XL: A Platform for Web Services

Daniela Florescu
XQRL, Inc.
dana@xqrl.com

Andreas Grünhagen
TU München
gruenhag@in.tum.de

Donald Kossmann
TU München and XQRL, Inc.
kossmann@in.tum.de

Abstract

This papers presents XL, a new platform for Web services. We have designed XL with three main goals in mind: (a) increase application developers productivity via high-level programming constructs for Web Services routine programming patterns, (b) achieve high scalability, security, and availability for Web services and (c) compliance with all W3C standards (e.g., XML, SOAP, WSDL) such that XL Web services can interact with any other Web services written in, say, Java or C#. We hope to achieve these objectives by providing the new XL programming model based on a simple core programming "algebra" that extends Milner's PI-calculus [21]. To optimize XL programs, we employ techniques from the design of database systems, compiler construction, and data flow machines, as well as techniques specially designed for Web Services. A demo of the platform has been shown at [14].

1 Introduction

Recently, Web services have been proposed as a new model to develop new applications and to integrate existing applications on the Internet. The basic idea is that autonomous software components interact by exchanging XML messages. This technology is particularly useful to implement processes that cross organization boundaries; examples are customer-relationship management, supply-chain management, e-procurement, portals, electronic market places, online shops, and games. In addition to the XML-family of standards, the W3C has established standards such as XML Protocol (i.e., SOAP) and WSDL, and it has started a working group on Web Service Architectures.

Proceedings of the 2003 CIDR Conference

To date, most Web services are defined using Java. Analyzing the first experiences, it has become clear that Java (or C# for that matter) are not always the right programming languages for this purpose. One of the most prominent reasons is that the XML type system is incompatible with the Java type system; as a result, a great deal of marshalling code needs to be written in order to convert XML into Java objects and vice versa; this code needs to be written in addition to the marshalling code that is necessary in order to bridge the (long-known) impedance mismatch between Java and SQL [10].

Another deficiency of Java is that it is not always appropriate to deal with failures or quality of service requirements in a network-centric environment. Furthermore, a great deal of functionality needed in most Web services such as logging, database access, security, authorization, transaction support is not part of the basic Java developer's kit. J2EE has been developed as an extension to Java that overcomes this limitation [19]. Indeed, J2EE provides a great deal of additional features, but the J2EE type system is still incompatible with the XML type system. Furthermore, J2EE is a rather complicated programming model so that it is significantly less popular than basic Java.

As an alternative, several other programming languages have recently been developed; examples are WSFL (IBM), WSCL (HP), XLang (Microsoft), and the WSCI developed by a consortium including BEA, Intalio, SAP, and Sun [36, 35, 32, 34]. These languages directly support XML and the Web services paradigm, but they are not as powerful and rich as Java.

This paper presents XL, a new programming language, and its implementation. Like WSFL, WSCL, XLang, and WSCI, XL directly supports XML, the other W3C standards, and the Web services paradigm. Like Java, it provides a very powerful programming model; in fact, a great deal of syntax for imperative constructs (e.g., loops) has been adopted from Java. Since XL supports all W3C standards and communicates with other Web services using messages, applications written in XL can communicate with applications written in other languages (e.g., Java) just as well as with other XL applications. Furthermore, XL is portable (like Java) and it provides high-level pro-

gramming constructs for routine work (e.g., logging and security).

The implementation of XL combines techniques from different fields of computer science; in particular, database systems, compiler construction, distributed systems, and data flow machines. XL programs are translated into a core algebra which is in turn optimized and then interpreted in a dynamic and flexible way. We believe that only through such a high-level interface as provided by XL, scalability, high availability, security, and ultimately quality of service can be guaranteed. Using languages like Java and current middleware architectures with many layers, such guarantees cannot be given because the level of abstraction in the programming model is too low and because calls to library functions must be treated as black boxes and cannot be optimized. In summary, we believe that a high-level programming model like XL will not only significantly increase the productivity of programmers, it will also improve performance and reduce administration and operational cost.

The XL project is still at the beginning. We have a demo that is available on the WWW [14], but both the language and the current state of the implementation are still under development. In addition, we are currently working on a debugger and tools to test Web services. Testing Web services is particularly challenging because all tests must be side-effect free; for Web services testing, a closed world or lab environment assumption is unrealistic because Web services communicate with Web services from other organizations. The debugger and test environment are not described in this paper; these tools are described in [28].

2 Relevant XML Technology

An important requirement in the design of XL is the compatibility with the existing state of the art in the XML and Web Services area. We detail the current state of the art in this section.

2.1 XML Abstract Data Model.

A very popular wrong myth related to XML is that "XML is just a syntax". While it is 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/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 [3] 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). The data is modeled using 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, element, attribute, comment, processing instruction, namespace) 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.

2.2 XML Schemas and the XML Type System

The data model describes only the basic composition of an ordered tree. The XML Schema [29] 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 [30] captures the essence of the structural information present in the XML Schema. The goal of the type system is threefold. First, it is possible to do *type checking*: given an expression and the type of the input data set, it is possible to detect statically if the expression will return errors on all (or some of) the valid instances of the input type. Second, 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. 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 of the 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 [30].

2.3 XML Expressions and XQuery

A complementary W3C standard deals with XML expressions and XML queries [27]. 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 XPath standard [37].

XML queries [27] 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 XML query expressions.

2.4 XML Protocol (SOAP)

An important standard for the Web Services area is the XML Protocol [26]. The initial focus of the XML Protocol (SOAP) standardization effort is to create a simple XML-based messaging protocol that can be ubiquitously deployed and easily programmed through scripting languages, XML tools, interactive Web development tools, etc. The goal is a Web enabled layered system which will directly meet the needs of applications with simple interfaces (e.g. getStockQuote, validateCreditCard), and which can be incrementally extended to provide the security, scalability, and robustness required for more complex application interfaces.

The XML Protocol is composed of the following four components: (a) an envelope for encapsulating XML data to be transferred in an interoperable manner that allows for distributed extensibility and evolvability as well as intermediaries, (b) a convention for the content of the envelope when used for RPC (Remote Procedure Call) applications, (c) a mechanism for serializing data representing non-syntactic data models such as object graphs and directed labeled graphs, based on the datatypes of XML Schema and (d) a mechanism for using HTTP transport in the context of an XML Protocol.

2.5 Web Services Definition Language

Finally, one of the major requirements for the development of Web services is the ability to describe their interface, the boundary across which applications (Web services user agents and Web services) communicate. Applications can then interoperate using this interface.

The Web Services Description Working Group inside the W3C has the task of defining such an interface. WSDL is composed of the following components: (a) the message: a definition for the types and structures of the data being exchanged, (b) the message exchange patterns: the descriptions of the sequence of operations supported by a Web service, and (c) the protocol binding: a mechanism for binding a protocol used by a Web service, independently of its message exchange patterns and its messages.

All the pieces described above (XML abstract data model, the XML schema and type system, the XML expressions, the Web Services messaging protocol and the Web Services abstract interface) constitute excellent building blocks for a Web Service infrastructure. Unfortunately, there is no real programming language for Web Services in the current state of the art, and we believe that the existing programming languages (e.g. Java, C#) or platforms (e.g. J2EE) are not appropriate for this task. We propose XL as an alternative programming language, specially designed for Web Services.

3 XL Programming Model

In this section, we describe how Web services can be defined using XL. We describe the fundamental design principles of the proposed programming interface, the notion of a Web service and how Web services interact in conversations, and the syntax of the most important programming constructs. An example is given in [13].

3.1 Design Principles

In [13], we gave a list of 17 desiderata that we believe to be important for a programming language that supports the definition and composition of Web services. Here, we would like to reiterate the five most important principles that drove the design of the XL programming language.

- XL should support a unique data model and type system: the standard XML one [27].

- XL should be expressive enough to describe the logic of most Web services.
- 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).
- With the help of XL, programmers should concentrate entirely on the logic of their application and not on implementation or optimization issues.
- XL must be compliant with all W3C standards and it must gracefully co-exist with the current Web services and infrastructure.

3.2 Web Services in XL

From our perspective, a Web service is any autonomous software component that is identified by a unique URI, understands SOAP messages, and whose actions can be described by WSDL. Every XL program will naturally meet these requirements and will take the burden from the application programmer to worry about standards like URI, SOAP, or WSDL. However, this definition of a Web service is very general and does not imply any particular programming model. In other words, a Web service could also be implemented in Java or any other language. Also, Web services written in different languages can communicate using SOAP and independently from the way they were defined.

Technically speaking, a Web service defined using XL generalizes the notion of an XQuery entity. The definition of a Web service in XL is composed of four optional parts:

- **Web Service Definitions:** As in XQuery, local functions and types can be defined as part of a Web service definition. Furthermore, schemas and namespaces can be imported. The syntax and semantics are the same as for XQuery.
- **Variable Declarations:** This part declares the internal data (state) of the Web service; in other words, this part specifies the *global* variables of a Web service. XL supports two kinds of global variables: *web-service instance* variables and *conversation-instance* variables. Both kinds of variables have a global scope, i.e they are accessible in the entire XL program. Web-service instance variables have a single instance per entire Web Service. An example of this kind of variable is the customer database of an online shop which has to be accessed across all conversations.

  Conversation-instance variables will have a different instance per conversation which the Web service participates in. These variables represent the *context* of a conversation (Section 3.3). Instances of these variables are created whenever a Web service starts a new conversation with other Web services. In an online shop each sales process would be represented by a single conversation. A variable in this context might reference for example the specific customer and the terms of payment which apply in this case.

  In addition, XL supports *local* variables which can be used in operations; these variables, however, are not declared globally. Such (traditional) local variables are instantiated for each block/action invocation and destroyed when the execution of the block/action is completed.

  The values of all the variables in XL are XML values, i.e. instances of the XML data model [3]. In other words, the value of an XL variable can be an XML document or the content of a SOAP message. The type of an XL value may (but it is not required to) be constrained using XML Schema [29]. However, it is also possible to deal with untyped XML data or with XML data for which the type is not known apriori.
- **Declarative Web Service Clauses:** This part contains invariants (integrity constraints) and other high-level, declarative directives that control the run-time behavior of the Web service; e.g., declarative error handling mechanisms, declarative discretionary access control, default actions, periodic actions, and declarative conversation patterns (Section 3.3). Invariants will typically constrain the values of the global variables, very much like database integrity constraints. Invariants can also be used to implement certain policies; e.g., for security. For instance, an invariant could express that a Web service is not allowed to interact with two different banks as part of the same conversation.
- **Operation Specifications:** This part describes the possible *actions* (or *operations*) supported by the Web service. In object-oriented terminology, the operations of a Web service correspond to the *methods* of a *class*. The statements that can be used in order to define the operations body are summarized in Section 3.4.

3.3 Conversations

As mentioned earlier, Web services communicate by the means of messages. A *conversation* is defined as a set of correlated messages exchanged between two or more participants in order to achieve a certain goal (e.g. business goal, information exchange). Conversations are uniquely identified by URIs; they are usually implemented by carrying the conversation URI in the envelop of each exchanged message.

A typical example for a conversation is a user session in an online shop. First, the user logs onto the system (the first message from the user to the system plus answer from the system). After that, the user buys something (the second message and answer), and finally the user determines a payment method (the third message and answer). Naturally, the third message can only be understood in the *context* of the first two messages.

Another example for conversations is an online auction. The auction site informs its customers about a new product which is for sale; the customers reply by sending bids; the auction site, in turn, informs customers about new bids and the status of the auction. Again, a bid can only be understood in the context of a whole auction and the messages exchanged for a particular auction are correlated. The auction example shows that more than two Web services can be involved in a conversation and that interaction patterns can be quite complex. Obviously, not all participants of a conversation are allowed to listen to all messages that are exchanged as part of the conversation; for instance, sealed bid auctions could also be implemented as a conversation.

XL supports the definition of Web services that participate in conversations in two ways. First, XL frees the programmer from manually managing the contextual data associated to a conversation via the *conversation-instance* variables. For example, for the auction site, the context of a particular auction is the product specification, the customer who wants to sell the product, the highest bid, the name of the customer who made the highest bid, and the list of customers that participate in a particular auction and would like to be informed about the status of the auction. Obviously, many auctions can be carried out concurrently at the auction site so that many instances of these variables need to be kept: one instance of each variable for each auction. When the auction site receives a message that is related to auction X and that involves the execution of the *closeAuction* operation, then the right instance of, say, the *highest bid* variable is automatically loaded and can be used in the definition of the *closeAuction* operation. For a customer, the context of an auction might simply be the highest bid and the information whether this bid was issued by the customer herself; this information might be need in, say, a *react* operation that can be defined at a customer's Web service.

The second way in which XL facilitates the conversation implementation is the declarative definition of the conversation patterns. It would be a great deal of work and error-prone if the programmers would have to manually handle the conversations and their URIs. Fortunately, in practice, there are only a handful of different patterns that define the way Web services participate in conversations. XL has a set of predefined patterns, and allows programmers to declare them; actions are performed automatically to implement them. For instance, if the implementor chooses a *Mandatory* conversation pattern, then each incoming message *must* contain a conversation URI (i.e., be part of a conversation); otherwise, an error is returned. Also, the mandatory pattern implies that the conversation URI of an outgoing messages (electronic payment) is the same as the conversation URI of the corresponding inbound message (a purchase order) that triggered the payment; in other words, the inbound and outgoing messages are part of the same conversation. More about XL conversation patterns can be found in [15].

3.4 XL Statements and Combinators

The body of an XL operation is described by statements, which are extensions of XQuery expressions [27]. In addition to classic imperative statements like variable assignment, conditional statements, loops, error handling and return statements, XL supports some XML specific update statements and some Web services specific statements (e.g., Web services invocation, logging, sleep). Finally, in addition to the classic imperative statement combinator (sequencing), XL supports other *statement combinators* borrowed from the workflow and dataflow theory (e.g., dataflow, parallelism, choice). In the following, we will briefly describe the most important statements. The full set of statements and statement combinators of the current design of XL is described in [13].

In the expressions of an XL statement all global variables declared in the *variable declaration* part of the Web service are in scope, as well as all the local variables defined in the current operation. Finally, for each operation, two local variables are implicitly defined: *$input* that is automatically bound to the body of the SOAP message that triggered the current operation execution and *$output* whose value implicitly constitutes the body of the response message.

Variable Assignment

The simplest statement is the assignment of a (global or local) variable. The syntax is as follows:

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

Any expression defined by the W3C XQuery proposal [27] can be used on the left side of an assignment. Local variables need not be declared before being used. The type or XML Schema of a local variable can optionally be set as part of the first assignment to this variable. (Global variables must be declared; the type of a global variable can optionally be set when the variable is declared.) As in Java, the scope of a local variable is the block where the variable is defined.

Update Statements

Unfortunately, XQuery does not yet provide expressions to manipulate XML data. Don Chamberlain et al. setup a working draft to extend XQuery in this respect [11] 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   $client
```

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

```
delete $client/creditcard[type="Visa"]
```

- *replace* in order to adjust elements (e.g., the telephone number)

```
replace $client/telephone with
  <telephone>(408)8901-23</telephone>
```

- *rename* in order to rename certain nodes (element or attributes)

```
rename $client/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  $client/telephone
after $client/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 recipient is expected to understand the message without an operation specification. In a synchronous call, the execution is halted until the called Web service finishes its execution and returns the result or an error which are both also wrapped in SOAP messages. 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.

As an example, consider the following synchronous service invocation that asks the online broker to buy 1000 SAP for at most €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.Broker.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 (reply or error message) 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 way of setting the envelop of a SOAP message explicitly. Such constructs could be useful in order to implement certain kinds of conversations, quality of service guarantees, 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 [22] have stabilized.

Conditional statements and Iterations

Just like most other programming languages, XL provides IF-THEN-ELSE statement, WHILE- and DO-WHILE loops (not shown here). The semantics are straightforward and the same as in other imperative programming languages. As for example the IF-THEN-ELSE listing below shows:

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

In addition to these common control statements XL supports a FOR-LET-WHERE-DO loop, with the following syntax:

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

The FOR-LET-WHERE-DO loop corresponds to FLWR expressions in XQuery [27], but it executes statements (with potential side effects) instead of evaluating side-effect free expressions.

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 SOAP message to the caller of the service. Just like variables and any other expression, the exceptions can be strongly typed optionally.

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. As in Java, a caught exception will trigger the execution of the associated statement.

3.5 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[33].

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>
```

Furthermore XL provides a set of additional combinators

? Simple error handling. If "statement1" fails, execute "statement2.".

| Nondeterministic choice. Execute either "statement1" or "statement2," but not both

|| Parallelism. Execute "statement1" and "statement2" in parallel.

& Dependency. Considering existing dependencies betweeen the statements an execution order is choosen by the system itself.

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 Optimization and Execution of Web Services

Having described the basic features of the XL programming model, we would like to turn to a presentation of the design of our platform to execute Web services. The XL platform has been designed with four major goals in mind: (a) high performance, (b) high reliability, (c) high security level and finally (d) very low or inexistent administration costs (i.e automatic optimization). We believe that the high level, declarative programming model of XL will enable us to achieve these goals, while they are very hard to achieve in an environment entirely based on a lower level imperative programming language like Java or C#.

Implementing such a platform for Web services involve combining techniques from different disciplines; most importantly, database systems, compiler construction, distributed systems, networking, and data flow processing. From database systems, we adopt the approach to transform programs in equivalent algebra expressions and to carry out transformations / optimizations on these expressions. From compiler construction, we adopt techniques such as dead code elimination and peephole optimization. Furthermore, an efficient platform for Web services will employ techniques like caching and process migration in a cluster of servers in order to achieve scalability. Finally, we advocate the use of pipelining and of data flow processing as much as possible; data flow processing has been studied, e.g., in projects like Ptolemy [25]. Exploiting data flow processing is one of many differences between our XL implementation and typical implementations for, say, Java.

We present in this section a couple of techniques that will lead us to achieving the four desiderata listed above. We note that not all the techniques presented in the following have been implemented and integrated

into our prototype. However, we are currently working on adding these techniques to the system and we hope that we will be able to carry out performance experiments soon.

4.1 Architectural Overview

The XL platform has two main components: (a) the XL compiler, and (b) the XL virtual machine. The role of the *XL compiler* is to translate the textual representation of an XL program into a *statement graph*. The statement graph is an abstract representation of the definition of an XL Web service, analogous to a database query execution plan, but adapted to *statements* and *programs* instead of *queries*. The statement graph uses a core "algebra", i.e. a set of simple, basic statements that are able to support the entire XL semantics.

The compilation is done in two steps. First, the textual representation of an XL program is translated by the *XL parser* into a naive (usually suboptimal) statement graph. Second, the *XL optimizer* transforms this naive statement graph into an equivalent but more efficient statement graph. The *equivalence* of statement graphs is defined in terms of the equivalence of their returned results, but also in terms of the equivalence of their impact on the context of the conversation where they are executed (e.g. variable updates, messages sent). If we think of a distributed conversation with different participating servers the context is not just the locally represented state of a single server but a conjunction of the states of all participating servers.

Furthermore, the optimizer adds annotations to the statement graph in order to specify precisely how each statement should be executed.

The statement graph produced by the optimizer is interpreted by the XL virtual machine whenever triggered by an incoming message. This process can be described as follows.

First, each incoming message is processed by a message handler. The XL message handlers correspond to the various modes of interaction with the XL virtual machine. Examples of such modes of interaction are: (a) *normal action* will invoke an operation of the Web service in the standard operation mode; (b) *error* which deals with inbound messages which are not understood; (c) *debug* which processes messages invoked by the XL debugger; and (d) *test* which simulates the actions of the Web service without any side-effects in order to test a Web service that invokes operations of another Web service [1].

Next, when the *normal action* message handler passes a SOAP message to the virtual machine, the virtual machine loads (if not cached already) the statement graph of the called operation from the database and executes it. The execution is done in a certain

[1]Debugging and testing are not subject of this paper; our initial design is covered in [28]

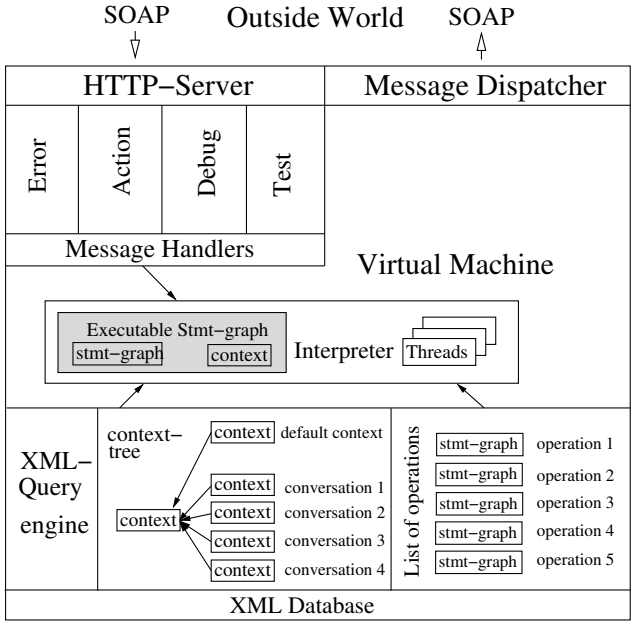

Figure 1: The architecture of the XL Runtime-System

context that contains all the information needed for the execution. The data produced as the result of the execution of the statement graph is sent back to the caller as another SOAP message. Figure 1 gives an impression of the XL architecture.

We would like to high-light three important features of the XL virtual machine. First, in order to execute statements in parallel, the virtual machine is multi-threaded. Second, the virtual machine is designed to be able to stream the intermediate data between statements; pipelining is a very important feature of our design. Third, in order to achieve scalability and high reliability, the XL virtual machine has been designed to support the migration of processes from one machine to another machine in a cluster (we expect that the platform will be installed on a cluster of servers).

Some statements and actions change the values of global variables; such statements involve interaction with an XML database that stores the values of these variables. Furthermore, the XML database stores all context information and an XML representation of the statement graph for each XL operation. Currently, we rely on a third-party database vendor for this purpose. In fact, at the moment, even a standard relational database system could be used for our purposes. In order to achieve very good performance, however, there must be a tighter coordination between the XL compiler, the VM and the database backend. This coordination is necessary to carry out certain kinds of optimizations (see Section 4.5), to exploit indexes of the database, and to operate efficiently in a distributed environment (e.g., in a cluster).

The right internal XML data representation is crucial for good performance in the virtual machine. Re-

call that all XL programs *only* manipulate XML data (the values of variables are XML documents) and that XL programs (i.e., statement graphs) and contexts are represented in XML. In the XL platform all XML data is represented as a vector of tokens; the XML parser generates a sequence of tokens for each XML document, in a similar way as a SAX parser generates events[2]. Representing XML data as vectors of tokens allows a very compact internal representation and it allows the processing of data in a stream-based way (Section 4.6). In many cases, such a vector never needs to be materialized; instead, the statements are implemented as iterators [17] and consume tokens in a pull-based manner. XML data lives through the entire process inside the XL virtual machine in this internal format, and the data is serialized back into an XML string only at the end of the execution of an operation when the result/answer is sent back as a SOAP message.

In the remainder of this section, we will first describe the algebra (i.e., the core XL statements, Section 4.2), the statement graph (Section 4.3) and the execution context (Section 4.4). Then, we will briefly sketch possible optimizations (Section 4.5) and the two most prominent features of the runtime system: stream-based processing (Section 4.6) and process migration in a cluster (Section 4.7).

4.2 XL Core "Algebra"

Although XL is a fairly powerful programming language, all constructs are supported by a very simple core algebra composed of six statements. This simplicity makes XL easy to optimize and to build highly reliant and secure platforms for XL. In some sense, these six statements can be seen as the pendant to the operators of the relational algebra that is used in order to implement SQL queries. As the operators of the relational algebra, these core statements can be implemented in different ways (e.g., pipelined and non-pipelined) and it is the responsibility of the XL optimizer to determine the best way to execute them.

- **Assignment:** As mentioned in Section 3.4, assignments bind an XL variable to an arbitrary XQuery expression.
- **Send:** This core statement evaluates an expression and sends the result to the target URI.
- **Receive:** This statement blocks and waits until it has received a specific message.
- **Wait:** This statement blocks until a certain event (e.g., an update to a variable) takes place.

[2]The major difference between our token stream and SAX is the fact that SAX events are propagated in a push fashion while our tokens are consumed in a pull mode.

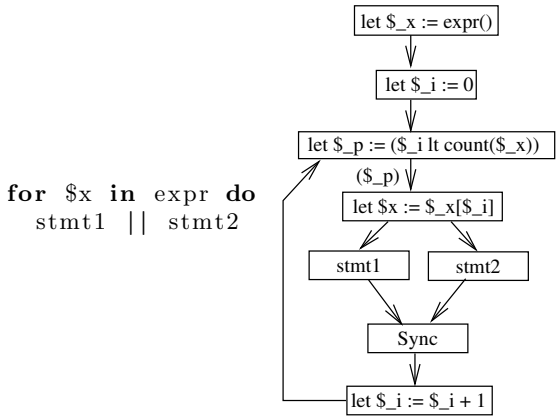

Figure 2: Statement Graph for a FWD- loop

- **Sync:** This statement is used to synchronize the parallel execution of statements.
- **Update:** We provide native support for the XML update statements described in Section 3.4.

4.3 Statement Graph

Each operation of an XL Web service is abstractly represented by a statement graph, similar in spirit to the statement graphs used for data flow analysis and code optimization (see [4]). The nodes of this statement graph are the core statements described above and the edges in the graph encode the order in which the statements are to be executed. An edge from statement S_1 to statement S_2 indicates that S_1 must be executed before S_2. The graph need not be fully connected; statements that are not ordered in the graph can be executed in parallel. Edges can be annotated by Boolean variables that indicate under which condition a statement is executed after another statement. This way, the statement graph encodes *if* statements and loops. Figure 2 gives a small example. The nodes of the statement graph are also annotated with additional information like the specific algorithm that has to be used to implement the statement and the estimated cost of the statement [3].

4.4 Execution Context

Each XL statement is compiled and executed in a *context*; the context contains the values of variables (e.g., local variables and conversation-specific variables) and it contains all other information needed to evaluate expressions (e.g., connections and pre-compiled queries).

The XL statements may need to evaluate XQuery expressions; such expressions also need an evaluation context that contains information like the imported schemas, namespace definitions, local functions and so on [27]. Such contexts can be generated dynamically

[3]not shown in figure 2

and are organized in a hierarchy. In such a hierarchy of contexts, an XQuery expression is executed in a particular context; if this context does not contain a specific definition used in the expression, then the parent context is inspected. The root of the context hierarchy, the base context, contains the definition of all built-in XQuery types, namespaces and functions [2]. This feature is exploited in the XL virtual machine in order to implement scopes of variables and conversations (i.e., organize the *conversational context*).

4.5 Optimization

The purpose of the optimizer is to transform the statement graph into a more efficient statement graph and to annotate the statement graph so that it can directly be executed. Some of the transformations applied by the optimizer are standard techniques from compiler construction (e.g., dead code elimination and detection of common subexpressions), some of these transformations are known techniques from the database camp (e.g., reordering of statements, parallelization, and control of data flow and materialization of intermediate results), and some of these transformations are specific to XL and our particular domain.

We are currently experimenting with the following techniques:

- **Common subexpressions:** Identify (sub-) expressions in different statements that are guaranteed to produce the same value. This is a classic technique studied in compiler construction and involves data flow analysis [4].
- **Reorder statements:** Identify statements which could be executed in parallel because they do not have any data dependencies. Furthermore, identify statements that operate on the same data in order to improve the temporal locality of the application.
- **Multi-query optimization:** In certain situations, it might be better to evaluate two expressions simultaneously rather than each expression individually. Such situations can be detected in XL because the compiler has a more global perspective on the application than a traditional database compiler (i.e. the entire program is analysed, not only an isolated query).
- **Batched updates:** Often updates to the same (or even different) variables can be batched and executed in one go, thereby saving round-trip messages to the database.
- **Identify local service invocations:** For such invocations, we could eliminate the (very high) overhead of the SOAP protocol. Moreover, the caller and the callee Web services can be optimized together in order to achieve better performance.
- **Identify local side-effect free service invocations:** If side-effect-free invocations are part of expressions, it might be possible to avoid these invocations altogether. Furthermore, the result of such an invocation can be cached.
- **If-then-else optimization:** First, if the conditions of nested *if-then-else* statements are disjoint, the order in which these conditions are evaluated can be tuned depending on the selectivities and cost of these conditions [20]. Furthermore, materialization and hashing can be applied to save costs to find the right branch of deeply nested *if-then-else* or *switch* statements. We expect that evaluating the conditions of *swich* and *if-then-else* will take up a substantial part of the cost to execute a Web service, considering trends like the personalization of Web services.

This is only an initial, incomplete list of possible techniques. Moreover, all of the techniques above have in some way or the other been discussed in the context of compiler construction, persistent programming languages, or database systems — which optimizations are particularly amenable and how they interact in our specific domain is still subject to future work. All these optimizations can be carried out very well in XL due to the high level of abstraction of the programming language and the very simple core algebra and statement graph that is used to implement the language.

4.6 Streaming Execution

Since XL was designed for data-intensive applications, the XL virtual machine makes excessive use of pipelined processing. In fact, the XL virtual machine can be characterized as a data flow machine. This approach is in the database tradition, but it is quite different from the way that, e.g., typical Java VMs are designed. Pipelining means that the "results" of one statement are passed piece by piece (i.e., token by token) to other statements. For this purpose, each statement is implemented as an iterator, as proposed in [17]. In order to understand what pipelining means in XL, we would like to interpret an edge between statements S_1 and S_2 as a set of pipelines (one for each variable) through which the values of each variable are passed token by token from statement S_1 to S_2. Statement S_2 consumes these tokens (if needed) one by one in its expressions, and pipes them (if unchanged) to the next statement(s) in the pipeline.

Almost all the core statements described in Section 4.2 can be implemented in such a pipelined fashion. The *assignment* statement, for instance, can be implemented by pipelining the results of the expression on the right side to the next statements and by carrying out successive inserts if the variable on the left side is persistent. The only exceptions from this rule are *event* and *sync* which require special attention.

The reasons for pipelining in the XL platform are essentially the same as in commercial database systems. First, pipelining provides faster initial response times; i.e., the system returns the first answers earlier. Second, pipelining reduces the main memory requirements and reduces costs to materialize intermediate results (e.g., on main memory or on disk) and/or the unfortunate swapping. Third, pipelining is a form of parallelism and therefore, potentially reduces the total response time. Fourth, pipelining is particularly important in the presence of bursts in the network: a service can start processing the first inputs while it is waiting for the rest of the input. Finally, pipelining is very helpful to avoid wasted work or even ensure that certain operations terminate in a similar way as lazy evaluation does for functional programming.

As an example, consider, the following code fragment:

```
!! infinite input from a sensor
let $a := (1, 0, 1, 0, 1, 0, ...)
if ( some $b in $a satisfies $b = 1) then
   let $output := "at least one '1'";
endif
```

Without pipelining, the execution of the first *let* statement would not terminate. With pipelining, the first 1 will be piped into the *if* statement which in turn immediately evaluates to true. As a result this operation will simply return the output *at least one '1'* (Existential qualification is very frequent in XQuery because it is implicit in almost all comparisons. For the purpose of clarity, we made it explicit in this example.)

Of course, pipelining is not always the best execution model and pipelining sometimes comes at an additional cost: overheads for handling individual tokens and synchronization if several statements consume the results of a statement. In particular, pipelining must be used with great care across iterations of a loop.

4.7 Process Migration Inside a Cluster

In order to achieve scalability and high reliability, we expect that the platform will be installed on a cluster of servers. In such a cluster, each server will run a separate XL virtual machine. The XL virtual machine has been designed to support the migration of processes. To migrate a process (i.e., conversation), simply the context of that conversation needs to be shipped to a different server because the context contains all the relevant information. Process migration is particularly important in this environment because interacting with external, autonomous Web services (and users) can result in message round-trip times and waiting times of several days. For complex operations, the virtual machine can also be instrumented to execute different portions of an activity on different machines in parallel, thereby exploiting pipelined and independent parallelism within an operation. Again, we believe that it is going to be easier to parallelize operations and to achieve scalability and high availability in a cluster using XL than using Java or J2EE due to the very simple and clean semantics (core algebra) of the XL programming model. This observation follows the tradition of relational databases that can be parallelized very well based on the relational algebra. However, we still need to prove our intuition.

5 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 W3C workshop on Web services [9] and the latest issue of the IEEE Data Engineering Bulletin on e-services [1].

In the industry, there have been a number of concrete proposals for new languages and frameworks related to our programming language proposal—most prominently, SUN's J2EE [19] and SunOne [31], and Microsoft's .NET initiative [24]. Compaq has developed the WebL language [33]; HP has developed the eFlow and eSpeak systems [7, 12], IBM is working on a language called Web Service Flow Language [36], and Microsoft has recently released their BizTalk Server 2000 [5] and XLang [32]. As mentioned in the introduction, these proposals either fall short to provide a full programming environment or they are too low-level and too complex in order to achieve our main objectives: increase the productivity of programmers and achieve better (automatic) optimization.

Web services are intimately related to workflow systems. There is of course an extensive literature on workflow systems (e.g., [23, 8, 16]). A great deal of the work already done in the area of workflow systems is applicable here, but has to be adapted to the particularities of the Web services context.

In the academic world, the notion of a service composition is based on a solid theoretical background consisting on the calculi developed first by Hoare [18], Milner [21] and more recently by Cardelli [6]. However, none of those languages and frameworks are consistent with the current XML and Web services state of the art, and the proposed approaches to implement those calculi are not designed for data-intensive and scalable applications.

6 Conclusion

We presented the design of XL, a new platform for Web services. XL provides a complete programming language that is fully compliant with the Web services paradigm and all related W3C standards (i.e., the XML family of standards, SOAP, and WSDL). One of the main principles is to provide a very high-level and mostly declarative programming model and to reduce complexity and the number of software layers (i.e., tiers) in a typical application system. This

way, we hope to increase the productivity of programmers, make optimization automatic, improve the performance and scalability of application systems. Although this has not been the focus of our work so far, we also hope to make it easier to obtain high availability and security / safety using our programming model and platform.

The project is still at the beginning. We are currently seeking for feedback on the design of the programming language. Furthermore, important concepts such as the transaction model are still open. We plan to fill in these holes as soon as some of the W3C standards (e.g., XML Protocol) have further stabilized. We have a prototype implementation of the platform (see [14]), but a great deal of engineering and experimentation is still necessary in order to achieve high performance and scalability. We are also planning to carry out pilot projects with customers and their applications using the XL platform.

References

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

[2] XQuery 1.0, XPath 2.0 Functions, and Operations Version 1.0. http://www.w3.org/TR/xquery-operators/, August 2002.

[3] XQuery 1.0 and XPath 2.0 Data Model. http://www.w3.org/TR/query-datamodel/, August 2002.

[4] J. Ullman A. Aho, R. Sethi. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1988.

[5] BizTalk.org. Biztalk initiative . http://www.microsoft.com/BizTalk/.

[6] L. Cardelli and R. Davies. Service combinators for Web computing. In *IEEE (TSE)*, 1999.

[7] 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.

[8] Vassilis Christophides, Richard Hull, Akhil Kumar, and Jrme Simon. Workflow mediation using vortexml. *IEEE Data Engineering Bulletin*, 24(1), 2001.

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

[10] 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.

[11] D. Chamberlin, D. Florescu, et al. Updates for XQuery. W3C Working Draft, Oct 2002.

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

[13] D. Florescu, A. Grünhagen, and D. Kossmann. XL: An XML programming language for web service specification and composition. In *WWW2002 Conference Proceedings*, 2002.

[14] D. Florescu, A. Grünhagen, D. Kossmann, and S. Rost. Xl demo. http://xl.in.tum.de/.

[15] D. Florescu and D. Kossmann. An XML Programming Language for Web Service Specification and Composition. Technical report, TU Munich, June 2001.

[16] Michael Gillmann, Wolfgang Wonner, and Gerhard Weikum. Workflow management with service quality guarantees. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Madison, Wisconsin, 2002.

[17] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

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

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

[20] A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimizing boolean expressions in object-bases. In *Proc. of the VLDB Conf.*, 1992.

[21] R. Milner. *Communication and Concurrency*. Prentice Hall, 1994.

[22] XML Protocol; Abstract Model. http://www.w3.org/TR/xmlp-am/, Jul 2001.

[23] C. Mohan. Workflow management in the internet age. Technical report, IBM Almaden Research Center, 2000.

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

[25] The Ptolemy Project . http://ptolemy.eecs.berkeley.edu/.

[26] Simple Object Access Protocol. http://www.w3.org/2000/xp/Group/, Jun 2002.

[27] XML Query. http://www.w3.org/XML/Query, Aug 2002.

[28] R. Rosette. Quality assurarance for web services. Diploma Thesis, TU München, 2002.

[29] XML Schema. http://www.w3.org/XML/Schema, May 2001.

[30] XQuery 1.0 Formal semantics . http://www.w3.org/TR/query-semantics/, August 2002.

[31] Sun. Sunone. http://www.sun.com/software/sunone.

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

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

[34] WSCI. Web service choreography interface. http://wwws.sun.com/software/xml/developers/wsci, 2002.

[35] WSCL. Web services conversation language. http://www.w3.org/TR/wscl10, Mar 2002.

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

[37] XML Path Language (XPath). http://www.w3.org/TR/xpath, Nov 1999.