2007

Alternate syntax for XSLT

David Love

Alternate Syntax for XSLT

David Love
Department of Computer Science
Rochester Institute of Technology
Rochester, NY, USA
dpl1926@cs.rit.edu

November 5, 2007

Abstract

XSLT is a transform language for XML that is defined over XML. In other words, XSLT is a language that performs transforms on XML documents, and XSLT programs are themselves XML documents. While XSLT is by nature a functional language, its definition as an XML application obfuscates this fact [15]. Previous research projects have taken the XML-Infoset and provided an alternate syntax in the form of S-expressions, along with providing languages to perform transformations of the new representation in manners similar to that of XSLT. For example, SXML / SXSLT performs this function by embedding said languages in Scheme [9].

XLove applies modern principles of object-oriented design, namely design patterns, to this problem. Xl is an alternate syntax for the XML-Infoset. It maintains a clear distinction between attributes and elements (while having a concise notation for namespaces). The syntax is built into a representation over the Document Object Model by observers responding to parsing events. Xlt is an alternate syntax for XSLT designed to emphasize the functional nature of the language. A set of visitors transforms the input Document Object Model tree into an output tree by mapping the Xlt abstract syntax tree to XSLT. The resultant document is a valid XSLT program over the Document Object Model which can than be directly executed or output as an XML file.

1 Introduction

It has been postulated that in essence, XSLT is a functional language [15]. The primary purpose of the combination of Xl and Xlt (from here on referred to simply as Xl/t) is to provide an alternate syntax for XSLT that better captures the semantic meaning of programming constructs and idioms than the current XML syntax. By providing such a syntax it is hoped that programmers familiar with various functional languages (Common Lisp, Scheme, Haskell, Arc, etc.) will be able to quickly and easily grasp the core concepts of XSLT. Additionally, if the time it takes to write a program is proportional to its length [3], then the removal of the extraneous text from the current XML syntax should allow programmers to develop prototypes far more quickly thereby encouraging a more bottom-up programming style (as befits a functional language). There are multiple projects seeking to provide a better interface for XSLT including (but not limited to) XSLScript [14] and NiceXSL [24]. Other projects have also taken a different direction, seeking to provide a full-blown XML processing language. SXML is perhaps the most ambitious of these, however, XDuce [8], and the work of Wallace and Runciman in Haskell [23] all deserve mention.

No discussion of XSL can be complete without talking about the Document Style Semantics And Specification Language (DSSSL) [4]. DSSSL was originally designed to provide a way of transforming and formatting SGML documents, but was later extended to handle XML and HTML. Widely considered the father of XSL, DSSSL is actually much more than that. The original design of XSL was accompanied by proposed changes to the DSSSL standard so that DSSSL would be a superset of XSL [1]. Since DSSSL stylesheets are in essence Scheme programs, the functional nature of such stylesheets is quite clear. However, DSSSL was developed before the standardization of many internet

technologies. Most notably, the XML-Infoset had yet to become a standardized representation of XML data. XSL was intended to focus on the new internet standards and address usability issues found in the then current DSSSL standard. The best concrete example of the relationship between XSL and DSSSL is an early prototype XSL translator named xslj [16]. Taking XSL stylesheets as input, xslj then translates such stylesheets into DSSSL stylesheets allowing for execution using Jade (James Clark's DSSSL engine).

Xl/t and xslj are in many ways similar in spirit. While xslj took the then non-standardized XSL proposal and provided an implementation using DSSSL, Xl/t seeks to define a new syntax for XSLT and provide an implementation using existing XSLT processors (most notably TrAX). This style of approach allows people to use existing and proven technologies for execution with the only new and unproven technologies are used for translation. While this does reduce efficiency, it allows a full engine to be developed in pieces. In fact, it is not a huge leap to imagine linking together multiple translation engines to choose the most stable platform for the execution of the resultant stylesheets.

1.1 Overview

XML has become a popular choice for the storage of data [12]. Its predecessors, namely HTML and SGML, focused more on the presentation of data, in either a visual or print form respectively. In order to take the data stored in an XML file and present it for other media, it is necessary to either annotate the data with certain display properties or apply certain transforms to said data. One of the methods for achieving this is the use of stylesheets.

Cascading Style Sheets (CSS) [17] are one way of annotating XML in order to format it for display. Because there is no transformation of elements occurring, oftentimes many workarounds need to be used in order to obtain the desired result. After all, their is no guarantee the underlying XML data model was designed for display. For example, XHTML is designed with the intent of being independent of display. This is why in many modern web pages a large number of div elements are introduced as a way of obtaining finer controls over the presentation to the end user. This clutters the XHTML document with elements that have no semantic meaning - they are simply used to control display.

However, an alternative exists in the form of the Extensible Stylesheet Language Family (XSL) [20], a key component of which is XSLT. By using XSLT to transform input objects into XSL-FO (formatting objects), a developer gains far more flexibility with regards to the end result. Since XSL-FO objects are designed explicitly for the purposes of display, setting properties is all that is necessary to define the presentation - no clever CSS hacks are necessary. XSLT has a syntax reminiscent of a functional language, however, it does not neatly fit within such a description. Additionally, it is specified as an XML application which can quickly become difficult for a human programmer to deal with. In order to make XSLT more accessible, the syntax should be specified independent of XML (in order to make the programs easy to parse for humans, not just computers), and focus should be placed on the functional aspects of the language. By doing so, XSLT becomes more than simply a stylesheet implementation with an irritating syntax. This new syntax would provide a fully fledged functional XML transformation language for programmers to work with.

1.2 Summary

Xl/t is the combination of two disparate pieces. The first is Xl, which defines an alternate syntax for the XML-Infoset. The implementation of this is a set of observers that parse an input document into a DOM representation. The second is Xlt. Xlt is a language that takes a DOM representation as restricted by the Xlt schema and transforms it into a DOM representation of XSLT. This is implemented as a series of visitors that perform the transformation. XLove is the glue that ties these pieces together. It takes a set of observers to parse an input document and a set of visitors to optionally transform that document. By passing Xl observers and Xlt visitors to an XLove object, we obtain a full implementation of the combination of Xl and Xlt, namely Xl/t. This maintains a separation between the syntax and any transformations that are done on the DOM representation. Internally, the Xlt visitors use an XLove object to parse the Xlt schema but perform no transforms on the resultant DOM representation. This is because the schema once parsed into a DOM representation from the Xl syntax is a valid XMLSchema.

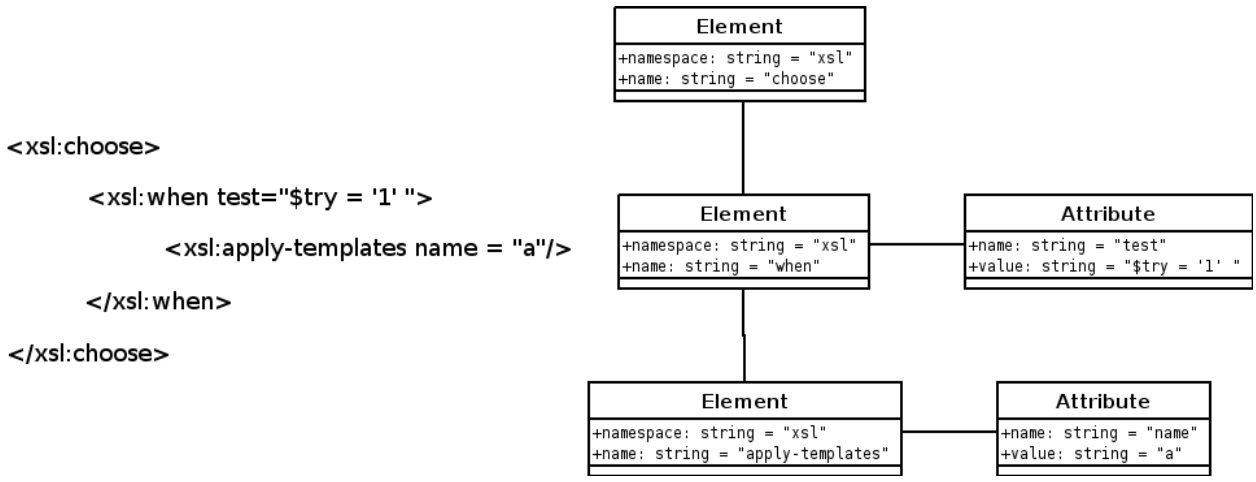

Figure 1: XML and Corresponding DOM

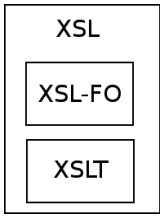

Figure 2: XSL Relationships

At the most coarse level, the goal of Xl/t is to generate an XSLT tree from an input program. However, a programmer using XLove as an interface to Xl/t is able to use existing XSLT/XML technologies, namely Java transforms (TrAX), XML parsing (SAX2), Java libraries for XML processing (JAXP), and XSLT-C. XLove is not a simple translator, rather it is a framework for developing languages whose abstract syntax tree is built over the Document Object Model (DOM) [18]. In particular, Xl/t builds such a tree and then transforms it into an XSLT tree over the DOM. This tree need not be output, but instead, can be accessed programmatically to pass the constructed tree to other modules, classes, or methods in memory.

A distinction must be made between XSLT and XSL-FO. XSLT is a transform language for XML defined as an XML application (i.e. uses XML as the language of expression). XSL-FO deals with presentation layout - in particular, formatting objects. These two standards along with the XPath standard form XSL, the Extensible Stylesheet Language. Xl/t only seeks to replace XSLT and does not encroach upon the domain of formatting objects. XPath is used with no alternate syntax provided (seeing as how XPath is a full language in its own right).

2 Xl/t Tutorial

The following sections illustrate some features and constructs in Xl/t through sample programs. There are two basic types of programs given. For the stylesheet elements of the language, some sample transforms on XML input data are shown. Also provided are more classic examples of sample programs (utilizing the empty XML file).

```
# Strip all of the text out of an xml document
[ :xlt:("http://www.cs.rit.edu/~dpl1926/xlt") ]
:xlt:main {

  :xlt:defun { :xlt:match { "*" }
    :xlt:lambda {
      :xlt:apply {
        :xlt:value { "node() | attribute::node()" } } } } }
```

Figure 3: Display Text in an XML Document

```
<?xml version="1.0"?>
<empty></empty>
```

Figure 4: Empty XML Document

2.1 Display Text in an XML Document

In order to display just the text from an XML document, a sample stylesheet is defined with a single function. The function is not bound to a name, but instead execution is controlled by a match guard. This match guard chooses any element (as indicated by the wildcard pattern) and takes the value of the element and any attributes.

2.2 Recursive Algorithm for Generating Factorials

A more classical functional programming example is a recursive algorithm for generating factorials. A ”main” method is defined by having a function whose condition for execution is a match on /. This executes the function once for the top-level root element of the document. Said function then calls the factorial function with the value of the variant x. As can be seen from the sample code, this factorial function uses an accumulator. The factorial function demonstrates the standard definition for a named function. Using let, a function is bound to a name, and again using let, names are bound to parameters. The fac function (which does the actual work of computing the factorial) is called with arguments. Scoping rules are apparent here - the value being taken for use as fac’s x parameter is factorial’s x parameter.

The fac function then utilizes the cond element in order to determine when to stop recursion. The base case is determined by the guard on x. We also see a common idiom in the second guard. In order to have if/else semantics, a guard containing the XPath function true is utilized. This should be similar to most Common Lisp users. Since XPath has support for much of the standard arithmetic of a language, Xl/t contains no native arithmetic support [7]. The existing XPath functionality is leverage to provide a complete programming environment.

2.3 Euclid’s Algorithm for Greatest Common Divisor

Another common recursive algorithm is Euclid’s algorithm for greatest common divisor. The general structure of the program is the same as the factorial program. The gcd function takes two parameters, x and y, and then determines the greatest common divisor of those parameters. In the cond, three guards exist. If x and y are equal, then we have a common divisor and we take its value. If x is less then y, we call gcd again with x bound to x and the value of y minus x bound to y. In all other cases, we call gcd with x bound to x minus y and y bound to y.

```
# Calculate factorials using a tail-recursive algorithm
# 5!
[ :xlt:("http://www.cs.rit.edu/~dpl1926/xlt") ]
:xlt:main {

  :xlt:output { "text" }

  :xlt:let [ name("x") ] {
    :xlt:variant { :xlt:value { "5" } } }

  :xlt:defun { :xlt:match { "/" }
    :xlt:lambda {
      :xlt:funcall [ name("factorial") ] {
        :xlt:let [ name("x") ] {
          :xlt:variant { :xlt:value { "$x" } } } } } }

  :xlt:let [ name("factorial") ] { :xlt:defun {
    :xlt:let [ name("x") ] {
      :xlt:variant }
    :xlt:lambda {
      :xlt:funcall [ name("fac") ] {
        :xlt:let [ name("x") ] {
          :xlt:variant { :xlt:value { "$x" } } }
        :xlt:let [ name("acc") ] {
          :xlt:variant { :xlt:value { "1" } } } } } } }

  :xlt:let [ name("fac") ] { :xlt:defun {
    :xlt:let [ name("x") ] {
      :xlt:variant }
    :xlt:let [ name("acc") ] {
      :xlt:variant }
    :xlt:lambda {
      :xlt:cond {
        :xlt:guard { :xlt:value { "$x = 1" }
          :xlt:lambda {
            :xlt:value { "$acc" } } }
        :xlt:guard { :xlt:value { "true()" }
          :xlt:lambda {
            :xlt:funcall [ name("fac") ] {
              :xlt:let [ name("x") ] {
                :xlt:variant { :xlt:value { "$x - 1" } } }
              :xlt:let [ name("acc") ] {
                :xlt:variant { :xlt:value { "$x * $acc" } } } } } } } } } }
```

Figure 5: Recursive Algorithm for Generating Factorials

```
# Greatest Common Divisor (using Euclid's algorithm)
# 36, 54
[ :xlt:("http://www.cs.rit.edu/~dpl1926/xlt") ]
:xlt:main {

  :xlt:output { "text" }

  :xlt:let [ name("x") ] {
    :xlt:variant { :xlt:value { "36" } } }

  :xlt:let [ name("y") ] {
    :xlt:variant { :xlt:value { "54" } } }

  :xlt:defun { :xlt:match { "/" }
    :xlt:lambda {
      :xlt:funcall [ name("gcd") ] {
        :xlt:let [ name("x") ] {
          :xlt:variant { :xlt:value { "$x" } } }
        :xlt:let [ name("y") ] {
          :xlt:variant { :xlt:value { "$y" } } } } } }

  :xlt:let [ name("gcd") ] { :xlt:defun {
    :xlt:let [ name("x") ] {
      :xlt:variant }
    :xlt:let [ name("y") ] {
      :xlt:variant }
    :xlt:lambda {
      :xlt:cond {
        :xlt:guard { :xlt:value { "$x = $y" }
          :xlt:lambda {
            :xlt:value { "$x" } } }
        :xlt:guard { :xlt:value { "$x < $y" }
          :xlt:lambda {
            :xlt:funcall [ name("gcd") ] {
              :xlt:let [ name("x") ] {
                :xlt:variant { :xlt:value { "$x" } } }
              :xlt:let [ name("y") ] {
                :xlt:variant { :xlt:value { "$y - $x" } } } } } }
        :xlt:guard { :xlt:value { "true()" }
          :xlt:lambda {
            :xlt:funcall [ name("gcd") ] {
              :xlt:let [ name("x") ] {
                :xlt:variant { :xlt:value { "$x - $y" } } }
              :xlt:let [ name("y") ] {
                :xlt:variant { :xlt:value { "$y" } } } } } } } } } }
```

Figure 6: Euclid's Algorithm for Greatest Common Divisor

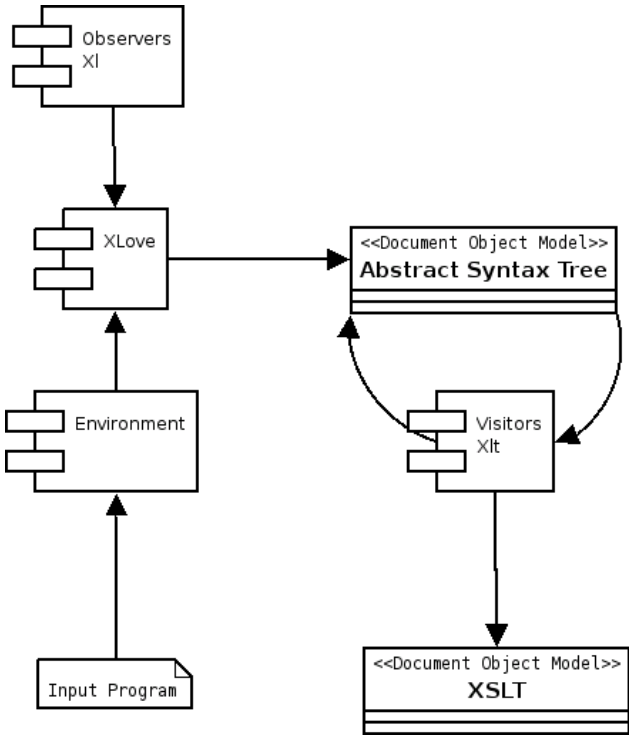

Figure 7: System Diagram

The documentation included with the XLove distribution contains explanations of each of the elements defined within Xlt. These descriptions are listed within the Xlt.Transform class (the inner class which defines the visitors).

3 Architectural Overview

The overall architecture of the system is fairly straightforward. An input program is processed through an environment. The provided environment, Console, accepts a program through either an InputStream or loads a program from a given filename. This environment is then accessed by the parser (through the XLove class) which uses a set of observers [6] to build the abstract syntax tree. This tree is validated using a schema that defines the accepted structure of an Xl/t document. A set of visitors [6] then walks the Xl/t tree transforming it into an XSLT (version 1.1) tree over the DOM.

3.1 Toolkits

Oops [13] is used as the primary toolkit for implementing the parser. It provides an easy to use preprocessor that takes a grammar specified in one of many different formats and generates a set of observers that recognize the grammar. It also provides for the concept of environments in which a program is input. XLove utilizes the extensibility of these environments to define a DocumentEnvironment (an environment which also contains a DOM representation of the accepted program). This is discussed in further detail in the section below regarding environments.

Dom4J [5] is utilized for two interrelated purposes. Primarily, it provides a friendlier set of utilities for building DOM trees as opposed to the standard JAXP API's. It does this while maintaining compatibility with those same API's. It is easy to retrieve a copy of a dom4j Document in terms of the standard JAXP Source interface. Secondarily, the toolkit includes a set of libraries for defining visitors over a DOM tree. By extending the provided framework, it becomes

```
java -jar xlove.jar -x examples/empty.xml examples/eratosthenes.xlt
```

Figure 8: Executing a Sample Program

simple to create a set of visitors that uses an Oops environment for handling input and output.

The third external toolkit used is the Sun Multi-Schema XML Validator [11]. The code provided by Sun Microsystems (in addition to the other projects distributed as part of the validator) allows easy validation of XML documents or DOM trees using a variety of schema formats. By validating the Xl/t tree against such a schema, it is possible to separate the definition of the syntax of Xl (the structure of a document) and the syntax of Xlt (the meaning of the document). This allows for additional languages to be specified that are defined in terms of Xl but for differing purposes than Xlt.

3.2 Component Modularization

The XLove package is divided into multiple sub-packages. The toplevel package, xlove, provides access to the translator. It allows a developer to retrieve an input program through the Console environment and utilizing a given set of observers and visitors, recognize the program and perform transforms upon it.

The package xlove.environment defines the Console environment, along with providing interfaces that are used for obtaining an input program (as well as directing output). This package insulates a developer from the internals of the Oops environment structure while making it easy to develop new environments for program execution (e.g. an applet instead of a console).

The Xl parser is defined in xlove.observers. This parser is defined in terms of the Oops Observers class which provides a collection of observers for recognition. While Xl uses an RFC style grammar, it is possible to use a variety of other grammars for defining such observers.

The VisitorFactory defined in xlove.visitors seeks to provide a similar interface for creating a collection of visitors as Oops provides for creating a collection of observers. The Xlt visitors are also defined here.

4 Design Documentation

As is visible from the System Diagram, the design of the XLove system should be extremely familiar to compiler writers [2]. The standard paradigm is used with some slight modifications. The structure of a program is accepted or rejected based on very straightforward criteria by the parser. A program needs only to satisfy the criteria of a valid Xl document in order for an abstract syntax tree (AST) to be built. Further syntactic analysis is delayed until the Xlt visitors run. One of the first things done by said visitors is to validate the AST against the Xlt schema. This checks for semantic correctness (including type checking). The logical structure defined by the schema must be obeyed in order for further processing to occur. Keyword validation (i.e. is said keyword a valid Xlt keyword?) is done utilizing reflection by the Xlt visitors (this will be discussed further in the visitor section).

4.1 XLove Programmatic Interface

For users simply interested in executing the interpreter, a basic main method is provided in the XLove class for explicitly this purpose. In order to execute any of the example programs provided in the sample implementation, one only needs to run the jar with a designated input document. The test target in the Makefile does exactly this.

```
String observers = "xlove.observers.Xl";
LinkedList visitors = new LinkedList();
visitors.add("xlove.visitors.Xlt");

XLove xlove = new XLove("program.xlt", observers, visitors);

TransformerFactory factory = TransformerFactory.newInstance();
Transformer transformer = factory.newTransformer(xlove.toSource());

transformer.transform(new StreamSource(new File("input.xml")),
new StreamResult(System.out));

System.out.print("\n");
```

Figure 9: Transforming an Input XML Document

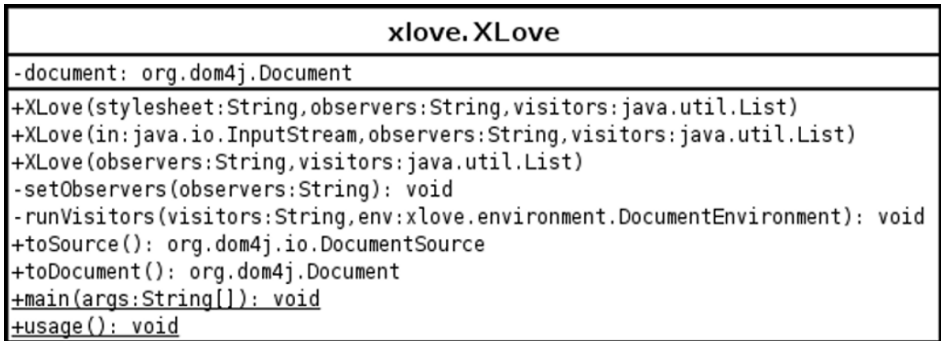

Figure 10: Package xlove

Embedding the interpreter in another program is a straightforward affair. The XLove class provides multiple constructors explicitly for this purpose. Moreover, the toSource method provides a mechanism for obtaining the resultant document as an implementation of the JAXP Source interface. Once again, the main method in the XLove class does just this, including optionally passing the resultant XSLT document directly to a transformer.

4.2 XLove Environments

A DocumentEnvironment is an extension of the standard Oops environment interface (edu.rit.cs.oops.Environment) that adds an org.dom4j.Document as part of the environment. This DOM tree represents both the AST and the resultant document post-transformation. The XLoveEnvironment is a concrete implementation of this interface. What it provides is an entire environment of execution for a program. This includes (but is not limited to) streams for input, output, and errors. Additionally, it provides access methods for obtaining resources for the program. The most notable of the resources it is used to obtain is the parser to be used, which is obtained through a standard Java property. As a side note, the Xlt visitors also utilize this functionality to grab values for variants defined as Java properties. The Console environment is a convenient wrapper that defines input as either a file or standard input, output as standard output, and error as standard error.

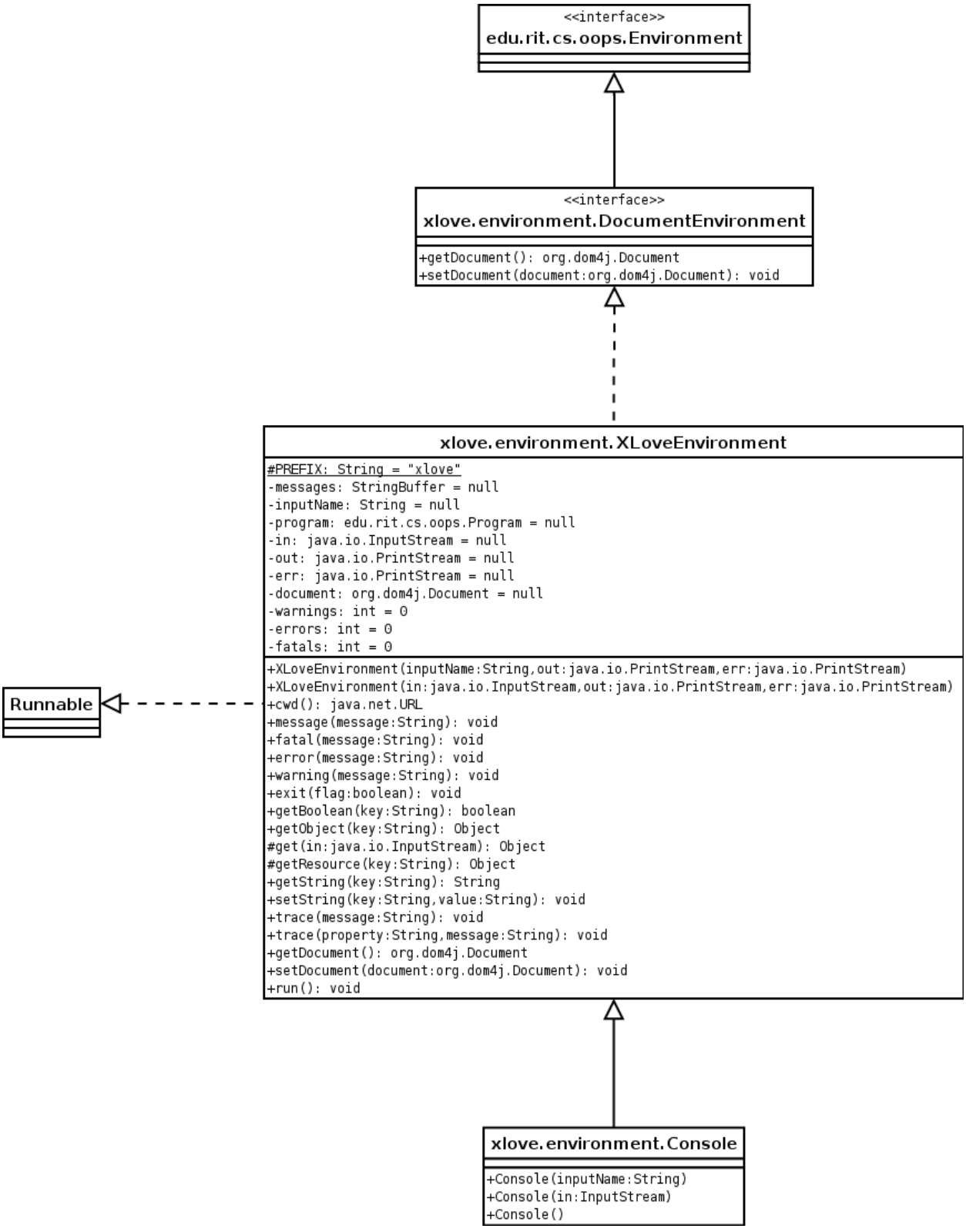

Figure 11: Package xlove.environment

```
root      : element;
element   : ( "[" ns* "]" )? name ( "[" attribute* "]" )? ( "{" child* "}" )?;
ns        : ":" `Id` ":" "(" 'String'? ")";
attribute : name "(" value ")";
value     : 'String';
child     : element | text;
text      : 'String';
name      : ( ":" `Id` ":")? 'Id';
```

Figure 12: RFC Grammar for Xl

4.3 Front-end Language

The Xl language is designed as a replacement for the XML-Infoset [19] (at least as far as the purposes of Xl/t are concerned). It has support for elements, attributes, text nodes, and namespaces. The namespace support is crucial for the analysis done by the visitors. The RFC grammar shown defines the syntax for Xl, namely how elements, attributes, text nodes, and namespaces are represented. Most notable about this grammar when compared to that of the full XML-Infoset is its brevity. No clever tricks are necessary in order to parse a document represented in Xl.

The greatest strength of the strategy of decoupling the syntax for specifying a document from the syntax of a document's logical structure deals with the nature of the presentation of information. An Xl document is a shorthand for describing the AST that will be built: nothing more, nothing less. This is visible in the schema definition for Xlt which utilizes Xl for its representation but is actually a standard W3C schema definition. Xlt actually instantiates an XLove object for loading and parsing the schema, and then passes the DOM representation to the schema validator.

4.4 Visitors Over the Syntax Tree

The visitors which run over the tree can be specified in the constructor for the XLove class. It is worth noting that no visitors need be run if simply using the Xl syntax as an alternate syntax for XML. The main method of the XLove class by default runs the visitors xlove.visitors.Xlt.

The Xlt class is a visitor factory that does many things besides simply providing the visitors for transformation. It provides a framework for doing schema validation as well. The actual visitors themselves are contained within an inner class. This class is responsible for checking that the namespace of the root element is xlt as well as validating against the schema. At this point, the input Xl/t document has already been parsed and is in the form of an AST over the DOM. This means that any XML schema can be used by the visitors for checking structural compliance. The Xlt schema is specified in Xl, but uses a standard XML schema as the target [21].

Which method is invoked for visitation is chosen based on reflection. This, along with the schema definition, is how keywords are defined for the language. If a method that matches the name of the DOM node that is being visited is defined within the visitor, then said method is used to visit that node. Each method takes two arguments - the element being visited and the visitor which is visiting the element. Each keyword is then responsible for determining how it is translated into the underlying XSLT. While elements such as cond are easy to translate (just change the element into an XSLT choose element), other more complicated constructs can be built. For example, the lambda construct ends up removing itself entirely from the output tree. Another more complicated example is the variant element. Variant can be a with-param or a param element depending on the context within which it is used. Also, if a variant is a top-level element, the visitor checks whether a property is set, and if so, uses the value of the property as the value of the variant. This allows a user to change the values of top level variant on the command line simply by setting a Java property.

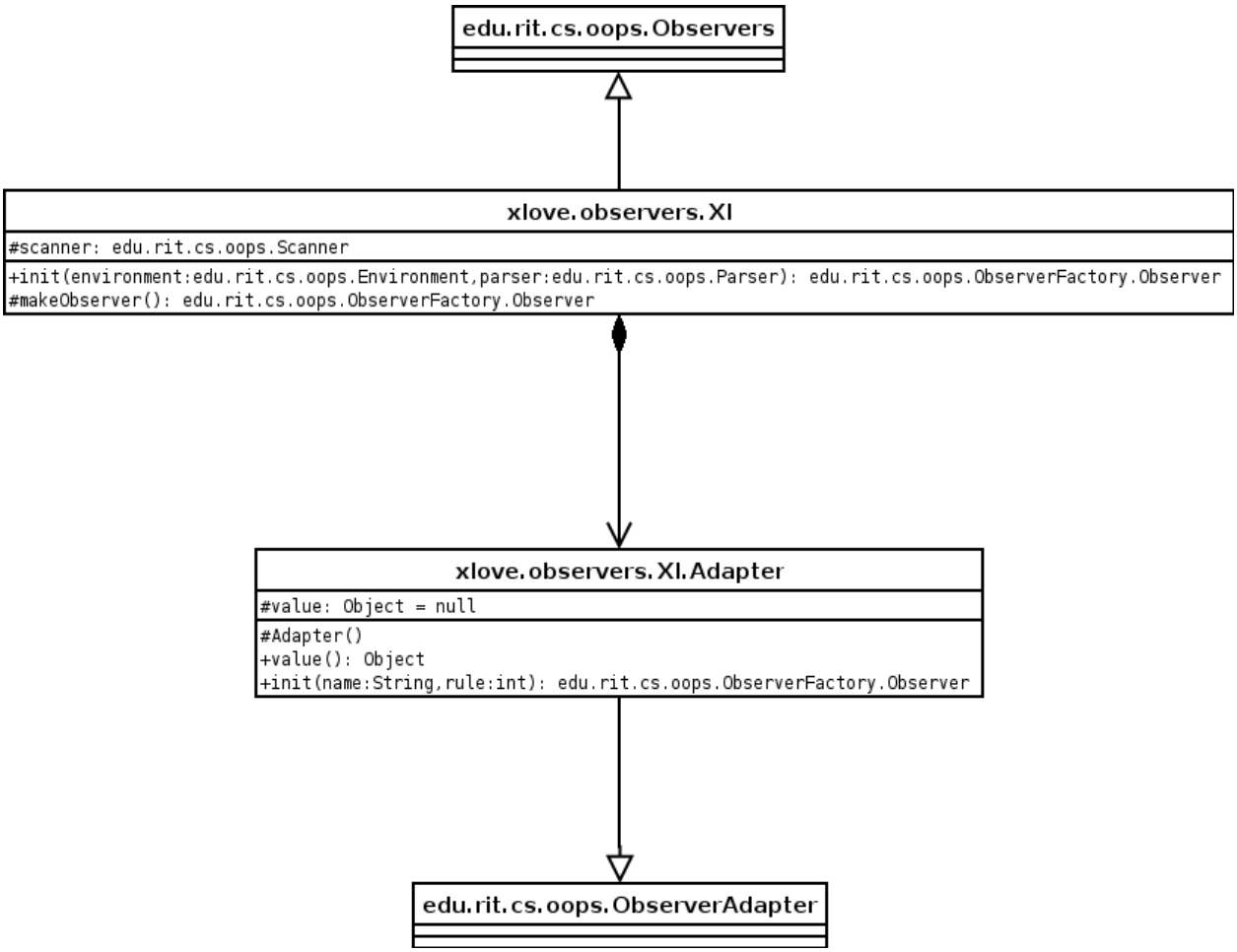

Figure 13: Package xlove.observers

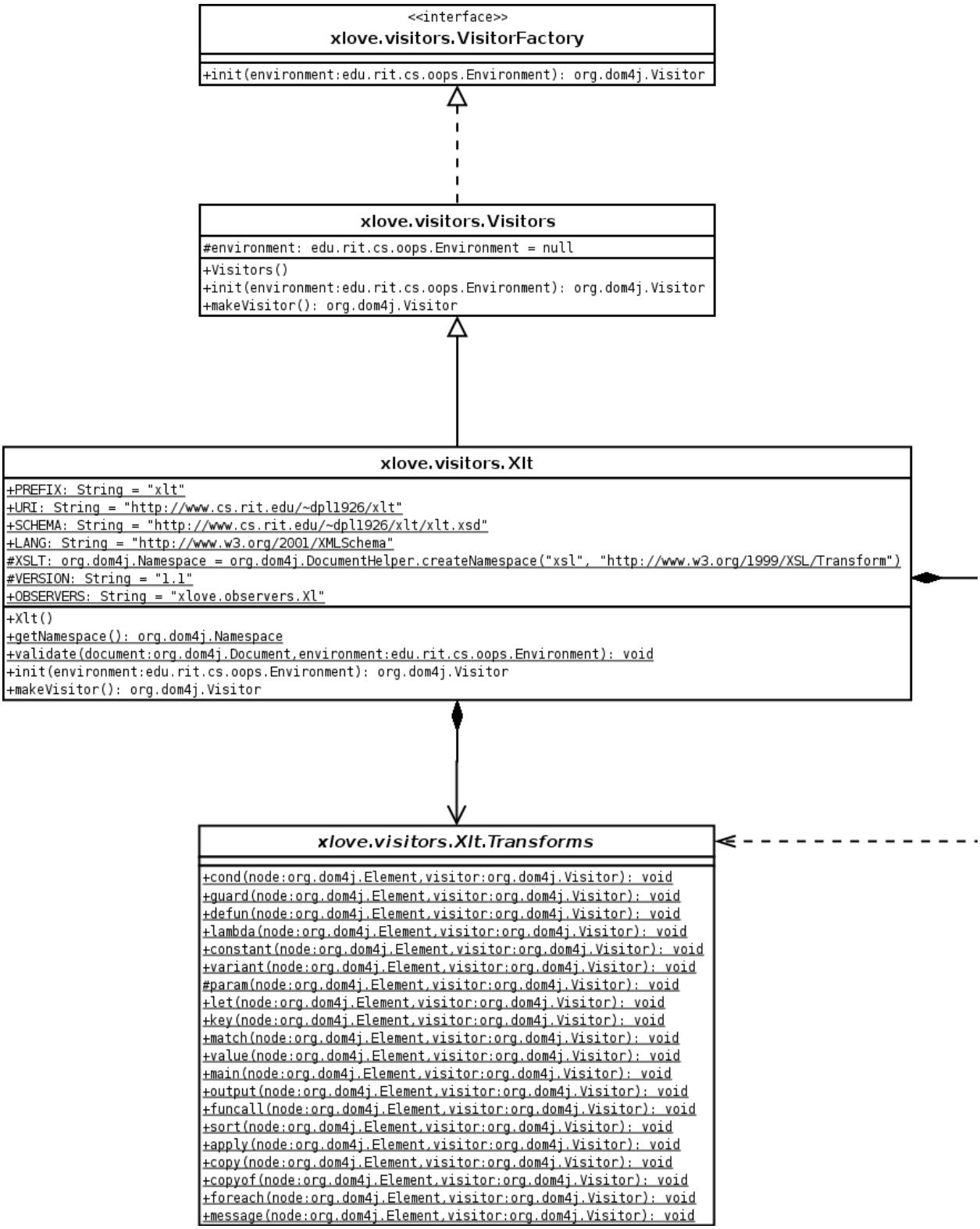

Figure 14: Package xlove.visitors

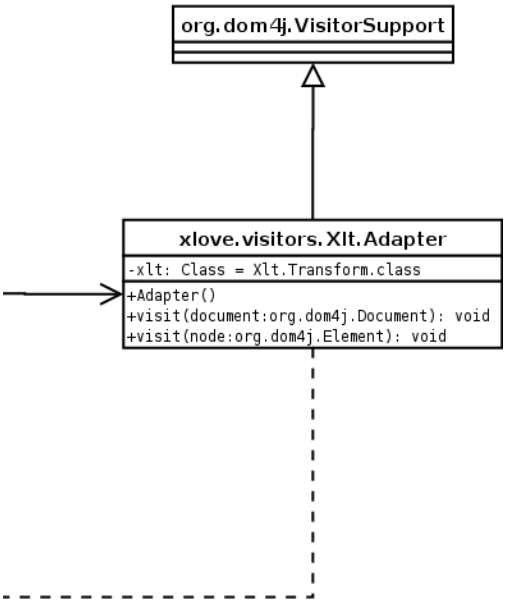

Figure 15: Package xlove.visitors - continued

4.5 Output Generation

The XLove class performs two different functions depending on the command line arguments. By default, the input Xl/t document (read either from a file or standard input) is translated into an XSLT document. The classes OutputFormat and XMLWriter from the Dom4J package are used to print the Document that results from the transformation process. Again, it is worth mentioning that for programmatic use, the XSLT document need never be printed. The second mode of operation utilizes this feature. A standard TrAX TransformerFactory and Transformer are instantiated. The transformer stylesheet is retrieved by using XLove's toSource method. The stylesheet is applied to the specified input document. For transforms that are independent of the input document (for example, the sample program for computing factorials), and empty XML file is provided. The results of the transform are then printed on standard output.

5 System Analysis

One of the major changes that has occurred in the world of XML since the beginning of this project has been the rise of the XQuery draft standard [22]. As of September 2005, the draft standard includes responses to the official Last Call Working Draft dated April 04, 2005. While it is possible that this draft may still be changed before it is fully adopted, various implementations of XQuery are already available. XQuery is a far more ambitious project than Xl/t. With that said, there are many high level architectural similarities between the project. XQuery also makes the distinction between the end user syntax of a program and the intermediary representation. For background, XQuery defines an algebra for representing XQuery programs to which an end user syntax maps. Early reference implementations allowed for an end user syntax that is the algebra (similar to the role XSLT plays in the Xl/t language). The same concepts of modularity are visible in both languages. XQuery goes even further, though. A new XPath standard (version 2.0) was specified as part of the XQuery working system as one example.

The major difference between this project and XQuery is that the two projects seek to solve very different problems. Xl/t only seeks to provide a programmer friendly syntax for transforms of XML documents. XQuery achieves this goal along with providing the ability for utilizing XML documents in a manner consistent with the extremely prevalent

usage of relational databases. Xl/t only views XML documents as documents, not as particular data stores. Hence, XQuery has the ability to perform such tasks as static typing. Additionally, XQuery has a far more sophisticated scoping system that can be used to prevent many runtime checks programmers usually have to perform when utilizing an XML document as a data store. Also, the preferred XQuery end user syntax, while having functional elements, is expressed in a very procedural manner. However, many of the virtues of functional programming are evidenced in XQuery (the ability for dynamic evaluation, binding of variables versus assignment, etc.).

The strongest point in Xl/t's favor versus XQuery is evidenced in industry. XSLT is an accepted and widely used technology. By using Xl/t, programmers are able to leverage existing infrastructure (in terms of actual application code) and simply add on new pieces in Xl/t. In this situation, it is most likely that Xl/t code will simply be a programmer convenience and at some point will be translated and output as XSLT. Also, Xl/t seeks to reuse existing standards and their implementations, as strongly evidenced by the use of the DOM, instead of developing an entirely new theoretical approach to viewing XML documents. Only time will tell if XQuery will replace the existing transformation techniques. But it is most definitely worth noting that XQuery allows programmers to solve entire classes of problems in a simple, easy to use syntax that Xl/t cannot easily solve.

In light of this, the high level design approach Xl/t takes is a valid one. It can perhaps be best viewed as a transitional step between the current industry standard approach and that used by XQuery. With the ability to define a new end user syntax for XQuery (as evidenced not only by the design, but by the existence already of multiple different end user syntaxes), it is entirely feasible to develop a functional syntax for XQuery without having to develop all of the infrastructure required in developing Xl/t. The XQuery design also offers the opportunity for much deeper optimization. While the design of Xl/t is fairly modular, it is not nearly as language agnostic as XQuery is.

6 Conclusions and Future Work

Looking at the raw syntax of Xl, it should seem eerily familiar to those who have studied early Lisp development. In McCarthy's original paper, there were two different syntaxes specified for Lisp [10]. S-expressions (which have been retained for over 40 years now) and M-expressions. The M-expression syntax never caught on (despite the beliefs McCarthy held at the time). The single, most straightforward reason for this is quite simple. S-expressions nearly eliminate the need for syntax. The only aberration is the loop function, which has been a subject of argument for many, many years. While initially an S-expression based language may be difficult to grasp, with proper style guidelines it is fundamentally easier to parse and understand - for humans and computers alike.

This is a case where SXML truly shines. Recognizing the value of S-expressions, embedding the XML-Infoset as a domain specific language (DSL) within Scheme is eminently logical. There is no need to write a new parser. This is perhaps the most fundamental weakness with Xl. Even assuming Java was still used as the implementation language for Xl, writing an S-expression parser is no more difficult that writing one for the M-expression variant used. With some Java Native Interface trickery, it would even be possible to implement Xl as a domain specific language within Lisp that ties into the existing Java infrastructure. This would have the great benefit of allowing existing Lisp programmers to easily and quickly deal with XML documents in a manner almost identical to SXML. Even further beyond that, it may be possible to use the existing SXML implementation and simply provide some shims to connect with the XLove system.

On a related note, the Xl/t sample programs provided are definitely longer than their Scheme / DSSSL counterparts would be. One large difference since the development of DSSSL is the advent of XML namespaces. This adds additional clutter to the source code of any Xl/t stylesheet. A reworking of the Xl/t implementation could potentially remove some of this cruft. In short, this should be an implementation detail hidden from users of the language unless they wish it explicitly exposed. If a stylesheet was written that used conflicting namespaces, the programmer should be able to optionally specify the namespace. They should not be required to specify the namespace of all elements used. These type of impediments to ease of programming are what Xl/t was designed to remove.

Assuming XQuery is the next logical successor to XSLT, an interesting project would be to target the XQuery algebra

in the same manner as XSLT is targeted in Xl/t. By using a DSL of XML embedded in a Lisp variant as the front end, programmers would be able to use the full power of Lisp to generate stylesheets that in the end would simply use existing XQuery processors to do the actual work of performing the transform. This would save on the extra work of implementing a full transform/query language (which for SXML is embodied in SXSLT).

In short, Xl/t represents a stepping stone between current technologies/standards and the next generation of XML processing. XQuery is close to being ratified. Whether or not it sees a large amount of adoption is still up in the air, but the general consensus seems to be "yes." The power of SXML and an S-expression based way of dealing with the XML-Infoset represents a new way of thinking about XML and providing interoperability across applications. As a DSL, SXML provides an easy way for Scheme programmers to quickly and easily use XML in their applications. Harnessing the power of both XQuery and SXML would represent the next logical step for a successor to Xl/t.

References

[1] S. Adler, A. Berglund, J. Clark, I. Cseri, P. Grosso, J. Marsh, G. Nicol, J. Paoli, D. Schach, H. Thompson, and C. Wilson. A Proposal for XSL. `http://www.w3.org/TR/NOTE-XSL.html`, 1997. Retrieved October 21, 2007.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] F. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1975.

[4] J. Clark. ISO/IEC 10179:1996 Document Style Semantics and Specification Language (DSSSL). `http://www.jclark.com/dsssl/`, 1996. Retrieved September 14, 2004.

[5] Dom4j. Dom4j. `http://www.dom4j.org/`, 2004. Retrieved September 14, 2004.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[7] E. R. Harold. *XML in a Nutshell: 2nd Edition*. O'Reilly, 2002.

[8] H. Hosoya and B. C. Pierce. XDuce: A Statically Typed XML Processing Language. *ACM Transactions on Internet Technology (TOIT)*, 3(2), 2003.

[9] O. Kiselyov. SXML. `http://okmij.org/ftp/Scheme/SXML.html`, 2004. Retrieved September 14, 2004.

[10] J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part 1. *Communications of the ACM*, 3(4), 1960.

[11] S. Microsystems. Sun Multi-Schema XML Validator. `http://developers.sun.com/dev/coolstuff/schema/`, 2003. Retrieved September 14, 2004.

[12] A. Schreiner. XML - Architecture, Tools, and Techniques. `http://www.cs.rit.edu/~ats/xml-2001-3/`, 2002. Retrieved September 14, 2004.

[13] A. Schreiner. Oops. `http://www.cs.rit.edu/~ats/projects/oops/edu/doc/`, 2004. Retrieved September 14, 2004.

[14] P. Tchistopolskii. XSLScript. `http://www.pault.com/pault/XSLScript/index.html`, 2000. Retrieved April 20, 2005.

[15] J. Tennison. *XSLT and XPATH: On the Edge*. M&T Books, 2001.

[16] H. Thompson. xslj: An XSL to DSSSL Translator. `http://www.ltg.ed.ac.uk/~ht/xslj.html`, 1998. Retrieved October 21, 2007.

[17] W3C. Cascading Style Sheets. `http://www.w3.org/Style/CSS/`. Retrieved September 14, 2004.

[18] W3C. Document Object Model (DOM). http://www.w3.org/DOM/. Retrieved September 14, 2004.

[19] W3C. Extensible Markup Language (XML). http://www.w3.org/XML/. Retrieved September 14, 2004.

[20] W3C. The Extensible Stylesheet Language Family (XSL). http://www.w3.org/Style/XSL/. Retrieved September 14, 2004.

[21] W3C. W3C XML Schema. http://www.w3.org/XML/Schema/. Retrieved September 14, 2004.

[22] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/. Retrieved September 28, 2005.

[23] M. Wallace and C. Runciman. Haskell and XML: Generic Combinators or Type-based Translation? *ACM SIGPLAN Notices, Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, 34(9), 1999.

[24] E. Willink. NiceXSL. http://nicexsl.sourceforge.net/html/overview.html, 2004. Retrieved April 20, 2005.