XL: An XML Programming Language

Daniela Florescu
INRIA
daniela.florescu@inria.fr

Andreas Grünhagen
TU Munich
gruenhag@in.tum.de

Donald Kossmann
TU Munich
kossmann@in.tum.de

Abstract

We present an XML programming language specially designed for the implementation of Web services. XL is portable and fully compliant with all W3C standards such as XQuery, XML Protocol, and XML Schema. One of the key features of XL is that it allows programmers to concentrate on the logic of their application. XL provides high-level and declarative constructs for actions which are typically carried out in the implementation of a Web service; e.g., logging, error handling, retry of actions, workload management, events, etc. Issues such as performance tuning (e.g., caching, horizontal partitioning, etc.) can be carried out automatically by an implementation of the language. This way, the productivity of the programmers, the ability of evolution of the programs, and the chances to achieve good performance are substantially enhanced.

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 (at least for a certain time), we expect that most data sources will eventually provide XML access for their published data.

Software vendors and standard bodies, like the W3 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, and, unfortunately, side-effect free languages like XSLT and XQuery are not powerful enough to describe the logic of complex Web services.

The concept of "Web services" became recently very popular; however, there is no clear agreed upon definition yet. By a *Web service* we understand an autonomous piece of software uniquely identified by an *URI* and that can interact with peer Web services via *messages* using *Internet protocols* like XML, XMLP or HTTP. Web services can participate in complex *conversations*. A conversation is an exchange of messages among a certain number of participant Web services. By exchanging messages during a conversation the participant Web services strive to achieve a certain (business) goal (e.g. place an order, place a bid, send a request for information). The Web services can, but they are not required to, maintain *contextual* or *historical* information about the messages exchanged as part of a certain conversation. The Web services can, in general, perform several *operations*, each being specialized to a certain task. Operations are invoked by XML messages using the emerging XML protocol [Prob]; the result of the execution is sent back to the originator using the same protocol.

As a typical example for a Web service, consider an online broker. The online broker can be identified via a Universal Resource Identifier (URI) and can be requested to perform a certain set of operations; e.g., login, buy stock, sell stock, etc. To log-on, an XML message containing the user's name and password is sent to the broker; this message is handled by the broker's *login* operation. As a result, the online broker will return an overview of the stock held and some other information; e.g., analysis and announcements. The login operation is the first operation that is invoked in a conversation with the online-broker. After logging onto the system, the user can send messages to the broker in order to carry out other operations (e.g., buy more stock) as part of the same conversation. The online broker might itself send messages to other services; e.g., the user's credit card company or bank in order to satisfy one of its own operations.

1.1 Current problems in Web services implementation

As of today, most Web services like the one described above are built using classic programming languages, such as Java or Visual Basic, and some kind of SQL-based RDBMS (e.g., Oracle and DB2), a mixture of paradigms that inherently implies a 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 objects (or the internal representation of another language of the sorts) 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 marshaled back and forth through JDBC-like interfaces to access and update the RDBMS. This is the infamous "database impedance mismatch" that triggered the development of object databases technology [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 the code of a Web application is indeed due to data marshaling. Moreover, a great deal of the plumbing and hand optimizations are hard-coded into the Java part of current applications. As an example, data caching in proxy objects in order to reduce 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 when the database schema evolves.

Language implementors and database manufacturers are making great efforts 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 6.) 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 across the three different paradigms.

In addition to the double impedance mismatch, programmers face another problem that drastically impacts both the productivity *and* the performance of Web services. The problem is that, very often, the application tier mixes in a dangerous way, in the same imperative language (e.g. Java), very different semantic actions. As a result, it becomes very hard, and hence very expensive, to evolve such a monolithic application. Examples of different semantic actions that are very often mixed in the same programs are: low level protocols manipulation, data validation, data transformation, data cleaning, real application logic, performance improvement mechanisms like caching, replication, partitioning or load balancing, error handling, etc.

In the case where those different semantic actions are strongly interleaved, the application evolution is almost impossible. Imagine the difficulty of changing the data model of such an application, where data spans three different paradigms and data instances are present in the three different layers and, potentially, cached or materialized in various places on the platform! Or imagine the difficulty of changing the error handling mechanism of the application! In many cases, programmers are obliged to rewrite large portions of the application when such needs arise. In addition, the performance of such applications is hard to guarantee when the natural changes in the data volume or the application workload arise. As everybody knows, the common solution today to the performance problem lies in the applications ability to achieve horizontal scalability, i.e. the ability of an application to scale with the number of physical machines that execute it. Achieving this famous horizontal scalability in the case of such a monolithic, semantically unclear application is, if not impossible, at least very hard and very expensive.

1.2 Our contribution

The alternative that we are pursuing with this paper is to introduce a new high-level programming language called XL for the specification of Web services. There are a couple of reasons why we believe that such a language can simplify the above problems. First, we believe that programmers should use a *single* data representation and type system to program a Web service. For obvious reasons we believe that this unique type system should be the XML one. Moreover, the language should be, as much as possible, *declarative* and *high level*.

We listed above a couple of semantic actions that are interleaved today, with unfortunate consequences. Among those, we believe that all the performance improvement mechanisms should not be programmed by hand, but derived automatically. A lesson that we all learned from the success of relational databases is that the ability to do automatic optimization is a key factor in the possibility to evolve an application and to be robust to the inevitable hardware or software platform evolution, or the environment evolution (data volume or statistics, application workload). Such a declarative, high level language can simplify or solve this problem, since it provides great opportunities for automatic optimizations like application transparent caching, replication, partitioning, or load balancing.

Another important feature of XL is that other semantic actions, like data validation, error handling, and scheduling, are as much as possible separated from the application logic itself, and, as much as possible, specified in a declarative manner[1].

Five fundamental 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. However, peer Web services should be able to exchange messages independently of their own internal specification language (e.g. Java or XL).

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, retry of actions, and periodic actions). Simple application logic should be expressed in a declarative way; imperative constructs should only be necessary for complex and unusual programming patterns.

4. With the help of XL, programmers should concentrate entirely on the logic of their application and not on implementation issues such 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.

[1]By declarativity we mean the programmers ability to specify *what* action do they want to perform and not *how* to implement it.

The rest of the paper is organized as follows. Section 2 describes the main design goals for an programming language for Web services. Section 3 gives a short reminder of existing XML related standards (XML data model, XML Schema, XML type system and XML expression language – XQuery). Section 4 details the design of XL. Section 5 gives a simple example, a publish & subscribe system. Section 6 describes the related state of the art. Section 7 concludes this work. In the Appendix, a more elaborate example is given: an auctioning system.

2 Desiderata for an XML Programming Language

In this section, we will describe a list of more specific requirements that drove our design of XL. Some of the requirements are derived from the global architecture in which 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], XML Forms [oWF], and XML Protocol [Prob].

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

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

4. **Message-based programming.** A Web service implemented in XL must communicate with other Web services via (XML) messages. Services are invoked via messages and results are also returned via messages.

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

6. **Platform independence, code mobility.** The language 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 must be modeled by the standard XML data model and type system [Que]. No other data models and type systems are 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 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 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 expect that such imperative constructs will be used less often than in traditional programming, given the particular nature of the applications we are targeting. Simple application logic should be expressed in a declarative way; imperative programming should only be necessary for complex application logic.

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., transactions.

13. **Universal naming for each component.** Each component (program, conversation, message, transaction) 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 The XML data model, type system and expression language

The first and one of the most important requirements in the design of XL was its graceful coexistence with the other XML related standards. By design, XL uses at its core the XML data model, type system, schema languages and expression languages standardized by the W3C. In this section we give a brief summary of those standards.

XML data model A very popular wrong myth related to XML is that "XML is just a syntax, and programmers cannot reason on a syntax". While it is indeed true that the original XML recommendation described only a *syntax* for data and documents, and not a *logical data model*, the W3C is currently in the process of standardizing such a logical, abstract data model for XML. The purpose of an abstract data model has been clear since the original papers of Codd in the 70's: it allows programs to achieve *logical and physical data independence*. In other words, programmers can concentrate on the abstract representation of data and they can ignore the real physical representation of the data. As a result, the physical data representation can evolve *without* any impact on the code of the applications itself. The huge advantage of this concept has been validated in the last 30 years by the success of the database industry. Fortunately, the XML standards did *not* ignore this important database heritage. The semantics of the current W3C's XML related programming languages (XSLT and XQuery) are described in terms of an XML abstract data model [M] that serves the same purpose as the relational data model for relational databases.

The W3C XML data model describes, in an E/R fashion, a set of entities present in an XML document and a set of relationships among them. The entities describe the data itself (e.g. nodes, values, sequences) and schema components. The data is modeled as very general mathematical structures, i.e. as *ordered trees of nodes*. The internal *nodes* have node identity and they can be of several kinds (e.g. document nodes, element nodes, attribute nodes, comment nodes, processing instruction nodes, namespace nodes) while the leaves of the trees, i.e. the values, can be values in the domains of the XML Schema basic types (e.g. integer, decimal, string, duration). Pivotal to the XML data model is the notion of *sequence*. One important property of the XML data model is that sequences are always flat; i.e., sequences of sequences are automatically unnested. Another important property of the XML data model is the ability to capture the *topological order* in nodes of the document. This order can be queried and exploited during the computation. Finally, an XML data model is also able to model *errors*, which are accepted intermediate results of various XML computations.

XML Schemas and the XML type system The data model describes only the basic composition of ordered tree. The XML Schema [Sch] describes *structural and content-based constraints* on the ordered trees. The XML Schema describes the simple types (with their accepted domain values and the accepted basic operations) supported by the XML data model, the definition of user-defined complex types and gives a basic support for user-defined integrity constraints (e.g. referential integrity constraints, lexical constraints).

The XML type system formally described in [sem] captures the essence of the structural information present in the XML Schema. The goal of the type system is threefold. First, it allows automatic *type inference*: given an XML expression (as described in the next paragraph) and the type of the input data set, the type system is able to intentionally (i.e., without executing the query on any particular data set) derive the type of the result. Second, it is possible to do *type checking*: given an expression and the expected type describing the input data set, it is possible to derive that the expression will return errors on all the valid instances of the input type. Finally, the type system is capable of testing the *type subsumption*. This is a useful feature for the following scenario: given an XML expression and the type describing the expected input data set, detect automatically if the result of the evaluation of the expression on all valid input data instances will be valid instances of a predefined expected output data type. More detailed information about the type system can be found in [sem].

XML expressions and XML queries A complementary W3C standard deals with XML expressions and XML queries[Que]. XQuery is a functional language. Like all functional languages, XQuery expressions are constructed using first order and second order function applications starting with variables and constants. Examples of first order functions are: logical, arithmetic, string manipulation, collection oriented operations like union, intersection and difference. Examples of second order functions are *map* and *sort*. Of particular importance are the second order FLWR expressions: they are XML expressions constructed based on a pattern that is akin to SQL's SELECT-FROM-WHERE queries. Like SQL queries, a FLWR expression has a special clause to define variables and their associated domains (the FOR clause in XQuery corresponds to the FROM clause in SQL), a special clause that filters variable bindings based on predicates (the WHERE clause in both languages) and a special clause that specifies how to construct the result (the RETURN clause in XQuery corresponds to the SELECT clause in SQL). Special expressions called *path expressions* are used in order to navigate in an XML tree; the syntax and semantics of path expressions are defined in the XPat standard [(XP].

XML queries [Que] are declarative, side effect free programs that manipulate XML data. A query is composed of a preamble containing function definitions, local type declarations, function declarations, XML schema imports, plus a main expression to be evaluated and returned as a result of the execution of the program. Unfortunately, the logic of complex Web services cannot be described using *only* declarative programs, or using *only* side effect free programs.

4 XL syntax and semantics

The programming language we propose in this paper extends the simple XQuery expression language to a full programming language, powerful enough to specify the logic of complex Web services. In this section we will describe the concepts of an initial design; these concepts provide a core functionality. More functionality (and syntactic sugar) are probably necessary to achieve wide acceptance. Moreover, the semantics of certain concepts (e.g., transactions or access control) need further investigation; they are omitted from this paper.

4.1 Web services in XL

A Web service in XL generalizes the notion of an XQuery entity. Like an XQuery entity, a Web service is identified by a unique URI (the target URI). Also, a Web service specification can contain a set of local function declarations plus a set of type definitions and schema imports. In addition, a Web service specification in XL can contain: (a) local data declarations, (b) declarative clauses, and (c) specifications of operations. We will describe these three novel components of XL in the following subsections.

The syntax of XL for Web services is as follows. Here and in the rest of the paper, keywords are denoted in bold-face and non-terminals are enclosed in angle brackets. Optional parts are denoted in square brackets. Comments are represented in italics. An asterisk is used if a clause can occur 0 or more times. The order in which the individual clauses occur is arbitrary; the individual clauses are separated by semi-colons.

```
service <uri>
     <FUNCTION DEFINITIONS>          !! inherited from XQuery
     <LOCAL DECLARATIONS>            !! inherited from XQuery and extended
     <DECLARATIVE WEB SERVICE CLAUSES >         !! new
     <OPERATION SPECIFICATIONS>          !! new
endservice
```

4.2 Web services local declarations

As in XQuery, a Web service implemented in XL can have local types and imported schema components. In addition, an XL Web service can declare local variables. Such variables hold only XML data and their potential values can be constrained by the XML type system. Two kinds of local variables can be declared in XL. The first kind of variable represents the internal state of the whole Web service; e.g., the customer database of an online broker. The second kind of variable represents the internal state of a particular conversation that the Web service is involved in; e.g., the session id when a user logs into the system or the maximum bid for an item in an auctioning system. The syntax for declaring these two kinds of variables is the following:

```
service <uri>
    !! function definitions, local types, schema imports
    ......
    !! state of the Web service
    ( let  [<type>] <variablename> [:= <expression>];)*
    !! state of a conversation of the Web service
    ([ context let [<type>] <variablename> [:= <expression>];])*
    !! declarative web services clauses and operations
    .....
endservice
```

All variables can be used and updated by each operation of the Web service. In this syntax, the “type” is the optional type constraining the type of the variable’s values, while “expression” is an XQuery expression describing the initial value of the variable. If no “expression” is given, then the variable is initialized to the empty sequence; if no “type” is given then the variable can be bound to any valid instance of the XML data model. Again, the difference between these two kinds of variables is that variables of the first kind are independent of conversations (i.e., they persist conversations) whereas variables of the second kind are associated to conversations (i.e., their life time is bound by the lifetime of a conversation). We will describe conversations and the semantics of such variables that form the context of a conversation in the next section when we detail the role of conversations.

4.3 Declarative Web service clauses

Essentially, this part contains a set of high level declarations that control the Web services global state, how the Web service operations are executed and how the Web services interacts with other Web services. The syntax for these clauses is as follows.

```
service <uri>
    !! function definitions and local declarations
    .....
    !! declarative web service clauses
    [ history]
    defaultoperation <operation>;
    [ unkownoperation <operation>;]
    [ init <operation>;]
    [ close <operation>;]
    ([ invariant <booleanExpression> throw <expression>;])*
    ([ on change variable invoke <operation>;])*
    ([ on event <booleanExpression>
               invoke <operation> [ with input <expression>];])*
    [ on error invoke <operation>;]
    [ conversationpattern ( required | ... | never);]
    [ conversationtimeout <durationExpression> [<operation>];]

   !! operation specifications
   ......
endservice
```

In the following, we will briefly describe the individual clauses. The meaning of the individual clauses will become clearer in the discussions and examples of the following subsections.

HISTORY: If this clause is specified, then all calls to operations of the Web service are automatically logged and recorded in an implicitly declared read only *$history* variable. The data automatically recorded in this variable includes for example the name of the operation that is called, the identifier (URI) of the caller, the value of the input and output messages, the timestamp when the operation was called and other statistical information that are important for the Web services tracing and monitoring.

Automatic logging is very useful for security reasons and in order to implement certain kinds of constraints. For instance, the *$history* variable could be used in the *debit* operation of a credit card company in order to constrain the number of transactions of a user per day.

DEFAULTOPERATION, UNKNOWNOPERATION these clauses declare the Web services behavior in cases when a message is sent to the Web service service and it is unclear which operation should process the message. The DEFAULT operation is executed whenever a message is sent to the service and no operation name is specified as part of the message. The UNKNOWN operation is executed if a message is sent to the server and the caller specifies the name of an operation which is not defined in the Web service. If no UNKNOWNOPERATION clause is given, then the default operation is used in such cases.

INIT, CLOSE these clauses specify a pair of operations that are automatically invoked when the Web service is created and destroyed, respectively. These operations can only be invoked once and they take no input.

INVARIANTS: In this clause, global Web services integrity constraints (or invariants) are defined. A Web service can define an arbitrary number of invariants. Typically, invariants are defined for stateful services and constrain the value of internal variables. Invariants, however, can also constrain the value of the *$history* variable and contexts of conversations. If at any time an invariant is violated, the statement that caused the violation is undone, an exception is raised, and the execution of the current operation is stopped if the exception is not handled. The exception that is raised when an invariant is violated is specified in the optional “exceptionExpression” part of the INVARIANT clause.

As an example, it could be specifiecd that all customers of the online broker must be older than eighteen years and that the balance of the account of each customer must be greater than 0. These two invariants would be defined as follows:

```
invariant   $customer/age > 18
             throw  <error> You are too young! </error>;
invariant   $customer/balance >= 0
             throw <error> Out of cash! </error>  ;
```

ON CHANGE In this clause, a simplified form of triggers can be specified. The semantics are straightforward: if the value of the *variable* changes, *operation* is called with an empty input. Changes to any variable declared in the Web services local declarations (Section 4.2) can be monitored in this way; likewise, changes to the *$history* variable (if declared) can be traced in this way.

ON EVENT This clause allows to declare more elaborate triggers and periodic tasks. Whenever, the *Boolean-Expression* evaluates to *true*, the operation is invoked. If an INPUT is specified, the corresponding expression is evaluated and passed to the operation as input. In many cases, the *BooleanExpression* will depend on some timestamp. For instance, the following clauses of our online broker example specify that dividends are once a year (October 1) and that fees are due every month. xf:currentDateTime() is the XQuery/XPath function that returns the current Timestamp; xl:createDateTimeSeq() is an XL function that constructs a sequences of timestamps, using $*$ as a wild card in the timestamp expression.

```
!! October 1, every year
 on event xf:currentDateTime()
              = xl:createDateTimeSeq("*-10-01-00:00")
 invoke addDividend;

!! every month
```

```
on event xf:currentDateTime()
             = xl:createDateTimeSeq("*-*-01-00:00")
invoke computeFee;
```

Note the semantics of the = operator in this example: the = operator is equivalent to an existential quantification according to the XQuery standard [sem].

ON ERROR INVOKE This optional clause specifies an operation that is called whenever an (other) operation of the Web service fails; e.g., if an INVARIANT is violated. In other words, if an operation raises an exception, this exception is passed as input to the operation specified in the ON ERROR INVOKE clause and the output of this operation is then returned to the client of the Web service. This way, application logic can be separated from error handling; in particular, all texts for error messages are employed by one operation only. As will be discussed in Section 4.5.2, exceptions can also be handled locally using TRY and CATCH statements. The operation specified in this clause is only called for exceptions that are not handled locally and would otherwise directly be returned to the client of the Web service.

CONVERSATIONPATTERN The CONVERSATIONPATTERN clause specifies in a declarative manner how the Web service interacts with other services as part of conversations. We assume here that the SOAP messages which are exchanged between Web services can carry the unique identifier (URI) of a conversation in their envelop[2]. It is very tiresome to specify for each message individually to which particular conversation it belongs (if any). Fortunately, there are only a handful of different patterns in which Web services typically interact and maintain conversations. Consequently, XL allows to specify the conversation pattern as part of the declaration of Web service. If such a pattern is specified, then the URI of the conversation is set implicitly whenever the Web service sends a message to another Web service. Currently, the conversation patterns supported by XL correspond one to one to the different kinds of scopes of transactions supported by J2EE [J2E]. These patterns are described in Table 1—for each pattern, two situations must be considered: (a) the ingoing message is not part of a conversation (defined as *none* in Table 1); (b) the ingoing message is part of a conversation (defined as *C1* in Table 1). As part of future work, we are going to assess these particular patterns and see whether they meet the requirements of typical Web applications.

For instance, the *Required* pattern has the following semantics: (a) if the Web service receives a message that has no conversation URI (i.e., is not part of a conversation), then the Web service will generate a new conversation URI and all other Web services it calls as part of processing the input message will be called using this new conversation URI. (b) If the Web service receives a message with a conversation URI, then all other Web services it calls as part of processing the input message will be called using the conversation URI of the input message.

The online broker is an example of a Web service that is based on the *Mandatory* pattern. Customers first invoke the *login* operation; after that, all other operations (e.g., *sell* and *buy* must be called as part of the same conversation. In that example, the context of the conversation is set; e.g., the session id variable which is declared in the CONTEXT LET clause of the Web service. This session id is used in the *sell* and *buy* operations in order to authenticate the user. An auctioning system as described in Appendix A is another example for a service that carries out conversations and keeps a state for each conversations.

Of course, a Web service can be involved in serveral conversations at the same time. Each conversation, however, follows the same pattern and has the same kind of context (the value of the context varies, of course).

[2]Such information is not yet taken into consideration by the current XML protocol but we expect that it will be added in near future.

Pattern	URI of Input Message	URI of Outgoing Messages
Required	none C1	C2 C1
RequiredNew	none C1	C2 C2
Mandatory	none C1	error C1
NotSupported	none C1	none none
Supports	none C1	none C1
Never	none C1	none error

Table 1: Conversation Patterns

CONVERSATIONTIMEOUT Finally, a timeout can be specified that terminates a conversation after a certain time since the last message exchanged as part of the conversation. An operation can be declared that is invoked if such a timeout takes effect. If a message is sent to a Web service after the time out, the Web service will assume that this message is part of a new conversation; in particular, the context of the (old) conversation is lost after the time out. For instance, if the time out of the online broker is set to ten minutes and a customer logs on and carries out no operations for ten minutes, then the user will have to log on again before buying or selling stock.

4.4 XL operations

Each Web service can perform multiple tasks, each described by an *operation*. As mentioned earlier, an operation is called every time a Web service receives a message. An operation, therefore, gets the content of a message as input, carries out a number of statements based on this input, and generates a message with the output. Consequently, unlike XQuery functions that can have multiple inputs and exactly one answer, XL operations have exactly one input and at most one output. Within every operation, two variables are defined implicitly: *$input* and *$output*. The *$input* variable is automatically bound with the content of the XML message sent to the operation. The value of the *$output* variable is computed in the implementation of the operation and automatically sent back as a message to the caller of the Web service. The execution of the operations can also throw exceptions which are sent back as XML messages to the caller or its agent.

In XL the specification of an operation is composed of the operation's declarative clauses and the operation body. The syntax of an operation specification is as follows:

```
operation <uri>::<name>
      <DECLARATIVE OPERATION CLAUSES>
      <OPERATION BODY>
endoperation
```

4.4.1 Declarative operation clauses

As for the whole Web service, the declarative clauses of an operation control the run-time behavior of the given operation. Some of the clauses are identical in syntax and semantics to those of the Web service and serve only to refine the global Web service behavior (e.g. HISTORY, CONVERSATION PATTERN, and ON ERROR INVOKE). We remark that the notion of an conversation timeout cannot be associated with a single operation. Other clauses like the PRECONDITIONS, POSTCONDITIONS, and NO SIDEEFFECT are specific only to operations, and we will describe them next. The syntax is as follows:

```
operation <uri>::<name>
    [ history; ]
    ([ precondition <booleanExpression> throw <expression>;])*
    ([ postcondition <booleanExpression> throw <expression>;])*
    [ on error invoke <operation>]
    [ no sideeffects; ]

    !! Operation Body
    .....
endoperation
```

PRECONDITION This is a condition that is checked before the first statement of the body of the operation is executed. If the condition fails (i.e., evaluates to the Boolean value FALSE), an exception is raised. The exception is specified in the THROW clause. Within the header of an operation, any number of preconditions can be defined. If there are several preconditions, these preconditions are evaluated in a random order.

Typically, preconditions will test certain properties of the *$input* variable; e.g., the existence of certain elements or the range of the value of certain elements. Preconditions, however, can also involve internal variables which are declared in the local declarations of the Web service (see Section 4.1). The precondition could also depend on some external conditions. The following precondition would specify that the *buy* operation can only be called while the status of the online broker is *open*:

```
precondition not( $status = OPEN)
       throw <error> Sorry, we are closed! </error>;
```

POSTCONDITION A postcondition is checked after the last statement of the operation has been executed. Typically, a postcondition will involve the *$output* variable but, again, any kind of Boolean expression can be used. If a postcondition fails, the given exception is raised. If more than one postcondition is defined, the postconditions are evaluated in a random order. If an exception is raised by a precondition, then no postcondition is evaluated. Likewise, postconditions are not evaluated, if an exception is raised within the body of the operation and the exception is not handled within the body of the operation.

An example for a postcondition is to validate the type of the *$output* variable before it is sent as a response to the caller of the *login* operation:

```
postcondition $output VALIDATES AS myns:customerinfo
   throw <error> Sorry, some internal error occured! </error> ;
```

NO SIDEEFFECTS This clause specifies that the operation has no sideeffects; i.e., the operation is an observer and does not change the internal state of the Web service or of any other Web service it might call. Operations that have no sideeffects can be invoked as part of expressions; otherwise, an operation cannot be invoked as part of an expression and it must be invoked as part of a statement. Using the syntax for service invocation described in Section 4.5.1 (see below), the following *if* statement that checks if SAP stock is lower than Euro 140 is only valid if the *getQuote* operation of *www.FServices.com* has no side effects. If not, the *getQuote* operation needs to be called in a separate statement and the result must be copied into a variable which may be used in the condition of the *if* statement.

```
if ( <symbol>SAP</symbol> -->
      HTTP://wwwFServices.com::getQuote < 140)  then do
  !!  buy SAP stock
endif
```

4.5 XL statements

XL extends the notion of XQuery expressions to statements. The body of an XL operation is described by such a statement. In addition to classic imperative statements like variable assignment, conditional statements, loops, error handling and return statements, XL supports some additional ones: some are XML specific (e.g. the update statements) and others are Web services specific (e.g. Web services invocation, logging, sleep). Finally, in addition to the classic imperative statement combinator (sequencing), XL contains other statement combinators borrowed from the workflow and dataflow theory (e.g. dataflow, parallelism, choice).

4.5.1 XL simple statements

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

Variable assignments The simplest statement is the assignment of a local variable. The syntax is as follows:

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

Local 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 block where the variable is defined (see Subsection 4.6). Expressions can be any expression defined by the W3C XQuery proposal [Que].

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

```
let $goodCustomer :=<person>
                     <name>Donald Trump</name>
                     <creditcard>
                       <number> 1234 4567 </number>
                       <expdate> 02/03 </expdate>
                       <type> Visa </type>
                     </creditcard>
                     <telephone>(408) 1234-567</telephone>
                   </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:

- *insert* in order to add new nodes to the XML hierarchy (e.g., an additional credit card element)
  `INSERT <creditcard>...</creditcard> INTO $goodCustomer`
- *delete* in order to delete nodes from the XML hierarchy (e.g., the Visa card)
  `DELETE $goodCustomer/creditcard[type=‘‘Visa’’]`
- *replace* in order to adjust elements (e.g., the telephone number)
  `REPLACE $goodCustomer/telephone WITH <telephone>(408) 8901-234</telephone>`
- *rename* in order to rename certain nodes (element or attributes)
  `RENAME $goodCustomer/name AS ‘‘fullname’’`
- *move* in order to move some XML nodes to a different location in the XML tree
  `MOVE $goodCustomer/telephone AFTER $goodcustomer/city`

The general syntax of the update statements is as follows:

insert `<expression>` (**into** | **before** | **after**) `<expression>`
delete `<expression>`
replace `<expression>` **with** `<expression>`
rename `<expression>` **as** `<expression>`
move `<expression>` (**into** | **before** | **after**) `<expression>`

Service Invocation Statements Probably the most relevant atomic statements in XL are those used for invoking other Web services; i.e., sending a message to another Web service. Often, the other Web service will be written in XL, but messages can be sent to any service that have a URI and respond to SOAP messages [Proa]. 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>[::<operation>]  [--> <variable>]
```

The semantics are straightforward. A message with the value of *expression* is sent to the Web service identified by *uri*. If a specific operation of that Web service should be called, then the name of the operation can also be specified. Otherwise, the default operation of the Web service is invoked. In a synchronous call, the execution is halted until the called Web service finishes its execution and returns the entire result (also wrapped in a 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 message is sent exactly once and in a best effort way. Quality of service guarantees and other specifications such as “at most once” or “at least once” which might become part of the XML Protocol recommendation [Prob] cannot be expressed in the current version of XL. We plan to extend XL accordingly once the XML Protocol recommendation has been completed.

As an example, consider the following synchronous service invocation that asks the online broker to buy 1000 SAP for at most Euro 140.00; the result is stored in the *$receipt* variable:

```
<order> <stock> SAP </stock>
        <limit> 140 </limit>
        <currency> Euro </currency>
        <amount> 1000 </amount>
</order>  -->
        HTTP://www.OnlineBroker.com::buy
--> $receipt
```

The syntax of an asynchronous call is similar to the synchronous one:

```
<expression> ==> <uri>[::<operation>] [==> <operation>]
```

In terms of the semantics: in this case the execution will not block and the program will immediately continue executing the next statement after the message to the called service has been sent. If the output (normal reply message or error) needs to be processed, then the name of the operation that will process the asynchronous result can be given as part of the call; this operation has to be a member of the Web service that originated the asynchronous call. Again, the message is sent exactly once and in a best effort way.

Currently, XL provides no syntax to set the envelop of a SOAP message explicitly. Such constructs could be useful in order to implement certain kinds of conversations and/or to implement distributed transactions and secure messages. We plan to extend XL in this way once SOAP and the emerging XML Protocol recommendation [Prob] have stabilized. Furthermore, XL provides no special constructs in order to implement multicasts. Again, we plan to extend XL in this direction as part of future work once XML Protocol has stabilized; it is conceivable, for instance, to extend the syntax such that variables with sequences of URIs can also be targets for service invocations.

Assertions Recall that it is possible to define preconditions and postconditions of XL operations (see Section 4.4). The more general concept is the concept of an assertion that can be executed at any point during the execution of an XL operation. Assertion statements are described using the following notation:

assert `<booleanExpression>` **throw** `<expression>`

If the Boolean expression evaluates to TRUE, the execution continues normally. Otherwise, an exception is raised. The second expression specifies the exception that is raised in this case; it can be any XQuery expression. Preconditions are special assertions which are carried out before the first statement of the body of an operation is executed. Likewise, postconditions are special assertions which are carried out after the last statement of the body of an operation has been executed. Assertions are only syntactic sugar and can easily be simulated by an ordinary IF statement.

4.5.2 Imperative statements in XL

Conditional statements Just like most other programming languages, XL provides an IF-THEN-ELSE statement in order to carry out conditions:

```
if ( <booleanExpression>) then do
    <statement>
endif
else do
    <statement>
endelse
```

The semantics are straightforward and the same as in other imperative programming languages.

Furthermore, XL supports the following SWITCH statement:

```
switch
  if (<booleanExpression>) then do <statement> end
  if (<booleanExpression>) then do <statement> end
  if (<booleanExpression>) then do <statement> end
   ...
 [ default do <statement> end]
endswitch
```

Typically, the Boolean expressions depend on variables such as *$input*. Again, the semantics are straightforward and the same as in programming languages such as C++ and Java. The Boolean expressions are checked from the top to the bottom until an expression evaluates to TRUE. The DEFAULT clause is optional.

Iteration statements XL supports three different kinds of loops: WHILE loops, DO-UNTIL loops, and FOR-LET-WHERE-DO loops, with the following syntax:

```
while (<booleanExpression>) do
   <statement>
endwhile
```

and

```
do
   <statement>
until (<booleanExpression>)
```

and

```
for <variable> in <expression>
let <variable> in <expression>
where <booleanExpression>
do   <statement>
```

The FOR-LET-WHERE-DO loop corresponds to FLWR expressions in XQuery [Que]. For instance, these loops are very useful if an application has a list of addresses and tries to send a message to each address.

Exception handling statements Web services implemented using XL signal failure by throwing exceptions - just as in Java or C++. The syntax of the XL statement that raises an exception is as follows:

```
throw <expression>
```

Here, expression can be any kind of XQuery expression. If the exception is not handled locally (see below), the execution of the operation terminates and the value of the expression (instead of the value of the *$output* variable) is returned as a message to the caller of the service. Just like variables and any other expression, the exceptions can be strongly typed optionally.

As an example, consider the following exception that signals that charging a credit card failed because the credit card has expired.

```
throw <error>
         <code>128</code>
         <description>Credit card expired</description>
      </error>
```

XL also adopts the Java syntax for catching exceptions. TRY is used to indicate a statement (or sequence of statements) in which an exception might be raised; CATCH is used to write code that reacts to exceptions. The syntax is as follows:

```
try
   <statement>
endtry
catch <variable> do
   <statement>
endcatch
```

The variable in the CATCH statement is bound to the value of the data carried by the exception that is raised while executing the statement(s) of the TRY statement. Like in Java, a caught exception will trigger the execution of the associated statement.

The current W3C standards (XQuery, XSLT, XML Protocol) do not yet have a clear common exception handling mechanism. On purpose, we under specified this feature of XL until the standards will clarify the same feature; later we will adjust our design of exception handling to the standard.

4.6 XL statement Combinators

Obviously, the body of an XL program can contain more than one atomic statement. There are several ways to combine statements. In the following “statement1” and “statement2” can refer to any atomic statement as the ones described in the previous sections or to 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 (or in parallel). If there is a cyclic dependency, then this combination of statements is illegal.

```
statement1  &  statement2
```

6. **Block.** As in C++ and Java, we use the following syntax to identify a block of statements. The body of an XL program, for instance, is formed as a block of statements. The scope of a variable is the block of statements in which the variable is used for the first time.

```
begin
    statement
end
```

4.7 Web Services specific statements

XL also provides a series of additional statements that are very helpful to implement Web services. We list them in the following.

Logging statement As mentioned in Section 4.1, there is an easy way to specify in XL that all calls to operations are logged in an automatic way - simply, the keyword HISTORY must be written in the declaration of a Web service. This way of automatic logging only involves calls to a Web service, the timestamp of the call, and the *$input* message sent by the caller. In order to write more information into a log, we propose the following syntax.

```
logpoint expr1 as name1, ... , exprN as nameN do
    statement
endlogpoint
```

As a result of the execution of this statement, the expressions 1 to N are evaluated and their values are inserted using the respective names into the *$history* variable. As an example, consider the following LOGPOINT statement that could be part of the *buy* operation of an online broker. The entries are written into the *$history* of the online broker and are placed just after the entries that are generated automatically for each call to the *buy* operation. The purpose of this log information is to trace the balance of the customer’s account and make sure that it is updated in the right way. The current balance is obtained by a lookup in the broker’s database using the customer’s id (given in the *$context* variable), the current quote of the ordered stock is obtained by calling a special financial service with the name of the ordered stock; the *getQuote* operation of that service must not have side effects (Section 4.4).

```
logpoint
  $customer[@id = $context/id]/balance as balance,
  $input  as order,
  $input/stock --> HTTP://wwwFServices.com::getQuote  as quote  do
   ...
endlogpoint
```

RETURN statement The RETURN statement terminates the execution of an XL operation and returns the current value of the *$output* variable.

HALT statement The HALT statement terminates the execution of an XL operation without returning any message to the caller. In the absence of a RETURN and HALT statement, the body of the XL operation will be executed and the content of the *$output* variable is returned after the last statement of the XL operation has been executed. (The *$output* variable is initialized to the empty sequence.)

NOTHING statement The NOTHING statement represents the empty statement useful in certain cases of workflow.

SLEEP statement The SLEEP statement stops the execution of an XL program for a certain duration. For instance, the following statement will stop the execution for 10 minutes:

```
sleep xf:duration(‘‘P10M’’)
```

The XQuery expression *xf:duration("P10M")* generates an XML value of type duration; in order to do this, it uses the function *xf:duration* defined in the XQuery built-in function and operation library.

WAIT ON EVENT and WAIT ON CHANGE statements Sometimes it is important to suspend the execution of a program until a certain event has happened. Examples for events are data updates, messages received, or certain points in time have been reached. For instance, the following statement will suspend the execution of an XL operation until the balance of the customer is more than 1000:

```
wait on event $customer/balance > $1000;
```

(If the balance raises never beyond 1000, then this XL operation will be stalled forever.)

Analogously, we propose a WAIT ON CHANGE statement that stops the execution of a program until the value of a variable has changed. For instance, the following statement will stop the execution until there is some change to the *$history* variable. This statement could be part of an operation that continuously monitors all the interactions of a Web service in order to, say, detect fraud.

```
wait on change $history;
 !! carry out fraud detection
 ...
```

Note that the following two statements are not equivalent, if the execution of a program should be halted until some given timestamp (xf:currentDateTime() is the XQuery/XPath function that returns the current Timestamp [F]):

```
 !! correct statement to wait for someTimestamp
wait on event
          xf:currentDateTime() = $someTimestamp

 !! incorrect statement to wait for someTimestamp
do
          nothing
until (xf:currentDateTime() = $someTimestamp)
```

The busy wait in this example does not work because there is no guarantee that the condition will be checked at every point in time.

RETRY statements A typical programming pattern in Web services is repetitive execution of statements until their successful completion, e.g. try to send a message until an acknowledgment is received. XL provides a convenient syntax to facilitate the programming of such patterns, as follows:

```
retry
    <statement>
[ maximum <intExpression> times [ throw <expression>]]
[ timeout after <durationExpression> [ throw <expression>]]
endretry
```

This statement will attempt the repetitive execution of the *statement* until the execution finishes without an exception, or at most a certain number of times (if a MAXIMUM clause is given) or for a maximum duration (if a TIMEOUT clause is given). For instance, the following statement will try to charge the visa credit card of a customer three times. Such a payment can fail temporarily for various reasons; e.g., if servers are overloaded. If all three times fail, however, it is assumed that there is something wrong with the credit card and an exception that indicates an illegal payment method is raised:

```
retry
     <payment>
        <info>  $customer/creditCardInfo  </info>
        <amount>  $order/volume  </amount>
      </payment> --> HTTP://www.visa.com::check
                 --> $receipt
  maximum  3 times
        throw <error> Illegal payment method </error>
endretry
```

5 Example: Publish & Subscribe in XL

In this section, we would like to demonstrate how simple it is to implement a publish & subscribe service using XL. This service has obviously two main operations: publish and subscribe. The *subscribe* operation takes an XQuery Boolean expression as input and the URI of a service that should be invoked whenever a document is published that matches the XQuery Boolean expression. The *subscribe* operation stores such subscriptions in the *$subs* variable so that the service can consider all subscriptions whenever new data is published. Consequently, the *publish* operation takes an XML document as input and simply iterates through all subscriptions: if the document matches to one of the subscriptions, the document is sent to the operation that is registered as part of the subscription. In order to find out which subscriptions match, the *publish* operation makes use of a built-in *eval()* function. *eval()* is not yet defined as part of the standard XQuery/XPath function library [F], but we expect this to happen; if not, we propose to define the *eval()* function as part of the XL specification.

This example is fairly simple and it does not involve any conversations or complex contexts as are necessary for more sophisticated workflows. We present a more complex example, an auctioning system, that demonstrates how workflows can be implemented in XL in Appendix A.

```
service HTTP://www.matchmaker.com
    namespace xf = "http://www.w3.org/2001/08/xquery-operators"

    !! Web service internal data - a subscription is a pair:
    !! (1) XQuery Boolean expression with £input as a free variable
    !! (2) URI of service to call if a match is found (no sideeffects!)
    let Subscription subs;

    !! Default operation is publish
    defaultoperation publish;

    !! Conversation Pattern:  no conversations are necessary
    conversationpattern NEVER

!! insert a subscription into the database
operation subscribe
  body
     !! check whether the £input has the right format
    precondition $input validates as Subscription;

    insert $input into $subs;
  endbody
endoperation

!! publish - this can be any kind of data
operation publish
  body
    for $q in $subs
    where eval($q/query)
    do $input --> $q/call
   endbody
endoperation

operation init
  body
      nothing
  endbody
endoperation

operation  close
   body
      nothing
   endbody
endoperation

endservice
```

6 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 Bulletin on e-services [deb01]. The main purpose of our work was to provide a clean basis for a new XML programming language for rapid development of Web services. Such a language will obviously not be built from scratch but using the knowledge and technology advancements accumulated in the last 40 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—most prominently, SUN's J2EE [J2E] and Microsoft's .NET initiative [.NE]. 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] and XLANG [Tha]. While there are many similarities between WSFL and Xlang on one hand and XL on the other hand, there are a couple of major differences. First, both WSFL and Xlang are XML programming languages in the sense that they have an XML *syntax*; our understanding of an XML programming language is a language that *manipulates* only XML data, independently of the syntax of the language itself. Second, both WSFL and Xlang are able to describe only how to *compose* existing Web service components (that are expected to be implemented using other languages), while XL is *complete* not only with respect to Web service composition but also *specification*. Using XL, it should be possible to implement complex Web services entirely *without* any need for any other programming language. Finally, and more importantly, XL adds to Web services the concepts of declarative behavior specification inherited first from relational databases and then from J2EE.

Another specification worth mentioning is the Business Process Model Initiative whose goal is to implement cross-organization processes and workflows on the Internet [BPM]. Moreover, 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]. Furthermore, the J2EE framework provides a number of features for service composition, conversations, database interaction, transactions, and security [J2E]. Recently, Sun Microsystems introduced the JXTA project on peer-to-peer computing to support distributed computing on the Internet [JXT]. SAP has recently announced an own effort to implement a Web services platform [Tec]. Finally, the notion of a service composition is based on a solid theoretical background consisting on the calculi 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.

7 Summary

In this paper we sketched the requirements and presented an initial design of an XML programming language whose 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 marshaling 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 syntax and semantics for transactions, support for more comfortable conversations, constructs to explicitly set the SOAP envelop of a message, implementation of multi-casts, and security aspects (e.g., authorization and encryption). 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. One particular issue is to find out what kind of meta-data is necessary in order to describe a Web service and make these optimizations possible.

The language we propose here is currently being implemented; we hope to have a demo available on the Web in spring 2002.

A Example: An Auctioning System in XL

In this appendix, we show how an auctioning system can be implemented using XL. The auctioning system has several operations. Bidders that would like to participate in future auctions can register using the *registerBidder* operation—after registration, they are automatically informed whenever a new auction starts (see the implementation of the *startAuction* operation).

The *startAuction* operation begins a new auction. As input the *startAuction* operation takes an opening bid; i.e., it sets a minimum prize. Furthermore, a description of the product needs to be given as part of the input to the *startAuction* operation.

An auction is then carried out as a conversation in which the auctioning system and potentially all bidders participate. Several such conversations (i.e., auctions) can be carried out concurrently; in particular, each bidder can simultaneously participate in several such conversations. In order to give a new bid, bidders invoke the *takeBid* operation of the auctioning system. If the bidders are also implemented in XL, bids are automatically associated to the correct auction if bidders *reply* to the invitation of the auctioning system because the bid is associated automatically to the right auction. When a new bid is entered, the other participants of the auction are automatically informed so that they can react. (Bidders that have registered to the system, but do not participate in an auction are not informed.)

An auction terminates after a period of 10 hours with no new bids. This period is specified in the CONVERSATIONTIMEOUT clause. The end of an auction is implemented by the *closeAuction* operation which sets the status of the auction to be closed and takes other actions such as informing the winner. The *closeAuction* operation is called implicitly if the auction is timed out, but it can also be called explicitly by another service. Of course, not everybody should be authorized to call this operation; extending XL for authorization is part of future work so that such restrictions are not implemented here.

```
service HTTP://www.auction.com

    namespace xf =
              "http://www.w3.org/2001/08/xquery-operators"

     !! Web service internal data
    let bidder;
    let id;
    context let auction

     !! entire Web service activity is monitored
    history;
```

```
     !! default operation is unknownOP
    defaultoperation unknownOP;

     !! all messages carry the incoming conversation URI
    conversationpattern required;

     !! a conversation cannot last more than 10 hours
    conversationtimeout xf:duration(‘‘P10H’’) closeAuction;

operation registerBidder
        !! new bidder has a URI so that he/she can be informed
       precondition exists($input/uri);

        !! new bidder is not in the database
       precondition empty($bidder[@uri = $input/uri]);

        !! new bidder is in the database
       postcondition exists($bidder[@uri = $input/uri]);

  body
      !! Check credit history of bidder
      ...

      !! Enter bidder into database
     insert $input into $bidder;

      !! Set output
     let $output : = <message> Thank you for registering
                               for "auction.com" </message>
  endbody
endoperation

operation  startAuction
        !! check if input is valid
       precondition $input validates as auctionInfo;

       body
           !! Increment the current id counter with one
          replace $id with <id> $id + 1 </id>;

           !! Set the contextual information
           !! specific to this auction
          let $auction := <auction>
                            <id>  $id  </id>
                            <status> OPEN </status>
                            <maxbid>
```

```
                          <bidder> nobody </bidder>
                          <amount> $input/minbid </amount>
                        </maxbid>
                        <productinfo>
                           $input/product
                        </productinfo>
                      </auction>;
     !! Inform all potential bidders of new auction
     <message>
        <text> We had a new bid for your item ! </text>
     </message> ==> $bidder/url
   endbody
endoperation

operation   takeBid

      !! bid has an amount
     precondition  $input validates as correctBid;
      !! auction is open
     precondition $auction/status = open;


      !! maxbid is at least as high as bid
     postcondition $auction/maxbid/amount >= $input/amount;

   body
     !! register new participant of the auction
     !! all participants are informed of new bids
    if ($auction/bidder <> $input/uri)
       insert $input/uri into $auction/bidder
    endif;

     !! Check bid:  Is it high enough?
    if ($auction/maxbid/amount >= $input/bid)
       throw <error> Sorry, but your bid
                     is not high enough. </error>
    endif

    &

     !! Check status of auction: Is it still open
    if ($auction/status = CLOSED)
      throw <error> Sorry, but the auction is closed. </error>
    endif

     ;
```

```
      !! Register new bid
     replace  $auction/maxbid with  <maxbid>
                                      <bidder>$input/uri</bidder>
                                      <amount>$input/amount</amount>
                                    </maxbid>;

     !! Inform everybody (except initiator) of the new bid  -- multicast!!!
     !! Do not wait for answers - bidders will answer by making new bids
     for    $q IN $auction/bidders
     where  $q <> $input/uri
     do     <newbid>
               <auctionId>$auction/id</auctionId>
               <amount>$input/bid</amount>
             </newbid>         ==>   $q;

      !! Set output confirm bid
     let $output := <confirmation>
                        <auctionId>$auction/id</auctionId>
                        <amount>$input/bid</amount>
                     </confirmation>
  endbody
endoperation

operation   closeAuction
   body
      !! close the auction
     replace  $auction/status with  <status> closed </status>;

      !! inform winner etc.
      ...
   endbody
endoperation

operation   unknownOp
   body
      throw  <error>The operation you request is not
                   available on this Web service ! </error>
   endbody
endoperation

operation init
   body
        nothing
   endbody
endoperation
```

```
operation close
    body
        nothing
    endbody
endoperation
endservice
```

References

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

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

[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.

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

[F] XQuery 1.0, XPath 2.0 Functions, and Operations Version 1.0. http://www.w3.org/tr/xquery-operators/.

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

[J2E] J2EE. Java 2 enterprise edition. http://java.sun.com/j2ee/tutorial.

[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.

[M] XQuery 1.0 and XPath 2.0 Data Model. http://www.w3.org/tr/query-datamodel/.

[.NE] .NET. http://www.microsoft.com/net.

[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.

[sem] XQuery 1.0 Formal semantics. http://www.w3.org/tr/query-semantics/.

[Tec] SAP Technology. http://www.sap.com/company/publications/fs_technology.asp?pressid=706.

[Tha] S. Thatte. Xlang overview. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm.

[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.

[(XP] XML Path Language (XPath). http://www.w3.org/markup/forms/.

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

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