaluated right to left. The reasoning behind this design is so that a pattern may be short-circuited if a false value is ever encountered. If a rightmost step of a pattern ever evaluates to false, the entire pattern is false and need not be evaluated any further.
The evaluate method proceeds like this:

The right most part of any step must be a node test, such as is this node named employee. For instance, in the pattern:

```
//*[company/branch[@city='NewYork']]/employee
```

the pattern is testing whether the current node is named employee and is a child of a branch element which has an attribute named city whose value is NewYork and the branch element is a child of a company element which can be found anywhere in the document. To evaluate this pattern left to right would mean:

1) find all company elements in the document

2) then find all the branch elements that are children of these company elements and also have an attribute named city which has a string value equal to ‘New York’

3) then find all the children elements whose name is employee of these branch elements

4) then determine if the current node is in the node set returned by this expression.

By evaluating the pattern right to left, if the current node does not have a node name of employee, the pattern is false, end of story, return false. Otherwise, continue evaluating each step right to left until the value is false, there are no more steps to evaluate or the root node is reached.
Instructions (XSLT Elements)

The XSLT specification outlines a number of instructions that must be implemented in a compliant XSLT processor. These instructions are identified by a prefix `xsl` that is associated with a namespace defined by the XSLT specifications. To simplify the discussion, all elements with a prefix of `xsl` are assumed to be an instruction. The extension mechanism of XSLT allows for new instructions to be defined for particular implementations of XSLT, but these must be associated with a namespace (and prefix) other than `xsl`.

In the XSLT Engine, the processing of instructions are initiated in the method `processCurrentStyleNode` located in the class XSLT, when a style node is encountered with the prefix `xsl`. When one of these nodes is encountered, an `XSLInstruction` object is created. The constructor of `XSLInstruction` takes the current context as a parameter. The current context includes the current style node, current source node, current result node and the XSLT wrapper class. The current context is stored in attributes of the `XSLInstruction`. The class `XSLInstruction` is a subtype of the class `Node`, so that the methods defined in this class will be available to `XSLInstructions`. Once the `XSLInstruction` has been instantiated in `processCurrentStyleNode`, its method `execute` is called. The method `execute` examines the node name of the style node with which it is associated and, using Perl’s pattern matching capacity, determines what type of instruction should be created. Once this is determined, the appropriate class is instantiated with the current context and its `evaluate` method is called.
There is a class defined in XSLTEngine for all of the instructions in the XSLT specifications. Thus `<xsl:apply-templates>` has a class called `ApplyTemplates`. All of these classes inherit from the `XSLInstruction` class. The only function that they inherit are the constructor, which is used to pass the current context to the newly instantiated subclass. Once one of these methods is created in the `execute` method of `XSLInstruction`, its `evaluate` method is called. The `evaluate` method is defined differently for each `XSLInstruction` subclass, according to the requirements of that instruction. Each `XSLInstruction` may update the result tree is appropriate.

**Functions**

Functions are allowable as parts of expressions and patterns. They are identified by having a legal XSLT name (QName), followed by open and closed parentheses. For example, `name`. Some functions are defined in the XPath specification and some are defined in the XSLT specification. Only those in the XPath specification are guaranteed to be in compliant XSLT processors.

There is a class `Function` in the XSLT engine. The class function serves a similar role as the class `XSLInstruction`. Whenever a new function is created, a new `Function` class is instantiated. The constructor of the `Function` class takes a function name, which is the type of function which should be created, an argument, which is the object on which the function should operate, and an optional argument list, if there is more than one argument to process. Then the method `evaluate` is called on the `Function` class, which instantiates a function object of the appropriate subtype, and
returns the value of calling the evaluate function of that newly created subtype. The implementation of each subtype will be specific to that function's purpose.

Variables

In XSLT, variables cannot be updated. There is no assignment operator. The value of a variable is assigned on in its inception and cannot be changed until the variable goes out of scope. Variables are created by means of the `<xsl:variable>` instruction. A variable instruction must have a name attribute that corresponds to the name of the variable. If there is no name attribute, it is an error. The value of a variable is created at run time either by the value of an expression located in a select attribute, or, if there is no select attribute, by means of the template body which is comprised of the elements which make up the children of the variable instruction. XSLT therefore uses dynamic type binding of a sort. In this implementation, variables are not explicitly bound to a type, but instead bound to a value. That value can be of any XSLT type, except in certain circumstances when a value of a given type is required (such as the requirement that a `Pattern` must return a `NodeList`). In these cases, dynamic type checking is performed.

Whenever an `<xsl:variable>` instruction is encountered by `processCurrentStyleNodes`, an `XSLInstruction` object of type `Variable` is instantiated and passed as parameters the current context. The value of the variable is immediately determined in the current context, according either to the expression identified by the `select` attribute, or if that attribute is not present, the template body contained within the variable instruction. Once the value of the variable has been
determined, the **Variable** object is added to the **LocalVarTable** residing on top of the **LVTStack**, which is stored in the **XSLT** wrapper object.

The **LocalVarTable** is a hash, as most Perl objects are. Therefore, variables are hashed into the **LocalVarTable**, with the name of the variable as a key. The value associated with key is an array, which is treated as a stack with Perl push and pop operations. The model for variable storage is therefore a stack-dynamic one, which supports recursive calls. This is important because XSLT is a highly recursive language. The addition of a variable to the **LocalVarTable** consists of a series of steps.

1) If the **Variable** being added has a unique name, an array (which is treated as a stack) is created. The newly instantiated Variable is pushed onto the stack and the stack is added to the Hash with the variable name as a key.

2) If a **Variable** with the same name has been previously declared, the new Variable is pushed onto the stack that has as its key that variable name.

A variable value is retrieved when a variable reference is encountered. Variable references are a type of expression and are preceded by a dollar sign (e.g. $varName). It is for this reason that the wrapper class **XSLT** must be sent as a parameter to the **Expr** class and all of its subclasses. Any expression may potentially contain a variable reference, and if it does, the **Expr** classes must have access to the **LVTStack**, which is contained in the wrapper **XSLT** object. The evaluation of a variable reference consists of checking to see if the current **LocalVarTable** has a hash entry with the name of the
variable reference. If there is such an entry, there may be a stack of variables with the same name. Since the last encountered `<xsl:variable>` instruction will be the top item on the LocalVarTable stack, that variable should be the first item popped off the stack and that variable should be the one in scope.

The retrieval of a variable takes advantage of the expression classes already implemented for scope checking. A variable is in scope if the variable reference is a following sibling, or the descendant of a following sibling of the Variable instruction. Therefore, before returning the popped variable, the current LocalVarTable runs an expression “following::*”, given as a context node the style node referenced by the variable. If the style node that contains the variable reference is in the NodeList returned by this expression, the variable is in scope.

In the method processCurrentStyleNode of the XSLT class, a type of garbage collection occurs after the processing of each node in the stylesheet. Once all the children of a given style node have been processed, any variables that are created as children of that style node will no longer be in scope. Therefore, a search is made of the current LocalVarTable for any variables that have as their referenced style node a child of the recently processed style node. These variables are removed from the current LocalVarTable. The program would continue to work if this process was not conducted, but the variable stack could grow unreasonably large in a large program. Therefore, this is a memory management issue.
Parameters

The processing of parameters is more complicated than the processing of variables. The complexity stems from 2 principal considerations. One is that there are more complicated rules for establishing the values of parameters than there are for establishing the values of variables. Parameters of a template are generally used as the result of an `<xsl:call-template>` instruction. These instructions may optionally have `<xsl:with-param>` instructions as their first children. If there the `<xsl:with-param>` instructions have any preceding siblings that are not `<xsl:with-param>` elements, it is an error. If the called template has `<xsl:param>` elements with the same name as the `<xsl:with-param>` elements, the value of that parameter is the value of the expression in the corresponding `<xsl:with-param>` as that expression is evaluated in the context of the `<xsl:call-template>` instruction. If the `<xsl:with-param>` does not have a select attribute, then its value is determined by the template body comprised of its descendants. If the `<xsl:with-param>` has no children, its value is the empty string. If the called template has an `<xsl:param>` that does not have a corresponding `<xsl:with-param>`, then the value of that parameter may be determined by a default select expression or template body, otherwise its value is the empty string.

In order to process parameters, the `<xsl:call-template>` instruction takes advantage of the preprocessing done for templates. First, all of the immediate children of the `<xsl:call-template>` style node are processed to see if they are `<xsl:with-param>` nodes. If so, a `WithParam` object is created for each one. The value of that `WithParam` object is determined by evaluating its select expression, or its
template body, in the current context. Then these WithParams are added to a WithParamTable. Then ParamTable of the called template is retrieved. The WithParamTable and ParamTable are passed to a LocalParamTable. Then the method updateLocalVariables is called in the LocalParamTable. The WithParam elements are compared with the ParamShells to determine which have the same name. For every WithParam that has a corresponding ParamShell, a new Variable object is created and pushed onto the corresponding LocalVarTable Stack. If there is a ParamShell which does not have a corresponding WithParam, the value of the ParamShell is determined by whatever default expressions or template bodies it has, as evaluated in the context of the <xsl:param>. Once these values are determined, a Variable object is created and pushed onto the appropriate LocalVarTable stack.

Expansion of XSLT Expression Evaluation

As mentioned above, the main difficulty in producing computational programs in XSLT 1.0 is the limited ability to produce complex user-defined data structures [21]. It is, however, possible to produce lightweight trees and capture them in variables. These lightweight trees are known as DocumentFragments. Although the type DocumentFragment has been eliminated in XSLT 1.1, the same concept remains and the same type of data structure is constructed in the same way. An example follows:

```xml
<xsl:variable name="lightTree">
  <xsl:element name="element1"/>
  <xsl:element name="element2">
    <xsl:attribute name="attrib1">attribVal</xsl:attribute>
  </xsl:element>
  <xsl:element name="element3"/>
</xsl:variable>
```
In XSLT 1.1 the example above would return an object of type *DocumentFragment*, which is the root node of the above tree. The XSLT 1.1 specifications call for the above example to return a *NodeList* containing only one node, the *Element*, root of the above tree.

The tree constructed by this code (in XSLT 1.0) is shown in Figure X. The XSLT 1.0 specifications are unclear on what operations can be performed on this variable, while in XSLT 1.1 they are explicit. The developers of XSLT interpreters have decided that path expressions cannot be run against *DocumentFragments* (at least Saxon 6.2.2, Xalan 2, MSXML3 and infoteria). In order to use this tree as a means of storing information, it is necessary to be able to access sub elements of the tree. This should be possible with an expression such as “$lightTree/element2/*”. This expression should return a *NodeList* containing the *Elements* named *element1* and *element2*. A good test to see if an interpreter is capable of this operation is to run the following program:

```
<xsl:stylesheet ...>
  <xsl:template match="/">
    <xsl:variable name="lightTree">
      <xsl:element name="element1"/>
      <xsl:element name="element2">
        <xsl:attribute name="attrib1">attribVal</xsl:attribute>
        <xsl:element name="element3"/>
        <xsl:element name="element4"/>
      </xsl:element>
    </xsl:variable>
  </xsl:template>
</xsl:stylesheet>
```
If the output of running this style sheet is not 2, the interpreter cannot run path expressions effectively against a DocumentFragment.

There may be some trepidation in allowing this type of expression. This could occur because the value of a variable may be of any data type. What, for example, would the value of the above expression be if $lightTree were a boolean value false? In the XSLTEngine, the value would simply be an empty NodeList.

The algorithm produced for constructing an XML Schema from an arbitrary XML document makes heavy use of passing DocumentFragments as parameters to named templates. The use of DocumentFragments as data structures requires that these document fragments be freely navigated and manipulated. This requires 2 fundamental operations on DocumentFragments: 1) path expressions must be applicable to DocumentFragments, so that specific nodes in a DocumentFragment can be accessed and 2) DocumentFragments must be convertible to NodeList. These are related in that if expressions can be applied to DocumentFragments, the result can be a NodeList.

The assumption is often made that path expressions will only be run against the source tree. If path expressions are somehow hardwired to only access the source tree, these types of expressions will either throw an error, or not run correctly. The XSLTEngine simply evaluates the variable, and if the variable occurs in a path
expression, an automatic type conversion is applied. If the value of the variable is a DocumentFragment, the current source node is changed to the DocumentFragment root and passed to the remaining steps in the path expression.

**Extension Elements**

In order to make manipulation of DocumentFragments easier, three extension elements have been created:

1) smb:append-copy
2) smb:replace-copy
3) smb:add-attributes

These are all instructions, which are executed like xsl elements, such as <xsl:copy-of> or <xsl:value-of>. They are all given the prefix smb to distinguish the namespace as distinct from the xsl namespace. The capability to process these instructions is built into the XSLTEngine.

These instructions were designed to conform to the basic XSLT processing model. Their functions follow closely the functioning of 2 built in XSL instructions: <xsl:copy-of> and <xsl:copy>. These instructions will be discussed in detail.

The function of <xsl:copy-of> should be explained. The syntax is <xsl:copy-of select=“Expression”>, where Expression is an XPath expression which returns either a NodeList or a Node. CopyOf will then make a new tree, which is appended to the current result node. When the result of an expression is a Node, the newly constructed tree has as a root a copy of the selected node. The descendants of this node are copies of all the descendants of the selected node, where
logical structure (parent, child, etc.) is maintained. Thus the new tree is isomorphic to the original tree. If the result of an expression is a NodeList, deep copies are made for each successive node in the NodeList, and the roots of each copied tree are appended to the current result node.

The current result node to which the copied tree will be appended depends on the context in which the CopyOf instruction is executed. If the instruction is executed outside of a variable declaration, the current result node will be in the final result tree. If the <xsl:copy-of> instruction occurs in the context of a variable, the current result node will either be a DocumentFragment node created by the <xsl:variable> instruction or possibly a new Element node, created by the <xsl:element> instruction, where the new Element node, has as an ancestor the DocumentFragment created by the <xsl:variable> instruction.

The <xsl:copy> instruction operates differently than the <xsl:copy-of> instruction. Copy makes a shallow copy of the current source node. That is, a copy of the current source node is appended to the current result node, but parent/child and other relationships (excepting attributes) are not retained. Only the current source node is appended. The Copy instruction cannot choose which node to copy via an expression, but can only copy the current source node. The current result node and current source node can be manipulated as previously outlined.

In the XSLT processing model, the source tree (the internal representation of the XML document to be transformed), cannot be updated, but it can be accessed. Accessed means that nodes in the source tree can be returned as values of expressions. These returned nodes can be used as information to construct the final result tree, but the
original source tree cannot be changed. Also, in XSLT variables cannot be updated. However, a new variable can be constructed based on the content of another variable. DocumentFragments can be constructed in variable declarations primarily with the use of three instructions:

1) <xsl:element> - which creates a new element node with characteristics established by accompanying attributes and children.

2) <xsl:copy-of> - described previously

3) <xsl:copy> - described previously

In the case of DocumentFragments, the content of a DocumentFragment, as saved in a variable, can be copied to a new variable by use of the instruction <xsl:copy-of select="variable"/>. The tree structure of the new variable will depend on the context in which the CopyOf instruction is called.

The following instructions were designed so as not to violate the fundamental aspects of the XSLT processing model. First of all, the instructions do not change the values of existing variables. Instead, they make a copy of some existing variable, along with some possible alterations. Next, they are not allowed to change the source tree. They operate by selecting target nodes to alter, but if these nodes are not part of a DocumentFragment (that is they are part of the source tree), an error is thrown.

AppendCopy

The purpose of AppendCopy is to create a copy of an existing DocumentFragment, with a selected tree appended to selected nodes of the original
**DocumentFragment.** That is, an updated copy of a `DocumentFragment` is saved into another variable.

**AppendCopy Syntax.** The syntax of `AppendCopy` is as follows:

```xml
<smb:append-copy target="Expression"
     select="Expression"/>
```

The prefix `smb` indicates that this is an extension element. The name of the instruction is `append-copy`. There are 2 attributes, with names `target` and `select`, the value of each being an expression.

1) **target** – the target expression selects the node(s) which will be appended to

2) **select** – the select expression selects the nodes or `DocumentFragments` that will be appended

The instruction is always an empty element. It can have no children.

**AppendCopy semantics.** `AppendCopy` does not alter variables, which means it cannot append nodes directly to a `DocumentFragment`. It creates a copy of a `DocumentFragment`, with copies of selected target nodes appended to with copies of nodes in the select expression.

As mentioned above, a context switch can be created with the instruction `<xsl:variable>`. If the value of the variable is specified by a template body, a new current result node of type `DocumentFragment` is created, and used to process the template body. `AppendCopy` can only be used in the construction of a variable via a template body. So, the instruction checks to see if the current result node is a
**DocumentFragment.** If it is, the instruction is being called in the context of a variable, otherwise, an error is thrown.

There are 2 expressions in **AppendCopy** – target and select.

1) **target** – the target expression selects node(s) of a **DocumentFragment**, which will be appended to in the generated copy

2) **select** – the select expression selects nodes, either from the source tree or a **DocumentFragment**, which will have deep copies made and appended to the target expression nodes

The target expression must evaluate to an **Element, DocumentFragment, or NodeList** composed entirely of **Elements** and/or **DocumentFragments**, or an error is thrown. These are the only types of nodes, other than **Documents**, that can have children, and therefore the only types of nodes that can be appended to. **Documents** are not accepted, because the source tree, style tree cannot be altered. The final result tree cannot be altered by this instruction. If the selected target node is an **Element**, the root of the tree that contains that **Element** is checked. If it is not a **DocumentFragment**, an error is thrown. The nodes of the target expression must be a **DocumentFragment** or **Element** that has as its root a **DocumentFragment** because only trees that are saved in a variable can be appended to.

The semantics of the instruction varies somewhat depending on the results of evaluating the target expression. If the result of the evaluation is a **DocumentFragment**, copies of all the children of the **DocumentFragment**, are appended to the current result node. The **DocumentFragment** that is the root of the
target expression is not appended, because a DocumentFragment cannot have a DocumentFragment child. Then the nodes of the select expression are appended to the current result node. If the selected target node is an Element, the nodes of the tree containing the target element are copied to the current result node, with nodes of the select expression appended to the nodes selected in the target expression. If the target expression is a NodeList, the trees containing target nodes are copied to the current result node in order, with appropriately appended nodes. Thus a target expression that contains nodes from 2 separate DocumentFragments will result in a new single tree, with the children of each DocumentFragment being the children of the newly created DocumentFragment.

The select expression can evaluate to any XSLT supported data type, except Attribute. Attributes do not exist in a parent/child relationship, and so cannot be appended. If the expression evaluates to a string, number or boolean, a CharacterData node is created with the string value of the selected node. If the expression evaluates to an element or DocumentFragment, a deep copy is appended to all of the target nodes. If the expression evaluates to a NodeList, each item in the NodeList is appended in order to all of the target nodes.

Figure 6 shows the result of the following XSLT code:

```xml
<xsl:variable name="z">.
  <smb:append-copy target="$x//A" select="$y//B"/>
</xsl:variable>
```
Figure 6: Append-Copy

**ReplaceCopy**

The purpose of **ReplaceCopy** is similar to **AppendCopy**. However, instead of appending a selected node to a target node, **ReplaceCopy** replaces a target node with a selected node.

**ReplaceCopy Syntax.** The syntax of **ReplaceCopy** is as follows:

```xml
<smb:replace-copy target="Expression"
select="Expression"/>
```

The prefix `smb` indicates that this is an extension element. Then name of the instruction is `replace-copy`. Then there are 2 attributes, with names target and select, the value of each being an expression.

1) **target** – the target expression selects the node(s) which will be replaced
2) **select** – the select expression selects the nodes or Document Fragments that will replace the target node(s)

The instruction is always an empty element. It can have no children.

**ReplaceCopy** semantics. **ReplaceCopy** does not alter variables, which means it cannot directly replace nodes in a **DocumentFragment**. It creates a copy of a **DocumentFragment**, with selected target nodes of the original tree replaced by copies of nodes in the select expression.

The error checking methods of **ReplaceCopy** are identical to those of **AppendCopy**. That is, the target node(s) must have as a root a **DocumentFragment**. The current result node must be a **DocumentFragment**. The nodes in the select expression can be of any type except **Attribute**. The node(s) in the target expression must be of type **Element** or **DocumentFragment**.

There are 2 expressions in **ReplaceCopy** – target and select.

1) **target** – the target expression selects nodes of a **DocumentFragment**, which will be replaced in the generated copy

2) **select** – the select expression selects nodes, either from the source tree or a **DocumentFragment**, which will have deep copies made and appended to the target expression nodes

The semantics of the instruction varies somewhat depending on the results of evaluating the target expression. If the result of the evaluation is a **DocumentFragment**, the entire **DocumentFragment** is replaced with a new **DocumentFragment** consisting of the nodes in the select expression.
If the selected target node is an **Element**, the nodes of the tree containing the target element are copied to the current result node, with nodes of the select expression replacing the nodes selected in the target expression.

If the target expression is a **NodeList**, the trees containing the target nodes are copied to the current result node in order, with appropriately replaced nodes. Thus a target expression that contains nodes from 2 separate **DocumentFragments** will result in a new single tree, with the children of each **DocumentFragment** being the children of the newly created **DocumentFragment**, except as replaced with nodes from the select expression.

The select expression can evaluate to any XSLT supported data type, except **Attribute**. Attributes do not exist in a parent/child relationship, and so cannot be used to construct a tree. If the expression evaluates to a string, number or boolean, a **CharacterData** node is created with the string value of the selected node. If the expression evaluates to an element or **DocumentFragment**, a deep copy is appended to all of the target nodes. If the expression evaluates to a **NodeList**, the selected nodes in the target expression are not added to the newly created tree, and in their place each item in the select expression **NodeList** is appended in order to the parent of target nodes.

Figure 7 shows the result of the following XSLT code:

```xml
<xsl:variable name="z">
    <smb:replace-copy target="$x//A" select="$y//B"/>
</xsl:variable>
```
AddAttributes

The purpose of AddAttributes is similar to AppendCopy. However, instead of appending a selected node to a target node, AddAttributes adds select attributes to a target node(s).

AddAttributes syntax. The syntax of AddAttributes is as follows:

```xml
<smb:add-attributes target="Expression"
            name="Expression" value="Expression"/>
```

The prefix smb indicates that this is an extension element. The name of the instruction is add-attributes. Then there are 3 attributes, with names target, name and value, the value of each being an expression.
1) **target** – the target expression selects the node(s) which will be have attributes added

2) **name** – the name expression determines the name of the attribute to be added

3) **value** – the value expression determines the value of the attribute to be added

The instruction is always an empty element. It can have no children.

Add Attributes semantics. `AddAttributes` does not alter variables, which means it cannot directly alter nodes in a `DocumentFragment`. It creates a copy of a `DocumentFragment`, with selected target nodes of the original tree altered by adding attributes.

The error checking methods of `AddAttributes` are similar to those of `AppendCopy`. That is, the target node(s) must have as a root a `DocumentFragment`. The current result node must be a `DocumentFragment`. The nodes in the name and value expressions can be of any type. The node(s) in the target expression must be of type Element.

There are 2 expressions in `AddAttributes` – target and select.

1) **target** – the target expression selects nodes of a `DocumentFragment`, which will have attributes added in the generated copy

2) **name** – the name expression evaluates to one of the XSLT supported data types. The string value is used as the name of the added attribute
3) **value** - the value expression evaluates to one of the XSLT supported data types. The string value is used as the value of the added attribute

The target expression must evaluate to an **Element** or **NodeList** of **Elements**. If the result of the evaluation is any other type, it is an error.

The name and value expressions can evaluate to any XSLT supported data type. The value that is used to create the attributes is the string value of the results of the expression, constructed by a call to the XSLT function string. There are a number of rules set out to define the operation of the string function (see [8]). For example, if the data type is the number 2, the string value is the string 2. If the data type is an element, the string value is the concatenation of string values of all **CharacterData** descendants, in document order, of that **Element**.

Figure 8 shows the result of the following XSLT code:

```xml
<xsl:variable name="z">
    <smb:add-attributes target="$x/\text{A}$" name="att" value="val"
</xsl:variable>
```
Simulation of Extension Functions with Standard XSLT

The SchemaEngine makes use of some extension functions, but these can be replaced with standard XSLT templates. The advantage of the extension functions is that they can accomplish the same result as standard XSLT methods with less code written, and there is a slight decrease in overhead. It might take several templates to accomplish what an extension function can, so using the extension function saves the time of template lookup and multiple instruction execution.

The functions of the above extension instructions can alternately be performed by manually constructing trees within the template bodies of `<xsl:variable>` instructions. The manual construction would involve invoking templates that append
nodes to the result node. If the template invocations come from within an
<xsl:variable> instruction, the result will be captured in the variable.

One could use the <xsl:call-templates> instruction, passing as a
parameter the node(s) to be appended, replaced or altered, or one could use the
<xsl:apply-templates select ="x" mode="y"> instruction. The use of
the mode attribute of templates can serve a similar function as the
<xsl:call-template> instruction. That is, templates that perform specific
functions can be selected according to the nature of the operation that is required.

The drawback to this approach is that a separate template must be written for
every possible node transformation, and a decision must be made for every node in the
tree as to whether that node is to be altered. This is a very labor-intensive process, and is
more easily accomplished with a single instruction.
There are 2 reasons for the construction of the SchemaEngine. First of all, as outlined above, schemas are very important to the full exploitation of XML technology. XML Schema is superior to DTDs, and so there is a need for a tool to infer XML Schema documents. The second reason is that XSLT is a new programming language and its capabilities and limitations need to be fully explored. The conventional wisdom is that XSLT is good for presenting XML documents, but is not suited for computational programs. The inference of a schema from an XML document is a problem that demands a highly computational program, and thus serves as an appropriate test bed for trying to develop a computational program in XSLT.

The SchemaEngine Implementation

Overview

The SchemaEngine is an XSLT program that is dependent on the XSLTEngine. Other XSLT interpreters will not successfully process the SchemaEngine program because the program relies on the ability to run path expressions against DocumentFragments and contains extension functions.

The inference process consists of providing three inputs, in the form of plain text
files, to the XSLTEngine, which then outputs the inferred XML Schema document. One may then run customization scripts on the XML Schema document to refine the schema as desired. See figure 9.

Figure 9: XML Schema Inference

In inferring a schema, the SchemaEngine must make a fundamental decision about each element in the XML document being processed. The decision to be made is whether that node merits a complex type definition or a simple type definition. The requirements for constructing each of these definitions are very different.

The primary concern in constructing complex type definitions is to represent the logical structure pertaining to a set of equivalently named elements. To infer this structure, one must interrogate the children and attributes of a collection of elements.
The primary concern in constructing simple types is to interrogate the character data children of the element, then either correlate the data to one of the built-in simple types, or construct a new type based on other simple type definitions.

The construction of complex type definitions will involve the construction of many other XML Schema components, such as model groups and attribute declarations, while the construction of simple type definitions minimally only involves the determination that the elements are simple types. This is because a valid simple type definition can be represented by an element declaration with no explicit simple type identified. The value of the type definition property of the element declaration will default to anySimpleType, which is valid for all simple types. A valid complex type definition however must adequately represent the relationships exhibited by the elements it describes.

Design of XML Schema Document Output

An XML Schema document is itself an XML document and can be described by a schema. There is, in fact, an XML Schema document for XML Schema documents. In order for an XML Schema document to be valid, it must conform to the rules in the schema for schemas [25].

There may be many XML Schema documents that are valid for a given XML input document. The SchemaEngine produces XML Schema documents that are valid, but conform to a certain style. There are no anonymous type definition components. All type definitions occur as top level elements (children of schema element). Element and attribute declarations are not given specified types, so that the type definition properties
of these declarations default to the ur-type base definition. The actual value of type definition name properties are equal to the element/attribute names of the elements/attributes they are meant to validate.

This format provides for valid XML Schema documents, but ones that are not sufficiently descriptive to capture all relevant details of most document classes. However, the improvement of produced XML Schema documents is accomplished through post-processing. Since XML Schema documents are themselves XML documents, post-processing can be accomplished through the use of custom XSLT scripts, that provide the specific output desired by the user. The generality of the initial output of the SchemaEngine is a conscious decision to allow for such customization. Additionally, the XSLT specifications provide the ability for one XSLT program to import templates from other XSLT programs through the use of the <xsl:import> instruction. Potential users could create their own custom scripts, import them into the SchemaEngine, and thereby customize its performance. An example of a customized schema is given in Appendix D.

Initialization

The SchemaEngine program is initiated by the creation of three variables, which are used throughout the remainder of the program. The first variable is created with the following instruction:

```xml
<xsl:variable name="XMLList" select = "//*"></xsl:variable>
```

This instruction creates a NodeList and associates it with the variable named XMLList. The XMLList NodeList contains every Element that is in the source
tree, or the document for which a schema is being created. The order of the elements in the NodeList is document order. In the context of the source document, this means elements that are encountered first in lexical order are placed first in the NodeList. Thus for the XML document:

```
<root>
  <element1>
    <element2/>
    <element3/>
  </element1>
  <element4/>
</root>
```

The NodeList will have the order: root, element1, element2, element3, element4. In the context of the internal tree representation of the document, this is a pre-order traversal. All the parent/child and attribute relationships are maintained in XMLList.

The second variable that is initialized is the XML Schema which is to be refined. For the time being, we will assume that this is an empty schema. That is to say, this is the only document for which a schema is being constructed. In that case, the variable declaration will look something like this:

```
<xsl:variable name="resultTree">
  <xsl:element name="schema"/>
</xsl:variable>
```

Appropriate namespaces and attributes can be added as desired.

The third variable is called count and is initialized to the value 1.
Processing model

The variable XMLList controls order of execution. This is accomplished using the third variable called count, which is used to iterate over the XMLList NodeList. The three initialized variables XMLList, resultTree and count are passed to a named template driver called process. This template looks at each node of XMLList in turn and makes a decision based on whether the current node in XMLList is represented in the DocumentFragment resultTree. On the first iteration (in the case of an empty schema), the DocumentFragment contains no type definitions, so the decision is made to add the current node in XMLList to the resultTree variable. This is accomplished by calling the named template addNode, passing the current node of XMLList (which is to be added), the resultTree (which is to be extended) and the count, which is updated at the end of execution. Within addNode, decisions are made on which type definition (simple or complex) would be appropriate for this node. Based on this decision, one of a number of other named templates is called, which will update the resultTree variable appropriately. When these templates are finished updating the resultTree variable, they make a recursive call to process. They pass the required three parameters. XMLList is passed as is. This variable is never changed. It must be passed as a parameter though so that its value is not lost. The second parameter is the updated resultTree and the last parameter is an incremented count.

On subsequent iterations, the named template process checks to see if a type definition has already been created for the current node in XMLList. If not, addNode is called again. If there is already a type definition in the result tree for the current
XMLList node (determined by looking at the name attributes of the type definitions), then the template process calls the template mergeNode with the same three parameters. The node (type definition) in resultTree that corresponds to the current node in XMLList, is compared to the current XMLList node, to determine if there are any differences in the two. The new information in the current node is used to create a new copy of the result tree, which takes account of the new information. The decisions on what changes to make to the result tree are made by other templates. Once the resultTree has been updated, the template process is called again, with the new result tree, the original XMLList, and an incremented count.

When the last element of XMLList has been processed, the resultTree is copied to the output (Result Node), and the program terminates.

Comparing New Nodes with Previous Nodes

There are many cases in the SchemaEngine where it is necessary to determine if a node currently being examined has already been defined or declared. This situation arises, for example, in the main process template for construction of type definitions, in the processChildren template for construction of model groups and in the processAttributes template for construction of attribute uses.

All these templates perform the task of identifying repeated nodes in the same manner. First a path expression is run against the DocumentFragment that represents the result under construction (e.g. the resultTree parameter in the process template). The result of the path expression is a NodeList of strings (or attributes) that contain the names of the relevant nodes in the DocumentFragment. The resulting
**NodeList** is compared to the string value of the currently examined node’s name. The comparison occurs within the `test` attribute of an `<xsl:when>` instruction. The string comparison follows the semantics outlined in the XSLT expression section of chapter 2.

For example, the `process` template has the following code that produces a `NodeList` containing the `name` attributes of all simple and complex type definitions in the `resultTree DocumentFragment`. The `resultTree` is the final schema under construction.

```xml
<xsl:variable name="attributeList"
  select="$resultTree//xsd:complexType/@name|
          $resultTree//xsd:simpleType/@name"/>
```

Once this variable has been set, the presence or absence of a type definition for the current node can be determined like this:

```xml
<xsl:choose>
  <xsl:when
    test="name($XMLList[$count])=$attributeList">
    <!--there is already a type definition for this node -->
  </xsl:when>
  <xsl:otherwise>
    <!--no type definition is present -->
  </xsl:otherwise>
</xsl:choose>
```

**Construction of Complex Type Definitions**

A key aspect of complex type definitions is the model group property. The model group specifies information about the element children of an element corresponding to the complex type definition.
There are three possibilities for the type of model group: all, choose or sequence. The model group property of a complex type definition is represented by an element with one of the above three names. This element must be a child of the complexType definition element that represents a complex type definition. The semantics of these possible model groups is described in the section on XML Schemas.

The SchemaEngine produces a model group of type sequence by default. This is because the sequence model group can reproduce the semantics of the all and choice model groups. Element declarations that appear as children of the model group element may be wrapped in particles. Particles have the properties minoccurs and maxoccurs. If an element appears only sometimes as a child of an element information item corresponding to a complex type definition, the particle corresponding to the child element can have the value of its minoccurs property can be set to zero. If the element occurs multiple times maxoccurs can be set to unbounded. By setting minoccurs to 0 when appropriate, other content models can be simulated by the sequence model group.

Construction of Model Groups. Therefore, when a complex type definition is first defined, the sequence of its element children is constructed in the sequence content model. The construction of a sequence content model is carried out in a template named processChildren. The operation of this template mirrors the operation of the SchemaEngine as a whole. There are three parameters to the template: sequence, children and childCount. The sequence parameter is a DocumentFragment, the children parameter is a NodeList and the childCount parameter is a
number. The `processChildren` template may be called from more than one other template, with the results varying on what initial values are passed to the template.

The `children` parameter is a list of all of the child nodes of the element for which a content model is being produced. The `childCount` parameter is initialized to 1 in the calling template, and is used to iterate over the `children`. Each node in `children` is processed in turn, updating the `sequence` variable by producing an element declaration in the form of `<xsl:element name="elementName">` where `elementName` is the node name of the corresponding element information item. The `Element` named `element` is then appended to the `sequence Element`.

After each child node is processed, the `childCount` variable is incremented and `processChildren` calls itself with the modified `sequence` parameter and the incremented `childCount` parameter. If the original `sequence DocumentFragment` is an empty element, the modified `sequence` will be a new model group. If the initial `sequence DocumentFragment` is another model group, the resulting `sequence` will be the old model group updated with new information. When the last node in `children` has been processed, the `processChildren` template copies the modified `sequence DocumentFragment` to the current result node. It is important that the initial call to `processChildren` be nested in a variable declaration, so that the `sequence DocumentFragment` be saved in a variable. Once the model group has been constructed, it is pasted into the complex type definition with either `<smb:append-copy>` or `<smb:replace-copy>`.
In the main process template, each node is examined and decisions are made as to what model groups are constructed for each node. If after creating a complex type definition, subsequent elements with the same node name are encountered in the source document, the children of the newly encountered node are compared with the previously defined type definition, and appropriate maxoccurs and minoccurs values are introduced.

If a previously defined simple type has the same node name as a subsequently encountered element information item that is a complex type, a new complex type definition is constructed. Then, element declarations (and associated particles) with the actual value of minoccurs set to 0 are added to a newly created model group. The newly created model group is then appended to the newly created complexType node that represents the complex type definition.

Construction of Attribute Declarations and Uses. The handling of attributes is performed in a manner similar to model groups. When a complex type definition is created for an element information item where no previous corresponding type definitions have been found, attribute declarations are created for each attribute of the element. Construction of an attribute declaration consists of creating an Element with these characteristics <xsl:attribute name="attributeName"> where attributeName is the name of the attribute. The attribute Element(s) are then appended to the complex type definition node complexType with <smb:append-copy> or <smb:replace-copy>.

During the processing of nodes in the variable XMLList in the template process, newly encountered element information items that are complex types, and
which have the same node name as a previously defined complex type (as determined by looking at resultTree), are compared. If an attribute does not appear in one or the other of the complex type elements, an attribute use is defined, with its associated attribute declaration. The attribute use is created by adding an attribute (named use) with actual value optional to the attribute declaration.

If a new element information item that is a complex type is encountered, which has the same node name as a previously defined simple type, a use attribute with value optional is added to all attribute declarations in the newly defined complex type.

**Construction of SimpleType Definitions**

When an element is encountered in XMLList that has no attributes or element children, it is a simple type. There are two possible avenues for dealing with this situation. A simple type definition can be constructed, or a top-level element declaration with reference to a built-in or user-defined simple type definition can be constructed. If a simple type conforms to one of the built-in XML Schema primitives, an element declaration is sufficient. For example:

```xml
<xsd: element name="myElement" type="xsd:string"/>
```

If one wishes to restrict or extend a primitive data type, the simple type definition is called for.

```xml
<xsd:simpleType>
  <xsd:restriction base="decimal">
    <xsd:totalDigits value="4"/>
  </xsd:restriction>
</xsd:simpleType>
```
In the initial construction phase, a named, to-level simple type definition is created for each simple type element. These definitions can be converted to element declarations in a post-processing phase, or if the user wishes, he can make more specific restrictions or extensions to the simple type definitions.

**Refining Schemas**

One objective for this project is to create a framework where XML Schemas can be refined given additional valid XML documents. The SchemaEngine accomplishes that objective.

The major data structure that is used for this program is a `DocumentFragment`, which is used to represent the schema under construction, model groups under construction and attribute declarations under construction. If instead of using an empty schema to initialize the `resultTree` variable, one uses a previously defined XML Schema, the result is a new schema which is a combination of the provided schema and one generated for a new XML document.

The `resultTree` variable is initialized by the XSLT function `document`, which reads a specified file, parses it, and returns the parsed document as a `DocumentFragment`. This value is then used to initialize the processing. The `resultTree` variable is treated throughout the SchemaEngine as a previously defined schema document, which is incrementally built according to new information provided by specific nodes, examined one at a time. There are no special measures that need to be taken for the SchemaEngine to refine a previously defined schema document except to provide that document as input to initialize the `resultTree` variable.
CHAPTER 6
DISCUSSION

The major goals set out for this project have been accomplished. It has been demonstrated that the XSLT programming language can be used to produce complex, computational programs. A modification to the XSLT language (the ability to run path expressions against DocumentFragments), that had not been part of XSLT 1.0, and was built into the XSLTEngine during the completion of this project, has shown itself to be a useful addition to the language. This is proven by the fact that the XSLTEngine is capable of interpreting a highly computational program using this expansion, and that the W3C has independently recommended the exact same modification to the language. Also a program has been written that performs a useful task, namely inferring and refining XML Schema documents from arbitrary XML document inputs. Additionally, this project demonstrates the power of the XML family of technologies.

Advantages and Disadvantages of the XSLT Approach

At this time, the primary disadvantage of undertaking large projects using XSLT is that XSLT is not a mature language. The specifications for XSLT have only been recently produced (and are still subject to change), so vendors have not had time to produce optimized implementations. Also, the language will be unstable for a while. These problems should disappear with the passage of time.

Using XSLT as a language to infer schemas has the philosophical advantages
already outlined, namely that using XSLT keeps everything in the XML domain. XSLT programs, XML Schemas and the XML input documents themselves are all XML compliant. The same tools can be used to handle all facets of the process.

This arrangement has further, concrete, benefits. Since XSLT is designed to handle XML documents, refining schemas becomes more straightforward. Original XML Schemas are easily represented in XSLT and can be used in the XSLT program without extra processing to convert the schema into some separate internally designed representation.

Keeping the whole process within the XML domain also allows for customization in great detail. The possibility of customizing restrictions or extensions on simple types is a perfect example, where users may want to impose restrictions of arbitrary rigor on an XML Schema document.

**Future Work**

The XSLTEngine does not fully implement XSLT, but only provides the capabilities necessary to complete this project. Future work should include making the implementation fully compliant with the XSLT standards. The framework for the XSLTEngine is such that to add most if not all of the remaining functionality required by the XSLT standards, all that must be done is to create new classes corresponding to XSLT instructions (elements) and functions not yet implemented. The processing model to handle these instructions and functions has already been implemented. Some notable instructions that should be added are `<xsl:key>`, `<xsl:import>` and `<xsl:apply-exports>`.
The SchemaEngine could be further improved by providing a library of templates that could be used to customize the inference of XML Schema documents. The feasibility of such an approach has been proven, but not very many such templates have been written. These library templates could be used in conjunction with an \texttt{<xsl:import>} instruction, or simply by running the templates against the product of the SchemaEngine.
APPENDIX A
EXAMPLE XML DOCUMENT

<recipeBook author="John Smith">
  <publisher name="BooksUnlimited" city="New York"/>
  <distributors>
    <BigBooks/>
    <BigBooks city="gainesville">
      <street>3rd Ave</street>
      <phone>555-1212</phone>
    </BigBooks>
  </distributors>
  <recipe contributor="bill">
    This is a CharacterData element
    <name>Hamburger Surprise</name>
    <ingredient>Ground Beef</ingredient>
    <ingredient> salt and pepper </ingredient>
    <time>15 minutes</time>
  </recipe>
  <recipe contributor="scott">
    This is a CharacterData element
    <name>Chicken Pot Pie</name>
    <ingredient>Cubed Chicken</ingredient>
    <ingredient>Pie Crust</ingredient>
    <time>2 hours</time>
  </recipe>
</recipeBook>
APPENDIX B
SAMPLE DTD FOR RECIPE BOOK EXAMPLE IN APPENDIX A

<DOCTYPE exampleDTD[

<!ATTLIST recipeeBook
  author  #REQUIRED
  >

<!ATTLIST publisher
  name   #REQUIRED
  city    #REQUIRED
  >

<!ATTLIST BigBooks
  city  #IMPLIED
  >

<!ATTLIST recipe
  contributor  #REQUIRED
  >

<!ELEMENT recipeeBook (publisher, distributor, recipe+)> 
<!ELEMENT publisher EMPTY> 
<!ELEMENT distributors (BigBooks+)> 
<!ELEMENT BigBooks (street, phone)?> 
<!ELEMENT street #PCDATA> 
<!ELEMENT phone #PCDATA > 
<!ELEMENT recipe (name, ingredient+, time)> 
<!ELEMENT name #PCDATA > 
<!ELEMENT ingredient #PCDATA > 
<!ELEMENT time #PCDATA > 

]/>
<xsd:schema version="1.0" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="recipeBook">
    <xsd:sequence>
      <xsd:element name="publisher">
        <xsd:attribute name="city"/>
        <xsd:attribute name="name"/>
      </xsd:element>
      <xsd:element name="distributors">
        <xsd:sequence>
          <xsd:element maxOccurs="unbounded" name="BigBooks"/>
        </xsd:sequence>
      </xsd:element>
      <xsd:element name="recipe">
        <xsd:attribute name="author"/>
        <xsd:attribute name="contributor"/>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
<xsd:complexType name="BigBooks">
  <xsd:sequence>
    <xsd:element minOccurs="0" name="street"></xsd:element>
    <xsd:element minOccurs="0" name="phone"></xsd:element>
  </xsd:sequence>
  <attribute use="optional" name="city"></attribute>
</xsd:complexType>
</xsd:schema>
APPENDIX D
POSSIBLE CUSTOMIZED SCHEMA DOCUMENT OF SCHEMA IN APPENDIX C

Note: Changes are in bold face

```xml
<xsd:schema version="1.0" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="recipeBookType">
    <xsd:sequence>
      <xsd:element ref="publisherType"/>
      <xsd:element ref="distributors"/>
      <xsd:element maxOccurs="unbounded" ref="recipeType"/>
    </xsd:sequence>
    <attribute name="author"/>
  </xsd:complexType>
  <xsd:complexType name="publisherType">
    <attribute name="city"/>
    <attribute name="name"/>
  </xsd:complexType>
  <xsd:complexType name="distributorsType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" ref="BigBooksType"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="recipeType">
    <xsd:sequence>
      <xsd:element ref="name"/>
      <xsd:element maxOccurs="unbounded" ref="ingredients"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```
<xsd:element ref="time">
</xsd:element>
</xsd:sequence>
<attribute name="contributor">
</attribute>
</xsd:complexType>
<xsd:complexType name="BigBooksType">
<xsd:sequence>
<xsd:element minoccurs="0" ref="street">
</xsd:element>
<xsd:element minoccurs="0" ref="phone">
</xsd:element>
</xsd:sequence>
<attribute use="optional" name="city">
</attribute>
</xsd:complexType>
<xsd:element name="street" type="xsd:string">
</xsd:element>
<xsd:element name="phone" type="xsd:string">
</xsd:element>
<xsd:element name="name" type="xsd:string">
</xsd:element>
<xsd:element name="ingredient" type="xsd:string">
</xsd:element>
<xsd:element name="time" type="xsd:string">
</xsd:element>
<xsd:element name="recipeBook" ref="recipeBookType">
</xsd:element>
<xsd:element name="publisher" ref="publisherType">
</xsd:element>
<xsd:element name="distributors" ref="distributorsType">
</xsd:element>
<xsd:element name="BigBooks" ref="BigBooksType">
</xsd:element>
<xsd:element name="recipe" ref="recipeType">
</xsd:element>
</xsd:element>
</schema>
APPENDIX E
THE SCHEMASENGINE

<xsl:stylesheet version="1.1" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:smb="http://www.smbuntin.org">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="/">
    <xsl:variable name="resultTree" select="document(schemaPrimer.xsd, c:\xml\src)"/>
    <xsl:variable name="XMLlist" select="/\*/"/>
    <xsl:variable name="count" select="1"/>
    <xsl:call-template name="process">
      <xsl:with-param name="resultTree" select="$resultTree"/>
      <xsl:with-param name="XMLlist" select="$XMLlist"/>
      <xsl:with-param name="count" select="$count"/>
    </xsl:call-template>
  </xsl:template>

  <xsl:template name="process">
    <xsl:param name="resultTree"/>
    <xsl:param name="XMLlist"/>
    <xsl:param name="count"/>

    <xsl:variable name="attributeList" select="$resultTree//xsd:complexType/@name|$resultTree//xsd:simpleType/@name"/>
    <xsl:choose>
      <xsl:when test="$XMLlist[$count]">
        <xsl:choose>
          <xsl:when test="name($XMLlist[$count])=$attributeList">
            <!-- merge two nodes if there is one already present-->
          </xsl:when>
        </xsl:choose>
      </xsl:when>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
<xsl:call-template name="mergeNodes">
  <xsl:with-param name="resultTree" select="$resultTree"/>
  <xsl:with-param name="XMLlist" select="$XMLlist"/>
  <xsl:with-param name="count" select="$count"/>
  <xsl:with-param name="node" select="$XMLlist[$count]"/>
</xsl:call-template>
</xsl:when>
<xsl:otherwise>
  <xsl:call-template name="addNode">
    <xsl:with-param name="resultTree" select="$resultTree"/>
    <xsl:with-param name="XMLlist" select="$XMLlist"/>
    <xsl:with-param name="count" select="$count"/>
    <xsl:with-param name="node" select="$XMLlist[$count]"/>
  </xsl:call-template>
</xsl:otherwise>
</xsl:choose>
</xsl:when>
<xsl:otherwise>
  <xsl:call-template name="output">
    <xsl:with-param name="resultTree" select="$resultTree"/>
  </xsl:call-template>
</xsl:otherwise>
</xsl:choose>
</xsl:template>

<xsl:template name="addNode">
  <xsl:param name="node"/>
  <xsl:param name="resultTree"/>
  <xsl:param name="count"/>
  <xsl:param name="XMLlist"/>

  <xsl:variable name="sequence">
    <xsl:element name="xsd:sequence"/>
  </xsl:variable>

  <xsl:variable name="attributeTree">
    <xsl:element name="attributeRoot"/>
  </xsl:variable>

  <xsl:variable name="nodeChildren" select="$node/*"/>
  <xsl:variable name="nodeAttributes" select="$node/@*"/>

  <xsl:choose>

    <xsl:when>
    <xsl:otherwise>
    <xsl:call-template name="addNode">
      <xsl:with-param name="resultTree" select="$resultTree"/>
      <xsl:with-param name="XMLlist" select="$XMLlist"/>
      <xsl:with-param name="count" select="$count"/>
      <xsl:with-param name="node" select="$XMLlist[$count]"/>
    </xsl:call-template>
    </xsl:otherwise>
    </xsl:when>
    </xsl:otherwise>
  </xsl:choose>

  <xsl:element name="xsd:sequence"/>
</xsl:template>

<xsl:variable name="attributeTree">
  <xsl:element name="attributeRoot"/>
</xsl:variable>

<xsl:variable name="nodeChildren" select="$node/*"/>
<xsl:variable name="nodeAttributes" select="$node/@*"/>

<xsl:choose>
<xsl:when test="$nodeChildren and $nodeAttributes">
  <xsl:variable name="newSequenceTree">
    <xsl:call-template name="processChildren">
      <xsl:with-param name="sequence" select="$sequence"/>
      <xsl:with-param name="children" select="$nodeChildren"/>
      <xsl:with-param name="childCount" select="1"/>
    </xsl:call-template>
  </xsl:variable>
  <xsl:variable name="newAttributeTree">
    <xsl:call-template name="processAttributes">
      <xsl:with-param name="attributeTree" select="$attributeTree"/>
      <xsl:with-param name="attributeList" select="$nodeAttributes"/>
      <xsl:with-param name="attribCount" select="1"/>
    </xsl:call-template>
  </xsl:variable>
  <xsl:variable name="topNode">
    <xsl:element name="xsd:complexType">
      <xsl:attribute name="name">
        <xsl:value-of select="name($node)"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:variable>
  <xsl:variable name="newNode">
    <smb:append-copy target="$topNode/xsd:complexType"
      select="$newSequenceTree"/>
  </xsl:variable>
  <xsl:variable name="newAttribNode">
    <smb:append-copy target="$newNode/xsd:complexType"
      select="$newAttributeTree/attributeRoot/attribute"/>
  </xsl:variable>
  <xsl:variable name="newResultTree">
    <smb:append-copy target="$resultTree/schema"
      select="$newAttribNode"/>
  </xsl:variable>
</xsl:when>
<xsl:when test="$nodeChildren">
  <xsl:variable name="newSequenceTree">
    <xsl:call-template name="processChildren">
      <xsl:with-param name="sequence" select="$sequence"/>
      <xsl:with-param name="children" select="$nodeChildren"/>
      <xsl:with-param name="childCount" select="1"/>
    </xsl:call-template>
  </xsl:variable>
</xsl:when>
target="$sequence/xsd:sequence/xsd:element[@name=$childName]"
name="maxOccurs" value="unbounded"/>
</xsl:variable>
<xsl:call-template name="processChildren">
  <xsl:with-param name="sequence" select="$newSequence"/>
  <xsl:with-param name="children" select="$children"/>
  <xsl:with-param name="childCount" select="$childCount+1"/>
</xsl:call-template>
</xsl:when>
<xsl:otherwise>
  <xsl:variable name="newNode">
    <xsl:element name="xsd:element">
      <xsl:attribute name="name">
        <xsl:value-of select="name($children[$childCount])"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:variable>
  <xsl:variable name="newSequence">
    <xsl:apply-templates select="$newNode" mode="attribute"/>
  </xsl:variable>
  <xsl:call-template name="processChildren">
    <xsl:with-param name="sequence" select="$newSequence"/>
    <xsl:with-param name="children" select="$children"/>
    <xsl:with-param name="childCount" select="$childCount+1"/>
  </xsl:call-template>
</xsl:otherwise>
</xsl:choose>
</xsl:when>
<xsl:otherwise>
  <xsl:copy-of select="$sequence"/>
</xsl:otherwise>
</xsl:choose>
</xsl:template>

<xsl:template name="processAttributes">
  <xsl:param name="attributeTree"/>
  <xsl:param name="attributeList"/>
  <xsl:param name="attribCount"/>
  <xsl:variable name="attributeTreeNames" select="$attributeTree/attributeRoot/*/@name"/>
  <xsl:choose>
    <xsl:when test="$attributeList[$attribCount]">
      <xsl:variable name="attributeName" select="$attributeList[$attribCount]/@name"/>
      <xsl:variable name="attributeValue" select="$attributeList[$attribCount]/@value"/>
      <xsl:for-each select="$attributeTreeNames">
        <xsl:choose>
          <xsl:when test=". = $attributeName">
            <xsl:attribute name="value">
              <xsl:value-of select="$attributeValue"/>
            </xsl:attribute>
          </xsl:when>
        </xsl:choose>
      </xsl:for-each>
    </xsl:when>
  </xsl:choose>
</xsl:template>
<xsl:choose>
  <xsl:when test="name($attributeList[$attribCount])=$attributeTreeNames">
    <xsl:call-template name="processAttributes">
      <xsl:with-param name="attributeTree" select="$attributeTree"/>
      <xsl:with-param name="attributeList" select="$attributeList"/>
      <xsl:with-param name="attribCount" select="$attribCount+1"/>
    </xsl:call-template>
  </xsl:when>
  <xsl:otherwise>
    <xsl:variable name="newNode">
      <xsl:element name="attribute">
        <xsl:attribute name="name">
          <xsl:value-of select="name($attributeList[$attribCount])"/>
        </xsl:attribute>
      </xsl:element>
    </xsl:variable>
    <xsl:variable name="newAttributeTree">
      <smb:append-copy target="$attributeTree/attributeRoot" select="$newNode"/>
    </xsl:variable>
    <xsl:call-template name="processAttributes">
      <xsl:with-param name="attributeTree" select="$newAttributeTree"/>
      <xsl:with-param name="attributeList" select="$attributeList"/>
      <xsl:with-param name="attribCount" select="$attribCount+1"/>
    </xsl:call-template>
  </xsl:otherwise>
</xsl:choose>

<xsl:template name="mergeNodes">
  <xsl:param name="resultTree"/>
  <xsl:param name="node"/>
  <xsl:param name="count"/>
  <xsl:param name="XMLlist"/>
  template name="merge"
  <xsl:variable name="originalNode" select="$node"/>
  <xsl:variable name="originalName" select="name($node)"/>
  <xsl:variable name="nodeChildren" select="$node/*"/>
  <xsl:choose>
    <xsl:when>
      <xsl:call-template name="processAttributes">
        <xsl:with-param name="attributeTree" select="$originalNode"/>
        <xsl:with-param name="attributeList" select="$XMLlist"/>
        <xsl:with-param name="attribCount" select="$count"/>
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
      <xsl:copy-of select="$attributeTree"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
<xsl:variable name="nodeAttributes" select="$node/@*"/>

<xsl:choose>
  <xsl:when test="$nodeChildren and $nodeAttributes">
    <xsl:variable name="attribResultTree">
      <xsl:call-template name="mergeAttributes">
        <xsl:with-param name="resultTree" select="$resultTree"/>
        <xsl:with-param name="node" select="$node"/>
      </xsl:call-template>
    </xsl:variable>
    <xsl:variable name="newResultTree">
      <xsl:call-template name="mergeChildren">
        <xsl:with-param name="resultTree" select="$attribResultTree"/>
        <xsl:with-param name="node" select="$originalNode"/>
      </xsl:call-template>
    </xsl:variable>
    <xsl:call-template name="process">
      <xsl:with-param name="resultTree" select="$newResultTree"/>
      <xsl:with-param name="XMLlist" select="$XMLlist"/>
      <xsl:with-param name="count" select="$count+1"/>
    </xsl:call-template>
  </xsl:when>
  <xsl:when test="$nodeChildren">
    <xsl:variable name="newResultTree">
      <xsl:call-template name="mergeChildren">
        <xsl:with-param name="resultTree" select="$resultTree"/>
        <xsl:with-param name="node" select="$node"/>
      </xsl:call-template>
    </xsl:variable>
    <xsl:call-template name="process">
      <xsl:with-param name="resultTree" select="$newResultTree"/>
      <xsl:with-param name="XMLlist" select="$XMLlist"/>
      <xsl:with-param name="count" select="$count+1"/>
    </xsl:call-template>
  </xsl:when>
  <xsl:when test="$nodeAttributes">
    <xsl:variable name="attribResultTree">
      <xsl:call-template name="mergeAttributes">
        <xsl:with-param name="resultTree" select="$resultTree"/>
        <xsl:with-param name="node" select="$node"/>
      </xsl:call-template>
    </xsl:variable>
    <xsl:variable name="newResultTree">
      <xsl:call-template name="mergeChildren">
        <xsl:with-param name="resultTree" select="$attribResultTree"/>
        <xsl:with-param name="node" select="$originalNode"/>
      </xsl:call-template>
    </xsl:variable>
    <xsl:call-template name="process">
      <xsl:with-param name="resultTree" select="$newResultTree"/>
      <xsl:with-param name="XMLlist" select="$XMLlist"/>
      <xsl:with-param name="count" select="$count+1"/>
    </xsl:call-template>
  </xsl:when>
</xsl:choose>
<xsl:with-param name="node" select="$originalNode"/>
<xsl:with-param name="oldChildList" select="$originalNode"/>
</xsl:call-template>
</xsl:variable>
<xsl:call-template name="process">
<xsl:with-param name="resultTree" select="$newResultTree"/>
<xsl:with-param name="XMLlist" select="$XMLlist"/>
<xsl:with-param name="count" select="$count+1"/>
</xsl:call-template>
</xsl:when>
<xsl:otherwise>
<xsl:call-template name="process">
<xsl:with-param name="resultTree" select="$resultTree"/>
<xsl:with-param name="XMLlist" select="$XMLlist"/>
<xsl:with-param name="count" select="$count+1"/>
</xsl:call-template>
</xsl:otherwise>
</xsl:choose>
</xsl:template>

<xsl:template name="mergeChildren">
<xsl:param name="resultTree"/>
<xsl:param name="node"/>
<xsl:variable name="nodeName" select="name($node)"/>
<xsl:choose>
<xsl:when test="$resultTree/schema/xsd:complexType[@name=$nodeName]">
<xsl:variable name="newResultTree">
<xsl:call-template name="addComplexTypeChildren">
<xsl:with-param name="resultTree" select="$resultTree"/>
<xsl:with-param name="node" select="$node"/>
</xsl:call-template>
</xsl:variable>
<xsl:copy-of select="$newResultTree"/>
</xsl:when>
<xsl:when test="$resultTree/schema/xsd:simpleType[@name=$nodeName]">
<xsl:variable name="newResultTree">
<xsl:call-template name="addSimpleTypeChildren">
<xsl:with-param name="resultTree" select="$resultTree"/>
<xsl:with-param name="node" select="$node"/>
<xsl:with-param name="simpleCount" select="1"/>
</xsl:call-template>
</xsl:variable>
</xsl:when>
</xsl:choose>
</xsl:template>
<xsl:copy-of select="$newResultTree"/>
<xsl:when>
<xsl:otherwise>
<xsl:copy-of select="$resultTree"/>
</xsl:otherwise>
</xsl:choose>
</xsl:template>

<xsl:template name="mergeAttributes">
<xsl:param name="resultTree"/>
<xsl:param name="node"/>
<xsl:variable name="nodeName" select="name($node)"/>

<xsl:choose>
<xsl:when test="$resultTree/schema/xsd:complexType[@name=$nodeName]">
<xsl:variable name="newResultTree">
<xsl:call-template name="addComplexTypeAttributes">
<xsl:with-param name="resultTree" select="$resultTree"/>
<xsl:with-param name="node" select="$node"/>
</xsl:call-template>
</xsl:variable>
<xsl:copy-of select="$newResultTree"/>
</xsl:when>
<xsl:when test="$resultTree/schema/xsd:simpleType[@name=$nodeName]">
<xsl:variable name="newResultTree">
<xsl:call-template name="addSimpleTypeAttributes">
<xsl:with-param name="resultTree" select="$resultTree"/>
<xsl:with-param name="node" select="$node"/>
</xsl:call-template>
</xsl:variable>
<xsl:copy-of select="$newResultTree"/>
</xsl:when>
<xsl:otherwise>
should not happen
<xsl:copy-of select="$resultTree"/>
</xsl:otherwise>
</xsl:choose>
</xsl:template>

<xsl:template name="addSimpleTypeAttributes">
<xsl:param name="resultTree"/>
<xsl:param name="node"/>

<xsl:choose>
<xsl:when test="$resultTree/schema/xsd:complexType[@name=$nodeName]">
<xsl:variable name="newResultTree">
<xsl:call-template name="addComplexTypeAttributes">
<xsl:with-param name="resultTree" select="$resultTree"/>
<xsl:with-param name="node" select="$node"/>
</xsl:call-template>
</xsl:variable>
<xsl:copy-of select="$newResultTree"/>
</xsl:when>
</xsl:choose>
<xsl:call-template name="processAttributes">
  <xsl:with-param name="attributeTree" select="$attributeTree"/>
  <xsl:with-param name="attributeList" select="$nodeAttributes"/>
  <xsl:with-param name="attribCount" select="1"/>
</xsl:call-template>
</xsl:variable>

<xsl:variable name="newComplexType">
  <smb:append-copy target="$newComplexNode/xsd:complexType"
    select="$newAttributeTree/attributeRoot/*"/>
</xsl:variable>

<xsl:variable name="newResultTree">
  <smb:replace-copy
target="$resultTree/schema/xsd:complexType[@name=$nodeName]"
    select="$newComplexType"/>
</xsl:variable>

<xsl:copy-of select="$newResultTree"/>
</xsl:template>

<xsl:template name="addSimpleTypeChildren">
  <xsl:param name="resultTree"/>
  <xsl:param name="node"/>

  <xsl:variable name="nodeName" select="name($node)"/>
  <xsl:variable name="nodeChildren" select="$node/*"/>

  <xsl:variable name="newComplexNode">
    <xsl:element name="xsd:complexType">
      <xsl:attribute name="name" select="name">
        <xsl:value-of select="name($node)"/>
      </xsl:attribute>
      <xsl:choose>
        <xsl:when test="$node/child::text()">
          <xsl:attribute name="mixed">true</xsl:attribute>
        </xsl:when>
        <xsl:otherwise>
          <!—do nothing -->
        </xsl:otherwise>
      </xsl:choose>
    </xsl:element>
  </xsl:variable>

  <xsl:variable name="sequence">
    <xsl:element name="xsd:complexType">
      <xsl:attribute name="name" select="name">
        <xsl:value-of select="name($node)"/>
      </xsl:attribute>
      <xsl:choose>
        <xsl:when test="$node/child::text()">
          <xsl:attribute name="mixed">true</xsl:attribute>
        </xsl:when>
        <xsl:otherwise>
          <!—do nothing -->
        </xsl:otherwise>
      </xsl:choose>
    </xsl:element>
  </xsl:variable>

  <xsl:variable name="sequence">
    <xsl:element name="xsd:complexType">
      <xsl:attribute name="name" select="name">
        <xsl:value-of select="name($node)"/>
      </xsl:attribute>
      <xsl:choose>
        <xsl:when test="$node/child::text()">
          <xsl:attribute name="mixed">true</xsl:attribute>
        </xsl:when>
        <xsl:otherwise>
          <!—do nothing -->
        </xsl:otherwise>
      </xsl:choose>
    </xsl:element>
  </xsl:variable>
</xsl:template>
<xsl:element name="xsd:sequence"/>
</xsl:variable>

<xsl:variable name="sequenceTree">
  <xsl:call-template name="processChildren">
    <xsl:with-param name="sequence" select="$sequence"/>
    <xsl:with-param name="children" select="$nodeChildren"/>
    <xsl:with-param name="childCount" select="1"/>
  </xsl:call-template>
</xsl:variable>

<xsl:variable name="newSequenceTree">
  <smb:add-attributes target="$sequenceTree/xsd:sequence/xsd:element" name="minoccurs" value="0"/>
</xsl:variable>

<xsl:variable name="newComplexType">
  <smb:append-copy target="$newComplexNode/xsd:complexType" select="$newSequenceTree"/>
</xsl:variable>

<xsl:variable name="newResultTree">
  <smb:replace-copy target="$resultTree/schema/xsd:simpleType[@name=$nodeName]" select="$newComplexType"/>
</xsl:variable>

<xsl:copy-of select="$newResultTree"/>
</xsl:template>

<xsl:template name="addComplexTypeChildren">
  <xsl:param name="resultTree"/>
  <xsl:param name="node"/>

  <xsl:variable name="nodeName" select="name($node)"/>
  <xsl:variable name="nodeChildren" select="$node/"/>
  <xsl:variable name="nodeChildNames" select="smb:name-set($nodeChildren)"/>
  <xsl:variable name="resultNode" select="$resultTree/schema/xsd:complexType[@name = $nodeName]"/>
  <xsl:variable name="resultNodeList" select="$resultNode/"/>
  <xsl:variable name="resultNodeAttribs" select="$resultNode/attribute"/>
</xsl:template>
<xsl:variable name="sequence">
  <xsl:element name="xsd:sequence">
  </xsl:element>
</xsl:variable>

<xsl:variable name="sequenceTree">
  <xsl:call-template name="processChildren">
    <xsl:with-param name="sequence" select="$sequence"/>
    <xsl:with-param name="children" select="$nodeChildren"/>  
    <xsl:with-param name="childCount" select="1"/>
  </xsl:call-template>
</xsl:variable>

<xsl:variable name="newSequenceTree">
  <xsl:element name="xsd:sequence">
    <xsl:for-each select="$sequenceTree/xsd:sequence/*">
      <xsl:choose>
        <xsl:when test="./@name = $resultNodeList">
          <xsl:copy/>
        </xsl:when>
        <xsl:otherwise>
          <xsl:variable name="newName" select="./@name"/>
          <xsl:element name="xsd:element">
            <xsl:attribute name="name">
              <xsl:value-of select="./@name"/>
            </xsl:attribute>
            <xsl:attribute name="minoccurs">0</xsl:attribute>
            <xsl:choose>
              <xsl:when test="./@maxoccurs">
                <xsl:attribute name="maxoccurs">
                  <xsl:value-of select="./@maxoccurs"/>
                </xsl:attribute>
              </xsl:when>
              <xsl:otherwise>
                </xsl:otherwise>
            </xsl:choose>
          </xsl:element>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:for-each>
  </xsl:element>
</xsl:variable>
<xsl:variable name="newComplexNode">
  <xsl:element name="xsd:complexType">
    <xsl:attribute name="name">
      <xsl:value-of select="name($node)"/>
    </xsl:attribute>
    <xsl:choose>
      <xsl:when test="$node/child::text()">
        <xsl:attribute name="mixed">true</xsl:attribute>
      </xsl:when>
      <xsl:otherwise>
        <!--do nothing-->
      </xsl:otherwise>
    </xsl:choose>
  </xsl:element>
</xsl:variable>

<xsl:variable name="newComplexType">
  <smb:append-copy target="$newComplexNode/xsd:complexType"
    select="$newSequenceTree"/>
</xsl:variable>

<xsl:variable name="finalComplexType">
  <smb:append-copy target="$newComplexType/xsd:complexType"
    select="$resultNodeAttribs"/>
</xsl:variable>

<xsl:variable name="newResultTree">
  <smb:replace-copy
    target="$resultTree/schema/xsd:complexType[@name=$nodeName]"
    select="$finalComplexType"/>
</xsl:variable>

<xsl:copy-of select="$newResultTree"/>

</xsl:template>

<xsl:template name="output">
  <xsl:param name="resultTree"/>
  <xsl:copy-of select="$resultTree"/>
</xsl:template>

</xsl:stylesheet>
BIBLIOGRAPHY


BIOGRAPHICAL SKETCH

Scott Buntin was born in Pensacola, FL, but grew up on St. Simon’s Island, GA. He received a Bachelor of Science degree in biology from the University of the South in Sewanee, TN. He received a Master of Science degree in biology from Georgia Southern University in Statesboro, GA, where he concentrated on genetic mapping of chromosomes in the *Neurospora crassa* fungus. He was an Adjunct Professor of Biology and worked in a microbiology lab at the University of Florida, where he met his wife, Fuhua Lu. He will receive a Master of Science degree in computer science from the University of Florida in Gainesville, FL. His concentration was on programming languages, with extensive work in the use of the XSLT programming language to infer XML Schema documents from arbitrary XML documents.