

XL: An XML Programming Language for Web Service Specification and Composition

Daniela Florescu

XQRL, Inc., dana@xqrl.com

Andreas Grünhagen

TU München, gruenhag@in.tum.de

Donald Kossmann

XQRL, Inc., TU München, kossmann@in.tum.de

Abstract

We present an XML programming language designed for the implementation of Web services. XL is portable and fully compliant with 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.) should 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.

Key words: XML, Web Service, Programming Language

1 Introduction

XML is the lingua franca for data exchange on the Internet. Among its many possible uses, XML-based languages are 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.

Since XML based services are provided via Internet, the term “*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 Protocol [26] or HTTP. Web services can, but they are not required to, preserve an internal state. In addition, Web services can participate in complex *conversations*. A conversation is an exchange of correlated messages among a certain number of participant Web services.

1.1 Deficiencies of Web Service Implementations

As a result most Web services are built using classic programming languages, such as Java or Visual Basic, and some kind of SQL-based RDBMS (e.g., Oracle or DB2), a mixture of paradigms that inherently implies a number of logically irrelevant but costly and error prone intermediate data-transformations. 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 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 [9].

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. For example, low level protocols manipulation and performance improvements are often mixed with data validation and real application logic. Furthermore, a WSDL interface must be constructed manually if a language like Java is used because such interface descriptions cannot be extracted automatically.

Furthermore, a WSDL interface must be constructed manually if a language like Java is used because such interface descriptions cannot be extracted automatically.

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 based on XML. Moreover, the language should be, as much as possible, *declarative* and *high level*. Tuning and optimisation should be carried out automatically, rather than hand-coded. Furthermore, the interface description (i.e., WSDL) should be generated and published automatically using such a language.

1.3 Requirements for a Web Service Programming Language

In the following, a list of more specific requirements that drove the design of XL is given. A comprehensive list is given in [14].

- (1) **High-level programming and service composition** 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. Moreover XL must allow the construction of high-level services out of the composition of more basic services which are not necessarily written in XL.
- (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) **Unique XML-based data model and type system** The data manipulated must be modeled by the standard XML data model and type system [23]. No other data models and type systems are allowed.
- (4) **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.
- (5) **Compliance with the W3C standards.** XL must be compliant with the XML W3C standards such as XML Schema [25], XQuery [23], XPath [7], XSLT [34], XML Forms [32], XML Protocol [31], and WSDL [6].

The following sections we will try to build up our picture of Web services. Section 2 gives a brief overview of related Web and XML standards. Section 3 explains how Web services are specified in XL. Section 4 gives a brief overview of the basic statements of XL. Section 5 illustrates XL Web services by giving a short example. Section 6 sketches the architecture of our implementation. Sections 7 summarises related work. Section 8 concludes this paper and shows avenues for future work.

2 Overview of Relevant Web Technology

2.1 Data Model

The W3C XML data model describes, in an E/R fashion [5], 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 a general mathematical structures, i.e., as *ordered trees of nodes* [25]. 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 and text nodes). 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).

2.2 XML expressions and XML queries

A complementary W3C standard, XQuery, deals with XML expressions and XML queries [23]. XQuery is a functional language. Like all functional languages, XQuery expressions are constructed using first order and second order function. 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*. Additionally XQuery includes FLWOR expressions which contain variable bindings (FOR- and LET clause), predicates (WHERE clause) and result construction (ORDER BY and RETURN clauses). FLWOR expressions are the XQuery analogon to SQL's SELECT statement. 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 XPath standard [7].

XML queries [23] 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.

2.3 XML Protocol (SOAP)

In the context of Web Services, SOAP and the XML-Protocol [31] provide a common XML-based message format. A simple but extensible messaging schema for inter-process communication is defined. Interface definitions and communication links should be easy to setup and to maintain or evolve. Possible enhancements as for example authentication or session management are foreseen and can be integrated into XML Protocol.

2.4 *Web Service Description Language (WSDL)*

As mentioned previously, interface description is a crucial part of deploying a Web service. The WSDL working draft [6] describes the message types, the service interface itself including the message exchange patterns to be used, and the transport-protocol bindings. The message exchange patterns include for example synchronous and asynchronous operation calls.

3 **Web services in XL**

3.1 *Design Philosophy*

The language XL is designed to provide the missing link between the various XML standards set by the W3C. By combining the type system, expressions and the service interfaces into a single language, XL takes advantage of existing standards without abandoning possible optimisations. In other words, XL extends the existing W3C standards in order to obtain a complete and uniform programming model for Web services.

3.2 *Web service declarations*

A Web service in XL generalises the notion of an XQuery entity. In addition, a Web service specification in XL is identified by a URI. XL Web services can contain a set of local function declarations plus a set of type definitions and schema and namespace imports. In addition, a Web service specification in XL can contain: (a) local data declarations, (b) declarative clauses, and (c) specifications of the Web service operations.

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>  
  < FUNCTIONS & LOCAL OPERATIONS >  
  < LOCAL DECLARATIONS >  
  < DECLARATIVE WEB SERVICE CLAUSES >  
  < PUBLIC OPERATION SPECIFICATIONS >  
endservice
```

Functions are defined in XL in exactly the same way as in XQuery. In the following subsections we will describe variable declarations, declarative Web service clauses, and public operation specifications which are specific to XL. The difference between local and public operations is that local operations can

only be invoked internally; i.e., only public operations are part of the WSDL description generated for an XL Web service.

3.2.1 Variable 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 variables represents the internal state of the whole Web service. These variables are instantiated once when the Web service is installed and persist the whole life time of the Web service. The scope of these variables is the whole Web service. An example is the warehouse database of an online shop.

The second kind of variables represents the internal state of a particular conversation that the Web service is involved in. Examples are the session id when a user logs into the system or the maximum bid for an item in an auctioning system. These variables are instantiated when the Web service joins a new conversation; in other words, when the Web service receives the first message with a specific conversation URI. 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.

This kind of variables can be used in the body of all operations of the Web service that participate in conversations; i.e., all operations that are able to receive messages that carry the URI of a conversation. The life time of such variables is bound by the life time of the conversation. Since the Web service can be involved in several conversations at the same time, multiple instances of such variables can exist at the same time; one instance of each variable for each conversation. In some sense, the set of all instances of these variables can be thought of as an array that is indexed by the URIs of conversations. In the *buy* operation of an online shop, for instance, a *session id* variable will be used in order to determine which customer invoked the *buy* operation; the right value (i.e., instance) of this variable will automatically be set using the conversation URI of the message sent from the customer to the online shop. 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 service
  ( context let [<type><variablename> [:= <expression> ] ; ) *

  !! declarative Web services clauses and operations ...
endservice
```

In this syntax, the “type” is an optional type constraining the type of the variable’s value, while “expression” is an XQuery expression describing the initial value of the variable. If no “expression” is given, then the variable is initialised to the empty sequence; if no “type” is given then the variable can be bound to any valid instance of the XML data model.

3.2.2 Declarative Web service clauses

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. In the following, we will briefly describe the individual clauses which are part of the Web service header declaration. The meaning of the individual clauses will become clearer in the discussions and the example given in Section 5. A more detailed description and a syntax definition is provided in [14].

HISTORY If this clause is specified, 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, and the timestamp when the operation was called.

DEFAULT- & UNKNOWNOPERATION These clauses declare the Web services behaviour in cases when a message is sent to the Web 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.

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 user-defined 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.

ON CHANGE In this clause, a simplified form of triggers can be specified. The semantics are straightforward: if the value of a certain variable changes, the specified operation is called with an empty input. Changes to any variable declared in the Web services local declarations can be monitored in this way.

ON EVENT This clause allows to declare more elaborate triggers and periodic tasks. Whenever a given boolean expression evaluates to *true*, a specified operation is invoked. If an INPUT is specified, the corresponding expression is evaluated and passed to the operation as input.

ON ERROR INVOKE This optional clause specifies an operation that is called whenever an 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.

CONVERSATIONPATTERN This clause specifies in a declarative manner how the Web service interacts with other services as part of conversations. There are many alternative models conceivable how to implement business conversations. As mentioned earlier, in our model we assume that the SOAP messages which are exchanged between Web services can carry a conversation URI in their envelop. Using this model, it would be 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 a 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 [18]. 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 the second column of Table 1); (b) the ingoing message is part of a conversation (defined as *C1* in the second column of 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 (*C2* in table 1). (b) If the Web service receives a message with a conversation URI, then all other Web services it calls

| <i>Pattern</i> | URI of | |
|----------------|---------------|------------------|
| | Input Message | Outgoing Message |
| 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

as part of processing the input message will be called using the conversation URI of the input message. Each single operation can overwrite this default pattern by specifying its own pattern.

The online shop 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., *buy*) must be called as part of the same conversation.

As mentioned earlier, a Web service can be involved in several conversations at the same time. For each conversation, the Web service maintains a separate context; i.e., a separate set of instances of each variable declared in a CONTEXT LET clause (see previous section). These variables can only be used if the ingoing message carries a conversation URI. Naturally, thus, such variables cannot be used if the conversation pattern is set to *Never*.

CONVERSATIONTIMEOUT 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. As part of the termination process, all the Web service's resources allocated to the conversation (e.g., context variables) are released. If a message is sent to the 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. An explicit termination of a conversation is thought about, but it is not included yet. We plan to add an explicit mechanism to terminate conversations as part of future work.

Terminating a conversation is a local event. In other words, other Web services can communicate in a conversation, even though one Web service has decided to bail out of the conversation

3.3 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 result in errors which are sent back as XML messages to the caller.

In XL the specification of an operation is composed of the operation's declarative clauses and the operation body. As for the whole Web service, the declarative clauses of an operation control the run-time behaviour 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 behaviour (HISTORY, CONVERSATION PATTERN and ON ERROR INVOKE). Other clauses like the PRECONDITIONS, POSTCONDITIONS, and NO SIDE EFFECT are specific to operations, and we will describe them next. As in the previous section the syntax definition is skipped here, but can be looked up in [14]. An example is given in Section 5.

PRECONDITION For each operation a set of preconditions can be specified which are checked before the first statement of the body of the operation is executed. If one of the conditions fails (i.e., evaluates to the boolean value FALSE), an exception is raised. This exception is specified in a THROW clause. 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.

POSTCONDITION Likewise a set of postconditions can be specified. Each condition 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 or in the body of the operation, no postcondition is evaluated.

NO SIDE EFFECTS This clause specifies that the operation has no side-effects; 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. Specifically, this clause makes it possible to invoke a Web service as part of the evaluation of an expression.

4 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). In the following sections the most important statements will be defined. A complete list is given in [16].

4.1 Basic statements

In this section we introduce some of the basic atomic statements that can be used in the body of an XL program.

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 4.2). Expressions can be any expression defined by the W3C XQuery proposal [23].

Update Statements Unfortunately, the W3C has not yet released a recommendation to manipulate XML data; there is only an initial proposal [10]. Once a recommendation has been released by the W3C, XL is going to adopt the syntax and semantics of that recommendation. 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 $customer
```

- *delete* in order to delete nodes from the XML hierarchy (e.g., the Visa card)

- delete** \$customer/creditcard [type=" Visa "]
- *replace* in order to adjust elements (e.g., the telephone number)

replace \$customer/phone **with** <phone>(408)8901-23</phone>
- *rename* in order to rename certain nodes (element or attributes)

rename \$customer/name **as** " fullname"
- *move* in order to move some XML nodes to a different location in the XML tree, while still preserving the internal structure and the node identifiers.

move \$customer/telephone **after** \$customer/city

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 has a URI and responds to SOAP messages [26]. 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.

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.

Conditional statements Just like most other programming languages, XL provides conditional statement like IF-THEN-ELSE or SWITCH.

```

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

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

```

The semantics are again straightforward. Note that the switch statement is more expressive than the switch statement in Java. The boolean expressions are checked from the top to the bottom until an expression evaluates to TRUE. At most one statement is executed — after that the *switch* statement terminates without considering any other boolean expression.

A more expressive switch statement is necessary in order to implement personalized Web services; i.e., Web services that react to messages from different kinds of users. In order to achieve good performance, an XL implementation should carry out optimisations depending on the boolean expressions.

Iteration statements XL supports three different kinds of loops: WHILE loops, DO-WHILE (not shown) loops, and FOR-LET-WHERE-DO loops, with the following syntax:

```

while ( <booleanExpression> ) for <variable> in <expression>
do
    <statement>
endwhile
    let <variable> in <expression>
    where <booleanExpression>
    do    <statement>

```

For WHILE loops, DO-WHILE the semantics are just the same as in Java. The FOR-LET-WHERE-DO loop corresponds to FLWOR expressions in XQuery [23]. 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, 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. In order to handle exceptions locally, XL adopts the try-catch syntax from Java.

Block We use the following syntax to identify a block of statements.

```
begin
  <statement>
end
```

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.

Several additional statements (e.g., log points or a sleep statement) are omitted at this place due to a lack of space. These statements are not just syntactic sugar. But as we are simply trying to give an impression what XL is about these statement can be skipped without too much loss. In [14] a full language description and several examples are provided.

4.2 *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. Compaq’s Web language WebL [30] uses similiar combinators.

- *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>
```

- *Failure.* If “statement1” fails, execute “statement2.”

```
<statement1> ? <statement2>
```

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

```
<statement1> | <statement2>
```

- *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>
```

- *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>
```

5 Example

The example below shows a Web service written in XL. This program could be part of the implementation of an online shop. Only the two operations *login* and *buy* are listed to give an impression of an XL Web Service definition.

This example demonstrates how conversation variables are implicitly set. In this case a client Web Service starts a conversation by calling the method *login*. Every time a new conversation is started a new instance of the context variable *\$customer* is created and initialised by default. In every further operation call in the conversation the same content is bound to the variable *\$customer*. The term *conversationpattern mandatory*; in the service header defines that for example the operation *buy* can only be called if a conversation has been started already by calling *login*. The operation *login* itself has the conversation pattern *requiredNew* and starts therefore a new conversation in any case.

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

  !! Web service internal data
  let $warehouseDB := <warehouseDB />;
  let $customerDB := <customerDB />
  context let $customer := <customer />;

  !! entire Web service activity is monitored
  history ;

  !! default operation is unknownOP
  defaultoperation unknownOP;

  conversationpattern mandatory;

  !! a conversation cannot last more than 10 days
  conversationtimeout xf:duration("P10D") logout;

  !! if an error occurs, logout is called
  on error invoke logout;

  operation login

  !! input variable is valid
  precondition $input validates as customerLoginSchema;

  !! customer needs a valid login
  precondition xf:exists(
    for $c in $input/customer,
    $db in $customerDB/customer
    where $c/id eq $db/id
```

```

        and $c/passwd eq $db/passwd
        return $c);

    !! a new conervation is started
    conversationpattern requiredNew;

body
    !! initialise the conversation variable
    let $customer := <customer>
                    <id> $input/customer/id </id>
                    <orders />
                    </customer>;

    !! return value
    let $output := 'Login successful'
endbody
endoperation

operation buy

    !! input is valid
    precondition $input validates as buySchema;

    !! check availability of product
    precondition for $p in $warehouseDB/products,
                    $o in $input/order
                    where $p/id eq $o/id
                    return $p/avialable gt $o/quantity;

body
    !! update the conversation variable
    insert $input/order into $customer/orders;

endbody
endoperation

operation unknownOP
    body
        nothing
    endbody
endoperation

operation logout
    body
        nothing
    endbody
endoperation
endservice

```

6 Implementation

In this section, we describe our prototype implementation of XL. The design of our prototype was driven by the following goals:

- **Distribution** The declarative nature of the XL-language offers various optimisation approaches. Among other things it should be possible to distribute the execution of a Web service over different computers.
- **Platform independence** In order to utilise existing resources it should be possible to execute XL-Web services on different platforms and operating systems.
- **Expressions** As the semantics of expressions (or any r-value) in the XL-language is defined according to the XML-Query definition, an XML-Query engine has to be integrated into the XL-Runtime-system.
- **Optimization** Provide the basis for powerful and automatic tuning and optimization. Optimization techniques for an XL implementation are described in [15].

The implementation of XL is in various ways different from other programming languages like Java:

- the virtual machine exploits implicit or explicit given parallelism. Synchronisation points are set automatically.
- the XL virtual machine handles the context of conversations.
- the XL virtual machine handles variables which do not fit in main memory (e.g., the customer database of an online shop).
- the XL virtual machine handles persistence and supports transactions.

6.1 Statement Graph

Each operation in an XL-Web service is translated into a directed and attributed graph which is the basis for all further considerations. The XL-syntax is mapped on to a set of basic statements. These basic statements reflect the command set of an interpreter and nodes in the statement graph. In the following, we describe the structure of the graph and the set of basic statements of our XL implementation

6.1.1 Parsing

Each statement is represented by a node in the statement graph. These nodes are connected by directed edges so called *combinators*. Each combinator is attributed by a (boolean) variable. If a statement has been executed each outgoing combinator is examined. If the associated variable evaluates to TRUE the succeeding statement has to be executed as well. Combinators can be attributed also with the negation of a variable; in this case, the next statement is executed if the value of the variable is FALSE. If the combinator is not attributed with a variable, the next statement is always executed. Each combinator therefore define a one to one predecessor-successor relationship

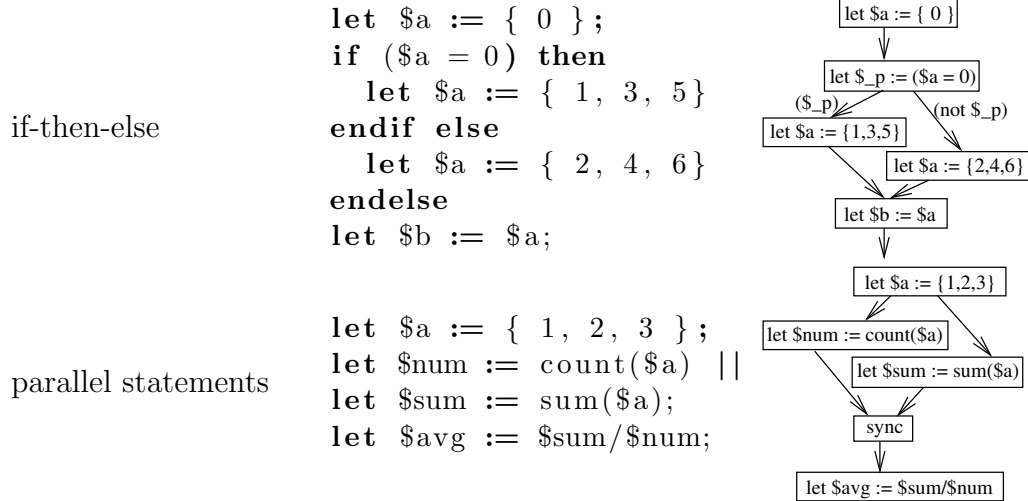


Fig. 1. Two example XL programs with the corresponding statement graphs

between single statements.

The statement graph is a very expressive representation of a program. A usual imperative if-clause can be expressed by an assignment which evaluates the if-predicate and two outgoing combinator, one carrying a temporary variable containing the predicate result and the other one its negation.

If two outgoing combinators carry the same boolean value both succeeding statements have to be executed. The statement graph does not imply any ordering between succeeding statements.

Figure 1 shows two short XL programs together with the corresponding statement graphs.

6.1.2 Basic statements

Assignment Probably, the most frequent statement is the assignment. As in any other language the assignment contains an r-value (or expression) which can be evaluated and an l-value (or variable) which stores the result.

Sending a message Sending a message requires two expressions, the content of the message itself and the destination of the message. The second expression provides a value which can be interpreted as an address according to XL. This can be either a URI or a URI with an operation name.

Receiving a message The receive-statement waits for a certain message to come. Typically this message is a response to a message sent before.

The correlation between sending a message and the associated return message is either implemented by adding a message-ID or by keeping an opened HTTP connection while waiting for an answer.

Wait statement The wait statement contains an expression and waits until this expression evaluates to TRUE. It is up to the interpreter to determine when to evaluate the expression.

Synchronise statement As the statement graph definition allows implicit

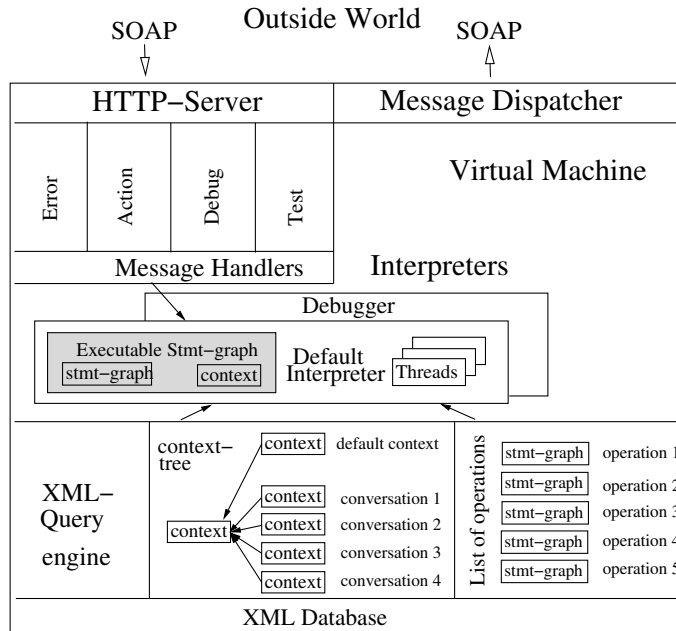


Fig. 2. The architecture of the XL Runtime-System

parallelism the different execution paths have to be synchronised. Even so exploiting the parallelism is left to the interpreter in the virtual machine the synchronisation is expressed in the statement graph. The synchronise statement schedules succeeding statements only if itself has been executed as often as there are direct predecessor statements.

6.1.3 Auxiliary Statement

Additionally to these basic statement some auxiliary statements are needed. As the initially parsed program is translated into a graph some information gets lost. To avoid information losses, these auxiliary statements are introduced.

Begin- and EndBlock (Variable Scope) So far the variable scope is not expressed in the statement graph. At runtime the virtual machine provides each statement with a context which contains among other things a set of variables which can be accessed by the statement. By adding Begin- and End-Block statements to the graph, this information remains part of the statement graph.

Update Statements Supplementary to the assignment we implemented a set of XML Update statements which do not add further expressiveness. In return altering XML content is much more convenient. These statements could be either mapped on to a JDBC Interface or implemented by natively in the XL system itself.

6.2 Runtime-System Architecture

Figure 2 shows the basic structure of the XL virtual machine. The XL communicates only by sending XML SOAP messages via HTTP. Each SOAP message can indicate its message type. Currently the XL prototype supports two different message types:

- operation call
- debug messages

Depending on this message type different message handlers are selected which continue processing the message. If the message type *debug* is specified, a special debugging interpreter is used which provides an interface for a remote debugging client. If no message type is specified a usual operation call is executed.

In case of an operation call the message handler initiates an appropriate interpreter. This interpreter initialises the input variable *\$input*, starts the execution, waits for the execution to finish and returns the result – either an error message or the content of the variable *\$output*.

The context information is contained in so called *control blocks*. These control blocks are nested inside each other in a tree-structure. Each control block contains a set of variables. At execution-time each statement holds a reference to one control block. As the statement is executed it can access all variables which are contained in its control block or any parent control block. The root control block therefore holds all global variables.

The auxiliary statements *Begin-* and *EndBlock* in the statement graph indicate where a new control block has to be used or an old one can be disposed.

6.2.1 Conversation-Handling

The basic feature of conversation handling is the context information which is associated with an existing conversation. Each SOAP message which is sent as part of a conversation has to contain a conversation-URI. The XL runtime-system has to map these URIs on to existing context representation (control-blocks). If an operation call as part of an existing conversation occurs the associated control-block is used as parent control block for this execution. By doing so the conversation variables contained in this control-block are joined by all operation calls in a conversation in one runtime-system.

If the service is not part of the conversation and the adequate conversation pattern is specified a new context has to be created and added to the URI-context map.

6.2.2 Interpreter

Each specific execution of XL-operations is controlled by a so called interpreter object. Each time an operation is to be executed the interpreter fetches the associated statement graph for this operation. The Interpreter creates a new context representation which inherits the global and conversation variables

| Dimension | Description | current Implementation |
|------------------|---|---------------------------------|
| Data granularity | The dataflow between statements can be implemented as whole variables or smaller entities. | whole variables |
| Execution Unit | Each interpreter might use a different internal representation of an XL program. The term <i>execution unit</i> denotes the smallest part of the XL program which can be executed by the specific interpreter without being interrupted. | single statements |
| Execution Point | In a distributed environment a statement or an operation can possibly be executed on different servers. | every thing is executed locally |
| Decision Time | It is up to the virtual machine to realize explicit or implicit parallelism. The decision which parts are to be executed concurrently can be made at compile or runtime. Likewise the decision on the execution point can be deferred until runtime | at runtime |

Table 2

Dimensions of the execution model

from a selected parent context. Then a dynamic binding between statement graph and context representation is generated.

The execution of statements is specific for our implementation of the XL-Interpreter. In our case the context binding is represented by so called exec-statements. Each of these exec-statements references a context and an associated statement. These exec-statements are scheduled on to a set of existing threads.

The interpreter fetches the initial exec-statement, appends it to a waiting queue and notifies the threads. One of the waiting threads fetches the exec-statement from the waiting queue and executes it.

Generally speaking execution models for XL can vary in different dimensions. The different dimensions we distinguish are shown in table 2.

By implementing different interpreters which vary in one or more dimension we can instrument different optimisation strategies.

6.3 XML Query Expressions

XML Query expressions are executed by an external query engine for XQuery. In fact the two systems are relatively loosely coupled as XL treats expressions as simple character-strings. These strings are passed the interface of the query engine which is closely related to the common JDBC interface for relational databases. Instead of parsing XQuery expressions XL creates XQuery-preparedStatements. To execute these statements XL has to provide a the context information namely the variables. In return the XQuery-statements

return the XML-result represented as a token-stream.

The loose coupling between the two systems certainly costs CPU-performance and optimisation possibilities are lost. On the other hand the XQuery engine can eventually be easily replaced by another implementation. The currently used XQuery engine is implemented by XQRL [33].

7 Related Work

The development and composition of Web services (or e-services) is currently a very active area in both industry and academic research. The main purpose of our work was to provide a clean basis for a new XML programming language for rapid development of Web services.

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 [18] and SunOne [27], and Microsoft’s .NET initiative [24]. Compaq has developed the WebL language [30]; HP has developed the eFlow and eSpeak systems [4, 11], IBM is working on a language called Web Service Flow Language [22], and Microsoft has recently released their BizTalk Server 2000 [1] and XLANG [28]. 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 behaviour specification inherited first from relational databases and then from J2EE.

The recently started language projects (Water [29], JWIG [20]) provide efficient methods for Web service specification, but still both languages do not solve the mismatches between XML, Java and databases.

Two other specifications worth mentioning are the Business Process Model Initiative whose goal is to implement cross-organisation processes and workflows on the Internet [2] and DAML-S whose goal is the automation of Web services using ontologies [8]. DAML-S accomplishes an automatic selection, composition, and interoperation of Web services. Like XL, DAML-S provides control statements to specify application logic. 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 [19]. Furthermore, the J2EE framework provides a number of features for service composition,

conversations, database interaction, transactions, and security [18]. Recently, Sun Microsystems introduced the JXTA project on peer-to-peer computing to support distributed computing on the Internet [21].

Finally, the notion of a service composition is based on a solid theoretical background consisting on the calculi developed first by Hoare [17] and more recently by Cardelli [3].

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.

Two recent and independent projects have a similar design philosophy and encourage us to pursue XL. BPEL [12] is very similar to XL. Like XL, BPEL declarative Web service description based on several XML standards. The Web Service Modeling Framework [13] focuses on interface description and the mediation of different data-types and processing concepts between different Web services.

On the WWW conference 2002 the language XL was introduced for the first time [14]. At the CIDR conference in January 2003 another XL publication ([15]) is about to appear. This paper focuses upon optimisation aspects of XL Web services. A detailed description of the language XL is given in the technical report [16].

8 Conclusion

This paper described the design and prototype implementation of a new programming model for Web services called XL. XL has a number of advantages compared to the programming models used in practice today (e.g., Java, C#, or Visual Basic).

- The programmer can focus on the description of the Web service. Using XL, marshalling between different data formats and data models is not necessary and optimization is automatic.
- XL is fully compliant with all W3C standards. Using SOAP, the integration of other Web services can be easily accomodated; even if these Web services are not written in XL.

There are several avenues for future work. Most importantly, there are a number of aspects that need to be integrated into the XL programming model; specifically, a distributed transaction model, life-cycle management, data push (pro-activeness), and syntactic constructs in order to facilitate authentication and authorization. Furthermore, we would like to enhance our prototype implementation in order to achieve best possible performance, availability, and security for all Web services written in XL.

References

- [1] BizTalk.org. Biztalk initiative. <http://www.microsoft.com/BizTalk/>.
- [2] BPMI.org. Business Management Initiative. <http://www.bpmi.org/index.esp>.
- [3] L. Cardelli and R. Davies. Service Combinators for Web Computing. In *IEEE (TSE)*, 1999.
- [4] 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, 2000.
- [5] Peter P. Chen. The entity-relationship model - toward a unified view of data. *TODS*, 1(1):9–36, 1976.
- [6] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. <http://www.w3.org/TR/wsdl>, Mar 2001.
- [7] Clark J., et al. XML Path Language (XPath). <http://www.w3.org/TR/xpath>, Nov 1999.
- [8] DAML Service Coalition. DAML-S: Semantic Markup for Web Services. <http://www.daml.org/services>.
- [9] 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.
- [10] D. Chamberlin, D. Florescu, et al. Updates for XQuery. internal W3C Working Draft, to appear, Dec 2002.
- [11] eSpeak. The Universal Language of E-Services. <http://www.e-speak.hp.com/>.
- [12] F. Curbera, F. Leymann, et al. Business Process Execution Language for Web Services. <http://www.ibm.com/developerworks/library/ws-bpel>, Jul 2002.
- [13] D. Fensel and C. Bussler. The Web Service Modeling Framework WSMF. <http://www.cs.vu.nl/dieter/wese/wsmf.bis2002.pdf>, 2002.
- [14] D. Florescu, A. Grünhagen, and D. Kossmann. XL: An XML Programming Language for Web Service Specification and Composition. In *WWW2002 Conference Proceedings*, pages 65–76, 2002.
- [15] D. Florescu, A. Grünhagen, and D. Kossmann. XL: A Platform for Web Services. In *CIDR 2003, Asilomar*, Jan 2003. <http://www-db.cs.wisc.edu/cidr/program/p8.pdf>.
- [16] D. Florescu and D. Kossmann. An XML Programming Language for Web Service Specification and Composition. Technical report, TU Munich, June 2001.
- [17] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
- [18] J2EE. Java 2 Enterprise Edition. <http://java.sun.com/j2ee/tutorial>.
- [19] JAKARTA. The JAKARTA Project. <http://jakarta.apache.org/>.
- [20] JMWIG. Java Extensions for High-Level Web Service Development. <http://www.brics.dk/JMWIG/>.
- [21] JXTA. The JXTA Project. <http://www.jxta.org/>.
- [22] Frank Leymann. Web Services Flow Language. <http://www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>, May 2001.
- [23] M. Marchiori, et al. XML Query. <http://www.w3.org/XML/Query>, Nov 2002.

- [24] Microsoft Corp. .net. <http://www.microsoft.com/net>, 2003.
- [25] Schema. XML Schema. <http://www.w3.org/XML/Schema>, May 2001.
- [26] SOAP. Simple Object Access Protocol. <http://www.w3.org/2000/xml/Group/>, Dec 2002.
- [27] Sun. SunOne. <http://www.sun.com/software/sunone>.
- [28] S. Thatte. Xlang overview. http://www.gotdotnet.com/team/xml_wsspecs/clang-c/default.htm.
- [29] Water. Simplified Web Services Programming. <http://www.waterlang.org>.
- [30] WebL. Compaq's Web Language. <http://www.research.compaq.com/SRC/WebL>.
- [31] Williams S., et al. XML Protocol; Abstract Model .
<http://www.w3.org/TR/xmlp-am/>, Jul 2001.
- [32] XForms Working Group. XForms: The Next Generation of Web Forms .
<http://www.w3.org/MarkUp/Forms/>, Feb 2003.
- [33] XQRL, Inc. <http://www.xqrl.com/>, Dec 2002.
- [34] XSL Working Group. Extensible Stylesheet Language XSLT.
<http://www.w3.org/Style/XSL/>, Jan 2002.