An XML Programming Language for Web Service Specification and Composition

Daniela Florescu
Propel
danaf@propel.com

Donald Kossmann
TU Munich
kossmann@in.tum.de

1 Introduction

XML is the lingua franca for data exchange on the Internet. Among its many possible uses, XML is ideal for publishing documents on Web sites, for storing catalogs in electronic market places, and for exchanging data between business processes. Even though some data sources will probably continue to use relational and object-relational database systems as a primary form of storage, we expect that most data sources will eventually provide XML access for their published data.

Software vendors and standard bodies, like the W3C consortium, have been very active in providing tools (XML parsers) and standardized languages (XSLT, XPointer, XPath, XQuery, etc.) for XML. So far, however, no imperative programming language has been proposed that is specifically tailored for building XML applications and Web services.

As of today most Web services are built using classic programming languages, such as Java and Visual Basic, and some kind of SQL-based RDBMS (Oracle, DB2), a mixture of paradigms inherently implying a fair number of logically irrelevant but costly and error prone intermediate manipulations. An XML Web application built on such technologies will have to deal with difficulties such as:

1. XML-Java mismatch: XML data must be converted into Java (or any other language of the sorts) internal representation as objects before it can be processed by the Java program. Likewise, Java objects must be converted back into XML data at the end of processing.

2. Java-Database mismatch: Java objects must be marshalled back and forth through JDBC-like interfaces to access and update the RDBMS, the infamous "database impedance mismatch" that triggered the development of object databases technology in the recent past[CM84].

While most people seem to have resigned themselves to using interfaces like ODBC and JDBC, the additional XML-Java "stuttering step" [Lam] might have a good chance to finally turn people's minds. It is not unusual that eighty percent or more of Web applications code is indeed due to data marshalling. Moreover, a great deal of the plumbing and hand optimizations are craftly hard-coded into the Java part of current applications. As an example, data cacheing in proxy objects, the ubiquitous astute way of reducing round-trips to the database system, is typically implemented at the application-level, making it very hard to extend these applications and guarantee their correct semantics whenever the inevitable DB schema evolution comes along.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Language implementors and database manufacturers are scrambling themselves to increase their products with “XML extensions” and to introduce automatic treatment for those chores whom programmers are currently dealing with manually. Indeed there are significant efforts both on the Java side, from database and third-party vendors in these directions (see Section 4.) However, we believe that the type systems of XML, Java, and relational database systems are simply too different and ultimately incompatible for productively building large scale applications that span accross the three different paradigmes.

The alternative that we are pursuing with this paper is to introduce XL (short for XML Language), a new high-level programming language for Web services. Five foundamental principles underlie the design of this new language:

1. XL should support a unique data model and type system: the standard XML one [Que].

2. XL should be expressive enough to describe the logic of most Web services. For most applications it should be possible to specify a Web service entirely in XL, without any need for Java or SQL.

3. XL should not just be complete with respect to Web service specification, but also comfortable to use. Hence, it should provide special constructs for important Web services programming patterns (e.g., logging and retry of actions).

4. with the help of XL, programmers should concentrate entirely on the logic of their application and not on implementation issues as performance aspects, data formats, or specific Internet and database protocols. Such aspects have to be dealt with in an automatic way and not via hand-coded and hard-coded solutions.

5. XL must be compliant with all W3C standards and it must gracefully co-exist with the current (Java-based) Web services and infrastructure.

In this paper we will outline the requirements for such an XML programming language and present a sketch of an initial design. Our design is based on familiar concepts of imperative programming languages as well as on those of parallel programming [Hoa85] and workflow management [Moh00b]. The long-term goal of this paper is to stimulate a discussion about the foundamental concepts and syntax of XL, and eventually to submit a proposal to the W3C with the help of as many practitioners and researchers as possible. Towards the end of this paper we will also indicate possible directions for future research to provide efficient implementations.

2 Requirements for an XML Programming Language

In this section, we will describe a list of specific requirements for XL. Some of the requirements are derived from the global architecture where a Web service specified using this language should be integrated.

1. **Compliance with the W3C standards.** XL must be fully compliant with the XML W3C standards such as XML Schema [Sch], XQuery [Que], XPath [XPa], XSLT [XSL] or XML Protocol [Prob].

2. **Business processes, Web conversations.** The language must provide constructs to implement business processes and, more generaly, it must support *conversations* between two or more Web services.

3. **Service composition.** XL should allow the construction of high-level services out of the composition of more basic services. It should also be possible to compose new services out of services not necessarely written in XL. It should be possible, for instance, to integrate Web services written in Java in a transparent and seamless way. The web services participating to a *conversation* have to be loosely coupled (i.e. changes in the implementation of one such service should not affect the other services that invoke or are being invoked by it).

4. **Message-based programming.** A web service implemented in XL should communicate with other web services via messages. Services are invoked via messages and results are also returned via messages.

5. **Location independent invocation.** In general web services should be uniquely identified using URIs. The invocation of a Web service should use the respective URI and be independent from the location where its code is stored or executed.

6. **Platform independence, code mobility.** The language we propose must be platform independent. It must be possible to run programs virtually on any machine that runs an interpreter for the language — independent of the operating system or the database system used.

7. **Unique XML-based data model and type system** The data manipulated should be modeled by the standard XML data model and type system [Que]. No other data models and type systems should be allowed.

8. **Optional strong typing.** Types for data components (e.g., variables) are optional. However, if a variable is associated with a type, strong typing must be enforced (i.e., type checking can be done at compile-time). Special constructs must be provided such that programmers can enforce properties of components dynamically if no specific type is statically associated with a component.

9. **Logical/physical data independence.** Programmers should be aware only of the logical structure of the XML data (i.e. the XML Schema) and they need not be aware of the physical representation of the data (e.g. DOM trees, SAX events, XML files or database tuples).

10. **High-level programming.** The language we propose must be high level and use declarative constructs whenever possible. The language must also support high-level exception handling and other special constructs to easily implement more complex services like logging, data lineage, time-triggered actions, etc.

11. **Imperative programming.** The language must provide standard imperative constructs like sequencing and iteration. However, we believe that given the particular nature of the applications we are targeting such imperative constructs will be seldom used.

12. **Transactions.** The language must provide constructs allowing programmers to specify sequences of actions to be executed in an isolated and atomic way; i.e., transaction.

13. **Universal naming for each component.** Each component (program, conversation, message, transaction, exception) must have a URI for data lineage and information traceability.

14. **Authentication, authorization and Security.** It must be possible to implement discretionary access control and, thus, restrict the use of a service.

15. **Automatic optimization.** The language design should enable and encourage automatic run-time optimizations and should discourage or even disallow low level hard-coded optimizations.

16. **Protocol transparency.** Accesses to a database and invocation of remote web services must be transparent. The protocols used (e.g., JDBC or HTTP) must be hidden in the implementation of the language.

3 Basic Concepts of XL

In the following, we will describe the concepts of an initial design of XL. These concepts provide a core functionality — more functionality (and syntactic sugar) are probably necessary to achieve wide acceptance. The semantics of certain concepts (in particular, transactions) definitively need further discussion and review — in this paper we will only discuss basic options for what we think are the most critical concepts.

XL Programs

Each XL program represents a Web service. In order to compose Web services, XL programs can send messages to other XL programs or any other Web service (e.g., written in Java) that understands SOAP messages (see the subsection on "Service Invocation" below). An XL program defines implicitly two variables: *$input* and *$output*. The variable *$input* is bound with the XML message sent to the XL program at the time the program is invoked. *$input* can also be bound by an XML Form [oWF], if the Web service is invoked by an interactive (human) interface. The variable *$output* contains the results and is automatically sent back as an XML message to the caller or an agent of the caller. Furthermore, XL programs can throw exceptions which are also sent back as XML messages to the caller or its agent.

An XL program is composed of two parts: a header and a body. The body is described by a statement (different kinds of statements and their possible combinators are detailed in the following subsections.) The header contains meta-data of the service. In particular, the header contains (optional) XML schema types for the *$input* and *$output* of the XL program and for exceptions. Declaration of such types is optional; in other words, the *$input*, *$output*, and exceptions can be untyped just like any other XML document can be untyped. In addition, the header of an XL program defines the globally unique URI of the XL program; XL programs are invoked using their URI. Furthermore, other meta-data that is necessary in order to implement transactions (see the subsection on transactions below), for security (e.g., the public key of the Web service for encryption), versions of the XL program, and for automatic optimization of XL programs (e.g., side effects of the program) can be specified in the header of an XL program. A more complete description of such metadata will be given and continuously updated in a forthcoming technical report [FK01].

The proposed syntax for the header of an XL program is as follows (optional parts are represented in brackets, keywords are represented in capital letters):

```
PROGRAM uri [typeOfInput] -> [typeOfOutput]
  [THROWS  typeOfException1]
  [meta data]
```

Basic Statements

In this section we introduce some of the basic statements that can be used in the body of an XL program. (Statements used for specific services will be discussed in the following subsections.)

Assignments, Local Variables, and Expressions The simplest statement is the assignment of a local variable. The syntax is as follows:

```
  LET [type] variable := expression
```

Variables need not be declared before being used. However, the (XML schema) type of a variable can optionally be set as part of the first assignment to this variable. The scope of a variable is the whole XL program. Expressions can be any expression defined by the W3C XQuery proposal [Que]. XQuery is very powerful and a great deal of the power of XL is gained by simply leveraging the expressive power of XQuery. XQuery expressions include normal XML elements, function calls, arithmetic expressions (i.e., use of built-in operators), XPath, and more complex queries similar in nature with the SELECT-FROM-WHERE constructs of SQL. As in XPath and XQuery, we denote variables with a $ sign.

As an example for a simple assignment, consider the following statement:

```
 LET \$harry :=  <person>
                    <name>Harry Potter</name>
                    <hobby>Quidditch</hobby>
                    <hobby>Broomsticks</hobby>
                 </person>
```

As mentioned earlier, typing is optional, but it is strictly enforced if it is used.

Update Statements Unfortunately, XQuery does not yet provide expressions to manipulate XML data. There are plans to extend XQuery in this respect and once a recommendation has been released by the W3C, XL is going to adopt the syntax and semantics of these expressions. In the meantime, we will use the following statements to manipulate XML data: (1) *insert* to add elements; (2) *delete* to delete elements; (3) *replace* to adjust elements; (4) *rename* to rename element tags. For instance, the following statement gives Harry an age:

```
INSERT <age>14</age> INTO \$harry
```

Service Invocation

Probably the most relevant atomic statements in XL are those used for invoking other Web services; i.e., send a message to another Web service. Often, the other Web service will be another XL program, but in effect it can be any service that responds to SOAP messages [Proa] or, more specifically, to the XML Protocol which is currently under development by W3C working group [Prob]. In the following, we will define Web services as any service that has a URI and that is able to listen and respond to messages according to the (emerging) XML protocol standard. Web services are invoked independently of the specific way they are implemented.

We propose two ways to invoke a Web service as part of an XL program: synchronous and asynchronous. The syntax of a synchronous call is as follows:

```
expression --> uri  [--> variable]
```

The semantics is rather straightforward. A message with the value of *expression* is sent to the service identified by *uri*. The execution is halted until the called service finishes it execution and returns the entire result (also wrapped in an SOAP message). If a *variable* is given as part of the call, then the body of the message returned by the called service is copied into this variable. The exact semantics of synchronous calls in the presence of transactions, will be discussed in a later subsection on transactions.

The syntax of an asynchronous call is as follows:

```
expression ==> uri [==> uriOfHandler]
```

In this case the execution of the XL program will not block and the program will immediately continue executing the next statement after the message to the called service is sent. If the output of the called service (errors or any other reply message) needs to be processed, then the URI of the Web service in charge with this processing can be given as part of the call. The exact semantics of asynchronous calls in the presence of transactions will be discussed in a later subsection on transactions.

Currently, XL does not explicitly support business conversations. In order to implement business conversations, every step of a conversation needs to be implemented by a separate XL program and there need to be special XL programs that keep track of the context of the conversation and invoke the XL programs that carry out the individual steps. Alternatively, contextual information about the current conversation can be carried out in the XML messages themselves. Furthermore, there is no explicit support for multicasts yet. In order to send a message to multiple services, each service must be individually invoked by a separate message. We plan to extend XL along these lines as part of future work.

Statement Combinators

Obviously, the body of an XL program can contain more than one atomic statement. There are several ways to combine statements in order to generate other statements, as explained below. In the following "statement1 and "statement2 can refer to any atomic statement as the ones described in the previous sections or the any combination of statements.

1. **Sequence.** The typical way to combine statements is by using the ";" symbol, like in C++ or Java. Thus, the following means that "statement1" is executed before "statement2."

   ```
   statement1; statement2
   ```

2. **Failure.** If “statement1 fails, execute “statement2.

```
statement1  ?  statement2
```

3. **Choice.** Execute either “statement1 or “statement2, but not both. Which one is executed is nondeterministic.

```
statement1  |  statement2
```

4. **Parallel execution.** Execute “statement1 and “statement2 in parallel. In other words, the order in which the individual statements are carried out is not specified.

```
statement1  ||  statement2
```

5. **Dataflow.** If there are data dependencies between “statement1 and “statement2 (e.g., “statement1 binds a variable that is used in “statement2), then execute the statement that depends on the other statement last. If there are no dependencies, then execute “statement1 and “statement2 in any order. If there is a cyclic dependency, then this combination of statements is illegal.

```
statement1  &  statement2
```

Access to Persistent Data

Similarly to Web services, any kind of persistent data that can be accessed by an XL program must have an associated URI. Examples for persistent data are tables of a relational database (viewed as XML), an XML document stored in the file system, a Web page provided by a Web server. In order to support access to persistent data, XL provides an *alias* statement that binds a local variable name to the persistent data referenced by a given URI. The syntax of this statement is as follows:

```
ALIAS  uri  AS  variable
```

If such a variable is involved in the rest of the program in an expression, then the persistent data will be queried. If the variable is on the left side of a LET statement, then a new value will be assigned to the persistent data (the old value of the persistent data will be lost). Furthermore, persistent data can be manipulated using the update statements described before (or the equivalent constructs once XQuery has been extended in this direction). For instance, if `HTTP://personDB.com/address` is the URI identifying the XML view of a table in a relational database containing contact information of wizzards, then the following sequence of statements will insert a new record into this database (note that wizzards use owls to send messages):

```
ALIAS  HTTP://personDB.com/addresses AS $contact;
INSERT <person> <name>Harry Potter</name>
                <owl>Hedwig</owl>
       <person> INTO $coordinates
```

Of course such updates will only work if the source of the persistent data (e.g., a Web server or a relational database) supports updates. If not, then the execution of such an INSERT statement will fail; i.e., an exception will be raised. Similarly, the execution of LET, DELETE, RENAME, and REPLACE statements can fail if the data source does not support updates. Web pages, for instance, will typically not be updatable, but, of course, Web pages can easily be queried.

An important question is how persistent data are created. To answer this question, XL provides special web services (part of a built-in Web services library) that can be invoked to create persistent data. Such a service takes the initial value or the expected Schema description of the persistent data as input XML message and returns the URI of the newly generated persistent data. Obviously, several such web services may exist, depending for example on the particular requested implementation for the persistent data (e.g. relational databases, XML files).

Our way to create and manage persistent data is very similar in spirit with the approaches proposed for persistent programming languages [AM95]. However, the novel Web services context imposes the reconsideration of a certain number of issues. For example, unlike most traditional persistent programming languages, we do not recommend for XL the concept of persistence by reachability. In other words, the newly created data persists even if all references to it are deleted, as we believe that automatic garbage collection is both not feasible and not desirable on the Internet.

Further Statements

As we previously mentioned, The XL language also contains additional statements necessary to implement basic services. We briefly list them here; the full syntax and semantics is described in detail in the upcoming technical report [FK01]. Those statements are: (1) exception handling (e.g. *raise, try-catch*); (2) retry the execution of failed statements for a certain period of time or for a certain number of times; (3) periodically invoke certain services; (4) (temporarily) halting the execution of a program for a certain time; (5) logging actions and the state of variables; (6) loops and conditions (e.g., *for, repeat, while, switch* statements); (7) assertions and invariants in order to dynamically check properties (e.g., types).

Transactions

One of the most important requirements for the composition of Web services is to support transactions; i.e., a sequence of statements that are carried out in an atomic and isolated way. In order to specify transactions in XL we propose the following syntax:

```
ATOMIC   statement   ENDATOMIC
```

Here, *statement* can be a sequence of statements or any other combination of statements. For each transaction a URI is allocated. The URI is passed along when a Web service is invoked as part of a transaction.

There has been a great deal of work since the seventies on distributed transactions and models for distributed transactions (see, e.g., [Moh00a]). All this work is relevant. In our initial design, we propose to enforce the full ACID paradigm for transactions. Enforcing an ACID paradigm involves the implementation of a two-phase commit protocol [BN96].

To implement a two-phase commit protocol, additional meta data for a Web service must be available. If a Web service A is invoked by an XL program B as part of a trasaction, the URIs of services that *prepare, prepareToCommit, commit,* and *compensate* the Web service A must exist. The purpose of those services is to take the URI of a transaction as input and carry out necessary actions in order to implement isolation and two-phase commit. If they do not exist, then the Web service A cannot be invoked as part of the transaction, and a compile time or at run-time exception is raised.

Likewise, transactional services must be supplied if persistent data is read or modified as part of a transaction. Obviously, special attention must be paid if a service is invoked asynchronously as part of a transaction; details of our solution are given in the upcoming technical report [FK01].

Finally, an important question concerns the semantics of nested *ATOMIC* statements. If an XL program A calls as part of a transaction T_1 an XL program B that itself defines a transaction T_2, then the *atomic - endatomic* statements of B are ignored; that is the statements of T_2 are carried out as part of the transaction T_1. In other words, nested *ATOMIC* statements are automatically flattened. An alternative would be to implement the original nested transaction model.

As mentioned at the beginning of this subsection, this is only an initial design. Ultimately, more flexibility will probably be needed. In some scenarios, for instance, two-phase commit is simply not feasible or not necessary. Also, other transaction models [Moh00a], specially designed for the composite, distributed Web services might be more useful. It is also conceivable that the W3C working group on XML protocol will recommend such a transaction model. Our purpose here is to give a starting point for further discussion and to provide a placeholder in our design.

Security

Currently, XL provides no constructs to authenticate the addressee of a message and to check whether the addressee is authorized to invoke the service. However, authentication and authorization can be implemented and integrated as specific Web services. For instance, an XL program that requires authentification would be invoked by sending it a signature as part of the *$input* message (e.g., in a special `<signature>` element). That XL program would then invoke a trusted Web service (provided by, e.g., VeriSign) in order to check the signature. In a future version of XL, we plan to provide special statements (such as those for logging) in order to help the developers of critical Web services that require authentification and authorization. For instance, such requirements could be declared in the header of an XL programs and the programmer could be relieved from the details of calling a service that checks signatures.

In order to protect messages from being trapped by third-parties, any implementation of XL will use standard encription mechanisms as proposed by the W3C XML Encryption working group [Enc]. Therefore, encryption will be automatic and XL programmers need not worry about the security of their messages.

4 Related Work

The development and composition of Web services (or e-services) is currently a very active area in both industry and academic research. Very good resources that address various aspects of this area are the recent W3C workshop on Web services [Con] and the latest issue of the IEEE Data Engineering Bulleting on e-services [deb01].

The main purpose of our work was to provide a clean basis for a new XML programming language for rapid developing of Web services. Such a language will obviously not be built from scratch but using the knowledge and technology advancements accumulated in the last 30 years.

In the industry, there have been a number of concrete proposals for new languages and frameworks related but not identical to our programming language proposal. Compaq has developed the WebL language [Web]; HP has developed the eFlow and eSpeak systems [CIJ+00, eSp], IBM is working on a language called Web Service Flow Language [WSF], and Microsoft has recently released their BizTalk Server 2000 [Biz]. There is also a Business Process Model Initiative with the goal to implement cross-organization processes and workflows on the Internet [BPM]. The state of the art in the Java world is to support XML via so-called servlets that translate (XML and HTML) requests into Java classes and back [JAK]. At the time of this writing, Sun Microsystems just introduced the JXTA project on peer-to-peer computing to support distributed computing on the Internet [JXT]. Finally, the notion of a service composition is based on a solid theoretical background consisting on the calculus developed first by Hoare[Hoa85] and more recently by Cardelli [CD99].

However, none of those languages and frameworks are totally consistent with the current W3C standards, and we believe that this is a mandatory condition for the success of such a programming language. There a number of (de-facto) standards which are currently under development and which are important requisites in order to enable the open composition of Web services; in particular SOAP [Proa], XML protocol [Prob], UDDI [UDD], and WSDL [CCMW].

5 Summary

In this paper we sketched the requirements and presented an initial design of an XML programming language whoose purpose is to render the implementation and the composition of new and existing Web services as easy as possible.

Developers should not worry about details of Internet protocols, database systems, and the infrastructure. Developers should also not worry about hand-tuning their applications or about marshalling particular data formats, but instead, they should concentrate on the application logic.

Our short-term goal is to foster discussions that will eventually come to a consensus for a complete design of such a language. Open questions involve the semantics of transactions, support for comfortable conversations, and security aspects (e.g., authorization and encryption). However, given the current great interest on Web services in industry and in academic research projects, we expect discussions to open very quickly and to be carried into W3C working groups. A more complete design of XL is given in a technical report [FK01]. This report is constantly updated and it contains more elaborated examples that demonstrate the power of XL.

In the long run, the implementation of XL and of a global infrastructure for Web service composition will involve a large number of new research opportunities; e.g., code mobility, automatic caching and optimization.

References

[AM95] M. P. Atkinson and R. Morrison. Orthogonally persistent object systems. *The VLDB Journal*, 4(3), 1995.

[Biz] BizTalk.org. Biztalk initiative. http://www.biztalk.org/home/default.asp.

[BN96] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan-Kaufmann Publishers, 1996.

[BPM] BPMI.org. Business management initiative. http://www.bpmi.org/index.esp.

[CCMW] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl.

[CD99] L. Cardelli and R. Davies. Service combinators for Web computing. In *IEEE Transactions on Software Engineering (TSE)*, 1999.

[CIJ+00] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M.-C. Shan. eFlow: a platform for developing and managing composite e-services. Technical report, Hewlett Packard Software Technology Laboratory, 2000.

[CM84] G. Copeland and D. Maier. Making Smalltalk a database system. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 316–325. ACM, 1984.

[Con] W3C Consortium. Workshop on Web services. http://www.w3.org/2001/01/WSWS.

[deb01] Special issue on Infrastructure for Advanced E-services. *Data Engineering Bulletin*, 24(1), March 2001.

[Enc] XML Encryption. http://www.w3.org/encryption/2001/.

[eSp] eSpeak. The universal language of e-services. http://www.e-speak.hp.com/.

[FK01] D. Florescu and D. Kossmann. An XML Programming Language for Web Service Specification and Composion. Technical report, TU Munich, June 2001. To appear at http://www3.in.tum.de/public/mitarbeiter/kossmann.html

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[JAK] JAKARTA. The JAKARTA project. http://jakarta.apache.org/.

[JXT] JXTA. Project JXTA. http://www.jxta.org/.

[Lam] L. Lamport. The temporal logic of actions. Technical report, Digital Systems Research Center.

[Moh00a] C. Mohan. Transaction processing and distributed computing in the internet age. Technical report, 2000.

[Moh00b] C. Mohan. Workflow management in the internet age. Technical report, 2000.

[oWF] XForms: The Next Generation of Web Forms. http://www.w3.org/markup/forms/.

[Proa] Simple Object Access Protocol. http://www.w3.org/tr/soap/.

[Prob] XML Protocol. http://www.w3.org/2000/xp/.

[Que] XML Query. http://www.w3.org/xml/query.

[Sch] XML Schema. http://www.w3.org/xml/schema.

[UDD] UDDI.org. Universal description, discovery and integration of businesses for the web. http://www.uddi.org/.

[Web] WebL. Compaq's web language. http://www.research.compaq.com/SRC/WebL/.

[WSF] WSFL. Web services flow language. http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.

[XPa] XML Path Language XPath. http://www.w3.org/tr/xpath.

[XSL] Extensible Stylesheet Language XSLT. http://www.w3.org/style/xsl/.