INAUGURAL - DISSERTATION

zur
Erlangung des Doktorwürde
der
Naturwissenschaftlich-Mathematischen Gesamtfakultät
der
Ruprecht-Karls-Universität
Heidelberg

vorgelegt von

Diplom Wirtschaftsinformatiker	Andreas Grünhagen
aus:	Karlsruhe
Tag der mündlichen Prüfung:	27. November 2006

Thema

The XL Web Service Language: Concepts and Implementation

Gutachter: Prof. Dr. Donald Kossmann

Prof. Dr. Barbara Paech

Abstract

The XL programming language has been build on two very simple premises. First, XML is the forthcoming language used to describe and communicate complex data. Second, services provided via the internet are complex but loosely coupled and use XML. Services are neither bound to a certain platform, computer, or application scenario. The interaction between service provider and consumer is based on availability and reliability of interface descriptions and the coherence to internet standards like HTTP and XML.
The XL language provides the means to easily describe complex services based on the XML data model, the XML query language XQuery and an XML storage model. In the following, the ideas behind the XL language, the language itself, and the XL runtime engine used as a prototype will be described in detail. Furthermore, different statement processing concepts, different usage scenarios and the non-functional requirements of the runtime engine itself are discussed.

Abstract

Die Entwicklung der Programmiersprache XL basiert auf zwei einfachen Grundannahmen. XML ist die meistbenutze Sprache zur Beschreibung und Kommunikation komplexer Daten. Im Internet bereitgestellte Dienste sind lose gekoppelt und basieren auf der Kommunikation von XML. Dienste im Internet sind weder an konkrete Umgebungen, Betriebssysteme oder Anwendungsszenarien gebunden. Die Interaktion zwischen Client und Server basiert auf der Verfügbarkeit verlässlicher und exakter Schnittstellen sowie den verwendeten Internet Standards, wie zum Beispiel HTTP und XML.
Mittels XL lassen sich komplexe Dienste in einer Programmiersprache beschreiben, die auf dem XML Datenmodell basiert und mit XQuery einer funktionale XML Sprache für Ausdrücke verwendet. Im folgenden werden die XL zugrundeliegdenden Ideen sowie der Prototyp der XL Laufzeitumgebung detailiert erläutert. Darüber hinaus werden verschiedene Anwendungsszenarien, funktionale wie nicht-Funktionale Anforderungen sowie verschiedenen Konzepte zur Befehlsausführung innerhalb der XL Laufzeitumgebung diskutiert.

Contents

Chapter 1

Introduction

With the current use of extensible markup language (XML) as both, a data representation and a communication means, the requirements for XML processing are becoming more complex. The thesis at hand describes the language design and the implementation of runtime environment for the XML processing language XL.
The XL programming language is designed to provide an easy to use, scalable, and efficient XML processing environment. In this brief introduction, some challenges of computer science will be pointed out, and how XML and XML processing could contribute to their solutions.

1.1 eXtensible Markup Language (XML)

Extensible markup languages (XML) denote a concept of semi-structured description languages. Data is structured using a small set of markup characters. Each XML document contains a tree structure of nested elements, labeled by corresponding start- end end-tags. An XML document c contains a single root element which in return can contain further elements, attributes and arbitrary text. Furthermore XML provides comments, processing instructions, and additional character data encoding structures. Listing 1.1 illustrates the different types of XML data structures. XML, like the example shown in listing 1.1, is easily readable be a human and easy to parse by a computer.
XML denotes a whole group of different languages, designed for various purposes. The following list contains just a small sample of XML languages picked from different domains, a much longer list of different XML languages can be obtained, for example, from the web pages of the World Wide Web Consortium (http://www.w3.org).

- HTML (HyperText Markup Language) [W3C06a], maybe the first well known markup language, used to describe the layout of webpages.
- RDF (Resource Description Framework) [W3C04b] describes arbitrary resources by associating simple properties and property values.
- SOAP (initially *Simple Object Access Protocol*) [W3C04c] message exchange standard used by Web services.
- Voice Extensible Markup Language (VoiceXML) [W3C05c]is a language designed to describe spoken languages. The typical use case of VoiceXML is the description of a dialog between a user and a speech recognition system used to place a purchase order.

```
<?xml version="1.0" encoding="UTF-8"?>
<part>
  <introduction author="Andreas">
   <!-- a Comment -->
    <eXtensibleMarkupLanguage/>
    <WebServices>
       nested text
    </Webservices>
  </introduction>
  <?process type="a processing instruction" ?>
   free text
</part>
```

Listing 1.1: *Example XML data, containing an XML prolog, nested XML elements, a comment, a processing instruction and nested text.*

The XML concept, of providing content and structure within the same document, is generally applicable. As the mentioned languages indicate, XML-related technology is used for all kinds of purposes:

- Database systems are enhanced by XML processing and storage capabilities (DB2 Extender [CX00] and Oracle XML DB Repository).
- Documentstandards used for wordprocessing are based on XML (e.g., Microsoft Word v12, or OpenOffice).
- Enterprise services software like, SAP R/3 Netweaver, use XML for inter-process communication (e.g., SOAP).

These examples illustrate the widespread use of XML. The flexibility XML offers is means to express complexity of business or user requirements. Due to the structure inside an XML document itself, interpreting the information (data and structure) is very easy. Information can be easily parsed, interpreted, extracted and transformed.
The advantageous features of XML are briefly outlined by the following bulletpoints:

Communication XML is means to ease communication and integration. XML languages can easily be processed either by a human or a machine. Standard techniques for describing XML schema information are available and can be applied to any communication pattern or use case.

Structure XML data is semi-structured, say, the XML data itself provides structure information even if no schema information is available.

Standardized XML is standardized by the World Wide Web Consortium (W3C) and it is being used. A big set of XML-related software is available: XML parser (Xerces at [Apa05b]), XQuery engines [FSC+03][FHK+03], advanced XML editors, and various XML software libraries.

The main advantage of XML is the possibility to easily describe complex structures, which, in the next step, can easily be interpreted by different processes with a minimum of prerequisite knowledge.

In this thesis a new XML processing language is presented. Instead of transforming XML data into a different representation (e.g., object-oriented or relational modell), the processing logic is expressed directly using the XML data modell. In chapter 4 syntax and semantic of the language XL are described, in section 8 the design of the XL runtime environment, the XL engine, is described.

1.2 Web Services

In the journal "*Communications of the ACM"*, David A. Patterson listed security, privacy, usability, and reliability (SPUR) as the current main challenges for computer and communication systems [Pat05]. Meanwhile, software development is not meant to be an art or a craft anymore, but an industry. The software development process is continuously tailored to be repeatable, predictable, or say, better controllable. Requirements do not address a single computer or application, but complex systems containing local and remote components, application logic, as well as different representations of data.
The basic idea of Web services is to compose a new service by combining loosely coupled applications. The term *loosely coupled* implies a late binding, possibly the components connected are determined at runtime by a look up at an UDDI server [UDD]. Typically, application communicate by sending XML messages using standard protocols, like HTTP [RF99]. In order to industrialize software engineering, Web services are one possible approach. A Web service is meant to be a well defined component, designed for being used in different contexts, by different users. Typically, a Web service provides information describing the technical interface as well as conditions and assumptions a service presumes. By doing so, Web services are one way of addressing the non-functional requirements listed by David A. Patterson.
Another concept, denoted by buzzwords like *service-oriented architecture* (SOA), is the virtualization of services.
If we turn to database programming, integrity constraints are checked, to some extend, by the database system itself. Even so this possibly reduces the performance, relieving the software developer from basic constraint checks pays off. If the same approach is applied to Web service, the service engine checks constraints set by a software developer. Compared to database applications, Web services address a different abstraction level and integrity constraints are more complex.
XML is a means to solve the problem. XML could be used to structure complex data, to describe properties, to ease communication and integration. In order to do so, an XML processing language, or more general, an XML processing environment has to be designed. The major requirements for an XML programming language are:

- a unique data model and type system: the XML standard [W3C04f].
- expressive enough to describe the logic of most Web services.
- comfortable to use. Hence, it should provide special constructs for important Web service programming patterns (e.g., logging, retry of actions, and periodic actions).
- the programmer should focus on the logic of their service and not on implementation or optimization issues. Declarative descriptions of state, conditions and side effects are preferable.

- it must be compliant with all W3C standards and it must gracefully co-exist with the current applications and infrastructure.

This following thesis is structured into 2 major parts and a final conclusion:

- Part I addresses the general aspects of XML processing and the concept of the XL language itself.
  - Chapter 2 describes in detail the motivation and the problem statement addressed by this thesis.
  - Chapter 3 describes the available XML processing technologies in general and, specifically, Web service related languages and concepts.
  - Chapter 4 describes language elements and syntax of XL.
- Part II describes the XL runtime system and the concepts used to implement the XL prototype.
  - Chapter 5 gives a brief introduction to the XL Runtime environment.
  - Chapter 6 describes different use cases of an XML processing language.
  - Chapter 7 provides an in depth view of the software architecture used to design the XL runtime environment.
  - Chapter 8 contains a detailed description of the XL engine itself.
  - Chapter 9 lists ideas on how to enhance and further develop the XL language.
  - Chapter 10 describes the performance experiments which were made and provides some basic figures.
  - Chapter 11 describes an alternative statement processing concept (pipelined execution).
- Part III provides a final conclusion and conceptual outlook on how to integrate complex service descriptions into a service framework.

Part I

XML Processing

Chapter 2

Motivation

XML is a data format and a markup language. The XML data is semi-structured, say, content and structure are mixed. XML data contains the content itself and information on how to interpret the content. XML data structures typically form a tree structure of nested elements. Furthermore XML provides means to define new structure by specifying the relationships between elements types.
In the following, the advantages of XML as a form of data representation are briefly listed:

- XML offers the possibility to create complex data structures on the fly. As the structure markup is provided together with the content, new elements can be added easily. Likewise, dispensable elements can be omitted. The possibility to flexible adapt existing data structures represents powerful means to capture software and data complexity.

  Furthermore XML provides several other features as for example an ordering concept (document order), a linking concept connecting separate documents or elements, and namespaces.

- XML can be processed easily. Due to the tree structure querying, transforming, and displaying XML data is easy. Furthermore XML itself and the processing concepts (i.e., parser, query languages, object model) are standardized. The XML data format and the processing concepts available are not restricted on a certain programming language, hardware platform, or operating system. Depending on the use case, the readability by a human user could be important as well.

The value of XML is based on the flexible data structure. Using XML is favorable if an application addresses heterogeneously structured data. Not surprisingly, this is the case for most real world data. Like Java objects, XML elements represent real world entities.

2.1 Problem Statement – XML processing

The problem statement of this thesis is split into two parts: problems addressed by the initial language design and the (following) design of the language runtime system itself. The following two subsections address these two aspects separately.

2.1.1 Language design

The processing of XML data still lacks a separation of concerns. By using a database system the separation of the different layers application, logical data representation and physical data representation could be easily achieved. For XML processing this is not the case. Processing XML still requires plenty of data transformations between either XML text, a generic document object model (DOM) [DOM04], Java Beans or relational tables. These data transformations are not only time consuming and error prune but also the separation of concerns is broken up in many cases. When developing an XML processing application a whole bunch of XML related technologies are available. Even so XML is meant to be the common exchange format, doing the cherry picking of XML technologies is difficult. This could be illustrated by a few problems common to XML processing:

- **Storage** Even so plenty of XML storage systems have been proposed, using either relational or native approaches, storing XML is still an active research area (see [DK05]).
- **Query Processing** Processing XML in general strongly depends on the XML representation used and the requirements of the use case. While in one case the application logic is implemented using Java Beans inside a J2EE framework [Sun05b], in another case application logic is expressed using XML Query [W3C04g].
- **Schema handling** In contrast to relational data, or say Java objects, the XML type information is optional. In return the use of XML Schema information depends on the technology used. While in some cases schema validation is optional ([DOM04]), it is mandatory in other cases ([Sun04a]).

As pointed out previously, a separation of concerns between physical storage, the type of the data and the application logic is still missing. The achievements of the J2EE framework [Sun05b], providing modularity, for the design of Java applications is still missing for XML applications. The XML Web service engine XL we set up does not solve the problems of storing XML, query processing, or schema handling, but we provide a high-level processing concept separating storage, query processing and type information.
The term *Web service* denotes a standalone application, providing XML-based services. The concept of loosely coupled application require a high level, declarative programming language describing the interaction and the application logic of services. Since XML data is the lingua franca of the Internet, XML processing has to be an integral part of a Web service framework. The problems addressed by a declarative XML processing language are listed in the following:

Storage Integration As for database applications, the separation of concerns for XML processing applications is a crucial element of software design. By separating the application logic and the XML storage layer, a flexible storage concept is possible. XML data in some cases serves as a data source in the style of a database while in other cases an XML element simply reflects a datastructure such as a Java object or C variable. It has to be possible to choose an XML representation depending on its usage or scope.

In order to achieve this separation of concerns, say, to hide the XML storage from the application programmer, the design of the XML language presented in this thesis does not address this issue. The scope and not the storage is an attribute of a data element. XML data is either persistent or not. In return, the runtime environment has to provide means to store data persistently.

XML processing The question is, how XML is used within a service architecture. If XML is used as a data representation format within the Web service engine, application logic has to process XML. Processing XML in this context means to use the features of XML, such as the nested tree structure and the document order, to capture the complexity of an application.
The XML language XQuery [W3C05d] is a functional query language providing declarative means to either query, navigate or generate XML data. Furthermore, the W3C is working on an XML Update proposal [CFR06].

XML processing applications include, for example, access either local and remote data, transactions, message and error handling. An XML processing framework (in our case, the XML processing language) has to provide means for a developer making the use of XML convenient. XML should not be an obstacle but an efficient concept of data representation and processing.

Furthermore, the XML processing framework has to provide by an an efficient runtime environment. Type system, query processing engine, and the storage of to be integrated into a common framework.

XML message handling If XML messages are used to simply perform an XML-based remote procedure call, the XML induced overhead is considerable. Instead of marshalling Java objects into XML messages and unmarshalling them at the other side, *real* XML messages should be sent. Real XML messages capture the complexity of an application by making use of the properties of XML.

Therefore, the XML processing language presented in this thesis has to provide an intuitive way of sending an responding XML messages. In return, the runtime environment has to implement the necessary marshalling transformations depending on the message format used.

Service Oriented Architecture The term *service oriented architecture* (SOA) denotes the concept of connecting loosely coupled applications via a network using XML messages. The topic of process communication is old and has been addressed previously several times (e.g., CORBA [Obj05], DCOM [Mic05a]). These approaches wrap the remote procedure call but do not provide the necessary integration between different programming concepts. Additional software layers were added, which neither improve scalability, performance nor maintainability.

XML offers the possibility to easily define new data formats. The key element of a SOA is the possibility to describe the interaction between different applications on an abstract level. Furthermore, applications are deployed on the Internet. In the Internet, connections are insecure and unreliable, communication partners are sometimes anonymous and not trustworthy. As a consequence, new means are necessary to describe the interaction between applications. Luca Cardelli defined an initial calculus [CD99] describing the basic primitives for expressing the application integration. If the XML based approach of SOA is reduced merely to a technical interface description of the service interface, no qualitative improvement compared to the previous approaches is achieved. The XML provided means to integrate different services, the semantically enhanced service description and its integration into an XML processing framework has not been achieved yet.

As a consequence, the runtime environment of an XML application has to provide the possibility to integrate different communication protocols easily.

Optimization The processing of XML data raises several optimization problems. The general term optimization typically addresses the non-functional requirement performance (see section 7.1). The following aspects have to be taken into account:

- **XML data representation** Depending on the XML processing technique, an appropriate type of XML data representation has to be chosen. Depending on the type of representation, query and transaction processing, persistent data storage, and necessary data transformation have to be taken into account. The different representation types are described in section 3.2.
- **Query Evaluation** If XQuery is used as an expression language, a XQuery runtime environment has to be integrated into the XML processing application, or say, embedded into the XML processing framework. The processing environment has to provide context information allowing the XQuery engine to optimize expressions. This context information could for example contain external variables, namespace declarations, user defined functions, index structures, or schema information.
- **Integration** The integration of other services by using the SOAP protocol [W3C04c] is necessary. Furthermore, it has to be possible to optimize services by integrating different services into a common processing framework. In some use cases the overhead generated by the loose coupling of the SOAP messaging protocol might be to expensive and a native integration into the XML processing environment becomes necessary.

Therefore, the XML processing language itself should be as declarative as possible. Optimizations have to be addresses by the XML processing environment and not the programmer of the application logic.

2.1.2 Runtime-System

The second aspect addressed by this thesis in part II is the implementation of an XL runtime system. In order to execute XL services, a runtime system is necessary which provides an integrated framework of compiler, XML storage, an XL virtual machine (executing the XL services), and for example a HTTP server.
The XL language defines (most of) the functional requirements of the runtime system. Furthermore, the non-functional requirements of the language framework have to be specified and put into practice.
Since the paradigm of a service oriented architecture implies a general applicability of services, the runtime environment itself has to be scalable in the first place. Section 7 describes the architecture and design of the XL runtime system, focusing on the nonfunctional requirement scalability.

2.2 Example Application – Online Shop

For the scope of this thesis a sample Web service application is used. This sample application has to serve several purposes. It has to represent the typical use-case of online XML Web

services. Applications providing an online service can be divided into *online transaction processing* (OLTP) and *online analytic processing* (OLAP). Since this thesis focuses on interactive applications providing dynamic services, the example used here belongs to the OLTP category. The second requirement for an example Web service in this thesis is the XML processing. The application used must illustrate either advantages and disadvantages of XML as a form of data representation and exchange format.

Application Scenario An online shop offers the possibility to buy books, proceedings, or journals, say any type of printed publication. After registering, a customer should have the possibility to query the offers according to a some criteria. The customer shopping cart is managed by the system. A registered customer can add or delete items from his cart. A purchase is completed by either going to the counter or by a timeout.
The actor of the sample application used here is predominantly the customer, say, the interface used by a potential customer is specified and implemented. Shop administration is neglected, for now. In this example, XML is used as a format of data representation and data exchange. The arguments of each operation are described by a single, possibly complex, XML element. In the following, the operations of the sample online shop are characterized in more detail:

- `registerCustomer` A new customer XML element is created and inserted into the customer database. The customer XML element includes a username, password, real name, address, account balance and the date of the last visit.
  A typical input XML parameter would be:

```
<customer>
    <uname>Andreas</uname>
    <passwd>abc</passwd>
    <fname> Andreas </fname>
    <lname> Gruenhagen </lname>
    <address>
        <street>Strasse</street>
        <city>Dorf</city>
        <zip>01234</zip>
        <country_id>11</country_id>
    </address>
    <phone>012/121333345543</phone>
    <email>here@home.com</email>
    <birthdate>0010-01-01</birthdate>
    <data>premium customer</data>
</customer>
```

- `queryItems` The item database is queried. The customer can specify various predicates, like price, item id, or keywords to be contained in the name or description. The way how user preferences are expressed certainly varies depending on the used query technologies (e.g., Java Beans, SQL, XQuery).

  The following listings provide some example messages sent to an XL Web service expressing the preferences by using query by example:

  - Search for a book with the title ”The Hitch Hiker’s Guide to the Galaxy”

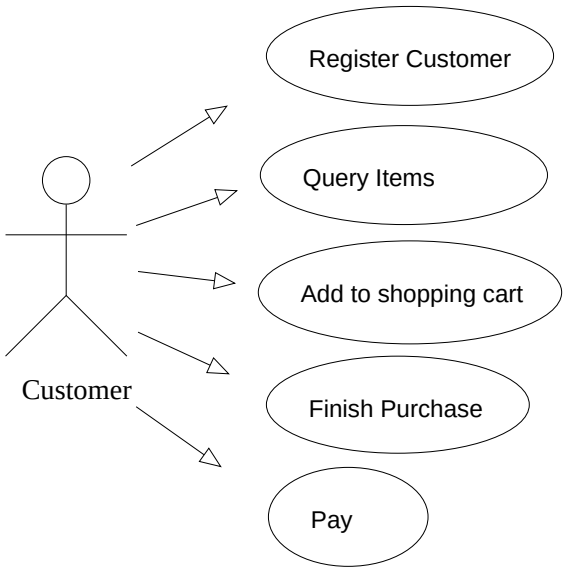


Figure 2.1: *Use case diagram for the example application "Online Shop"*

```
<query>
   <type>book</type>
   <name>The Hitch Hiker's Guide to the Galaxy</name>
</query>
```

– Search for items which contain the word ”warehouse” in their description

```
<query>
    <subject>warehouse</subject>
</query>
```

- `addToCart` By calling this operation the user can either start a new shopping session and add an item or add another item to an existing shopping cart. A valid username and password have to be provided and the requested item has to be available. If this is the case, a new lineitem is added to the variable representing the shopping cart. Additionally, the customer database needs to be updated (last visit) and available stock in the item database is adjusted automatically.

- `goToCounter` This operation completes a shopping session by creating an order XML element which is inserted into a persistent variable. The order XML element includes the customer name and address, the ship type, and for example an order date. Finally, the account balance of the customer has to be adjusted.

- `pay` After purchasing several items at the XL store a customer is requested to pay. In the XL shop this simply includes an update of the corresponding account balance XML element in the customer database.

This shop application contains readonly operations as well as update operations. The terms ”readonly” and ”update” refer to the state of a service.

2.2.1 Functional Requirements

In the following the functional requirements of the different operations are explained in more detail.

- **Query Items:** Parameters: query by example xml:

```
<item>
   <type>book</type>
   <keyword>Harry Potter</keyword>
</item>
```

  All items with the title "The Hitchhiker's Guide to the Galaxy" are returned.

```
<item>
  <type>book</type>
   <author>Douglas Adams</author>
   <title> The Hitchhiker's Guide to the Galaxy</title>
</item>
```

  User preferences, or say, the parameters of a database query, on the Web are usually specified by filling out a form. The preferences expressed in the examples above resemble the *query by example* concept.

- **Add reservation** A customer can make a reservation for a certain performance. The customer has to specify the performance itself, a name, the number of places. In return a ticket-Id is returned. This ticket-id uniquely identifies the reservation.
  A performance should not be overbooked. A user authentication is not required.

- **Query reservation** Used by the theater to query the reservations made.

- **Update reservation (optional)** Used by the theater to either delete or update reservations

- **Update performances (optional)** Used by the theater to either delete or update performances

2.2.2 Non-Functional Requirements

- **Event description** Each item (e.g., book, journal, CD, etc.) is described by a set of attributes. Some attributes are common to all items like title and date, but some attributes are specific for a certain item or a certain item type. The exact description schema of the different item types should be flexible.

- **Service integration** It must easily be possible to integrate the new service into an existing application. Service invocation should be based on common standards.

- **Performance** The service should execute each operation of the service within less than x sec given a typical workload.

- **Storage** It must be possible to either represent the state of the shopping service persistently by using a rel. database, or transiently in main memory.

The corresponding XL program is listed in section 4.6

Chapter 3

XML- & Web-Technologies

3.1 Core XML

XML is a data markup language based on simple character files (characters are atomic text units [ISO03b]). The term *markup* referrs to the use of the verb mark text in order to structure a text and to provide instructions or annotations for a compositor or typist on how a particular text should printed. The markup provides instructions on how the contained elements are to be interpreted.
The core XML standard itself simply specifies the general entities which can be part of an XML document, as for example XML elements, processing instructions, comments, and whitespace [W3C04a]. If data can be successfully parsed according to the XML core standard, it is *well-formed*.
The design goals for XML are described very briefly by the XML Core Working Group and shall be quoted here [W3C04a]:

1. XML shall be straightforwardly usable over the Internet.
2. XML shall support a wide variety of applications.
3. XML shall be compatible with SGML [ISO86].
4. It shall be easy to write programs which process XML documents.
5. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
6. XML documents should be human-legible and reasonably clear.
7. The XML design should be prepared quickly.
8. The design of XML shall be formal and concise.
9. XML documents shall be easy to create.
10. Terseness in XML markup is of minimal importance.

An XML *document* starts with a prolog XML tag naming for example the XML version and the encoding used. After the prolog, an XML document contains one root element. An XML element is delimit by corresponding start- and an end-tags. The following example illustrates the XML syntax:

```
<?xml version="1.1" encoding="UTF-8"?>
<employee xmlns="http://myontology.com/people"
          xmlns:iso="http://iso.org/std"
                    id="0815">
  <name>Andreas</name>
  <section>computer science</section>
  <iso:id> x-01234 </iso:id>
</employee>
```

This XML text models an employee entity. Start- and end-tag of an XML element are delimited by the two brackets: `< [ name ] >`. The closing end-tag contains an additional slash: `</ [ name ] >`. Each name is defined within a namespace, which could be either a default namespace, or an explicitly defined namespace identified by an URI [BL94]. In order to combine different namespace a prefix can associated with a namespace. In the example above, the tag names ”employee”, ”name”, and ”section” are defined in the namespace identified by the URI ”http://myontology.com/people”. The prefix ”iso” is associated to a different namespace identified by the URI ”http://iso.org/std”.
As illustrated by the previous example, XML contains nested elements. The content of an element could be plain text, nested child elements, or a mixture of both. Like in HTML, mixing child elements and text arbitrarily is allowed in XML. Another component of the XML datamodel are attributes. Like child elements, attributes can be added to an XML element by including the name - value pairs into the start-tag. Like elements, attributes have a name within a namespace. The values associated to attribute names are restricted to plain text and are delimited by apostrophes. Attributes could either be used to add additional content, like employee id attribute in the previous example, or attributes could be used to declare namespaces or link elements by references.
Furthermore, the XML data modell includes several additional components:

- XML comments:

  ```
  <!-- this is an XML comment -->
  ```

- XML Processing instructions providing additional information for applications processing the XML data.

  ```
  <?xl build_index="no" ?>
  ```

  In the context of an XL or XQuery application processing XML, a processing instruction could be used to exclude certain elements from an index structure.

- Character Data. In order to embed arbitrary character data which should not be interpreted as XML markup, the character data element could be used. This could be used to include XML data into another XML document without interpreting it as markup. Furthermore, binary data could be included into an XML document by using a Base64 encoding [Jos03].

```
<![CDATA[ <?xml version="1.1" encoding="UTF-8"?> ]]>
```

Predecessor of XML was the *Standard Generalized Markup Language* (SGML) [ISO86]. SGML used to be less strict on several aspects compared to XML. SGML is more flexible since it does not enforce the strictly nesting of element. On the contrary, a document type definition is mandatory in SGML while it is optional in XML.
The XML standard does not make any assumptions on how to evaluate the the data itself. XML provides means to easily define new data structures conforming to the XML core standard. Since the first W3C recommendation for XML [W3C98], a huge set of XML related standards has been setup. In the following a brief overview of XML technology will be given.
By strictly nesting elements XML enforces a tree structure for each element. Using XML, structuring heterogeneous data is very simple. The hierarchical tree structure of XML can be used to express various relationships (e.g., listing or grouping of elements, part of relationships, aggregations). XML elements can be combined almost arbitrarily, an XML element can contain arbitrary content as long as it is well-formed XML. Additionally, XML provides a notion of identifiers and references, which could be used to add links inside one document, as well as between separate documents.
The drawback of XML is the overhead caused by the embedded structure information. The markup code describing the data structure requires space. If the represented data is structured very strictly, using the same pattern repeatedly, the repeated markup code is unnecessary redundancy. A typical example of rigidly structured data is a JPEG picture [Wal91]. JPEG files contain a sequence of transformed pixel color values. Since each 8×8 block of pixels is treated uniformly, additional structure information is not required. Unlike pixel graphics, vector graphic pictures are not as rigidly structured and consequently use XML [W3C05b].

3.2 XML Representation

3.2.1 Data Representation

As for any processed data, XML as it is stored on hard disc has to be transformed into a representation used by the particular programming language. The term *data transformation* denotes the translation from one conceptual modell into another one. If for example a Web service is implemented using Java, the XML data has to be translated into a Java object modell.
Usually, data transformation are cumbersome and avoided when possible. Data transformations are:

- **difficult and error prune** In order to apply a data transformation the corresponding type definitions in both type concepts have to be kept in sync. If another XML element is added to an XML schema, the corresponding Java class definition and the mapping between both representations has to be adapted.
  XML data can be represented by a generic Java object modell, but for most use cases the generic XML object modell is not suitable. Usually, you do want to represent an XML element ”employee” not as a generic XML Java object, but as an instance of an employee class. The different XML representations are discussed in section 3.2.

  The XML to Java transformations, say, XML employee element to Java employee object, is lossy as certain features of one type system cannot be translated into the other one. For example, a Java object employee does not contain a document order as expressed

by an employee XML element. Another example is the type inheritance used by the XML Schema type system [W3C06b]. XML Schema allows to derive new type by either restricting or extending existing types. If a new type is defined by extending an existing XML element, the new type cannot serve as a substitute for the base type.

Comparing XML and Java as a type of data representation is difficult. Even though, both are used to represent data and are used to substitute each other in many cases, XML and Java were designed for different purposes and different use cases. Typical use cases for XML data are: parse, store, or send and receive via HTTP. XML data . The data within a Java objects, the attributes, are used to describe the state of an object. Both of these two descriptions are oversimplified, but still the different approaches of how data representation could differ is illustrated.

Further problems are caused the different access patterns or the different access granularity of the type systems. If an XML data source is used, but the XML data inside the application is represented by objects, e.g., Java Beans, the multi user synchronization at the data source is difficult. While one application addresses for example a single item in a product database another application could update or fetch the whole product database represented in the XML source as an individual XML element ”products”. The data representation in the application logic typically does not reflect the tree structure of the XML data.

- expensive Data transformations have to be considered expensive, as they typically do not provide a view on the data but create a deep copy. Each data transformation adds another software layer implementing the transformation. The actual costs in terms of time or memory consumption depend on the transformation applied.

XML data can be represented differently depending either on storage type available (e.g., main memory, hard disc, database) and the use cases. It has to be distinguished between three basic concepts used to represent and to process XML: text (characters or large object), parsing events, or as a tree modell. In the following these three concepts are discussed.

3.2.2 Text

The initial representation of XML is usually text stored as a sequence of characters. For text documents, like HTML Web pages [W3C06a] or SVG [W3C05b] pictures, the text representation of XML is suitable. Documents are typically processed as a whole. For sending XML via network connections, the text representation is the simplest to process. Furthermore, the text representation of XML is human readable and can be edited by basically every texteditor.

3.2.3 Event Based

Due to the simple structure of XML (nested elements, start and end tags) the text representation can be parsed easily. Since XML has simple nested structure, only a limited number of possible parsing events are possible (e.g., *start-document*, *start element*, or *end element*, *processing instruction*, etc..).
The concept of representing XML by using events is to provide an interface callback functions each processing a certain event type. The event based XML representation therefore also

denotes a certain processing concept. The *Simple API for XML* [SAX04] library uses this concept. The advantages and disadvantages of this concept are straightforward. Since the XML data is not represented as a whole, by processing the XML parsing events as they occur XML data flows through the process. Memory or storage requirements are reduced, since only the currently processed event needs to be represented in the process.
The disadvantage of using a callback functions is you cannot navigate. The XML parsing events are generated on the fly in document order. It is not possible to navigate within the XML data. Parsing events have to be processed in the sequence they occur.
In order to overcome this disadvantage parsing events are materialised. Each event is stored using a token. XML data is represented by a sequence of tokens which could be used to replay the initial parsing events. A token sequence can be either represented in main memory, or could be stored by a database. A single token represents the parsing event, including context information. In XML, the name of a single element is part of a namespace. This namespace is typically not given in the element itself but abbreviated by a prefix. The association between prefix and namespace is knowledge provided by previously parsed events. If the tokens provide the necessary context information, as for example the namespace or parent and sibling information, it is possible execute path expressions by simply navigating in the token sequence. Additionally, it is possible to use indices in order to select certain elements without processing a complete XML document. The use of token sequences makes it possible to enrich the XML data with additional information:

- type information could be added, for example by including additional token
- further information required by the processing engine (e.g., XQuery) could be added

Furthermore, the representation of token sequences could be implemented efficiently by reusing token objects.
The XQuery engine used by XL uses a token based XML representation [FHK$^+$03].

3.2.4 Object Model

Based on the event processing model described in the previous section a standard document object model (DOM) has been set up for Java and ECMAScript [DOM04]. Other languages adapted the model later on. The object model is generic model for representing arbitrary XML data. In Java, generic classes are provided representing the different components of the XML data, as for example document, element, attribute etc. In order to represent XML data using DOM a SAX interface is used to generate a tree of objects representing a certain XML document or element.
The document object model contains all properties of initial XML data, including parent child relationships, attributes and document order. But, for many use cases the DOM has several major drawbacks. The complete XML document, represented as text, has to be parsed in order to create a DOM representation. Parsing XML is time-consuming, but since a transformation has to be applied it cannot be avoided. DOM is a generic XML representation, evaluating expressions, like XPath or FWLOR-XQuery expressions, is combersome since every step has to be programmed explicitly. Expression evaluation is not declarative. Furthermore, DOM does not provide an advanced type information. The content of simple XML elements is plain text. Primitive types, like integer, double, or date are not supported. In order to apply indexing schemes, XQuery additionally requires to maintain node identifiers.

3.2.5 Persistent XML Storage

Since XML is being used widespread thoughout several application domains, the persistent storage of XML data became more and more important. Two main types of systems could be distinguished, namely:

- native XML data stores, such as Natix [KM00] or XML Transaction Coordinator [HH04].
- relational database system enhanced in order to store and manipulate XML (e.g., DB2 xml extender)

The mapping of XML data to the underlying storage has to overcome two basic problems addresses by several publications. First, the document order is a property of XML data which has to be maintained within the different representations. In a native XML database, document order is typically a build in feature. Since relational data has no implicit ordering, the order information has to be explicitly added and maintained throughout several update operations. Different ordering schemes have been proposed by for example [TVB+02] and [OOP+04].
The second problem which has to be addressed in order to design an efficient XML storage is the access granularity. The two extremes could be described as follows:

- XML data is stored as a whole document in the file system or as a character large object (CLOB) in a database. Lock and access granularity are very coarse grained. Typically, existing relational database systems take this approach. The DB2 Extender [CX00] provides means to either store XML as a single object or shredded into several relational tables containing the content and not the XML markup anymore.
- XML data is broken up into many small elements (e.g., tokens [DK05] or tuples in a relational table [FK99]). Lock and access granularity are very fine grained.

Depending on the targeted use case one the two alternative approaches serves better. In [KM00] and [DK05] the authors trade between the two extremes by providing an adaptive XML storage.

3.3 Query Processing and Typing

3.3.1 XML Schema information

The XML core standard is enhanced by two different schema specification languages.

DTD The initial document type declaration (DTD), described in [W3C98], was designed to describe a tree structure implemented by a XML documents. DTDs specify the grammar of an XML document, containing for example element type and attribute list declarations. If an XML document conforms to the restriction specified by a DTD, it is called *valid*.

XML Schema The XML *schema* standard [W3C06b] in many ways supercedes the initial DTDs. XML schema provides means to specify either simple or complex types and associating these types to qualified names (QNames) within a namespace. In contrast to DTDs, XML schema is a type system, as is distinguishes between the name of a type specification and the actual tag name in an XML document conforming to the type.

XML schema contains a set of 19 build in primitive, say, atomic types, as for instance string, boolean, float, and date. For specifying complex types, XML schema provides

different means: sequences of subelements, alternative subelements, ordering or occurrence restrictions. Furthermore XML schema includes anonymous types, nil values, and type annotations. XML schema itself is described using XML syntax.

The following listings provides two small example XML schema definitions:

- Body temperature type:

```
<simpleType name='bodyTempCelsius'>
  <restriction base='decimal'>
    <totalDigits value='3'/>
    <fractionDigits value='1'/>
    <minInclusive value='35.0'/>
    <maxInclusive value='42.5'/>
  </restriction>
</simpleType>
```

- Enumeration type "size"

```
<xsd:simpleType name="size">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="XS"/>
    <xsd:enumeration value="S"/>
    <xsd:enumeration value="M"/>
    <xsd:enumeration value="L"/>
    <xsd:enumeration value="XL"/>
  </xsd:restriction>
</xsd:simpleType>
```

XML schema provides means to define abstract type definitions and to derive new types. New types can either be created by restricting existing types, as shown in the examples above, or by extending given types. In the definition of a new XML schema type it can be specified whether new type can be derived by either extension or restriction. Instances of complex super-types, by default, cannot be substituted by instances of derived types.

3.3.2 XQuery

XQuery is versatile functional query language addressing XML data processing [W3C05d]. XQuery is a declarative language encompassing XPath expressions, XML element construction, element grouping and ordering, strong XML Schema typing, user defined functions etc. Certainly XQuery is inspired by previous languages, as for example database query languages (e.g., SQL [ISO03a], OQL [CB00]), other XML query languages (e.g., XQL [RLS98], Quilt [CRF00]). For navigation XML documents, XPath [W3C99] is used in XQuery. In the following paragraphs, the important components of XQuery will be briefly introduced. A detailed description of the XQuery language is given on the W3C webpages [W3C05d], includes also features omitted here (e.g., error handling, modules).

Constructors & Types

XML Constructors XQuery distinguishes between direct and computed element construction. For direct construction, XML data could be simply provided in standard XML notation

```
<example range="ae">
  <small> abcde </small>
  <text> { $var/data } </text>
</example>
```

```
element example {
      attribute range {"ae" },
      element small   { "abcde" },
      element text    { $var/data }
}
```

Listing 3.1: *XML element construction*

(left part of listing 3.1). Pairs of curly brace character { ... } are used to embed arbitrary expressions into the created XML data. In listing 3.1 a path expression using an external variable $var illustrates the concept. At runtime, the content of the external variable has to be provided.

The computed element construction is shown on the right of listing 3.1. In this case a set of explicit keywords (e.g., element, attribute) is used to specify the different XML elements, attributes, etc. Computed XML construction could be usedfor example to generate XML elements whose name or namespace is determined dynamically at runtime.

Like the XML elements and attributes shown in listing 3.1, processing instructions, namespace declarations, or comments can be constructed likewise.

XQuery Type System XQuery is a strongly typed language with a type system based on [W3C06b] as described by the previous paragraph. Based on the basic types, XQuery contains a set of sequence types. Additionally XQuery defines a notation, on how to specify sequence types. The term `element()` refers to an arbitrary element node, `element(*, xns:exampleType)` refers to an element with an arbitrary name and a type annotation xns:exampleType. The term `item()+` refers to a sequence of one or more nodes or atomic values.

Path & Sequence Expressions

A path expression is a sequence of navigation steps within the tree structure of XML elements. The path expressions navigates within the XML tree structure and returns as a result an arbitrary length sequence of elements, or possibly subtrees, in document order.

In total XQuery defines seven forward and five reverse axis explained in table 3.1. Each step in an XPath expressions, using one of the twelve axis, can be associated with a predicate. Each step in a path expression is executed by evaluating a sequence of input nodes. Each axis type specifies a set of nodes in the XML tree relative to the current input node. Table 3.1 lists the possible axis types. For the more frequently used axis steps an abbreviated more convenient syntax is defined.

Typically, predicates are used to specify the name of a target node, or to select certain nodes within a sequence. The expression "*$db/item[weight lt 10]/name*" illustrates the use of predicates, within path expressions. In this case, the names of all items with a weight less then 10 are selected. Additionally to the predicates on the element names, predicates are embedded into path expressions by using square brackets. The same expression using the non-abbreviated syntax would look like: "*$db/child::item[child::weight lt 10]/child::name*"

Notation	Abbreviation	Result	Implementation
$x/child::*	$x/*	All direct child elements of the context node of the variable $x	mandatory
$x/descendant::*		All descendants of the context node of variable $x (transitive closure of the child axis)	mandatory
$x/descendant-or-self::*	$x//*	The context node of variable $x and all descendants	mandatory
$x/parent::*	$x/..	The parent node of the context node of variable $x	mandatory
$x/self::*		The context node of variable $x	mandatory
$x/attribute::*	$x/@*	All attributes of the context node of variable $x	mandatory
$x/ancestor::*		All ancestor nodes of the context node of variable $x (transitive closure of the parent axis)	optional
$x/ancestor-or-self::*		The context node of variable $x and all ancestors	optional
$x/following-sibling::*		All in document order following siblings of the context node of variable $x	optional
$x/following::*		All descendants of the root node which in document order follow the context node of variable $x	optional
$x/preceding-sibling::*		All in document order preceding siblings of the context node of variable $x	optional
$x/preceding::*		All descendants of the root node which in document order precede the context node of variable $x	optional

Table 3.1: *XPath navigation axis*

Arithmetic & Logical Expressions

XQuery supports the arithmetic expressions using addition, subtraction, multiplication, division, and modulus. For evaluating an expression, the two operants are evaluated, atomized, and casted to double values. If either of the two operants contains a sequence of more then one node, an error is raised. If either the two operants contains an empty sequence, the result is a empty sequence. The expression *$item/price - $item/discount* determines the atomic value of the two operants and returns a double result value. Note, in this example the two operants price and discount are implicitly atomiced. Say, the start- and end-tags (in this example <*price*> ... </*price*> and <*discount*> ... </*discount*>) as well as attributes are removed and the content atomized.
Similar to arithmetic expressions, logical expressions using the keywords "*and*" and "*or*", and value comparison is possible in XQuery. The path expression *$db//item[price lt 100 and weight lt 10]* for example selects all item whose price is less then 100. In this example, a path expression, using the abbreviated syntax for the descendants-or-self axis (the double slash: *//*), is combined with a predicate.

FLWOR Expressions

A complex feature of XQuery are the so called FLWOR expressions (pronounced "flower"), corresponding to the five keywords for - let - where - order by - return. The syntax of FLWOR expressions is described as follows[1]:

```
( for $Var [at $i] in <expression> (, $Var in <expression>)*
| let $Var := <expression> (, $Var := <expression>))+
( where <expression> )?
(order by   <expression> (, <expression>)*)?
return <expression>
```

For and Let The *For* and *Let* clauses bind XML nodes to local variables within the XQuery expression. The *For* clause evaluates a subexpression (for-expression) and binds iteratively each returned node to a local variable. This local variable can be used to iterate through the sequence of elements returned by the expression. The optional term *at $i* allows to specify an index variable. If specified the index variable contains the postion of the currently processed element within the sequence of elements returned by the for-expression. The *Let* clause evaluates a subexpression and binds the the complete result to the local variable. *For* and *Let* clauses generate a sequence of tuples. Each tuple represents a set of variable bindings which are valid for all subsequent clauses . Figure 3.3.2 illustrates the different semantics of dependent for and let clauses.

Where The *Where* clause specifies a predicate evaluated for each generated set of variable bindings.

Order By If an *Order By* clause is specified, a value based ordering is applied. The value which determines the ordering is specified by an expression using the variable bindings defined by the *For* and *Let* clauses. If no *Order By* clause is specified, the nodes have to be returned in document order.

[1] the exact syntax of FLWOR expressions is given in [W3C05d], section 3.8

Input variable $X :=

```
<x>
  <a> <b>1</b>
  </a>
  <a> <b>2</b>
  </a>
</x>
```

```
for $A in $X//a
let $B := $A//b
return $A, $B
```

```
<a><b>1</b></a>,<b>1</b>
<a><b>2</b></a>,<b>2</b>
```

```
for $A in $X//a
let $B := $X//b
return $A, $B
```

```
<a><b>1</b></a>,<b>1</b>,<b>2</b>
<a><b>2</b></a>,<b>1</b>,<b>2</b>
```

Figure 3.1: *Different FLWOR expressions, the left expression contains a dependent supexpression in the let clause. The right expression has a different let clause, therefore the content of $B does not depend on $A.*

Return The *Return* clause finally specifies the generated output. The *Return* clause is evaluated once for each set of variable bindings which pass the *Where* clause.

XQuery is currently, May 2006, a W3C Candidate Recommendation. Recent changes and ideas were presented in a SIGMOD record 2005 [EM05]. Currently several implementations of XQuery standard are available, as for example Galax [FSC+03], Saxon [Sax06], and the query engine used by XL BEA/XQRL [FHK+03].

3.4 Extensible Stylesheet Language

The *Extensible Stylesheet Language (XSL)* denotes a package of functional XML related languages [W3C05a]. XSL is used to transform XML (XSL Transformations) and to format XML (XSL Formatting Objects) according to rules.
XSL Transformations (XSLT) allows to formulate recursive rules. Each rule contains a predicate, which matched against the input XML data. If the specified predicate is evaluated to true, some new XML data is generated. XSLT uses of the language XPath [W3C99] to express the predicates.
Formatting Objects (XSL-FO) part of XSL provides mean for transforming XML data into a more usable form for human perception. This could be for example plain text, a postscript document, or a picture.
In the open source Apache project, an implementation of XSL is available [Apa05b].

3.5 Service Oriented Architecture

The term *service oriented architecture* (SOA) denotes a software architecture composed of loosely coupled services connected by message passing interfaces. Since the common terms

service, *message*, or *architecture* are used very frequently in different contexts, an exact definition is difficult. In this thesis the definition setup by the W3C [W3C04d] are used. In the following sections, the relevant terms are defined.

3.5.1 Architecture

The term *architecture* denotes the *fundamental organization of a system, expressed by its components, their dependencies, and the principles governing its design and evolution* [IEE00]. The architecture describes the abstract concept used to either build or use a system. An architecture has to:

- remain valid, independent of the perspective (user, developer) or the implementation
- contain all major entities and their relationships
- sketch the *big picture*.

3.5.2 Service

The W3C defines a service as an "abstract resource that represents a capability of performing tasks that represents a coherent functionality" ([W3C04d], section 2.3.2.10). Associated to a service, several further elements have to be defined. In order to perform a certain task, a services is invoked and executed. Services could incorporate a persistent state, The services term definition abstracts from:

- How a service is invoked: synchronous or asynchronous
- Where a service is invoked: local or remote
- How a service is implemented: used programming languages, or hardware

In the following the term *Web service* denotes a service provided to remote clients on a network, say, the Internet, using common standards.
The Web service concept, say, the Service oriented Architecture, is based on the idea of simple autonomous applications communication by exchanging XML messages.

WSDL

The technical interface of a Web service is simple compared to the complex interface of classes in object oriented programming.
The public Web service *interface*, as it is described by WSDL [W3C01], contains a set of operations.
The WSDL interface description is split up into five different aspects each describe by a corresponding XML elements:

- **types** Type definitions based on XML schema [W3C06b]. Types are named and are used to describe input and output messages of a service.
- **messages** An abstract message specification naming the different message parts.

- **portType** A port type describes the abstract signature of an operation. An operation is identified by a unique name within the interface. WSDL distinguishes between four different invocation patterns (one-way, request-response, solicit-response, notification). Depending on the pattern the type of input message is specified. Furthermore, it is possible to specify the type of a potential error message.

- **binding** The binding defines the protocol, the message type, and all further information required to invoke a certain implemention of a port type. Since the same Java service could be invoked either by SOAP messages or Java RMI (Remote Method Invocation), several different bindings for a port type can exist.

- **service** The final service element aggregates a set of related ports in order to provide a Web service. A port specifies a concrete address of a binding.

The listing 3.2 provides an example WSDL document describing the interface of the small online shop application described in section 2.2. Using the Axis toolkit provided by the Apache project [Apa06], WSDL description for given Java Bean can be generated easily.

SOAP Messages

The SOAP messaging protocol [W3C04c] is currently a widespread standard for XML based messaging. SOAP messages have to be well formed XML, valid according to the SOAP schema. SOAP messages are structured very straightforward, a message contains an optional header and a mandatory body element. The initial design of SOAP addressed a use case of messages being received and send via several intermediate servers to a final destination. Header entries were meant to address intermediate servers forwarding the message and the final receiver. The body element contains the actual payload of the message and only addresses the final receiver of a message. This distinction is partly obsolete since intermediate router on the SOAP level are rarely used.
The typical use case of SOAP message is a remote procedure call execute on a Web service engine. SOAP header elements are typically used for authentication or conversation handling. The SOAP body element contains the encoded operation call itself including the invocation parameters and the operation name. Depending on the encoding of SOAP message, the operation name could be part of the message (*RPC* encoding) or it can be transmitted by different means (*document* encoding), for example as a HTTP header element. The SOAP standard explicitly does not specify the content of the body element. The encoding only distinguishes between the *literal* and *encoded* style, either the XML encoding is provided by the application itself or by the SOAP framework. Typically, the combination RPC / encoded is used.
SOAP header and body together are wrapped into a SOAP envelope. In section 4.3 some example SOAP messages are provided.
The SOAP message standard serves well as an all purpose XML messaging protocol. SOAP messages could be transmitted using different network protocols (commonly HTTP [RF99], but also SMTP [Pos82]), binary attachments are also possible.
The advantage of SOAP is its flexibility, any XML data, representing either plain text or a complex data structure, can be wrapped into a SOAP envelope. The advantages SOAP provides depend on the targeted use case. Compared to a simple Java remote procedure call, SOAP message itself adds considerable overhead required to serialize Java objects into XML an back. XML SOAP messages are an integration device connecting different applications.

```
<?xml version="1.0"?>
<definitions name="MyShop"
targetNamespace="http://myshop.com/shop.wsdl"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns="http://schemas.xmlsoap.org/wsdl/">
 <types>
    <schema targetNamespace="http://myshop.com/shop.xsd"
           xmlns="http://www.w3.org/2000/10/XMLSchema">
       <element name="queryType">
            <complexType>    <sequence>
               <element name="type" type="string"/>
               <any minOccurs="1" maxOccurs="unbounded"/>
               </sequence>  </complexType>
       </element>
       <element name="queryAnswerType">
            <complexType> <sequence>
                <element name="item" minOccurs="0"
                                     maxOccurs="unbounded">
               <complexType> <any minOccurs="1" maxOccurs="1"/>
               </complexType>
                 </element>
               </sequence> </complexType>
        </element>
        <!-- further type definitions -->
     </schema>
 </types>
 <message name="queryItemsInput">
      <part name="body" element="queryType"/>
 </message>
 <message name="queryItemsOutput">
     <part name="body" element="queryAnswerType"/>
 </message>
  <!-- further message definitions -->
 <portType name="queryItemsPortType">
    <operation name="queryItems">
       <input message="queryItemsInput"/>
       <output message="queryItemsOutput"/>
    </operation>
 </portType>
   <!-- further port type definitions -->
  <binding name="ShopSoapBinding" type="queryItemsPortType">
     <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http"/>
       <operation name="queryItems">
        <soap:operation soapAction="http://myshop.com/queryItems"/>
           <input> <soap:body use="literal"/></input>
           <output><soap:body use="literal"/></output>
        </operation>
    </binding>
    <service name="MyShop">
        <documentation>My online shop</documentation>
        <port name="MyShopPort" binding="ShopSoapBinding">
           <soap:address location="http://myshop.com/"/>
        </port>
    </service>
</definitions>
```

Listing 3.2: *WSDL document describing an online shop service*

- XML Protocol *SOAP* at http://www.w3.org/2000/xp/Group
- Web Services Description Language*WSDL* at http://www.w3.org/2002/ws/desc/
- *Business Process Execution Language (BPEL)* at http://www.ibm.com/developerworks/library/ws-bpel/
- *Semantic Markup for Web Services (OWL-S)* at http://www.daml.org/services/owl-s/1.1/
- UDDI

3.6 Web Application Server

The application server domain is very big. All different types of J2EE servers from different vendors (e.g., BEA Weblogic [Jac03] or IBM WebSphere [IBM]) could be summed up by the term application server. In the following two subsection just the two most relevent projects, each providing a new programming language and a runtime system, are described.

3.6.1 $C\omega$

$C\omega$ (pronounced "C omega") is a experimental language designed by Microsoft [Mic05b; BMS05], targeting the ease of use for a programmer. Like in XL, programming should be easy, available concepts are reused and extended by additional features. $C\omega$ extends the concepts used by C# in several ways:

- The .NET common type system (CTS) is enhanced for handling data streams. $C\omega$ data streams are lazy lists, generated on demand. The dot operator '.', used in Java for example to navigate between objects, is extended to navigate and to express operations applied to each list element.
- $C\omega$ bridges the gap to XML by providing some implicit mappings between XML data and $C\omega$ objects.
- $C\omega$ does not include XQuery, but it provides declarative means to filter and navigate within data structures in a declarative way.
- $C\omega$ hides concurrency from the programmer. By declaring methods being invoked either synchronous or asynchronous, programmer do not have to care about managing different threads or semaphores. $C\omega$ provides a simple signal - slot mechanism to either send messages asynchronously or to wait for an call in return.

Like XL, $C\omega$ includes several interesting aspects relieving the programmer from handling technical details. Syntactically, $C\omega$ is build as an extension of the language $C^{\#}$.
$C\omega$ does not adapt the XML type system, say, XML is a foreign type system which has to be transformed into the $C\omega$ type system. XML is merely used as a means of communication and not used to capture complexity and or to express processing logic.

3.6.2 Business Process Execution Language for Web Services (BPEL4WS)

Business Process Execution Language for Web Services (BPEL4WS) [IBM03] uses the concept of an abstract interface definition, as described by WSDL [W3C01]. BPEL4WS combines existing partner links by using a set of service related primitives. The basic primitives available in BPEL4WS language include: sequences of statements (*sequence element*), concurrently executed statements (*flow elements*), control flow statements like switch or while, wait statements (*pick* element), as well as assign, send, and receive statements. BPEL4WS uses a pure XML syntax in order to specify a program.
A BPEL4WS program contains four basic sections:

- **variables** The variables used by the BPEL4WS program are declared in initial section. The type of a variable can be WSDL message type or any arbitrary XML schema type. Variables specified as a message type could be used as input or output of an operation call. Variables are not initialized.

- **partnerLinks** A partner link represents a connection to a different service used by the current BPEL4WS program. A partner link has a name, a type like the WSDL port type, and a role. The role specifies the part the described service plays in a certain interaction (e.g., customer, creditor). Partner links could be grouped by using a so called partner element.

- **compensationHandler** In order to provide transactional properties it is possible to use so called compensationHandler in BPEL4WS. Associated to a certain statement the corresponding undo operation is specified.

- **faultHandler** Like the C++ try-catch block, a fault handler used in BPEL4WS catches a certain error type.

- **correlationSets** Correlation sets describe in BPEL4WS the conversation handling used to implement stateful services.

- **process** All previous declarations and the actions provided by the service themself are contained in a process element. As mentioned previously, basic imperative constructs are available. Even though BPEL4WS could be extended the designated expression language is XPath. Complex expressions, like SQL [ISO03a] or XQuery [W3C05d], are not integrated into BPEL4WS.

The basic concept of BPEL4WS is to integrate different existing services at an abstract level. The processing logic expressed by a BPEL4WS program is focused on the integration of existing applications. Since typically only the integration logic is expressed by the BPEL, most of the business logic is encapsulated by further service invoked by the BPEL engine. Consequently, BPEL does not provide an integrated XML storage, since all necessary data is meant to be held by the underlying services.
Considering the overhead caused by XML SOAP messages, a fine grained integration of interactive applications could be difficult.
Currently implementations of the BPEL4WS standard are available from Oracle and IBM.

Chapter 4

XL Language

4.1 Introduction

XL is a forth generation programming language for XML processing. The XL language provides means to create, query and store XML data. The language is designed to be simple and straightforward, a programmer should directly use XML, not just for communication, but as a means to capture complexity. Like a complex Java class, XML data represents a complex structure which ought to be used by a programmer. In XL, XML is directly used and not hidden behind wrappers. In XL, literally speaking, everything is XML. XL adopts an XML type system based on XML Schema [W3C06b] and the XML query language XQuery [W3C05d]. In XL, the application logic is expressed using XML variables. XL could be briefly categorized by the following bulletpoints

- XML programming language, all variables contain XML. XML data, processed by XL, is not transformed into a different type system.
- XML query language and type system. The processing logic is directly expressed using the XML query language XQuery [W3C05d] and the XML Schema type system [W3C06b].
- Simple, easy to program, focuses on web service and not the XML marshaling code
- Standard compliant, programmer do not have to care about different XML related standards

The table 4.1 illustrates the differences between the different approaches used to build a common XML processing application like a Web service. A comparison is difficult, since all three address different use cases and can only be compared at a very abstract level:

- XL language as it is described by this section.
- BPEL (3.6.2)
- a Java based application using a common J2EE application server.

The criteria used to compare the different approaches are explained in detail in the previous problem section of this thesis, section 2.1.

Criteria	XL	BPEL	J2EE [Sun05b]
Storage Integration	Integrated XML storage	out of scope (see 3.6.2)	Object relational mappings are available for Java (e.g., [Hib05]), but not XML specific
XML Message handling	Integrated XML messaging		Integrated XML messaging, but Java specific
Service oriented Architecture	Integrated processing framework based on an XML data modell	BPEL integrates existing services, but does not provide an complete framework	Modularized processing framework based on an Java data modell, integration effort depends on the modules used. XML is merely used as an messaging format.
Optimization	the integrated processing framework provides a view of an entire service, therefore more optimizations are feasible	Optimization only at the integration level	Due to modularized architecture, optimization at the service level is possible. X

Table 4.1: *Comparison of alternative XML processing frameworks*

4.2 Web Services in XL

A Web service in XL generalizes the notion of an XQuery entity. In addition to a query, a Web service is identified by a unique universal resource identifier (URI) [BL94] – the target URI of a message. Like an XQuery entity, a Web service specification can contain a set of local function declarations plus a set of type and namespace definitions. The scope of the declared functions and namespaces is Web service itself, functions, namespaces, and namespace prefixes can be used within every XQuery expression in the Web service. In addition, a Web service specification in XL can contain:

1. local data declarations representing the state of a Web service
2. service-specific declarative clauses
3. specifications of the Web service operations

The concept expressed by the XL syntax is to combine existing C- or Java-like imperative programming elements (e.g., if, while and for), declarative XML processing elements as in XQuery, and Web service specific synchronous and asynchronous service invocation. XL is based on the ideas of a service calculus published by Luca Cardelli and Rowan Davies [CD99]. In the following, keywords are denoted in bold-face and non-terminals are enclosed in angle brackets. Optional parts are denoted in square brackets. Comments are prefixed by two exclamation marks and 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. In XL, as in XQuery, variables are always prefixed by a dollar sign $. The non-terminal `< expression >` for example represents an arbitrary XQuery expression.

```
service < URI >
  < NAMESPACE / FUNCTION DEFINITIONS >
  < LOCAL DECLARATIONS >
  < DECLARATIVE WEB SERVICE CLAUSES >
  < OPERATION SPECIFICATIONS >
endservice
```

In the following subsections we will describe namespace and function declaration, the declarative Web service clauses, and the operation specification.

4.2.1 Namespace and Function definition

Functions are defined in XL in exactly the same way as in the prolog of an XQuery entity (chap. 4.12 [W3C05d]).

```
service < URI >
  !! namespace / function definitions
  ( declare function < qname > ( <parameter-list> )
4   {
      < expression >
    };   )*
  ( declare namespace < name > = < URI >;   )*
  [ declare default namespace < name > = < URI >; ]
```

```
9   !! ...
endservice
```

A namespace declaration simply associates a small prefix with a URI identifying the namespace. Again, XL adopts the XQuery syntax (chap. 4.15 [W3C05d]). Furthermore XL provides a default namespace declaration to be used within all XQuery expressions.

4.2.2 Web services local declarations

To represent a state, an XL Web service can declare local variables. Such variables hold XML data and their potential values can be constrained by the XML type system. The term *local* in this case means local for the service itself, as these variables can be accessed by every operation of the service but are not visible to the outside.
The syntax for declaring these XL variables is the following:

```
service <uri>
  !! function definitions, local types,
  !! schema imports ......

5   !! state of the web service
  ( let [<type>] <variablename>
              [:= <expression>]; )*

  !! state of a conversation of the service
10  ( context let [<type>] <variablename>
              [:= <expression>]; )*

  !! declarative web services clauses
  !! and operations ....
15 endservice
```

Listing 4.1: *Variable declaration in XL*

XL distinguishes between two different kinds of local variables by introducing an additional variable scope. The first type of variable represent the internal state of the whole Web service (lines 6 and 7 in listing 4.1). These variables are instantiated once, when the Web service is installed and persist the whole lifetime of the Web service or possibly longer. The scope of these variables is the whole Web service. A typical global variable of a Web service is for example a customer database, or a RDF document describing items in a Web shop.
The second type of variable represent the internal state of a particular conversation that the Web service is involved in (lines 10 and 11 in listing 4.1). Examples are the session id when a user logs into the system, the maximum bid for an item in an auctioning system, or a history of past queries in a product database. 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. The term *conversation* denotes a set of correlated messages exchanged between different services. A conversation has neither got any central state nor a definite start or end time. A service participates in a conversation by simply sending a message including the unique identifier (URI) of a conversation.
The idea behind the XL conversation concept is to provide convenient means for implementing conversations between services. The XL conversations concept can be compared to the ses-

sion beans in J2EE [Sun05b], as XL, J2EE implicitly sets the context for executing operations depending on message properties.
We assume here that the SOAP messages which are exchanged between Web services can carry the conversation URI in their envelope. Alternatively, the URI could be included as an HTTP parameter.
This kind of variable 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 lifetime of such variables is bound by the lifetime 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 broker, 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 broker. (Obviously, this conversation URI should not be public in this example.)
In this syntax, the `<type>` is the optional type constraining the type of the variables values, while `<expression>` is an XQuery expression describing the initial value of the variable. If no expression is given, then the variable is initialized to the empty sequence; if no type is given then the variable can be bound to any valid instance of the XML data model.

4.2.3 Declarative Web service clauses

Essentially, this part contains a set of high level declarations that control the Web services global state, how the Web service operations are executed and how the Web services interacts with other Web services.
These declarative clauses target typical aspects of Web service programming as for example event handling, asynchronicity, and default actions. Depending on the actual use case many more declarative attributes associated to a service could be added. You might think of attributes used in the context for semantic web applications ([Dav04]), such as service category information or detailed textual interface description.
The syntax for these clauses is as follows:

```
service <uri>

  !! function definitions
  !! .....
  !! declarative web service clauses
  [history ;]
  [defaultoperation <operation> ;]
  [unkownoperation <operation> ;]
  [init <operation> ;]
  [close <operation> ;]

  ( invariant <booleanExpression>
       throw <expression> ;)*

  ( on event  <booleanExpression>
```

```
        invoke <operation>
      [ with input <expression>] ;)*

    [on error invoke <operation> ;]
    [conversationpattern
        (required | ... | never) ;]
    [conversationtimeout
        <durationExpression> [<operation>] ;]

    [conversationclose operation> ;]

    partner <name> := wrapper {
         language =  <external language>;
                  import = <external library>;
                  mapping = <mapping specification>;
                  [  language specific parameters ]
    }

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

Listing 4.2: *Service declaration clauses in XL*

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

History If this clause is specified, then all calls to operations of the Web service are automatically logged and recorded in an implicitly declared read only *$history* variable. The data automatically recorded in this variable includes for example the name of the operation that is called, the identifier (URI) of the caller, the value of the input and output messages, the timestamp when the operation was called, and other statistical information that are important for the Web services tracing and monitoring.
History logging could be useful for legal or security reasons in order to either document events or to implement certain kinds of constraints. Application service providers could for example easily restrict the number of accesses per day by using this type of automated history logging.
The *$history* variable itself can be used in the same way any other XL variable is used in XQuery expressions in XL operations. Furthermore, the *$history* variable could be store persistently.

Default- & unknownoperation These clauses declare the Web services behavior 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. If no UNKNOWNOPERATION clause is given, then the default operation is used in such cases.
If neither of the two options is specified the service should not return any answer at all.

Init, Close These clauses specify a pair of operations that are automatically invoked when the Web service is created and destroyed, respectively. These operations are only be invoked once by the engine itself and they take no input.
In case such declarations are missing, the *init* and *close* operations are used.

Invariants In this clause, global Web services integrity constraints (or invariants) are defined. A Web service can define an arbitrary number of invariants. Typically, invariants are defined for stateful services and constrain the value of internal variables. Invariants, however, can also constrain the value of the *$history* variable and contexts of conversations. If at any time an invariant is violated, the statement that caused the violation is undone, an exception is raised, and the execution of the current operation is stopped if the exception is not handled. The exception that is raised when an invariant is violated is specified in the optional "exceptionExpression" part of the INVARIANT clause.
As an example, it could be specified that all customers of an online shop must be older than eighteen years and that current stock of each item store should not drop below a certain level. These two invariants would be defined as follows:

```
invariant    $customer/age > 18
 throw  <error> You are too young!
        </error>;

invariant $items//stock/[current < minimum]
  throw  <alert>  replenishment required </alert>;
```

Instead of raising an exception, a direct error handling operation could be invoked using the ON EVENT clause.

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

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

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

on event $items//stock/[current < minimum]
  invoke replenishment;
```

Note the semantics of the = operator in this example: the = operator is equivalent to an existential quantification according to the XQuery standard [W3C04g].

On Error Invoke This optional clause specifies an operation that is called whenever an (other) operation of the Web service fails; e.g., if an INVARIANT is violated. In other words, if an operation raises an exception, this exception is passed as input to the operation specified in the ON ERROR INVOKE clause and the output of this operation is then returned to the client of the Web service. This way, application logic can be separated from error handling; in particular, all texts for error messages are employed by one operation only. As will be discussed in Section 4.4.2, exceptions can also be handled locally using TRY and CATCH statements. The operation specified in this clause is only called for exceptions that are not handled locally and would otherwise directly be returned to the client of the Web service.

Conversationpattern This clause specifies in a declarative manner how the Web service interacts with other services as part of conversations. The term *conversation* denotes a sequence of correlated messages, which share a commen context.
There are many alternative models conceivable how to implement business conversations. As mentioned earlier, in our model we assume that 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 [Sun05b]. These patterns are described in Table 4.2 on the following page.
For each pattern, two situations must be considered: (a) the ingoing message is not part of a conversation (defined as - in the second column of Table 4.2); (b) the ingoing message is part of a conversation (defined as C_1 in the second column of Table 4.2). For each of these cases the pattern specifies wether:

1. the invoked operation should be executed in the same conversation (defined as C_1 in the third column of Table 4.2)
2. the invoked operation should be executed in a new conversation (C_2)
3. the invoked operation should not be executed a part of a conversation (-)
4. error, the current operation either requires or forbids a conversational context (*error*)

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. The operation call itself is not a part of this new conversation (C_2 in table 4.2) (b) If the Web service receives a message with a conversation URI, then all other Web services it calls as part of processing the input message will be called using the conversation URI of the input message. The operation call itself is executed within the context of the given conversation (C_1 in table 4.2)
For all operations of a service a default pattern can be set, while each single operation can overwrite this by specifying its own pattern. Each single operation can overwrite this default pattern by specifying its own pattern.

Pattern	URI of Input Message	URI of Outgoing Messages
Required	- C_1	C_2 C_1
RequiredNew	- C_1	C_2 C_2
Mandatory	- C_1	*error* C_1
NotSupported	- C_1	- -
Supports	- C_1	- C_1
Never	- C_1	- *error*

Table 4.2: *Conversation Patterns*

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., *sell* and *buy*) must be called as part of the same conversation.
As mentioned earlier, both of these Web service require 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 messages 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*.
In XL the conversation URI is not visible to the programmer, say, the programmer is not meant to set the conversation URI explicitly.

Conversationtimeout & Conversationclose Finally a timeout can be specified that terminates a conversation after a certain time since the last message exchanged as part of the conversation. An operation can be declared that is invoked if such a timeout takes effect. If a message is sent to a Web service after the time out, the Web service will assume that this message is part of a new conversation; in particular, the context of the (old) conversation is lost after the time out. For instance, if the time out of the online broker is set to ten minutes and a customer logs on and carries out no operations for ten minutes, then the user will have to log on again before buying or selling stock.
Alternatively an operation can be specified, which explicitly terminates the conversation it is called in. Say, after the close operation has been executed the context associated to the current conversation is removed.
As several instances might participate in a conversation, the conversation itself is not necessarily terminated if a single Web service quits.

Partner Wrapper XL offers the possibility to integrate external applications by using a plug-in. Instead of sending a SOAP message, a local application is invoked directly. In the service header the wrapper for an external call is specified.

```
partner $Employees = wrapper {
            language = "JAVA";
            import          "examples/java/system.jar";
            mapping = "examples/java/mapping.xml";
            class = "system.office.Employees";
    }
```

Each warpper is associated to a certain name, as for example $Employees in the example above. The wrapper element contains several subelements:

- **language** The language clause specifies the type of plug-in to used. Currently XL only support Java plug-in.
- **import** The import clause specifies the external library to be loaded. As for the Java plug-in, a Jar file is specified.
- **mapping** The mapping file specified by this clause defines who the XML type system used by XL is mapped to the type system of the specific application invoked. The syntax of this file depends on the targeted type system.

Additionally to these three elements further language specific information can be added. In order to invoke call a Java method, the class name has to be specified as well.
The listing below gives an example of a Java wrapper specification:

```
partner $Employees = wrapper {
            language = "JAVA";
            import "file:/examples/java/system.jar";
            mapping = "file:/examples/java/mapping.xml";
            class = "system.office.Employees";
    }
```

In order to invoke the Java application, the Java archive (Jar-File), specified by the import clause, has to be loaded. The mapping file specifies the Java to XML and back translation. In XL the Castor toolset [cas05] is used as a data binding framework between XML and Java.

4.3 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.
The XML listing below shows a sample XML SOAP message.

```
<?xml version="1.0" encoding="UTF-8"?>
<ns0:Envelope xmlns:ns0="http://schemas.xmlsoap.org/soap/envelope/">
    <ns0:Body>
        <queryItems>
            <items>
               <type>book</type>
                <author>Douglas Adams</author>
            </items>
```

```
        </queryItems>
    </ns0:Body>
</ns0:Envelope>
```

For this message the so called RPC/literal style is used. The name of invoked operation is specified by the XML tagname inside the SOAP body, in this case the operation *queryItems* is called. In the RPC-style the actual parameter of the operation call is specified by the content of the tag naming the operation. As for the example given above, the operation *queryItems* is invoked, the input variable `$input` has the initial value

```
  <items>
    <type>book</type>
     <author>Douglas Adams</author>
 </items>
```

In XL, the content of the message is directly used as input for the operation. XML is not used to wrap Java objects but as a direct form of data representation.
Alternative SOAP message dialects are possible as well (e.g., RPC/encoded, document/encoded).
The signature of an XL operation is described by its name and a list of arguments. Each argument is again described by its name and optional by a type according to the XQuery datamodel [W3C04f].
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 also 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. The syntax of an operation specification is as follows:

```
operation <name> '[' ([ <type> ] < variablename> )*   ']' {
   < DECLARATIVE OPERATION CLAUSES >
   < OPERATION BODY >
}
```

Unlike common Java based XML applications, XML deserialization and serialization in XL is very simple. Since the content of input variables of operation in XL is XML no additional XML to Java marshalling code is required.
Currently, all XL operations visible as part of the public interface of the service. Yet, XL does not include the notion of private or protected operations. If the XL language is enhanced to address security issues, restricted visibility has to be included.

4.4 XL Statements and Combinators

4.4.1 Declarative operation clauses

As for the whole Web service, the declarative clauses of an operation control the run-time behavior of the given operation. Some of the clauses are identical in syntax and semantics to those of the Web service and serve only to refine the global Web service behavior (HISTORY, CONVERSATION PATTERN and ON ERROR INVOKE). We remark that the notion of an

conversation timeout cannot be associated with a single operation. Other clauses like the PRECONDITIONS, POSTCONDITIONS, and NO SIDEEFFECT are specific only to operations, and we will describe them next. The syntax is as follows:

```
operation <uri>::<name> [' ([ <type> ] < variablename> )*  ']' {
 [history ;]
 (precondition  <booleanExpression>
          throw  <expression> ;)*
 (postcondition <booleanExpression>
          throw  <expression> ;)*
 [on error invoke <operation> ;]
 [no sideeffects ;]
 [conversationpattern
      (required | ... | never) ;]
 !! Operation Body
 !! .....
}
```

PRECONDITION This is a condition that is checked before the first statement of the body of the operation is executed. If the condition fails (i.e., evaluates to the Boolean value FALSE), an exception is raised. The exception is specified in the THROW clause. Within the header of an operation, any number of preconditions can be defined. If there are several preconditions, these preconditions are evaluated in a random order.
Typically, preconditions will test certain properties of the *$input* variable; e.g., the existence of certain elements or the range of the value of certain elements. Preconditions, however, can also involve internal variables which are declared in the local declarations of the Web service (see Section 4.2). The precondition could also depend on some external conditions.
Preconditions could be used for different purposes: to dynamically check the state of a conversation, to check criteria for the input message or for debugging purpose. The following precondition would specify that the *buy* operation can only be called if the customer is logged in:

```
precondition not( $status = "LOGGED_IN")
  throw <error> Sorry, you have to login in first </error>;
```

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

```
postcondition $output validates as
              myns:customerinfo
  throw <failure> Sorry, $output
                  is invalid! </failure>;
```

NO SIDEEFFECTS This clause specifies that the operation has no sideeffects; i.e., the operation is an observer and does not change the internal state of the Web service or of any other Web service it might call. Operations that have no sideeffects can be invoked as part of expressions; otherwise, an operation cannot be invoked as part of an expression and it must be invoked as part of a statement (described in the next section).

4.4.2 XL statements

XL extends the notion of XQuery expressions to statements. Each statement alters the state of the in a well defined way. The body of an XL operation is described by a set of statement. Like C or Java, XL supports the common imperative statements like the assignment, if and while statements, a switch statement, and a for $\cdots$ do statement for iterating through sequences. Additionally XL supports some XML specific statements (e.g., update statements) and Web service specific elements (e.g., logging, service invocation). 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).

XL simple statements

In this section we introduce some of the basic atomic statements that can be used in the body of an XL program. As in Java, each simple statement is terminated by semicolon ';'.

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 4.4.3). Expressions can be any expression defined by the W3C XQuery proposal [W3C05d].
As an example for a simple assignment, consider the following statement:

```
let $employee :=<employee>
                  <id>4711</id>
                  <name>Donald Duck</name>
                  <department>Sec-01</department>
                  <position/>
                </employee>
```

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

Update Statements Unfortunately, XQuery does not yet provide expressions to manipulate XML data. There are plans to extend XQuery in this respect and once a recommendation has been released by the W3C, XL is going to adopt the syntax and semantics of these expressions. In the meantime, we will use the following statements to manipulate XML data:

- *insert* in order to add new nodes to the XML hierarchy (e.g., an additional credit card element)

```
update $customer insert <creditcard>...</creditcard>
into   $customer/payment
```

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

```
update $customer delete $customer/creditcard[type="Visa"]
```

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

```
update $customer replace $customer/telephone with
     <mobil>(550)4901-01</mobil>
```

The general syntax of the update statements is as follows:

```
update <variablename>
 [  insert <expression> ( into | before | after ) <expression> |
    delete  <expression>      |
    replace <expression>  with <expression> ]
```

The initial update clause specifies the updated variablename. This is necessary, since this name cannot be extracted unambiguously from the following expressions. Afterwords, the update operation itself is specified (insert, delete, or replace) and an expression is given specifying the position of the update within the updated variable.
Note: The possibility of updating XML variables managed by XL in such a declarative way is an important feature of XL. The XL programmer does not care about how the updated variable is represented, the variable $customer in the example above could be either stored in main memory or a relational database.
Like database applications using SQL, XL relies on an XML store implementation. The generic store interface used by XL provides interfaces for updates. It is up to the XL engine, or say the engine combined with the store implementation, to figure out how to execute the updates.

Service Invocation Statements Probably the most relevant atomic statements in XL are those used for invoking other Web services; i.e., sending a message to another Web service. Often, the other Web service will be written in XL, but messages can be sent to any service that have a URI and respond to SOAP messages [W3C04c]. 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:

```
send_sync <expression> --> <uri-expression>[::<operation>]
                [--> <variable>]
```

The send statement is to be read form left to right. A message with the value of *expression* is sent to the Web service identified by *uri-expression*. The target URI of the message is again specified by an XQuery expression. The used arrows '–>' indicate the dataflow to a remote service and possibly back to a local variable. 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.
Note: Invoking an default operation is a typical "*Web*" requirement. As for Webpages index.html is default name, a remote service could be invoke without knowing the specific operation. Since the SOAP message protocol is used to wrap remote procedure calls of for example Java applications, the operation name cannot be simply omitted.

In a synchronous call, the execution is halted until the called Web service finishes its execution and returns the entire result (also wrapped in a SOAP message). If a *variable* is given as part of the call, then the body of the message returned by the called service is copied into this variable. The message is sent exactly once and in a best effort way. Quality of service guarantees and other specifications such as "as often as possible" or "at least once" which might become part of the XML Protocol recommendation [W3C03] cannot be expressed in the current version of XL.
As an example, consider the following synchronous service invocation that asks the online broker to purchase 1000 SAP for at most €140.00; the result is stored in the *$receipt* variable:

```
  <purchase_order>
          <pid>01234</pid>
          <stock> SAP </stock>
          <currency> Euro </currency>
          <amount> 1000 </amount>
  </purchase_order>  -->
      http://www.OnlineBroker.com::purchase
           --> $receipt
```

Since the data being sent here is XML, like the SOAP message itself, no expensive transformations have to be applied. The SOAP message generated by the send statement above is shown in the XML listing below (SOAP style RPC / literal):

```
<?xml version="1.0" encoding="UTF-8"?>
<ns0:Envelope xmlns:ns0="http://schemas.xmlsoap.org/soap/envelope/">
  <ns0:Header>
      <M_TYPE ns0:mustUnderstand="1">XL_SYNCOPERATIONCALL</M_TYPE>
      <M_ID       ns0:mustUnderstand="0">10981</M_ID>
  </ns0:Header>
  <ns0:Body>
       <purchase>
            <purchase_order> <stock> SAP </stock>
                                     <limit> 140 </limit>
                                     <currency> Euro </currency>
                                     <amount> 1000 </amount>
             </purchase_order>
       </purchase>
    </ns0:Body>
</ns0:Envelope>
```

A typical answer returned by an XL Web service is shown in the following listing:

```
  <?xml version="1.0" encoding="UTF-8"?>
<ns0:Envelope xmlns:ns0="http://schemas.xmlsoap.org/soap/envelope/">
   <ns0:Header>
      <M_ID ns0:mustUnderstand="0">10981</M_ID>
   </ns0:Header>
    <ns0:Body><purchaseResponse>
                          <pid>01234</pid>
                          <returnCode>200</returnCode>
                          <desc>order successful</desc>
                      </purchaseResponse>
    </ns0:Body>
</ns0:Envelope>
```

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

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

In terms of the semantics: in this case the execution will not block and the program will immediately continue executing the next statement after the message to the called service has been sent. If the output (normal reply message or error) needs to be processed, then the name of the operation that will process the asynchronous result can be given as part of the call; this operation has to be a member of the Web service that originated the asynchronous call. Again, the message is sent exactly once and in a best effort way.
Currently, XL provides no syntax to set the envelop of a SOAP message explicitly. Such constructs could be useful in order to implement certain kinds of conversations and/or to implement distributed transactions and secure messages. We plan to extend XL in this way once SOAP and the emerging XML Protocol recommendation [W3C03] have stabilized. The SOAP standard version 1.2 [W3C04c] does not specify how a conversation ID is to be included into the message, SOAP does not define a certain keyword. In XL, as recommended by the SOAP primer, include an conversation ID as a header element in the SOAP message.
Furthermore, XL does provide an implicit notion of multicast: if the expression specifying the destination returns a sequence, a message is send to each of them. If it is a synchronous operation call, all return values are concatenated and assigned to the given local variable. Note: the return values are not ordered. Multicast messages have not yet been addressed by the XML protocol working group. As you can see in the sample SOAP messages shown above, the initial SOAP message contains two optional XL specific headers, a message ID and a message type. Depending on the used message protocol the message id is required to associate return messages to the correct message sent before. For HTTP, a message ID is necessary, since a bi-directional network connection is used.
XL distinguishes between different message types:

- **Synchronous operation call** An operation is invoked and the result is returned to the caller.
- **Asynchronous operation call**An operation is invoked and the result is passed to a callback operation, if specified.
- **Other message types** Except for the operation calls, different message types could be used and handled by XL as well. In order to either administrate, deploy, or debug a Web service, communication between a user interface (locally or remote) and the Web service engine is necessary. Depending on the specific use case, a whole set of different message types are conceivable.

Furthermore, the XL engine offers the possibility to define a default message type, say, if the header attribute is omitted, the default type is assumed.

Assertions Recall that it is possible to define preconditions and postconditions of XL operations (see Section 4.3). The more general concept is the concept of an assertion that can be executed at any point during the execution of an XL operation. Assertion statements are described using the following notation:

```
assert (<booleanExpression>) throw <expression>
```

If the Boolean expression evaluates to TRUE, the execution continues normally. Otherwise, an exception is raised. The second expression specifies the exception that is raised in this case.

Imperative statements in XL

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

```
if ( <booleanExpression> ) {
     <statement>
} else {
     <statement>
}
```

The semantics are straightforward and the same as in other imperative programming languages. Furthermore, XL supports the following SWITCH statement:

```
switch {
 if ( <booleanExpression> )
     { <statement> }
 if ( <booleanExpression> )
     { <statement> }
 if ( <booleanExpression> )
     { <statement> }
 // ...
 [default { <statement> }]
}
```

Again, the semantics are straightforward. 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. (In C++ and Java, *break* statements are used for this purpose.) The DEFAULT clause is optional.

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

```
while ( <booleanExpression> ) {
  <statement>
}
```

and

```
do {
  <statement>
} while ( <booleanExpression> )
```

and

```
for <variable> in <expression>
 {   <statement>  }
```

The FOR-DO loop corresponds to FLWOR expressions in XQuery [W3C05d].
The given expression is evaluated and returns a sequence of elements, for example, lineitems of a purchase order. The map statement binds each individual sequence element to the variable name and executes the statements given in the DO part of the statement. The listing below illustrates the usage of the FOR-DO statement:

```
for $item in $order/lineitems
do {
 update $productDB replace $productDB/item[$item/id eq id]/stock
      with <stock> {$productDB/item[$item/id eq id]/stock
                          - $item/quantity } </stock> ;
}
```

This FOR-DO statement executes an update operation for each lineitem of an order. Like the XQuery FLWOR expression, the FOR-DO statement iterates through a sequence of nodes. The XL statement enhances the XQuery expressions by executing a set of statements for each sequence element. The iterator variable ($item in the example above) acts like a cursor used to process the sequence elements individually. A cursor is not an L-value itself, but represents a pointer on a single sequence element. In the current XL version, pointers are not possible – the iterator variable contains a deep copy of the sequence element. For a later XL version, updatable cursors would be an interesting idea.

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 an error message to the caller of the service. The SOAP standard defines *Fault* element in order to signal an exception, instead of a SOAP *Body* element containing a common return value. The SOAP fault element include an error ”code” and a ”reason” subelement, describing the fault. Just like variables and any other expression, the exceptions can be strongly typed optionally.
As an example, consider the following exception that signals that charging a credit card failed because the credit card has expired.
The following listing illustrates the the throw statement. An unauthorized access is signaled by raising an exception:

```
throw <error>
            <code>401</code>
            <protocol>HTTP<protocol>
            <description>Unauthorized</description>
            <resource>member.xml</resource>
        </error>
```

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

```
try {
  <statement>
 } catch(<variable>)  {
  <statement>
 }
```

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

4.4.3 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.
XL should be free to combine statements either as a sequence of statement, a set of parallel statements, a combination of, or simply leave it to the runtime system which statements to execute.

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>
```

Even so all these combinators are useful, the most frequently used combinator is still the sequence. Hence, XL adopted the semicolon ';' as the standard combinator, every simple statement in XL has to be terminated by a semicolon. The other combinators can be appended as a postfix after the semicolon.

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.

```
{
  <statement>
}
```

4.4.4 Web Services specific statements

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

Logging statement As mentioned in Section 4.2, there is an easy way to specify in XL that all calls to operations are logged in an automatic way - simply, the keyword HISTORY must be written in the declaration of a Web service. This way of automatic logging only involves calls to a Web service, the timestamp of the call, and the *$input* message sent by the caller. In order to write more information into a log, we propose the following syntax.

```
logpoint <expr1> as <name1>,
                 ...
         <exprN> as <nameN>
 {
  <statement>
 }
```

As a result of the execution of this statement, the expressions 1 to N are evaluated before and after the execution of the statement (or series of statements) and their values are inserted each time into the *$history* variable using the respective names.

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

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

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

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

```
sleep xf:duration("P10M")
```

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

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

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

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

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

Note that the following two statements are not equivalent, if the execution of a program should be halted until some given timestamp (xf:currentDateTime() is the XQuery/XPath function that returns the current Timestamp [W3C04h]):

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

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

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

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

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

This statement will attempt the repetitive execution of the *statement* until the execution finishes without an exception, or at most a certain number of times (if a MAXIMUM clause is given) or for a maximum duration (if a TIMEOUT clause is given).

For instance, the following statement will try to charge the visa credit card of a customer three times. Such a payment can fail temporarily for various reasons; e.g., if servers are overloaded. If all three times fail, however, it is assumed that there is something wrong with the credit card and an exception that indicates an illegal payment method is raised:

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

4.5 Related Work

This section provides a brief overview of the related work on Web service related programming languages.
The service oriented architecture and the development and composition of Web services is currently a very active area in both industry and academic research.

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 [Sun05b] and SunOne [Sun], and Microsoft's .NET initiative [Mic06b] including C# and $C\omega$. Compaq used to developed the WebL language [Web], IBM is working on a language called BPEL [IBM03] (Business Process Execution Language for Web Services, see 3.6.2), and Microsoft has recently released their BizTalk Server 2000 [Mic06a], XLANG [Tha01] and $C\omega$ [Mic05b] (see 3.6.1).

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 (which are expected to be implemented using other languages), while XL is *complete* not only with respect to Web service composition but also *specification*.
The approach used by BPEL allows the programmer to express processing logic in BPEL itself. But still, the services themself, or say, the core of the processing logic cannot be implemented in BPEL. 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 Web Service Modeling Framework [FB02a] focuses on interface description and the mediation of different data-types and processing concepts between different Web services.

Two other specifications worth mentioning are the Business Process Model Initiative whose goal is to implement cross-organisation processes and workflows on the Internet [BPM] and DAML-S whose goal is the automation of Web services using ontologies [Coa]. 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 [JAK]. Furthermore, the J2EE framework provides a number of features for service composition, conversations, database interaction, transactions, and security [Sun05b]. Sun Microsystems introduced five years ago already the JXTA project on peer-to-peer computing to support distributed computing between PC, servers and even PDAs on the Internet [JXT]. A very good guide on how to build a Web service using common tools like WSDL [W3C01], SOAP [W3C04c], and UDDI [UDD] is the architecture guide from Sandeep Chatterjee and James Webber [CW03].

Finally, the notion of a service composition is based on a solid theoretical background consisting on the calculi developed first by Hoare [Hoa85] and more recently by Cardelli [CD99].

However, none of those languages and frameworks are totally consistent with the current W3C standards, and we believe that this is a mandatory condition for the success of such a programming language.

On the WWW conference 2002 the language XL was introduced for the first time [FGK02]. At the CIDR conference in January 2003 another XL publication ([FGK03]) appeared.

4.6 Example Application

To illustrate the XL language in a real life use case, a small Web shop was implemented in the way the TPC-W benchmark defines it. In the following sections out small application will be described and a commented XL source code listing is provided. At the end of this section (on page 64), the objectives of the XL language design described in section 2.1 are reviewed.
The same XL program is used to in section 10.3 to perform a complex benchmark. In this section it will used to give an example of how a complex XL program looks like.
The XL shop sells *items*. Items in our case are represented by a complex XML element containing several subelements. Customers can register, query the item database and add items to a shopping cart. A purchase is completed by going to the counter. An additional checkout operation without finishing the purchase order is included in this example.

On the following pages the complete XL program is given. The listing is intercepted at several points and a detailed description of the previous listed code is given.

```
!! a service declaration
 service http://myshop.com:10000/
```

```
    !! default namespace used by XL, incl. XQuery
 5  defaultnamespace "http://xl.informatik.uni-heidelberg.de/TPCW/";

    !! Web service internal data
    let #element()* $items := doc("items.xml")/items;
    let #element()* $countries := doc("countries.xml")/countries;
10  let #element()* $customers := <customers></customers>;
    let #element()* $authors := doc("authors.xml")/authors;
    let #element()* $orders := <orders></orders>;
    let #element()* $money_orders := <money_orders></money_orders>;
    let #xs:integer $order_id := 0;
15  let #xs:integer $id := 0;

    context let #element()* $cart := <cart>
                                       <state>init</state>
                                       <lineitems/>
20                                     </cart>;
```

The lines 8 to 15 contain the declaration and definition of the global variables of the service. Typically, the global variables are stored in a database, the expressions provided here simply initialise the variable. In line 15 a context dependent variable is defined. In this example the variable $cart holds the shopping cart of an individual customer. Each instance of this variable is bound to a specific conversation, say, a certain purchase.

```
    !! entire Web service activity is monitored
    history;
    !! initializes the service
    init initOp;
25  !! unknown operation
    defaultoperation unknownOP;
    !! terminates a conversation
    conversationtimeout goToCounter;
    !! default conversation pattern
30  conversationpattern mandatory;
```

The declaration section of this service contains the following elements (see section 4.2.2):

- Since the keyword *history* is specified (line 22), all operation calls are logged in a global variable `$history`.
- The specified init-operation is called once at the startup time of the service. In our example the init-operation is called *initOp* (line 24).
- The unknown-operation handles an error case. If a message is received, specifying an unknown operationname, this operation will be invoked instead. In this XL service this operation is called *unknownOP* (line 26).
- The keyword *conversationtimeout* was initially meant to specify a timeperiod after which a conversation is closed. In this example we abuse the keyword in order to specify a close operation. If the operation *goToCounter* is invoked in this service, it is executed within the context of this conversation. Afterwards the conversation is closed (line 28).

- In this service the default conversation pattern is *mandatory*. By default every received message has to be part of a conversation (line 30).

```
!! register new customer
operation registerCustomer [ #element()* $input ] {

35      conversationpattern never;

        precondition (\xf:not( xf:empty($input/uname))\)
                throw <error>username is
                     missing in $input: {$input/uname}</error>;
40      precondition (\xf:empty( $customers/customer[uname
                                        eq $input/uname])\)
                throw <error>username {$input/uname/text()}
                   exists already: {$customers/customer/uname}</error>;
        precondition (\xf:not( xf:empty($input/passwd))\)
45              throw "password is missing in $input";
        precondition (\xf:not( xf:empty($input/fname))\)
                throw "fist name is missing in $input";
        precondition (\xf:not( xf:empty($input/lname))\)
                throw "last name is missing in $input";
50      precondition (\xf:not( xf:empty($input/address))\)
                throw "address is missing in $input";
        precondition (\xf:not( xf:empty($input/email))\)
                throw "email is missing in $input";

55      body {
          update $customers insert  <customer>
                          { $input/uname }
                          { $input/passwd }
                          { $input/fname }
60                        { $input/lname }
                          { $input/address }
                          { $input/phone }
                          { $input/fax }
                          { $input/email }
65                        <since> {xf:current-dateTime()} </since>
                          <last_visit />
                          <discount>0.0</discount>
                          <balance>0.0</balance>
                          { $input/birthdate }
70                        { $input/data }
                          </customer>
                    into $customers;
          let $output := "Thank you for registering";
        }
75    }
```

The operation *registerCustomer* inserts a new customer into the global variable *$customers*. The predefined variable *$input* contains the required information related to the new customer. The a typical content of *$input* would be for example:

```
<customer>
 <uname>Asterix</uname>
 <passwd>AG_</passwd>
 <fname> Asterix </fname>
 <lname> Gallier </lname>
 <address>
    <street>via 1</street>
    <city>Gallisches Dorf</city>
    <zip>01234</zip>
    <country_id>11</country_id>
 </address>
 <phone>012/121333345543</phone>
 <email>asterix@comic.fr</email>
 <birthdate>0010-01-01</birthdate>
 <data>premium customer</data>
</customer>
```

In this example, the preconditions listed before the operation body validate the input variable and define appropriate error messages if a validation fails. The operation *registerCustomer* is not part of a particular purchase, therefore the conversation pattern for this operation is set to "*never*"

```
  !! add a new lineitem to the shopping cart
  operaton addItemToCart [#element()* $input] {

     precondition (xf:not( xf:empty($input/customer_uname)))
80             throw "customer_uname is missing in $input";
     precondition (xf:not( xf:empty($input/id)))
               throw "id is missing in $input";
     body {
         let  #element()* $inp := $input;
85
        if($cart/state/text() eq "init") {
           !! initialize $cart
           update $cart replace $cart/state
                    with <state>created</state>;
90
           update $cart insert <total>0.0</total>
                   into $cart;

           if(xf:not(xf:empty($input/address))) {
95         !! if a shipping address is specified in $input,
           !! use this one
               update $cart insert $input/address
                           into $cart;
```

```
            } else {
100         !! if no shipping address is given, use
            !! the address specified in $customers
               update $cart insert
                   op:item-at($customers/customer[uname
                   eq $input/customer_uname],1)/address
105              into $cart;
            }
         }
         let #element()* $customer := <customer/>;
         if(xf:empty($cart/customer_uname)) {
110
            let $customer := for $c in $customers/customer
                             where $c/uname
                                   eq $input/customer_uname
                             return $c;
115         if(xf:not($input/passwd
                      eq $customer/passwd)) {
                    throw <error>wrong password: xf:not(
                      {$input/passwd} eq {$customer/passwd}:
                           { $customers/customer/uname/text() },
120                        { $input/customer_uname/text() }  </error>;
            }
            update $cart insert $input/customer_uname
                         into $cart;
            update $cart insert <id>{ $order_id }</id>
125                      into $cart;
            let $order_id := $order_id + 1;

          } else {
               let $customer := $customers/customer[uname
130                 eq $input/customer_uname];
                if(\xf:not($input/passwd eq $customer/passwd)\) {
                    throw <error>wrong password:
                       xf:not({$input/passwd/text()} eq
                              {$customer/passwd/text()}:
135                    {xf:not($input/passwd/text()
                        eq $customer/passwd/text())} </error>;
               }
          }

140       if(xf:empty($items/item[id eq $input/id])) {
               throw "Item does not exist";
          }
          let #element()* $item := $items/item[id
                                   eq $input/id];
145
          print <debug>{$items/item[id eq $input/id]/stock}
                                   lt {$input/quantity}</debug>;
```

```
            if( $item/stock lt $input/quantity ) {
150             throw <error>Item needs some replenishment
                 {$item/stock} lt {$input/quantity}:
                 {$item/stock lt $input/quantity}</error>;
            }
            update $items replace $items/item[id
155                                   eq $input/id]/stock
               with <stock> {    integer($item/stock)
                               - integer($input/quantity) } </stock>;

            let #xs:double $subtotal :=
160              ($input/quantity * $item/price)*(1.0 - $customer/discount);
            let #xs:double $tax := $subtotal * 0.0825;
            let #xs:double $shipcosts := 3.0 + (1.0 * $input/quantity);
            let #element()* $lineitem := <lineitem>
                                  { $input/id }
165                               { $item/title }
                                  { $item/backing }
                                  { $input/quantity  }
                                     <subtotal>{ $subtotal }</subtotal>
                                     <tax>{ $tax }</tax>
170                                  <shipcosts>{  $shipcosts }</shipcosts>
                              <total> { $subtotal + $tax + $shipcosts } </total>
                             </lineitem>;

            update $cart insert $lineitem
175                    into $cart/lineitems;

            update $cart replace $cart/total with
                 <total>{xf:sum($cart/lineitems/lineitem/total)}</total>;

180         update $customers replace $customers/customer[uname
                                     eq $input/customer_uname]/last_visit
               with <last_visit>{xf:current-dateTime()}</last_visit>;
            let $output := $lineitem;
        }
185 }
```

The operation *addItemToCart* is invoked once for each lineitem within a purchase order. If this operation is invoked for the first time it initializes the context variable *$cart*. The variable *$input* in this case contains the user id, the user password, and the information specifying the item to be ordered. An example of *$input* for this operation is given below:

```
<input>
  <customer_uname>Asterix</customer_uname>
  <passwd>Obelix</passwd>
  <id>5</id>
  <quantity>1</quantity>
</input>
```

The customer *Asterix* orders one item with the id 5. The conversation variable $cart contains a shipping address. This address could be set by including it into the input variable of the addItemToCart operation (lines 94 to 99), otherwise the address included in the global variable $customers is used (lines 99 to 106). Between the lines 109 and 138 the given user password is checked, and the customer user name is set in the conversation variable $cart. At line 140 it is checked, whether the global variable $items contains the specified item id. At line 143 the ordered item element is selected from the global variable $items and stored in a local variable. Line 146 contains a print statement used for debugging purposes. At line 149 it is checked, whether sufficient stock of the item ordered is available. If available stock is not sufficient an error is raised. After checking the stock the order can be fulfilled and the global variable $items is updated (line 154). Between the lines 159 and 172 the new lineitem entry is created and added to the shopping cart (line 174). At line 177 the total sum of the all lineitems included in the shopping cart is updated. Finally, at line 180, the time of last visit of the current customer is updated in the global variable $customers. If successful, the newly created lineitem element is returned to the caller. If an error occurs, an exception is raised and returned as a SOAP fault element to the caller. The XQuery builtin-function item-at is used sometimes (e.g., line 103) to return only the first result tuple an terminate the evaluation of the expression afterwards. It is only used for optimisation purposes in this example.

The implementation of this operatin illustrates how complex processing logic can be expressed based on XQuery and the XML data modell. At the same time, this implementation emphasizes the importance of efficient update expressions based on the XML data modell. In our example application the addItemToCart operation is frequently invoked and has to be considered expensive in terms of runtime performance. Especially the update operations on the global variables $items and $customers require exclusive locks on the corresponding XML elements. Depending on the XML storage the applied locking granularity varies and thereby the associated costs for either reading and writing of global and conversational variables increase.

```
   !! query items
   !! required input:
   operation queryItems [#element()* $input] {

190    conversationpattern supports;

   body {
     let $output :=
           for $item in $items/item
195        where (    xf:empty($input/title)
                   or $item/title eq $input/title)
           and    (    xf:empty($input/publisher)
                   or $item/publisher eq $input/publisher/text())
           and    ( xf:empty($input/subject)
200                or xf:contains($item/subject,$input/subject/text())
                   or xf:contains($item/title,$input/subject/text())
                   or xf:contains($item/desc,$input/subject/text()))
         and      ( xf:empty($input/isbn)
                   or $item/isbn eq $input/isbn)
205      and      ( xf:empty($input/discount)
                   or $item/discount lt $input/discount)
         and      ( xf:empty($input/price)
```

```
                or $item/price lt $input/price)
        and    ( xf:empty($input/id)
210              or $item/id eq $input/id)
        and    ( $item/stock gt 0.0)
        return <product>
                 { $item/id }
                 { $item/title }
215              { $item/publisher }
                 { $item/author_id }
                 { $item/subject }
                 { $item/pub_date }
                 { $item/isbn }
220              { $item/discount }
                 { $item/price }
                 { $item/desc }
               </product>;
    }
225 }
```

The operation *queryItems* provides a query interface in order to search for specific items. The content of the variable $input is matched with the item descriptions in the global variable $items. Matching in this case means, to compare the tag names of all child elements of $input and to compare the corresponding content of the specific child elements. For each child element describing an item in the global variable $items it is checked whether a corresponding element is included in $input. If this is the case, the content of the two child elements are compare.
If the variable $input contains for example:

```
<query>
   <isbn>3423244887</isbn>
   <price>12</price>
</query>
```

all items with the given ISBN number and a price of less then 12 are returned. Note, the type of comparison applied to the content of the child elements depends on the tag name, say, the type of the element. The subject element, for example, is expected to contain a keyword. It is checked, whether the given keyword is contained in either subject, title, or description text of the items returned.
Furthermore, it is checked whether still some stock is available for the returned items. Items which are not available anymore should not be reported (line 211).
This example illustrates how a complex processing and query logic could be expressed efficiently by using XQuery and the XML data modell.
In this example the operation queryItems implements the conversation pattern *supports*. If the received message is part of an conversation, the operation is executed within the context of this given conversation, say, within the context of a specific shopping session. Otherwise, the operation is executed outside a conversation.

```
!! query OrderStatus
operation orderStatus {

body {
```

```
230         let $output := $cart;
    }
   }
```

In order to retrieve the current state of a purchase order the operation *purchaseStatus* is added. The operation returns the state of a purchase order by returning the content of the conversation variable $cart. Consequently, this operation has to be invoked as part of a conversation.

```
   !! finish Order
    operation goToCounter [#element()* $input] {
235
     body {
      !! if the cart was only initialized but
      !! no lineitem added an empty message is returned
        if($cart/state/text() eq "init") {
240           return;
        }
        if(xf:empty($cart/lineitems/lineitem)) {
              return;
        }
245     !! set the default ship type
        let $shiptype := "Mail";
        !! initialize the order variable
        let $order := <order> </order>;

250     !! create new order id
        let $order_id := $order_id + 1;
        let $oid := $order_id;

        !! update the state of the purchase order
255     update $cart replace $cart/state
        with <state>pending</state>;

        !! if a shiptype is specified in the input message,
        !! use it
260     if(xf:not(xf:empty($input/shiptype))) {
            let $shiptype := $input/shiptype/text();
        }
        !! for each item in the shopping cart, the the global
        !! items database, represented by the global variable
265     !! $items, is updated. The XML element volume is increased
        !! by the number of ordered items.
        for #element()* $li in ($cart/lineitems/lineitem) {
           let $vol := $items/item[id eq $li/id]/volume;
           update $items replace $items/item[id eq $li/id]/volume
270                  with <volume>{$vol + $li/quantity}</volume>;
        }
        let #element()* $address := $cart/address;
        if(\xf:not(xf:empty($address/country_id/text()))\) {
           update $address replace $address/country_id
275         with op:item-at($countries/country[id/text()
```

```
                    eq $address/country_id/text()],1)/name;
      } else {
         update $address delete $address/country_id;
      }
      !! create an order description
      let $order := <order>
                      <state>pending</state>
                      <id>{$oid}</id>
                      <shiptype>{$shiptype}</shiptype>
                      <uname>{ $cart/customer_uname/text()}</uname>
                      <name>{
                        for $c in $customers/customer
                            where $c/uname/text()
                                  eq $cart/customer_uname/text()
                        return
                          xf:concat(xf:concat($c/fname/text()," "),
                           $c/lname/text())}
                      </name>
                     {$address}
                     { $cart/lineitems }
                     { $cart/total }
                     <date>{xf:current-dateTime()}</date>
                    </order>;
      !! update the globale order variable
      update $orders insert $order into $orders;
      !! update the global variable $customer and calculate the
      !! new account balance
      let $balance := $customers/customer
                  [uname eq $cart/customer_uname]/balance - $cart/total;
      update $customers replace $customers/customer
                  [uname eq $cart/customer_uname]/balance
            with <balance>{ $balance }</balance>;

      !! return the order description
      let $output := $order;
      !! for debugging purposes, print the order
      print <output>{$output}</output>;
        }
    }
```

The operation *goToCounter* finishes a purchase order and closes the corresponding conversation. a order description, including an order-id, is generated and added to a global order database (variable $orders). The global customer (variable $customers) and the item database (variable $items) are update as well. In the $input variable a dispatch or shipping type can be specified, the default value is ”Mail”. As an order acknowledgement, the generated order description is returned to the customer. A typical order description looks like:

```
- <order>
    <id>15</id>
    <shiptype>Mail</shiptype>
```

```
    <uname>andreas5</uname>
    <name>Andreas Gruenhagen</name>
    <lineitems>
      <lineitem>
         <id>5</id>
         <title> The True History of Chocolate </title>
         <backing/>
         <quantity>1</quantity>
     <subtotal>17.50</subtotal>
     <tax>3</tax>
     <shipcosts>1.44</shipcosts>
         <total>21.94</total>
     </lineitem>
    </lineitems>
    <total>21.94</total>
    <date>2006-05-17T16:42:51.2060000Z</date>
  </order>


315   !! finish Order
    operation pay [#element()* $input] {

          conversationpattern never;

320     precondition (xf:not( xf:empty($input/uname)))
              throw <error>uname is missing in {$input}</error>;
        precondition (xf:not( xf:empty($input/passwd)))
              throw <error>password is missing in {$input}</error>;
        precondition (xf:not( xf:empty($input/cc)))
325           throw "creditcard information is missing";
        precondition ($customers/customer[uname/text()
                    eq $input/uname/text()]/passwd/text()
                     eq $input/passwd/text() )
              throw "wrong password";
330     precondition (xf:not( xf:empty($input/amount)))
              throw "amount is missing in $input";
        body {

          let #element()* $i := $input;
335
          let #element()* $customer :=  for $c in $customers/customer
                                        where $c/uname eq $i/uname
                                        return $c;

340       if(\xf:not($i/passwd/text() eq $customer/passwd/text())\) {
            throw <error>wrong password:
                xf:not({$i/passwd} eq {$customer/passwd}:
                {xf:not($i/passwd/text() eq $customer/passwd/text())}
              </error>;
```

```
345             }


                let #xs:double $balance := $customer/balance + $i/amount;
                update $customers replace
350                     $customers/customer[uname eq $i/uname]/balance
                                with <balance>{$balance}</balance>;
                update $money_orders insert <money_order>
                                            { $customer/fname }
                                            { $customer/lname }
355                                         { $i/amount }
                                            { $i/cc }
                                           </money_order>
                                into $money_orders;
                let $output := <answer>Thank you for visiting XL.</answer>;
360         }
        }
```

The operation *pay* adds a specified amount to the account balance of the specified customer and generates new element money_order in the corresponding global variable.

```
        (: Bestsellers
365     operation getBestsellers [#element()* $input] {
            conversationpattern supports;

            body {
                let #element()* $elements :=
370                  \fn:subsequence(for $i in $items/item
                      order by double($i/volume) descending
                      return $i, 1, 3)\;
                let $output := for $item in $elements
                               return <product>
375                                     { $item/id }
                                        { $item/title }
                                        { $item/publisher }
                                        { $item/author_id }
                                        { $item/subject }
380                                     { $item/pub_date }
                                        { $item/isbn }
                                        { $item/discount }
                                        { $item/price }
                                       </product>;
385         }
        }
```

The operation *getBestsellers* illustrates again the use of XQuery. The list of items available in the database are order by the purchase volume for each of the individual item. The XQuery function subsequence then returns the first three items.

```
        operation unknownOp {
```

```
        body {
390         let $output :=
              <error>Operation not available</error>;
        }
    }

395 operation initOp {
        conversationpattern supports;

        body {
            nothing;
400     }
    }

endservice
```

The two operations *unknownOp* and *initOp* are empty in this example. The operation unknownOp is invoked if an undefined operation name is specified in the by SOAP message (see 4.2.3), it returns a standard error message. The operation initOp is invoked the runtime environment itself at startup time.

Requirements Review As a retrospective for this section, the aspects listed within the problem statement of this thesis (section 2.1) shall be evaluated:

Storage Integration Storage integration is hidden from the XL application developer and reduced to merely a configuration topic. Typically, global and conversation variables are backed by a persistent storage.

XML processing Since XL uses the XML data modell, XML processing is not a hindrance anymore. Using the XML query language XQuery within an high level service description language like XL provides a new quality of XML processing. The XML data modell can be directly used as a modelling concept, and not a plain form of data representation anymore.

Service Oriented Architecture Using XL, a service can be setup without using a large scale framework, like J2EE [Sun05b] or .NET [Mic06b]. The very general approach expressed by the term service oriented architecture, everything is a service, is not met only defining a new language. Additionally, the general applicability, say, scalability and performance has to be achieved. This aspect will be addressed as a major requirement in section 7.1.

Optimization The optimization aspect is not addressed by the language design but by the underlying runtime system implementing the language. Technical aspects, like index structures, caching or scheduling strategies in an XL cluster are hidden from the programmer.

Part II

The XL System

Chapter 5

Introduction

This part addresses the design of the XL Web service engine. A Web service engine provides a framework to execute Web services. For a programmer, writing a service is made easy, as common frameworks provide not only rich toolset (e.g., synchronous or asynchronous message passing, persistence) but also prescribe a certain programming language (e.g., Java) or a programming concept (e.g., object oriented programming).
The XL engine is not so much different in many ways, XL thinks of the world being XML. Each entity – messages, variables, data – is XML. The XL engine provides a runtime system (RTS) for XL programs. The XL RTS parses XL programs and translate them into an internal representation. The internal representation is interpreted when an XL operation is invoked. Furthermore, the XL engine includes the context handling, an XQuery engine, a HTTP server, and a persistent XML store.
The XL engine yet does not provide a set of distinct modules which can be used individually but an integrated XML processing framework which can be customized.
In contrast to the requirements for the XL language itself (section 2.1), the requirements for the XL runtime environment strongly depend on the usage scenario. In the following chapter the different scenarios XL is applied to are described. The term *scenario* depicts a set of applications with similar requirements, and use cases. Within a XML processing scenario typical messaging patterns occur.

Chapter 6

Scenarios & Use Cases

The number of possible scenarios an XML processing language could be used for are very diverse. The question is: which role plays XML in information technology ? XML is meant to integrate, to represent, and to describe data. A programmer implementing an XML processing application needs to have some tools available, which enable him to focus on the application logic implemented using an XML data representation.
The processing scenarios for XML are very diverse and requirements vary. In the following subsections three usage scenarios are sketched to signify the wide range of possible Web service applications.

6.1 Mobile Device

Mobile devices, such as small laptops, personal digital assistants (PDA), embedded devices, or even mobile phones interact in very heterogeneous environments. Different network connections are used (e.g., bluetooth, WLan, GSM), different platforms, different security constraints. Furthermore, applications on mobile devices are complex, different communication patterns occur and a more complex context management is necessary. The two following bullet points illustrate these two aspects:

- **Push versus Pull** A mobile phone actively announces its position to a server, information is ”pushed” by the mobile device. Likewise, a server managing a set of embedded devices in a house (e.g., fridge, heating, lighting) typically queries all devices at boot time (devices could be on, off, or standby) . In this case a service on the device has to be provided which returns for example the current state. Information is ”pulled” by the server.

- **Context Management** In an ubiquitous computing environment, containing multiple adhoc network connections, the state of a mobile device is not only the information available rightaway (e.g., location, owner of the device, temperature), but also more complex aspects like user objectives (e.g., user goes to the mensa, the lecturing hall). In the *Communications of the ACM* Joëlle Coutaz et al. [CCDG05] address the meaning of context in a mobile environment.

XML could be a tool used to express the complexity of mobile applications.
On mobile devices, such as PDAs, XML processing is necessary for several reasons. As a client for Web services provided by a server, a mobile device needs to generate XML messages and it

must be able to process the XML answer. Furthermore, XML is a possible data representation to be used within a PDA in order to represent, for example, a state shared by several applications on one or several PDAs. A state could either be given by a static configuration set externally or it could be a dynamic data structure used to communicate between applications.
In each case, the restriction on a mobile device and the use cases are different compared to an XML processing application on a server. A mobile device typically has limited CPU performance, main memory, and network bandwidth. Furthermore, on mobile devices commonly functionality is restricted and the availability of software is limited.
The use cases for XML processing on a PDA could be characterized as:

- no concurrency or a very low level of concurrency
- the degree of user interaction is higher compared to server XML applications
- since complex and time consuming operations are usually executed on a large scale server instead of a PDA, a limited query functionality compared to a server is sufficient on a PDA.
- uncertainty and unreliability of mobile applications.

6.2 Server

The second XML processing scenario depicted here is a server providing a Web service. Naturally, here the requirements are different compared to those on a mobile device. A Web service is defined as an application receiving an XML message and returning XML as an answer. Depending on the expected workload and the type of XML message different environments for executing a Web service are suitable. Possible parameters determining the characteristics of Web services are:

- state of the service and its representation
- number, size, and complexity of the expected XML messages
- application characteristic: update frequency, size of application

In the following we define the server scenario as a single computer (single or multi-processor) providing a complex Web service to a possibly large number of clients. Typically such a service relies on an underlying database. All operation calls are processed on the computer providing the service. Typically a service participates in several conversations with different clients and other servers.
The scenario is characterized as follows:

- persistent state representation and database transactions
- concurrent operation processing
- concurrent conversations
- provision and evaluation of interface descriptions (e.g., WSDL [W3C01])
- integration of other services.

6.3 Cluster

The third scenario for XML processing extends the previous standalone server by using a cluster solution. In contrast to the single server another dimension is added which triggers several new requirements:

- a suitable load-balancing strategy must be applicable. Depending on the type of application, either round robin or more sophisticated scheduling methods (e.g., data-placement aware or *sticky sessions*) could be the best choice.
- distributed state representation and storage
- a single point of failure must be avoided

Distributed processing is a very complex issue, but it is not XML-specific. As for the scope of this thesis, distributed processing, including all its consequences, is a requirement for a Web service engine. It must be possible to consider locality, among others, as a property of either data or functionality and to consider this during compilation and optimization.

Chapter 7

Architecture & Design

The concept of a service oriented architecture implies a general applicability. Hence, XML processing should be possible in as much uses cases as possible. If the requirement of a service oriented architecture (section 2.1) is to be put into action, a common XML processing concept for different scenarios is necessary. In the following, the requirements for the runtime environment itself are described.

7.1 Requirements of a Web Service Engine

By combining the requirements of an XML processing language (section 2.1) and expected usage scenarios, the requirements for a runtime environment can be defined.
As the different use cases indicate, specifying the requirements for a Web service engine is difficult. In the following subsections we distinguish between functional requirements (What should be done ?) and the non-functional requirements (How should something be done ?). The requirements of a Web service engine are motivated by its use cases.
Even though XML is meant to be the common exchange format, doing cherry picking of XML technologies is difficult:

- **Storage** Although plenty of XML storage systems have been proposed, using either relational or native approaches, storing XML is still an active research area.
- **Query Processing** The application logic processing XML in general heavily depends on the XML representation used. While in one case the application logic is implemented using Java Beans inside a J2EE framework [Sun05b], in another case application logic is expressed using XML Query [W3C04g].
- **Schema Handling** In contrast to relational data or say Java objects the XML type information is optional. In return the use of XML schema information depends on the technology used. While in some cases schema validation is optional ([DOM04]), it is mandatory in other cases ([Sun04a]).

As Web services are deployed in non-standard use cases, scalability becomes crucial. Deploying a J2EE server for instance on a mobile device to process SOAP messages would impose considerable overhead. In the following the requirements (functional and non-functional) for a Web service engine are discussed as they are motivated by the usage scenarios.

7.1.1 Functional Requirements

Functional requirements in general specify the functionality of the software, in our case, the functionality of a Web service engine. Functionality defines, literally speaking, what should be done. Which functionality has to be provided for either a programmer, an administrator, or for other services using a Web service.
In most XML processing scenarios a common set of functional requirements (FR) exists, XML data need to be parsed, transformed and possibly stored. The FR of a Web service engine could be summarized as follows:

XQuery Processing Depending on the usage scenario it should be possible to customize the XQuery engine in order to provide just the required subset of XQuery. On a small PDA device XPath and XML element construction is sufficient while in other usage scenarios user defined types and schema evaluation are necessary.

It could be distinguished between for example: path expressions, schema evaluation & type checking, complex types, element construction, user defined types & functions.

The significance of XQuery as a query language depends on how XML-centric the service engine is. If XML is merely used for external communication, XQuery certainly is not as important as for example in XL.

XML Representation In order to provide complex services state information, typically represented by a set of XML documents, needs to be managed and used within a Web service engine. The underlying XML store has to provide operations for initializing, updating and querying the XML as well as locking functionality to synchronize concurrent accesses.

An XML store could provide a persistent storage, possibly based on a relational database, a native XML store or even a filesystem. Providing a storage layer within a service engine is not the big challenge, but to integrate persistence seamlessly into the programming concept. For the Web service developer, persistence should merely be an attribute of data, not an obstacle[1].

Messaging A Web service engine must be able to support a set of different messaging formats (e.g., SOAP [W3C04c], Java RMI) and different communication patterns (e.g., synchronous, asynchronous, call back).

In addition to network messaging, local operation calls have to be possible as well. It has to be possible to easily integrate local applications (e.g., Java classes or C++ libraries, OS services, other Web service engines) into the Web service engine.

The different approaches to integrate other services or applications imply several differences:

- Different processing overheads. Parsing and transmitting an XML SOAP message over a network connection is expensive compared to a Java-only RMI call or a invoking a local Java application.
- Different functionality. Some services provide transactional functionality (Undo or Rollback operations) or type safety. Web service do not necessarily provide any guarantees.

[1] The container-managed persistence used by enterprise Java beans is one approach [Sun05b].

- Information availability. The application integration relies on the availability of meta information. Meta information could describe the service interface (e.g., WSDL [W3C01]), availability, legal information, etc. Some meta information can only be used by a software developer himself, but some information could also be utilized by the Web service engine directly.

The Web service engine has to be able to deal with these differences.

Network Different network protocols (e.g., HTTP [RF99], SMTP [Pos82], BEEP [Net03], or non-standard protocols) could be provided. For some use cases it might even be necessary to deactivate network communication completely (because it is not needed or for security issues).

Standard Compliance The provided Web services must be compliant with the major Internet-related standards such as HTTP [RF99], XML[W3C04a], XML/Schema [W3C06b], SOAP [W3C04c], WSDL [W3C01], UDDI [UDD].

Web service engines could be rated as an enabling technology providing a framework to build arbitrary applications. A Web service engine itself provides a set of means for the development of XML applications. Since usage scenarios vary considerably, the service engine has to be scalable with reference to the deployed functionality.

7.1.2 Non-Functional Requirements

The non-functional requirements (NFR) specify *how* a certain functionality should be provided. NFRs describe the qualities of the given software, for example security, performance, scalability, maintainability, reliability, or usability. NFRs are objectives, typically set up during the software analysis phase. The NFRs themselves have to be documented, including and the dependencies between different NFRs. Finally, it is necessary to evaluate the achievement of objectives. For being able to determine this achievement, NFRs have to be expressed by means of measurable variables. The NFR performance for example could be expressed by runtime figures or the main memory footprint. Typically, achieving a higher level of security requires more checks, more additional code to be executed. As a consequence, these two NFRs could be in conflict.
In general, the non-functional requirements are far more difficult to specify, implement, and evaluate. The XML processing engine requires efficient scalability with regard to the depicted usage scenarios.
In the two following paragraphs the NFRs scalability and performance are discussed.

Scalability As the different use cases described in section 6, indicate, XML processing applications are deployed on all kinds of platforms. The diversity of usage scenarios and requirements for Web services listed in the previous sections require a scalable Web service engine.
The term *scalability* in general expresses the ability to adapt a given system with reference to another, functional or non-functional, requirement. The term scalability has to be distinguished from the term flexibility. Compared to scalability, the term *flexibility* denominates a more general requirement. Flexible software can be customized, updated or configured depending on specific requirements. Scalable software denominates to possibility to easily integrate new resources or functionality into a given framework. Flexibility incorporates further non-functional

requirements, as for example usability, maintainability not addresses by scalability. Nevertheless, both terms cannot be delineated separately.
Scaling functionality has been addressed by the previous section 7.1.1.
Scaling a system means, for instance, to improve the performance, improve the security and so on. In any case improving implies to change a system in order to meet a certain requirement. Since NFR strongly depend on each other, improving a system with reference to one requirement could either improve or impair a system according to other requirements. One could say scalability is a Meta-NFR, specifying whether one could customize a system for dealing with changes of other NFRs . In this section, scalability with reference to NFRs is examined. A typical NFR for a Web service is performance. Depending on the actual application and the usage scenario, performance is measured differently: response time, throughput, start up time, memory or power consumption.
On an application server, scalability is for example achieved by using resource pooling or caching. In a distributed environment, the number of slaves used to scale up a cluster, is a means to scale the system.
The key element of a scalable architecture is a flexible granularity. The term *granularity* denotes the level of complexity within the architecture. Flexible granularity is achieved, if it is possible to either split or merge components. By doing so, very coarse grained components can be created, providing a simpler interface with possibly less functionality. Alternatively, it must be possible to design fine grained components providing a complex interface.
It is neither possible to foresee every future requirement nor to implement all current requirements without conflicts. Scalability is achieved if guidelines are setup on how to adapt a system. Naturally, these guidelines have to be defined on a very abstract level.
The XML store for example could be used to illustrate granularity in the XL architecture. The XML store in XL encapsulates XML data instances used within the engine, whereas the data itself can be either held in main memory or backed by a relational database. On a mobile device, a simple but lightweight main memory store could be sufficient. Use cases for a simple store are merely to retrieve, update, and delete data at certain positions. Correspondingly, the store interface is small, say, very coarse grained.
On a server the simple store would do its job, but for scaling performance a more sophisticated, fine grained interface is necessary. Several enhancements are possible, as for example: the store could provide meta information for the XQuery engine (e.g., size, dynamic schema information, index structures), multiuser synchronization and possibly different isolation levels in the relational database beneath the store.
In a cluster environment a distributed XML store requires an even finer grained interface. In a distributed XML store data location is an attribute of a store element. Access to remote XML store elements could be implemented using the common proxy-stub pattern, again this would require a finer grained store interface compared to a standalone server.

NFRs for a Web service engine

The key element of scalability is to identify components which could be either split up or merged. As for a Web service engine, interface changes must not require changes in architecture.
If a single architecture is to encompass different use cases, scalability is required. If for example persistent storage is not needed on mobile devices, it must be possible to switch it off – and reduce the memory footprint in return. If a more efficient communication protocol can be used

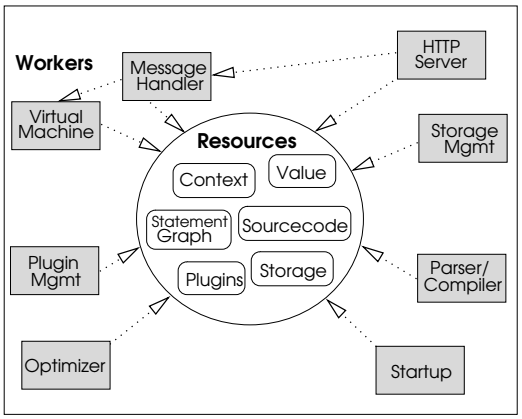


Figure 7.1: *Worker components use resources. This figure illustrates the conceptual model of the patterns used in the XL engine. Resources represent the state of the engine while workers use either these resources, or other workers to provide a service internally or externally.*

in a cluster, it must be possible to integrate it. Therefore, scalability is the most important non-functional requirement for a Web service engine.
The architecture principles we present are based on common design patterns and provide the necessary adjustability to build a scalable system.
While some top down approaches on how to consider NFR [MCN92] are available, they do not provide suitable design patterns. What is needed are software design patterns, to be used throughout the whole system, which by themselves provide scalability.
By using the design patterns we propose, active and passive components within the architecture are separated. Scalability is achieved by identifying these active and passive components and documenting their dependencies. In the following sections these common design patterns are identified and applied to all modules in the system.
In our concept, the term *active* denotes a component which is started, executed, and stopped afterwards. In Java terms, it implements the Runnable interface. *Passive* components in return, represent for example, a certain state or an object being used by an active component. In the following, the term *worker* will refer to an active component. The term *resource* will refer to a passive component Figure 7.1 illustrates the conceptual model of the patterns used in the XL engine. Passive resources represent the state of the engine while active workers use either these resources, or other workers to provide a service internally or externally.

7.2 Design Principles

By using the resource (passive) and worker (active) patterns, several design questions are raised which are relevant for a scalable system. For each module the following decisions need to be made:

- Identify active components (workers) and passive components(resources) necessary to provide the required functionality:
  - **Active Components** Functionality can be delegated to subworkers which in return could be called either synchronously or asynchronously.

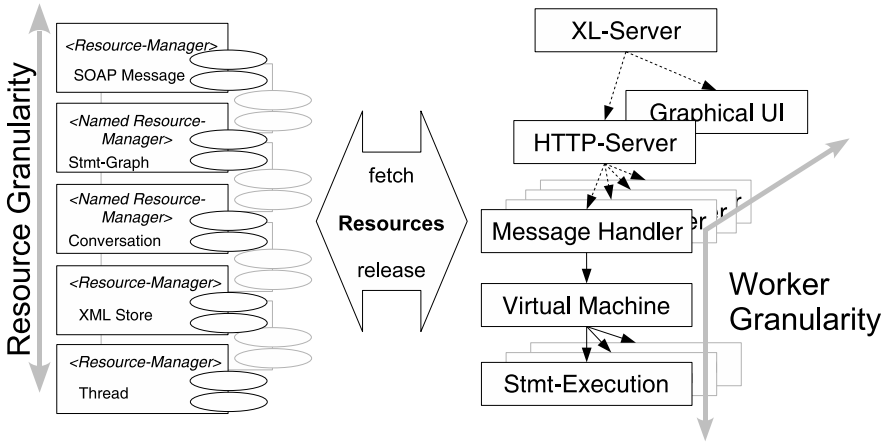


Figure 7.2: . *Resources, depicted on the left, are used by a tree of workers shown on the right. In order to scale the engine, either resources or workers could be merged. Workers could be either merged with their parent / child worker or with sibling workers.*

  - **Passive Components** Resources required by the worker. Each resources could be handled by a dedicated resource manager directly or it could be part of a bigger resource which therefore needs to be locked.

- Dependencies between workers and resources must be described.

In section 7.1.2 we defined the term scalability as the ability to adapt a system with reference to another NFR. In order to adapt a system to meet a certain objective, flexibility is needed. Flexibility in software design is determined by the interfaces available, complex interfaces provide flexibility. In order to design a scalable system one must therefore be able to adapt interfaces used for interaction between components. After identifying these resources one could decide on how to group or how to split them. By doing so, one scales the complexity of a resources.
Design patterns like resources and workers are not new and are used frequently. In the Linda programming language [Gel85], tuples correspond to our resource concept, but are used for executing distributed processes. As for the architecture of the XL engine, these patterns are a means of building a scalable system. Once the resources required by a certain worker are identified, it can be decided on how to group them. Likewise, workers themselves can be merged.
Figure 7.2 illustrates the scalability provided by the worker / resource pattern. Resource could be merged or split up, depending on how they are used: resource used by the same worker each time could be merged. Workers using other workers and resources effectively build a usage tree. In this tree, sibling workers could be merged to create simple, but more coarse grained interfaces. Likewise, workers in a parent - child relationship could be merged.

7.3 Patterns

In the following, the major design patterns to be used within a Web service implementation are described. In this section, the term *architecture* denotes the fundamental organization of a system, expressed by its components, their dependencies, and the principles governing its design and evolution ([IEE00]). The term *module* denotes a part of the system which provides

a certain functionality. In a Web service engine for instance, the virtual machine, the storage or the HTTP-Server are modules. The term *component* denotes the most general element within our architecture model. A component can use and can be used by other components.

Resources Types Resources are the objects within the architecture, resources are used by other components. Resources for example represent a certain state, provide access to the operating system, or simply represent the result of an evaluated expressions. Resources are in a sense passive, as they do not do anything – resource are not active components. In general resources typically are:

- "*heavy-weight*" – meaning expensive to create
- sparse – their number could be limited
- reused – it could make sense to reuse resources, even if holding a pool of resources implies additional costs.
- managed according to some policy

Together with each resource a corresponding resource manager needs to be specified. The resource manager creates and disposes resources according to some policy. Managing components in our architecture means, that creating and disposing resources is delegated to an associated resource manager. Resource managers typically:

- are shared among several threads, access to resource managers is always synchronized.
- can be resources themselves.
- can be combined to reduce the lock granularity.

The idea behind using a resource manager is not only the pooling and reusing of resources, but also the synchronization between threads. While a resource manager is accessed, synchronization is necessary, but once a certain resource is exclusively locked by a component it can be used without any further synchronization.
In our terminology, acquiring or releasing a *lock* denotes setting or releasing the semaphore protecting this resource. In the following paragraphs the two generic resources types are described.

- **Stateless Resource** Stateless resources are pooled objects. Any component which needs an anonymous resource fetches one from the corresponding resource manager, uses the resource and releases it afterwards. While it is being used, the resource is locked exclusively by a component. After returning a resource to the pool, it does not contain any state. Figure 7.3 shows the state graph of resources on the left.
  Configuring a system means to define a strategy on how to handle each of the different resources. This could mean for example:
  - do not pool a resource, create it on demand and dispose it as soon as it is not needed anymore.
  - create a pool of resources with a resource specific garbage collection strategy

  Typical stateless resources in a service architecture are for example value storage, database connections and threads available in the system.

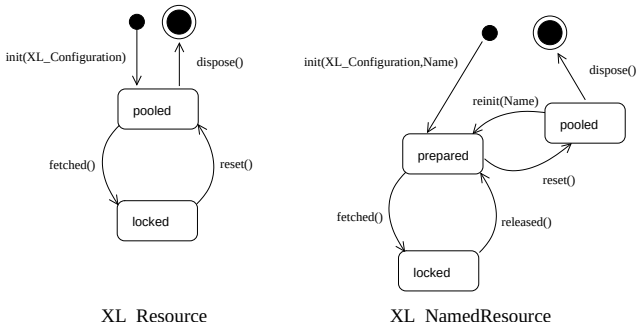


Figure 7.3: *The UML state diagram for a stateless resource and named resources*

- **Named Resources** A special group of resources are *named resources*. Each named resource has a name (ID) which uniquely identifies it. Named Resources typically provide context information associated to their name. Compared to the stateless resource the named resource requires a different resource manager. Named resources:

  – could be either locked in a shared or exclusive mode.

  – need to be explicitly released, i.e., the information associated to a certain name is not needed anymore.

  – After releasing named resources, they could be reused, possibly with a different name.

  Figure 7.3 shows the state graph of named resources on the right. Just as the stateless resources the named resources have to be considered '*heavy-weight*" and sparse – they are worth being reused. Typical state resources in a service architecture are for example session context information, operation bytecode or persistent variables.

Resource Management Corresponding to the two resource types, the basic resource managers are defined. Based on these types, more sophisticated resource handling strategies could be deployed:

- In order to implement a queuing or scheduling strategy (FIFO, LRU) new types of resource managers have be added. A resource scheduler does not create or dispose resources, but impose a certain scheduling strategy considering for example specific attributes of the handled resource (e.g., for priority queuing).

- Acquiring locks could be bound to a certain predicate. For example a lock on a certain resource is granted if the worker requesting it can present a lock on another resource of a certain type (for authorization or authentication, security is an issue).

Resource managers are the only synchronization points within the service architecture.
When locks on resources are held and released, deadlocks can occur. For deadlock detection and prevention literature provides several algorithms (Silberschatz et al. [SGG03]) which could be applied if necessary.

Worker The active components in the system are *workers*. Each worker is characterized as follows:

- "*light-weight*"
- implements the Runnable interface
- fetches, uses and release resources as they are needed.

Each worker is created on demand, as it is started, it provides a certain functionality by either performing some operation or by starting subworkers. To execute a workers, a set of worker-specific resources is required (e.g., a thread). Resource allocation is nonpreemptive. To start a new subworker either the currently owned thread of a worker is passed to a subworker by just calling the execute function, or a new thread resource is fetched to start a subworker asynchronously. In the XL engine, the following components are considered workers: the HTTP server, the XL virtual machine, and the optimizer.
Scalability is achieved if one can specify components which could be either merged or split up. The structure set by the resource and worker patterns (worker using other workers and resources) help identifying these components.

Chapter 8

The XL Engine

This section describes the implementation of the XL Web service engine in detail.

Each of the following subsections addresses a certain part (or module). In each case the used concepts, the separation into resources and workers used within the module, and the assumption are explained and design alternatives are discussed. The purpose of this section is to describe the implementation of the XL engine, explain how the design patterns introduced in the previous section are used to implement the XL engine. Syntax and semantics of XL are described in section 4.

Figure 8.1 provides an overview architecture picture of the XL engine. The arrows labeled M-in and M-out illustrate the flow of messages through the engine. An incoming message is received by, for example, a HTTP server. In a cluster environment, a scheduling strategy is applied before parsing the message. In the XL runtime system, depicted in the center of figure 8.1, we distinguish between different message types (e.g., synchronous operation call, asynchronous operation call, debug message), each handled by a different message handler. Inside the runtime system we distinguish between several different modules: a virtual machine containing a pool of threads, context representations holding the references for global and conversational variables, an XQuery engine, and an XML store. The XL compiler is displayed at the bottom of figure 8.1, separated from the XL runtime system.

The XL engine is implemented using the object oriented programming language Java [Sun05a]. By using Java the XL engine uses the features provided by Java, as for example the garbage collection. The resources and resource managers used by the XL engine circumvent the Java garbage collection by pooling and reusing resources. Resource management is expensive in any case, either by the Java virtual machine or by XL. Each time the resource manager is accessed or an object is disposed, synchronization between the different threads becomes necessary. Implementing a new resource management inside XL has several advantages compared to the Java garbage collection: the granularity of the resource management can be adapted depending on the resource granularity.

In the following sections the terms *class*, *method*, or *object* refer to the corresponding Java terms.

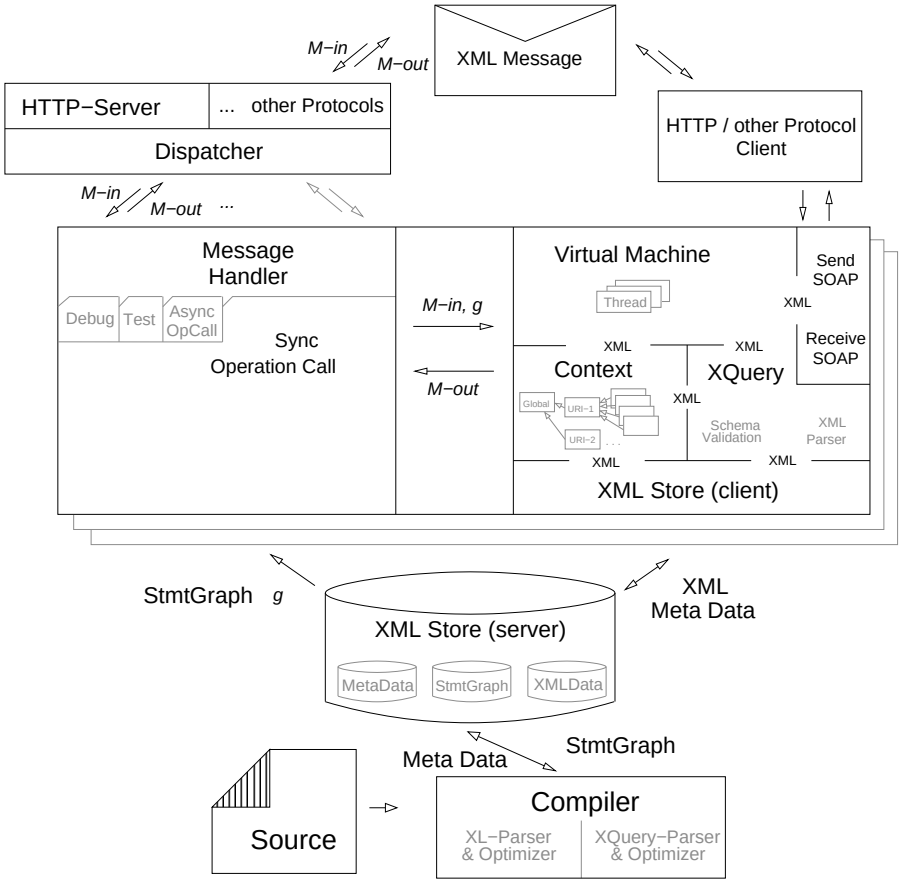


Figure 8.1: *Architecture of the XL Engine.*

8.1 Compiler & Statement Graph

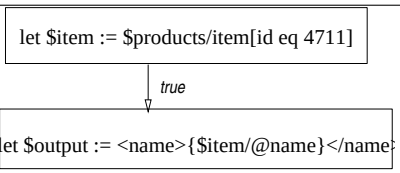
8.1.1 Parser (Worker)

As with any programming language, the XL source code has to be parsed and compiled. Therefore, a standard parser (generated by JavaCC [VS05]in our case) parses the XL code and generates a Java object model representing the given XL program.
Each syntax element of the XL language is represented by a corresponding Java Bean class derived from one of the two of generic classes XL_Statement and XL_Expression. The if-clause, for example, is represented by an object containing a conditional expression and one or two command blocks representing the true or false cases. A block command is represented by a simple list of specific command objects. The root of the parsed service is a service element which contains for example the URI of the parsed service and the list of Web service operations. An operation in return is again represented by a single object.
The XL object modell created by the parser contains for each syntactical element of XL a specific class. These classes only represent the parsed program as it is. Each object representing a syntactical element (e.g., assignment, send statement) contains line information indicating the line number in the originally parsed program. Line numbers are necessary for building an XL debugger. The classes of the of this XL object modell are plain Java Beans providing *get* and *set* methods for their specific subelements. Each Java class representing a syntax element of XL is derived from the base class XL_Stmt.
The object modell representing a parsed XL service is the interface between the parser and the subsequent compiler and optimizer. In the XL engine, different parsers can be used[1] as long as the same object modell is generated.
In the next processing step this object model is translated into a statement graph as described by the next paragraph.

8.1.2 Statement Graph (Named Resource)

The initial object model is translated into a directed graph of XL statements. The nodes of this graph represent a semantic operation interpreted by the XL virtual machine (VM). The edges in the statement graph represent the control flow between statements. Each edge is labeled by a conditional XQuery expression. After executing a statement all expressions connected to outgoing edges are evaluated. If an expression evaluates to ”true”, the subsequent statement has to be executed as well. Table 8.1 shows the basic imperative programming elements and the corresponding graph structure.
The statement graph is a directed, attributed graph. Sequences of statements are represented as a sequence of connected statements. Conditional terms, like if-statements, are expressed as a branch: two outgoing edges in the statement graph (combinators) labeled with two alternative predicates, say, the true and false cases of an if statement (see table 8.1). Likewise, parallel statements can be expressed very easily, after an initial statement, two outgoing combinators are labeled with identical predicates, for example TRUE. After the initial statement has been executed, an XL virtual machine evaluates the predicates associated to the outgoing combinators, if a predicate evaluates to TRUE, the subsequent statement is scheduled for execution. If more then one statement is scheduled, a virtual machine can execute them concurrently or sequentially in arbitrary order.

[1] Parsers generated by different parser generaters, e.g., JavaCC [VS05] or AntLR[Par05]

XL Sourcecode	Statement Graph
`let $item := $products/item[id = 4711];` `let $output := <name> {$item/@name} </name>;`	

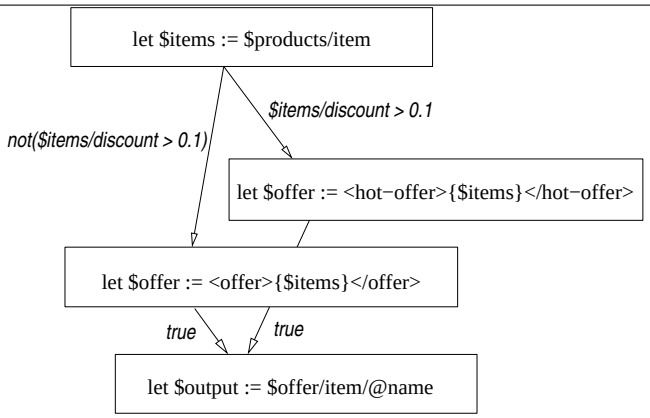
`let $items := $products/item`
`if(items/discount > 0.1) {`
`  let $offer := <bargain> {$items} </bargain>;`
`} else {`
`  let $offer := <offers> {$items} </offers>;`
`}`
`let $output := $offer/item/@name;`
	

Table 8.1: *XL programs and the corresponding statement graphs*

In the following, the term *statement* denotes a semantic operation supported by the XL virtual machine. These statements represent the logical algebra of the XL virtual machine.
It has to be distinguished between the XL algebra, representing the logical algebra, and the physically implemented XL virtual machine. In the XL implementation, some statements of the logical algebra are implemented several times, a service invocation can either implemented by sending an XML message or by performing a local operation call. Furthermore, depending on the physical implementation of the XL algebra, additional physical statements have to be added (e.g., a synchronize statement) or several logical statements are grouped together into one physical statement (e.g., pipeline statement).
The XL algebra contains the following statements:

Assignment The assignment evaluates an arbitrary XQuery expression (R-Value) in a given context and associates the result with the variable name given on the left side. If a local variable of this name does not exist in the given context, it is created. In assign statements, XL always makes a deep copy.

Send Statement The send statement send a SOAP message to a given address. Two expressions have to be evaluated, one providing the content to be send and one specifying the URI of the targeted service and the operation to be called. The message is send at best effort using the specified protocol – if supported by the XL VM. The actual message format (meaning the SOAP-dialect being used) is determined by the XL VM itself and is not meant to be changed by the user. The XL send statement does not support multicast messages.

In order to ensure the correct correlation between corresponding messages the XL engine uses a two internal variables. Each time a message is sent, an internal global variable (`$_id`)[2] is increased by one, and the new value is assigned to a temporary variable used by a send- and the corresponding receive statement. Internally the local variable links the send- and receive statement. Furthermore, the integer value identifying the messages internally is added as a the SOAP message header element.

Receive Statement As counterpart for the send statement the receive statement waits for a certain message. Typically the expected message is the reply to an previously send message. The received message is evaluated and if a variable name is given the content of the reply messages is associated it.
An asynchronous operation call is expressed in the XL statement graph by a single send statement. A synchronous is expressed by a send and a subsequent receive statement.

Send and receive statements in XL only support the HTTP protocol [RF99] using the SOAP message format [W3C04c].

Begin- and Endblock Statement The scope of variables in XL programs are represented by these two statements. Conceptually, these statements are not necessary. These statements are used because implementing the code generation becomes easier if begin- and End-Block statements are explicitly represented.

Map Statement Compared to the other statements the map statement is fairly complex. The motivation behind the map statement is to execute statements once for each element within a sequence of nodes specified by an expression. The typical use case of the map statement is to send an email for each entry in a customer list.
The map statement is the corresponding element in the abstract statement graph.
The map statement evaluates an expression which returns a sequence of elements. This sequence of elements can be materialized and stored in a internal variable. Alternatively, the iterator executing the XQuery expression is held open until the expression either completely evaluated or execution context is disposed. XL currently uses the latter approach. Each time the map statement is executed it advances in the initial sequence by one element and binds the current element to a given variable name. Additionally, a local boolean variable is set to true or false whether the end of the list is reached or not.

Using this map-statement and a conditional goto-statement evaluating the boolean variable the XL for statement (see section 4.4.2) is implemented in the logical and physical algebra.

An example of a for-statement being used in an XL program is given in section 4.6, at line 267 on page 60 a for-statement recalculates foreach purchased item the volume of remaining items in stock.

Throw *Throw* statements in XL are used by the programmer usually in case of an error. The normal execution is aborted and error handling is instantiated. A *throw* statement has a single expression parameter, which meant to describe the error raised. XL does not apply an error type handling, like Java does. The XL throw statement signals the runtime system an error condition. The runtime system either continues the execution in a catch block of

[2] internal variables in XL always have names starting with an underscore '_'

a try-catch element or the execution is aborted and the exception is communicated to the caller.

Print For the convenience, a print statement was added to XL. The print statement evaluates an XQuery expression and prints the value on standard out.

Sync Statement In the statement graph, parallelism is expressed simply by creating two outgoing combinators, labeled with the same predicate.

Additionally to these basic statements the XL includes a set of further statements which either apply an optimization or add convenience for the XL programmer:

Update Statements Any update operation can be expressed using plain assign statements. But if you think of updating a single element (e.g., a lineitem in a big purchase order), an assignment would impose a considerable overhead. Instead, a small set of update statements is defined, modeled after the emerging XUpdate standard [CFR06]. Each update statement updates exactly one variable. Each update statement initially names the update variable and then the specific update operation (insert, delete, or replace). The syntactical update statements described by the XL syntax (section 4.4.2) directly correspond to the XL statements executed by the VM. How each of the update statements is actually executed depends on the implementation of the actual storage used by the updated variable.

- **Insert** The *insert* statement inserts new XML data at a certain position into an XML node specified by a certain variable. The following statement, for example, inserts a new lineitem into a purchase order:

  ```
  update $order insert <item>{$data}</item> into $order/items
  ```

  First, the XL engine evaluates the expression specifying the insert postion, *$order/items* in this case. The insert postion is specified as an XML element, the new content is inserted either *into* (into the current element), *before* (before the start element), or *after* the element. The new . Furthermore, for ”insert into” it is distinguished between two different postions: *into first* (after the start element – default), and *into last* (last child element of current one – default) .

  Insert position and new content are passed to the store interface.

- **Delete** The *delete* statement deletes an XML element within the specified variable. The position of the element is specified by a XQuery expression.

  ```
  update $order delete $order/items[id eq "4711"]
  ```

- **Replace** The *replace* statement replaces an XML element. Replace postion and new content are specified as XQuery expressions.

  ```
  update $order replace $order/items[id eq "4711"]/amount
    with <amount>{$order/items[id eq "4711"]/amount * $discount}</amount>
  ```

  In the example above, an XML element ”amount” is replaced by a new element. Note: in this example the relevent XML element ”amount” has to be located twice the variable order: once for specifying the replace position and second time for creating the new element. Since this is a very frequent use case, evaluating the path expression twice should be avoid (for example by introducing a $this variable, referencing the update postion)

For all update statements the insert position has to be an XML element. Due to the properties of XML storage used, it is not possible to replace just the content of an XML element.

Operation Call In XL, an operation is invoked by sending an message. Sending a message is a concept of generating an event and broadcasting it. The Web service engine has to choose the way of expressing and communicating the event depending on the targeted recipient. The Web service paradigm of loosely coupled applications implies a late binding, say, the Web service engine evaluates the WSDL description [W3C01] of the targeted service and selects the appropriate messaging protocol and format. In XL the URI of an invoked Web service is specified by an XQuery expression, network protocol (HTTP) and messaging format (SOAP) are not variable. XL does analyse service invocations at compile time and distinguishes between three different invocation statements. The remote service invocation is expressed by a send and a subsequent receive statement. By using these statements XL requires the remote service to understand XML, say, SOAP messages. The SOAP messaging protocol implies a considerable overhead, XML messages have to be generated, a network connect has to be setup, the message is transmitted and parsed and so on. In the following the two additional statements used to optimize service invocation are described:

- **Local Operation Call** In order to avoid this overhead for local operation calls an new statement was added, which executes a local operation call without the SOAP detour. In the XL virtual machine a new execution context is selected and the specified operation is invoked within the same Java virtual machine.
- **Java Operation Call** XL supports the integration of external applications

Graph generator (Worker) As for any programming language, the XL compiler contains several phases. After parsing the XL program, the generated object modell has to be transformed into the corresponding statement graph. The approach used by XL is a simple replacement: the object within parser generated object modell are treated as nodes in the statement graph. An if-clause, represented by an XL if-object contains a reference to the if predicate, a block of statements representing the TRUE-case, and optionally another block of statements. The graph generator removes the if-object and replaces it by a branch. Two combinators are created, one labeled with the if predicate, the other one with the it negation. In a second step every predicate evaluation is turned into an assignment to a temporary local variable. The combinators in the final statement graph are always labeled by variables, or negations of variables. The transformation applied to an if-clause is illustrated in figure 8.2.
In same style, iteratively every object generated by the parser is replaced and a complex statement graph is created. Note: as these replacements are applied, syntactic sugar is removed. A possibly complex XL operation is represented by a statement graph containing a only a small number of different statement types. Thereby, no information is lost, but certain context information is difficult to retrieve afterwards. In the statement graph, begin and end of nested if-clauses can be only be determined by complex graph analysis. If this type of structure or dataflow information is required throughout the optimization process, the statement graph can simply enhanced by specific comments or NOP statements (no-operation) providing the required information. Depending on the later execution modell, for example meta information on the data or control flow could be added.

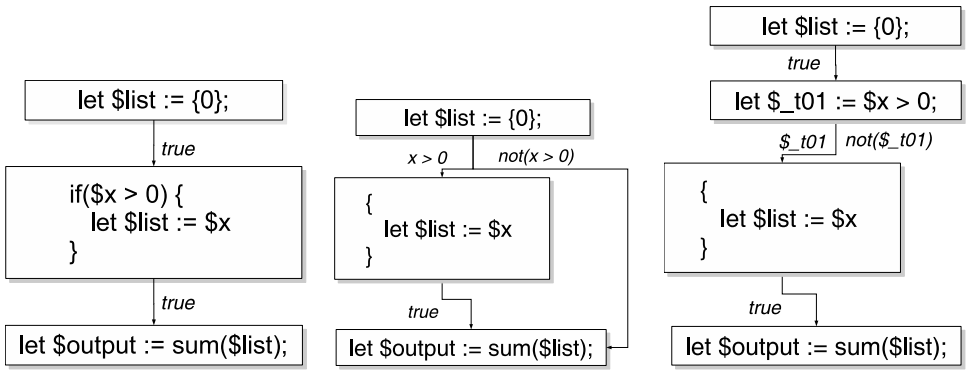


Figure 8.2: *XL statement graph generator, an if-object as is replaced by statements and combinators*

8.1.3 Optimization

Based on the statement graph standard compiler techniques can be deployed as they are described in [ASU86]. The optimizer analyses the XL statement graph and applies platform independent transformations, which reduce either runtime or memory consumption. The output generated by the optimizer is again a valid statement graph. The XL optimizer contains a set of rules, each specifies a certain conditions and a transformation of the graph. An optimization controller selects an strategy on which rule are to applied. A more detailed description of the XL optimizer is given in [Wit03].

Code generation

Since the XL engine is implemented in Java, the statement graph has to be transformed into a representation which easily can be interpreted by an Java VM. Code generation maps the abstract statement graph onto the XL virtual machine. Naturally, code generation depends on the features of the virtual machine and vice versa. The generated code has to use the means the underlying virtual machine provides. The means of a virtual machine are the processing modell, including the processor, its registers, and for example the different processing pipelines of the CPU. For XL, not a physical processor is used but a virtual machine which is based on an XQuery engine. XL is a high level interpreted language, the resources managed by the XL engine are not registers but XML content.
The alternative processing concepts applicable for XL are described in detail in section 8.4. Alongside, the necessary code generation is explained.

8.2 Expressions

The XL language uses XQuery [W3C05d] for any type of expression. Since XL variables could either be very small or very big, the XQuery engine and its integration into the XL runtime system is difficult. Variables are physically represented differently, main memory or relational database. Variables have different sizes, single byte character to large scale, database style variables of Mega or Gigabyte sizes. Furthermore, variables are used differently within XL. Some variables are initialized once and never updated, while others are updated very frequently. Variables could be changed completely or just be modified by replacing some elements. Likewise, variables could be read as a whole or just single elements in a large document.

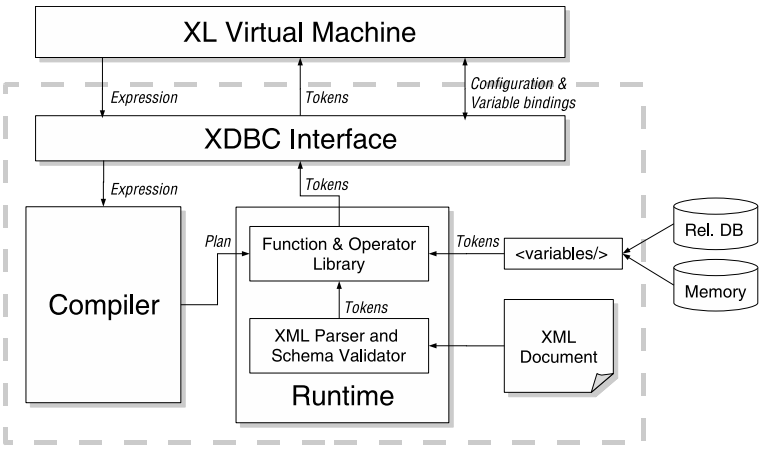


Figure 8.3: *Architecture of the BEA XQuery Engine used by XL (source [FHK+03]).*

In order to integrate the XQuery engine into XL a generic approach was taken. For XL, XQuery expressions are merely black boxes: an expression uses a set of input variables and returns a data of a certain type if evaluated. XL, for now, does not analyze the expressions on how variables are used. .

The BEA XQuery engine used by XL provides a JDBC-like interface called XDBC [FHK+03], containing a so called Connection element which represents the execution context of an expression. Similar to JDBC, executing an XQuery expression starts by creating an XDBC connection. Each connection provides a method to create prepared expressions which are executed in the context of this connection. Like JDBC, one connection should only be used by one thread at a time. The JDBC-Connection concept is enhanced in XDBC by establishing a parent-child relationship between connections, the child connection inherits all context information from its parent (namespace, function definitions or import declarations).

Figure 8.3 gives an overview of the BEA XQuery engine. For each expressions in XL, a *prepared expression* object is created[3]. The prepared expressions are created using the context information available within the given XDBC connection. The expression is optimized according to the type information, namespace, and function definitions within the connection. In order to evaluate an XQuery expression in XL, an XDBC connection and prepared expression objects need to be created. Like the JDBC PreparedStatements, the XDBC prepared expressions interface allows it to set external variables. The term 'external' in this case refers to external variables for a single expression, say, variables managed by the XL engine. Before evaluating an expression, the XL engine sets the corresponding variables inside the expression. Furthermore, the prepared expression provides type information describing the expression it represents.

Since neither the connection itself nor the prepared expressions hold state information after being used, they can be reused. Speaking of XL design patterns, an XDBC connection and prepared expressions are a stateless resource managed by the XL engine, like a JDBC connection pool. Since in XL, typically, an operation is executed completely, all expressions used in an operation are grouped together into one resource (class `XL_RTContext`). If the same operation is invoked a second time, the prepared expressions and the XDBC connection could be reused. If the same operation is invoked concurrently a second time, a second set of prepared expressions and a connection are required. In the statement graph, it is not distinguished be-

[3]Like JDBC, XDBC refers to compiled expressions as *PreparedStatements*. In order to avoid name conflicts with XL statements the term *prepared expression* will be used in this thesis.

tween different expression types. Each expression is represented by an object created by the parser (class `XL_Expression`), which only contains the expression string and methods returning the variablenames used by the expression. The link between the prepared expression, containing the compile query plan, and the static expression in the statement graph is created by an integer. Since the number of expressions used in an XL operation is fixed, the corresponding prepared expression is specified by an integer index value in an array of prepared expressions.
The BEA XQuery engine is based on the common iterator modell [Gra93]. Expressions are translated into a tree of operators, implemented using the iterator pattern. Each iterator consumes 0 to N input iterators, applies a certain algorithm (e.g., sort, map, or join) and generates a sequence of output elements. The iterator pattern is designed to process sets of elements using a pull concept. The iterator concept is modular, has low memory requirements, and avoids, due to the pull concept, unnecessary computations. Data essentially flows through the iterator tree.
The XML data model used by the query engine, a Token sequence representation, is described in section 3.2.3.
XL uses the same XML data model for representing any type of XML data. Since every expression in XL is XQuery, XL set the external variables used by an expression and initiates the evaluation. The result of evaluated expression is represented by a token iterator, the expression is evaluated by consuming the iterator. Typically, XL consumes a token iterator and materializes the token sequence in a main memory token store.

8.3 Context

As for all programs executed by a computer, XL operations require a context. This context holds the dynamic information necessary to execute the individual statements in the statement graph. The virtual machine uses the context to interpret the statements.
In order to evaluate an XQuery expression, context information is necessary, for example: namespace definitions, user defined functions, and variables. Variables, or more specific, XML data is managed by a store object described in section 8.5. The context associates the XML data with variable names.
As described in section 8.2 executing an XQuery expression in the XL engine is a two step process. First a prepared expression object is created which is executed afterwards. As a prepared expression object for each XQuery expression in an operation is created, a query plan is set up which depend on the context information provided.
The variables of an XL Web service are represented by a tree structure:
global Context (global Variables) → conversation contexts (conversation Variables) → operation context (local Variables)
Global variables are accessible from every operation, conversational variables are only accessible form operations being executed within a certain conversation and local variables are only accessible within the operation itself.
The distinction between different variable scopes is necessary, since the actual representation of a variable depends on its scope. Depending on the usage scenario, global and conversational variables are possibly persistent and need to be backed by a database. Database access is provided by a generic store object which contains methods for initializing, retrieving and updating variables. The store interface is described in more detail in section 8.5.

Workers & Resources The context representation is split in two resources:

- *Variables* Variable references for the different scope levels
- *XQuery context* The context for executing XQuery expressions is defined by a connection object provided by the XQuery engine.

Discussion The handling of store resources in XL depends on the scope of the variables they are containing. Store objects containing global variables are plain resources, the individual store simply represents a connection to the storage beneath it. To execute an operation inside a conversation, the store resource containing the conversational variables has to specifically reference the variable instances of the given conversation. To execute several operations concurrently within the same conversation several store objects corresponding to the conversation are required. Each conversation is therefore managed by named resources, identified by the conversation URI, which in return manages all store objects for this specific conversation.
Although the two resources, query context and variable scope, are managed separately, they conceptually depend strongly on each other. Especially for query optimization the characteristic of the XML data representation, determined by the variable scope, is important. Performance characteristics of the storage types certainly differ, therefore, accessing a small XML element in main memory is fast, but does not provide any index support.
If the storage properties are considered when optimizing query expressions, these two components need to be treated as a common resource. A prepared expression should only be used in combination with the storage it has been optimized for.

8.4 Virtual Machine

The XL virtual machine (VM) executes XL operations. According to the XL concept of workers and resources the VM is a worker. When an operaton call is issued, a new instance of a virtual machine is created, the required resources are fetched, and the specified operation is executed. Before explaining the different processing concepts, the resources available to the VM have to be explained:

Statement Graph The statement graph of an operation represents the compiled XL operation.

The statement graph, or respectively the generated code for a specific VM, could either be serialized in order to store it on a hard disc or a database. Currently, the XL statement graph is kept in main memory. The statement graph is translated into a physical algebra interpreted by the VM.

Store In order to execute an operation, data needs to be managed. In this context, the term data refers to the content of a variable. A *store* in XL refers to, in database terms, a connection to a database. A single store, or say, database connection, is used to back a set of values, each representing the content of one XL variable – referred to as *XL_Values*. Via this store, XML data can be retrieved from the database and is transformed into the token sequence representation used by the XQuery engine. Furthermore, the store provides methods to perform either of update operations specified in section 8.1. If the used store supports ACID transactions, read and update operations on a set of XL_Values can be handled as a database transaction. Either a single statement, a set of statements or a complete operations is executed as a transaction. Each XML variable stored in a persistent database is uniquely identified by an ID, usually an URI composed out of the

URI of the service itself, the variable name, and possibly some more information (e.g., a conversation id). At start-up time a Web service engine recreates these URIs and uses them to restored the values.

Context The *context* necessary to execute an XL operation contains dynamic information necessary to execute the statement graph. This includes for example the URI identifying the global and conversational variables, the XQuery execution context, and the prepared expressions.

Naturally, the physical algebra depends on the processing concept used by the VM. The term *processing concept* denotes the basic concepts and entities used by the VM to execute an operation. A VM selects a single entity, say, a statement, an expression, or a variable, uses it and continues afterwards. More general, resources within the VM are fetched and released at a certain granularity level. The processing concept defines the basic entities and their scheduling granularity.
Instead of machine code instructions an registers, the basic entities an XL VM deals with are XML data, XQuery expressions and XL statements. As for XL, we distinguish between three different processing concepts briefly described in following:

- **Statement Scheduling** Each statement of the statement graph is interpreted individually. The VM fetches the graph and schedules the initial statement. Statements are scheduled by appending them to a wait queue. In order to execute the statements the VM contains a set of threads, each thread fetches a statement from the waiting queue, executes it, evaluates the predicates associated to the outgoing combinators of the statement and schedules the corresponding subsequent statements. The very fine granular scheduling of individual statements offers the biggest flexibility: Concurrent execution of statements, as expressed by the logical algebra, is exploited easily. If two outgoing combinators of a statement are attributed with 'TRUE', both subsequent statements are added to the wait queue. If enough threads are available in the VM, the statements are fetched by different threads and executed concurrently. If several branches in the statement graph are executed concurrently, synchronization is necessary. If two branches of the statement graph are executed concurrently, the two execution paths have to be merged again.

  Scheduling decision on how or where to execute a statement or an expression are made on a per statement basis. This strategy is favourable if the effort necessary to execute a single statement is high compared to the synchronization costs due to the scheduling decision itself.

- **Operation Scheduling** The second execution strategy is to schedule complet operations. A statement graph representing an XL operation is interpreted by a single thread. In order to interpret the statement graph efficiently, it is transformed into an array of statements. The edges in the statement graph are represented by conditional or unconditional GOTO commands within the statement array. The execution of the statement array starts at the first statement. Two integer variables are used to navigate in the statement array: a program counter (pc) containing the index of the current statement and a next pointer containing the index of the next statement to be executed. The virtual machine assigns the new default value to the next pointer ($next = pc + 1$) and executes the current statement. Afterwards, the value of next is assigned to the program counter pc. The goto statements perform a jump within the statement array by overwriting the next pointer. The execution

of the statement array id finished when next pointer contains an index value bigger then the array length, typically arraylength + 1.

By serialising all statements into an array, the execution of a single statement and the navigation between statements can be implemented very efficiently. On the contrary, concurrency expressed in a statement graph is not exploited.

In most use cases, scheduling individual statements imposes a considerable overhead. Unless single statements are very expensive to execute the operation scheduling approach

Discussion The subworker interpreting the actual statement graph (called *XL_Execute* in XL) relies on how the statement graph is represented in the implementation. As for the current implementation, the XL program is expressed as an array of basic statements and conditional goto statement – the physical algebra of XL (see section 8.1). The array of statements is interpreted by performing a look-up in a Java case-switch statement depending on the statement type.
A different concept to execute an XL program could be to stream data between statements. Instead of executing each statement separately, statements are executed iteratively. A variable set by one statement is piped into the following statements while the first statement is executed. Pipelined processing of data is common for database systems using a pull concept based on the iterator pattern. The advantages of a pipelined processing concept are a reduced memory requirements, a faster response time, and data is read on demand when it is actually needed. The actual benefit of pipelined data processing depends on the workload and the type of storage being used. The pipelined processing concept is described in section 11.

In the final code generation step the statement graph is translated into an array of commands executed by the XL VM.

Workers & Resources Parsing and compiling a Web service is carried out by a sequence of workers:

- **Parser**: parses XL-Source code and generates an object model of the given program
- **Pre-Processor**: performs replace operations on the previously generated object model before and removes the syntactic sugar. Afterwards the object model representing an XL program is translated into a graph of basic statements.
- **Optimizer**: The optimizer uses standard compiler techniques ([ASU86]) to optimize XL programs. Depending on the optimization scope, the optimizer addresses either single operations within a Web service separately (like the pre-processed) or a whole Web service is optimized as a single entity.
- **Code generation**: code generation finally addresses each operation again separately.

Each operation in an XL program is represented by a named resource:

- **Statement graph**: The statement graph is uniquely identified by its name. This named resource contains all at compile time generated information related to a specific operation.

During the compilation process this includes for example the pre- and postconditions which are eliminated by the pre processor. After compiling this resource references the generated XL

program representation, which can be evaluated by the XL Virtual Machine. The statements graph resource bundles a set of other resources, as for example single statement objects or prepared XQuery expressions.

Discussion By using different workers for sequentially creating and altering the same resource statement graph, it is easy to add or replace components. In XL for instance two alternative parsers exist both generating the same type of object model representing an XL program.
In the current XL version, all workers are invoked synchronously, and executed sequentially by the same thread.

- After compiling an XL operation the resource representing this operation usually not changed anymore until it is disposed. Unless selfmodifying code is required.
- If overloading of operation names needs to be added, a more complex identifier for the named resource has to be generated.
- The op-resource contains a set of further components which could be treated a a separate resources:
  - the statements themselves which are generated either by the parser or the preprocessor
  - the XQuery expressions
  - variables

  Since during compiletime all these components cannot be addressed individually but could only to be used within in their context, it does not make sense to treat them a separate resources. During compile time an XQuery expression is described merely by the variables it uses.

Workers & Resources Several worker components are involved in the expressions handling:

- parsing and compiling expressions
- evaluating expressions

Either parsing, compiling or evaluating expression is actually implemented as a separate worker in XL, but integrate into either the XL compiler itself or the virtual machine executing XL operations.
A compile expression is represented by a resource:

- XDBC connection
- prepared expression

In XL, the prepared expression themselves are combined together with the corresponding XDBC connection into one big resource representing the XQuery part of the context for executing an XL operation.

Discussion Just like database connections, XDBC connections are time consuming to create and therefore a connection pooling strategy needs to be setup.
Whether a connection and the associated prepared expressions have to be treated as separate resources or not depends on the flexibility required in the XL engine. In the initial XL version no dynamic XQuery expressions existed. In order to minimize the effort during execution, a connection and all prepared expressions of one XL operation were combined and handled as one big resource.
Dynamically created expression cannot necessarily be reused and therefore a pooling strategy as the one above does not make sense. In this case XDBC connection and prepared expressions need to be treated as separate resources.

8.5 XML Storage

The storage interface used by XL provides methods to initialize, retrieve and update XML data. As for the Java database interface JDBC [Sun01] the storage beneath XL distinguishes between a so called *store* object, representing the connection to a specific storage and a *data source*, representing some specific XML data inside the storage.
The actual storage used could be either a main memory XML store, a relational database or possibly a native XML storage system. A data source represents an arbitrary XML content according to the XQuery data model [W3C04f] within the context of a given store. The data source maps the physical representation of the XML on to the token stream used by the XQuery engine [FHK+03] inside XL.
In order to provide a transactional functionality, the generic store interface is enhanced by a *transactional store*. The transactional store adds commit and rollback features as well as explicit value locking in shared or exclusive mode. The synchronization is achieved by using the transactional features of the storage below, as for example the relational database. The XL engine itself does not do any synchronization while accessing global variables. In XL the execution of either a single statement or a whole operation can be a database transaction. In the current implementation of XL, executing statement corresponds to a transaction.

Workers & Resources The XML store in XL is implemented as a resource in any case, while the type of resource depends on the actual scope of the data.

Persistent Storage Persistent XML storage is implemented as a named resource. The data represents a state which can be loaded and is identified by a URI.

Transient Storage Transient XML data in return is a stateless resource even if it is actually stored in a database. Transient XML storage is allocated, used and released, but it does not hold a certain state after an XL operation is finished.

8.6 Communication & Message Handling

The network layer inside the XL Web service engine wraps the used messaging protocol (e.g., HTTP [RF99], BEEP [Net03]) and provides means for sending and receiving messages. The purpose of the network layer is straightforward: the received content is parsed, translated into an internal representation and passed to the message handling layer. In a cluster environment, a

dispatcher forwards the message from the master node to one of the slave nodes without parsing it.
XL distinguishes between different messages types, as for example a synchronous operation call, an asynchronous operation call, or a debug message. Each of these different message types is handled by a specific worker.

Workers & Resources HTTP server, dispatcher and the different message handler each implement the worker pattern. The following resources are used:

SOAP messages Internal representation of a parsed SOAP message

Threads The Java thread are treated as a resource in XL.

Discussion The design decision one has to make inside the network layer is about when to invest CPU into a message. The network layer of a Web service engine is basically the entry point into the system implemented by a worker. By restricting the resources available inside the network layer the load processed by a system can be easily scaled. By for example restricting the number of working threads inside the network layer, the level of concurrency inside the processing engine can be controlled.

8.7 User Interaction

The term *user interaction* refers to the interaction between a software developer or an administrator and the Web service engine. The term *user* denotes in this section a software developer, an admin, or a different application using the engine, a user does not invoke a Web service but develops, deploys, or manages it. The primary use case you typically have in mind for a Web service engine is to start and stop a service. This minimal level of interaction could be enhanced in three different ways:

- update the configuration of a service
- update a service
- debug or monitor a service

The interaction between the administrator and an Apache Webserver ([Apa05a] in the default configuration) for example is fairly simple: to update the configuration of a webserver a central configuration file is changed and the server ”gracefully” restarted. Updates of services are done by by-passing the Webserver completely (e.g., replace a Perl-script on the file system).
For a complex Web service a more sophisticated strategy is necessary:

- online updates of either the configuration and the service itself have to be possible without causing data loss or down time – if possible.
- remote administration and monitoring have to be possible
- administration not only includes human interaction, but also the interaction between a different XL engines, for example a master server in an Web service cluster which starts and stops slave servers on demand.

Interaction can be addressed by applying the common Model-View-Controller pattern. Each of the different clients (a simple graphical user interface, a complex development environment, a makefile, or a different server) has a different view of the Web service engine.
The interaction between the engine and a user is implemented in XL using two elements: a central class providing the necessary access methods for a user interface (called *XL_RTS* in the following) and a set of configuration classes. The configuration class objects are read only for the Web service engine and contain all parameters configuring the engine at runtime. In order to interact with the XL engine the user updates the configuration classes and invokes the appropriate method of the XL_RTS object. The XL engine has to be notified of a change in the configuration, depending on the type of change it could be necessary to restart the engine. One configuration parameter for example is the type of HTTP-Server to e used: XL currently contains a very small XL specific HTTP server just providing the HTTP-Post method or an integrated HTTP server used by the Tomcat servlet engine [Apa05c]. If the type of HTTP server is changed in the configuration class, the XL engine has to be restarted.
The following list describes the different configuration classes and the parameters they contain. The labels (printed bold) denotes the name of the corresponding Java class:

XL_ClusterConfig If XL is deployed on cluster, a special cluster configuration is necessary. In the simple cluster scenario used by XL (a single master server and a set of slaves), each slave acts like a common XL server – master and slave do not exchange any meta information after start up. The XL_ClusterConfig class only contains parameters used by the master XL server:

- slaves(*set*): a set of names identifying the slave computers
- timeout(*int*): a timeout used for the master slave communication
- slave configuration (*file*): configuration file used by the slave servers
- user and password on the slave computers (*strings*)

XL_CompilerConfig The XL compiler contains a set of rules used to optimize the statement graph. In the compiler configuration each of these rules can be switched on or off (only boolean parameters).

XL_Environment The XL_Environment provides environment information:

- the build date, the Java compiler used to build the XL files, and the operating system XL was compiled on.
- the XL home directory and the XL logo displayed by a graphical user interface (if available)

The XL_Environment section is meant to provide information available at compile time. It cannot be changed dynamically at run time.

XL_LogConfig The XL_LogConfig class describes configures the log messages generated by the XL engine. The following parameters could be set:

- log level (*int*): The type of log-entry generated by the engine can be configured by a 32-bit integer value. Each of the different log types is switched on by setting the corresponding bit – which limits the number of different log levels to 32. The following

log types exist: not specified error, XML parser error, XL debug parser messages, XL scanner error, XL message handling error (e.g., SOAP), XL pre-processing errors, XL optimizer messages, statement execution, expression execution, statistical information (used for performance experiments), typing errors, XL operation start and stop messages, network errors.

- log file(*File*): if specified, all log messages are printed into a file.
- WIPS interval (*int*): WIPS is the abbreviation for *Web Interactions per Second* used by the TPC-W benchmark [Tra02]. In XL, the number of interactions per second is calculated by waiting until a certain number of interaction occurred. Then, the elapsed time is calculated and the WIPS figure is printed.

XL only distinguishes between different log entry types and not the severeness of the log message (e.g., info, warning, exception, error). Furthermore, log message are not formated according to certain criteria.

XL_RTSConfig The central configuration of the runtime system (RTS) is represented by the XL_RTSConfig class. This class contains a big set of parameters configuring the system as a whole:

- isServer (*boolean*): Depending on the parameter the XL engine starts a HTTP server. If this parameter is set to FALSE, the XL engine cannot messages via a network connection. Default is TRUE
- isWSDLServer (*boolean*): if set to TRUE the engine provides a WSDL [W3C01] description of the service.
- port (*int*): Network port number used to receive incoming messages. This parameter could be used to explicitly overwrite the port specified in the XL program.
- inputFile (*File*): the filename of the XL program itself.
- out (*Writer*): Java Writer class used to print the log messages. Default is standard out.
- displayGUI (*boolean*): Display a simple Java Swing user interface for managing the XL engine.
- min- max number of threads (*int*): Minimum and maximum number of threads used by the XL runtime system.
- soap (*string*): Internally XL distinguishes between different implementations of the SOAP message protocol. This parameter is used to change the implementation used.
- parser (*string*): Internally XL distinguishes between different XL parser implementations. The parser used by XL is determined by this parameter.
- server (*string*): Internally XL distinguishes between different HTTP server implementations. The server used by XL is determined by this parameter.
- queueRequest (*boolean*) and numberOfRequestWorkers (*int*): Depending on the HTTP server used by XL, different message handling strategies are possible. At the level of the HTTP server itself, this parameter distinguishes between a best effort and a more sophisticated queueing strategy.
- version (*float*): Versionnumber of the current XL runtime system.

- xldir (*string*): the current root directory used by XL.
- defaultMsgHandling (*string*): XL distinguishes between different message type (e.g., operation call, debug message). If a SOAP message does not contain type information the default message type, specified by this parameter is assumed.
- cluster (*boolean*): Is set to TRUE if the XL engine is part of a cluster (master or slave)
- slave (*boolean*): If set to TRUE if the XL engine is a slave inside a cluster

XL_ValueConfig The XL_ValueConfig class configures the XML storage used by the XL engine. XL_ValueConfig references different config object, each addressing a different XML storage type: main memory store, or the different JDBC storage's. The set of parameters depends the specific storage.

XL_VMConfig The XL_VMConfig class configures the XL virtual machine. The following parameters are included:

- conversationTimout (*boolean*): If this parameter is set to TRUE the XL virtual machine does consider a conversation timeout clause otherwise conversation timeouts are ignored.
- maxNumberOfConversations (*int*): Maximum number of concurrent conversations.

XL_CallConfig Other then by sending a message to the Web service engine, the user can invoke an operation on a deployed Web service by specifying the parameters either as command line parameters or the graphical user interface. Since the interaction between user interfaces and the Web service engine is limited on the central engine interface (XL_RTS) and the config classes, an additional config class is added. The XL_CallConfig class contains several parameters configuring a service invocation:

- name of the invoked operation
- input and output values
- conversation URI
- several boolean parameters:
  - acquire time (XL measures the runtime of the operation call)
  - printOutput: print the return value of the invoked operation.
  - wait: if set to false the XL engine terminates after invoking the specified operation.

In order to initialize the XL engine the configuration of all parameters is described by an XML config file:

```
<?xml version="1.0" encoding="UTF-8"?>
<xl-config>
  <section name="RTS">
      <parameter>
          <name>isServer</name>
          <value>true</value>
```

```
      </parameter>
      <parameter>
          ...
      </parameter>
       ...
   </section>
   <section name="cluster">
   ...
    </section>
    ...
</xl-config>
```

Like the previous description of the config parameters, the XML config file is grouped into several sections. Each section contains a set of parameters which carry a name and a value. Likewise a different, more intuitive, schema could have been used. The XL engine is configured by first parsing and evaluating the given init file, which is specified by a commandline parameter -init. The engine extracts all parameter values and initialises the corresponding config classes. After evaluating the config file all other commandline parameter are evaluated as well. By specifying a certain config value as a separate command line parameter the values specified in the config file are overwritten. Individual parameters can also by changed via the graphical user interface.
Conceptually, the interaction between an administrator and a service engine is not different then any other service invoked by a client. An administrator is simply a special type of client interacing with the engine. Technically, an administrator requires access to properties of the service engine not available for a common service. The configuration resources, as they are described by the previous listings, could be queried and update using the same expression language and invocation schema as common XL variables. The necessary action, like restarting the engine or halting a certain service could be implemented using the techniques already available within XL. If for example the the *isServer* attribute is updated the subsequent operation (e.g., the HTTP server is started or stopped) could be expressed by an XL clause: *on update $server-config* $\cdots$.
If the configuration is available as a common resource within the XL engine queries and updates can be managed almost like any other resource in the system. Certainly, providing the configuration as a resource does create a security problem. Authentication and authorisation have to be integrated into XL using standard encryption and right management techniques. The XL resource management could be enhanced in order to provide additional security. For example: locks (shared or exclusive) an certain resources could be set only if a lock on a corresponding authentication resource is acquired before.

8.8 Cluster

In a cluster, a service is not provided by single computer anymore but by a set of computers. The single computers in a cluster have to be organized and managed in order to distributed the workload efficiently.
The objective for building an XL cluster was not to implement another Web service cluster but to prove, or say to exploit, the scalability of the XL Web service engine. The question therefore is, whether the clustered execution of XL services could be achieved without changing the basic software architecture of the engine.

In order to implement a clustered version of the XL engine two straight forward approaches were taken. The simple cluster solution build for XL is a master - slave architecture containing a single master forwarding the individual messages on to a set of share-nothing slave servers. The second approach used to distribute XL services is similar to the first one, but uses a global XML store accessed by all slave servers via a remote JDBC connection.

Workers & Resources In the XL cluster architecture each slave computer is resource managed by the XL master server. The master server starts and stops the single slave installations and distributes the slave configuration. As described by section 8.7, the XL engine is configured by an XML config file containing different sections containing set of parameter name-value pairs.
In the cluster the central master server uses a simple intra cluster communication protocol to communicate with the slaves. The protocol used by XL is stateless and contains in the initial implementation only the very basic commands to start, stop and configure the slaves. The protocol is implemented like the common HTTP, the first line send contains a version number and the name of the command (e.g., start). The following lines contain parameters, separated by an empty line form the optimal payload of the message. In the XL cluster solution, communication only take place between master and slaves.
A more sophisticated XL cluster solution could be implemented if an appropriate intra cluster protocol is used which allows more complex communication patterns. In the Web service scenario everything is a service. Consequently, the intra cluster communication could be implemented using the same techniques used to provide the service itself. In this case each cluster computer provides a set of cluster-internal services managing the distributed execution.

Chapter 9

Future work (on the XL engine)

After implementing a basic XL engine, the further growth of the language and its engine have to be addressed. The language design must provide the possibility to enhance, say, grow the language according to the different requirements.

9.1 XL Plug-In

In order to implement a sophisticated XL cluster solution or, for example, a Java-XL integration, the XL language has to be enhanced by further statements.
In order to implement a cluster using the standard messaging methods available in XL, it would be necessary to access internal information of the engine using a common XL program (which is not possible right now). If arbitrary Java classes are to be integrated into XL, an efficient Java plug-in has to be developed. Either of the two enhancements could be easily achieved. Instead of adding different features a common integration strategy has to defined. In the following, several aspects of integrating a plug-in into XL are addressed. In order to illustrate the ideas, the cluster communication implemented by a special plug-in is used.

- **Syntax** The XL syntax should not be altered in order to integrate new functionality. In order to stick to the initial concept of sending messages the current syntax used to express sending synchronous and asynchronous messages should be used. For example, the URIs of conversations hosted on a certain slave inside a cluster could be determined by sending a message to the corresponding XL-slave engine:

  ```
  <request type="query">
    <item>conversations</item>
  </request> --> CLUSTER://slave-1.mycluster.com/ --> $slave1
  ```

  It is up to the XL engine to provide the corresponding protocol implementation on the client and on the server side if necessary.

- **Data Transformation** If the XML data modell of XL is left, which is usually the case if plug-ins are used, data transformations are necessary. The XML data provided in the statement has to be transformed into the value domain of the plug-in, e.g., into Java objects. The generated data does not have to be a Java object, but since XL is implemented in Java, a handle representing the data in the Java VM is necessary. Since XL is implemented in Java, the data transformations have be implemented in Java as well. The

questions is how to allow the XL programmer to specify a new plug-in, including the necessary data transformations.

- **Configuration** Additional plug-ins to be used within the XL language have to be included into the configuration of the XL engine. The configuration specifies the associated protocol to be used within the service invocation URI, the data transformation classes, and the Java class implementing the plug-in itself. For each of the required classes abstract interfaces could be easily defined, since the call of a plug-in corresponds to a usual send statement in XL.
  Additionally, it has to be specified whether the invocation of a certain plug-in involves sideeffects or not. If sideeffects occur, the associated undo operation has to be specified as well. Whether the undo operation is part of the configuration or the specific operation call itself, has to be discussed.

- **Execution** Since the plug-in integration into XL has to be generic, the invocation of a plug-in inside XL could be simply achieved by defining abstract interfaces. If arguments are passed to the plug-in back and forth, the data-transformation classes are invoked. The execution of the plug-in itself has to be implemented by invoking the specified plug-in class. If the specific plug-in requires additional context information, the XL engine can provide readonly context containing runtime information.

9.2 Security

The issue security has not been addressed in this thesis (and by XL) at all. The non-functional requirement security has several different aspects (data integrity, privacy, authorization, etc..), which cannot be discussed here.
For one reason, XL could be regarded to be on safe side: Java is considered a safe language [Lon05], as, for example, pointer manipulation is not possible and array and string boundaries are checked implicitly. On the other hand, many Web service have to be considered a security leak as messages send using HTTP bypass every firewall. Since firewalls are used to control the in- and outgoing traffic, bypassing it maybe easy but does not provide security.
Web services, exposing a public interface to a possibly hostil environment, require a detailed security concept, including for example domain, user, or network protocol specific security policies. A Web service engine, such as XL, should provide means to express these different policies.

Chapter 10

Performance Experiments & Results

The performance chapter of this thesis contains three parts:

- Section 10.1 describes the different environments used in the experiments.
- Section 10.2 explains the microbenchmarks of a Web service engine.
- In section 10.3 a more complex application scenario based on the TPC-W benchmark [Tra02] is used to evaluate the XL engine as a whole.

In order to prove the scalability of the XL engine, we deployed XL on three different platforms: a PDA, a standalone server, and a Web service cluster. XL is implemented in Java and certainly benefits from the portability Java provides. The XL source code and the underlying storage together contain about 700 Java classes. XL uses an XQuery engine described by Florescu et al. [FHK+03]. The disadvantage of using the Java programming language is the memory overhead caused by the Java virtual machine. The Java virtual machine allocates at least 20 to 30 MByte of main memory. Additionally the overhead imposed by the XML representation used

10.1 Experimental Environment

PDA On a mobile device only a limited functionality is available. This is due to the hardware restrictions on a mobile device or the limited availability of software components. On the PDA used (HP iPAQ h4150) only a Java SDK version 1.3 was available and thus only a limited XQuery functionality could be used (e.g., no XML schema validation). Since a PDA is usually sufficient to run in single user mode, the synchronization features and the sophisticated resource handling of XL could be stripped down. If value representation and XQuery context objects cannot be accessed concurrently and are being reused the resource management could be stripped down to create a single object which is created and reused.

Since a PDA typically does not require a persistent storage itself but relies on the storage of a remote serve, a main memory storage to be used with in the engine on the PDA is sufficient.

For our measurements we used an HP iPAQ h4150 with an Intel XScale PXA255 (400 MHz) processor running Windows CE 4.20 and 64 MByte RAM.

Standalone Server The standalone server is the default usage scenario of a Web service engine. This XL configuration provides the full functionality as described in section 4, including XQuery, concurrent conversation handling and a persistent storage. The Web service engine is configured for example by choosing the representations for different variable scopes (global, conversation, local) and by defining the different resource handling strategies.
For our measurements, global variables were backed by a local relational database (MySQL Version 4.1.5-gamma). The default resource handling strategy in XL is straightforward: resources are created on demand and returned to a resource pool after being released. Thereby, the resource handling strategy is configured by two parameters, these being the maximal number of resource and the maximal pool size of a certain resource type.

As standalone server, a Pentium 4 CPU (2.80 GHz, 1 GByte RAM) running Linux, kernel 2.4.20, was used.

Cluster The third and possibly most complex scenario is a Web service provided by a computer grid or a cluster. In this case the state of the Web service could be either represented by a single central database or by a distributed database. As for the standalone Web service the variable scope is a means to partition the state of a service. Conversational data, representing the state of a single conversation, could be stored in local databases while global data shared among nodes in a cluster is backed by a central database.

The XL cluster setup contains a central master and a set of slaves. The master receives the incoming messages and forwards them to one of the slaves using either a round robin or a *sticky conversation* (messages in a conversation are always forwarded to the same slave) strategy. The XL master server forwards HTTP messages without parsing the XML content. The persistent variables are stored in a remote relational database accessed by the slaves.

In the XL cluster for master and slave nodes the same type of computer is used as for the standalone server.

10.2 Microbenchmarks

This section depicts a set of basic runtime figures measured inside the XL engine which are necessary to recognize bottlenecks and to customize a system for a certain use case.

10.2.1 SOAP Messages

An important part within a Web Service engine is the generation of SOAP messages. Each time an external Web Service is called a new SOAP message has to be created by serializing the internal XML representation into a character String. In return for the de-serialization of a SOAP message it is necessary to parse the character string and to translation into an internal message representation.

Platform	Parsing SOAP Message (ms)			Generating SOAP Message (ms)		
Message Size (Byte)	500	5000	10000	500	5000	10000
XL (PDA)	131.49	928.33	1355.65	145.48	346.11	571.03
XL (server or cluster)	0.83	15.59	31.90	0.61	3.47	6.65
Apache Axis	5.05	19.25	35.10	2.28	5.47	8.35

Table 10.1: *Time(ms) to parse and to write SOAP messages of different sizes using either XL or Apache Axis 1.2alpha.*

Since the XL engine uses XQuery not only for expression inside the XL programs but also for internal XML handling, SOAP messages are generated by a set of XQuery expressions. To parse XML, a BEA XML parser is used and an incoming message is evaluated again by a set of XQuery expressions to extract the SOAP elements.
Table 10.1 shows a comparison of XL on a PDA and a server versus the Apache Axis SOAP (1.2alpha, [Apa06]) package. The Axis package is not part of XL but provides reference figures XL can be compared to. The runtime figures are the measured time in milliseconds it takes to generate a complete SOAP message in each of the environments. These figures do not include network latencies or any service invocation time but only the time required to generate the SOAP message represented by a character string ready for sending via a network socket.
In Table 10.1 the size of the SOAP message is sized from 500 to 10000 Bytes. A 500 Byte SOAP message could be considered fairly small carrying just a few Bytes of actual payload while 10000 Byte represent a big message in our test cases. The payload of the messages itself consists of small XML elements, each of them containing approximately 1 to 20 characters text. Especially for small messages the Apache Axis package has a considerable overhead due to the Java reflection features used. The runtime required by the Axis package for either parsing or generating messages were dependent on the complexity of the message. In our example complex SOAP messages were generated using several dozens of Java Beans. Simple SOAP messages turned out to be faster using Axis. For the PDA the runtime figures are of course slower, but still the 0.346 seconds to generate a 5000 Byte SOAP message are acceptable on a mobile device.

10.2.2 Operation Call

Invoking an operation of a service is probably the most frequent element of Web service programming. The overhead related to a service invocation differs considerably, especially in the Web service domain. Depending on the use case, a service invocation could be an inlined function call in C or Java or a heavy weight SOAP call, possibly including a lookup at an UDDI server and high network latencies.
For XL this section provides figures on how big the operation call overhead is and how it is pieced together. The generation and parsing of SOAP messages shown in the previous subsection is one part of it. As described in section 8.4 executing an XL operation requires a statement graph, an XQuery context and a set of store resources representing the connection to the underlying storage. In the current implementation, the XQuery context and store resources are combined into a single resource being managed by XL.
Since all resources are pooled, the overhead of creating an XL execution context, say fetching

Operation	SOAP message size (Byte)	Server internal (ms)	Standalone Server (ms)	Cluster (ms)	PDA internal (ms)
empty operation	500	0.05	1.59	1.934	6.70
operation x { **body** { **let** $output := $input; } }	500	1.07	2.07	4.40	6.65
	5000	2.08	18.77	22.29	51.02
	10000	3.53	39.05	43.19	99.88
operation x { **body** { **let** $output := for $b in $input//b order by integer($b) return $b; } }	500	1.11	2.08	4.68	Not possible on the PDA
	5000	25.18	40.92	43.05	
	10000	56.77	86.76	88.56	

Table 10.2: *Time(ms) to execute operations on a XL engine internally, on a standalone server an a cluster (including the SOAP messaging overhead).*

the required resources, is very low. On a standalone server executing an empty operation internally takes about 0.05 milliseconds, if resources in the pool are available.
Table 10.2 shows the execution times for two stateless service operations. The measures are performed on the XL server itself excluding any networking overhead. The column entitled "Server internal" contains the XL internal execution time of the operation without message handling. The standalone server column presents the execution time of the operation on the server measured at the level of the HTTP server. The cluster column likewise shows the execution time on a simple XL cluster. In the cluster scenario, a single master XL server forwards all HTTP messages to a single slave server. The PDA column contains the internal execution time of the operations on the PDA itself.
The differences between the columns illustrate the overhead due to message handling compared to the actual execution time of a statement. Especially for small operations, SOAP messages impose a considerable overhead.
By comparing the two operations in table 10.2 the required time to execute a more complex XQuery expression is illustrated. In the the second example selected here, a sequence of XML elements is extracted form the `$input` variable and ordered according to certain predicates. The bigger the `$input` variable, the more elements have to be ordered.

In order to demonstrate the scale-up which can be achieved using an XL cluster, we executed the complex ordering expression shown in table 10.2 on a standalone server and a small cluster. The achieved scale up is expressed by the number of operations executed per second (OpS) for different workloads. In table 10.3, the number of clients determine the load being put on a server. By comparing the columns in table 10.3, we can illustrate the scale up achieved by scaling the cluster. What is remarkable is the overhead due to the forwarding of messages as well as the scale-up for a higher load on a bigger cluster.

Number of Clients	Standalone Server (Ops)	Cluster (Ops)		
		1 slave	2 slaves	3 slaves
5	80.60	67.38	104.47	121.40
10	71.92	58.13	107.15	141.72
50	72.04	56.63	115.01	167.98

Table 10.3: *Operation calls per second (OpS) for a stateless service on a standalone server and a cluster (SOAP message size: 5000 Byte).*

Operation	Size of `$global` (Byte)	Store representation used for `$global`		
		Main Memory (ms)	local MySQL (ms)	remote MySQL (ms)
operation x { **body** { **update** $global insert < XXX/ > into $global; } }	500	0.10	39.12	47.21
	5000	0.09	44.05	63.94
	10000	0.11	51.88	57.02
operation x { **body** { **let** $x := $global; } }	500	0.16	3.62	10.47
	5000	1.30	14.15	26.93
	10000	4.10	26.29	45.82
operation x { **body** { **update** $global delete $global; } }	500	0.08	27.55	29.04
	5000	0.08	30.28	39.80
	10000	0.10	36.48	37.34

Table 10.4: *Time(ms) to execute XML update statements using different stores*

10.2.3 Storage

In XL a generic XML storage interface is used which can be implemented using different XML representations. As for the current XL implementation either a main memory XML store or a persistent store based on a relational database (MySQL, Version 4.1.5-gamma) could be used. Table 10.4 shows the execution times measured for different update statements performed on the two store implementations. These figures form the baseline for a set of possible improvements as for example the use of schema information, XML indexing, or caching.
The runtime figures shown in table 10.4 represent an small insert into a global variable, the usage of a global variable, and deleting the content of a global variable. In each case the size of the global variable is varied from 500 to 10000 Byte. These figures show the overhead due to the persistent database. It has to be mentioned, that the XQuery engine used here is not aware of the database beneath it.
There are several questions related to the store which are still a research issue and cannot be an-

swered here. One example would be how to efficiently support XQuery expressions by indices and how to integrate query processing and database efficiently.
An important aspect of every database application is the isolation level, or the locking of data inside the database respectively. The level of concurrency is determined by the mapping of the XML data on to the relational schema. In the current implementation of the store each time a variable is accessed (read or write) the whole variable is locked in either shared or exclusive mode.

10.3 Complex Benchmark (TPC-W)

In order to demonstrate XL in a more complex scenario a shop application in XL has been setup. Since, to our knowledge, no SOAP-based Web service benchmark exists we used the TPC-W benchmark [Tra02] as a scenario reference. The objective of this section is for one to identify bottlenecks in a more complex scenario and to prove the applicability of the XL concept.
The XL shop sells *items*. Items in our case are represented by a complex XML element containing 21 subelements including id, name, description, price, discount, stock and packaging information. Customers can register, query the item database and add items to a shopping cart. A purchase is completed by going to the counter, which generates a purchase order to be stored persistently.
Below, the operations are characterized in more detail:

- `registerCustomer` A new customer XML element is created and inserted into the customer database. The customer XML element includes a username, password, real name, address, account balance and the date of the last visit.

- `queryItems` The item database is queried. The customer can specify various predicates, like price, item id, or keywords to be contained in the name or description.

- `addToCart` By calling this operation the user can either start a new shopping session and add an item or add another item to an existing shopping cart. A valid username and password have to be provided and the requested item has to be available. In this is the case, a new lineitem is added to the variable representing the shopping cart. Additionally, the customer database needs to be updated (last visit) and available stock in the item database is adjusted automatically.

- `goToCounter` This operation completes a shopping session by creating an order XML element which is inserted into a persistent variable. The order XML element includes the customer name and address, the ship type, and for example an order date. Finally the account balance of the customer has to be adjusted.

- `pay` After purchasing several items at the XL store a customer is requested to pay. In the XL shop this simply includes an update of the corresponding account balance XML element in the customer database.

The XL shop example includes a readonly operation `queryItems` which is called very frequently and some costly operations including several updates like `addToCart` and `goToCounter`. The load generator we use contains a set of threads, each representing one customer. Each customer registers, queries the items, adds the returned items to his shopping cart and moves to

Number of Clients	Standalone Server without database	Standalone Server using local MySQL	Cluster using remote MySQL
1	22.96	5.60	3.82
10	51.57	6.40	5.90
50	49.35	7.01	6.34

Table 10.5: *Operation calls per Second (OpS) on a standalone server and a cluster for different workloads.*

the counter. After several shopping sessions the customer pays the bill by calling the operation `pay`. In order not to make the example too complex our customers actually buy every item returned by `queryItems` and between two operation calls each customer threads waits 10 ms. The load generator is implemented in Java using the Apache Axis package [Apa06]. The performance of the XL engine is expressed by the number of operation calls per second (OpS) measured on the server.
Performance is dominated by the synchronization inside the storage while accessing global variables. If the state of a Web service is represented by XML data, locking a whole variable, as it is done in the current XL storage implementation, is too expensive. More advanced XML locking techniques (e.g., [HKM04]) need to be applied.

Chapter 11

Pipelined Processing

In many cases the processing of a Web service incorporates database accesses or other services being called. Like database cursors, each of them typically provides a sequence of results to be processed. This part of this thesis introduces a streamed processing concept which iteratively consumes different data sources and is capable of executing a whole service via streaming.

11.1 Motivation

A Web service language describes the interface and the processing logic of the service. In the following *streaming* denotes a different processing concept. Instead of materializing variable content locally, the generated results are passed on to subsequent statements which depend on the previously set variable. The XL code example below illustrates a typical use case. The lineitems included in an XML purchase order are extracted from the variable `$input` and assigned to a local variable `$items`. The succeeding statement inserts certain lineitems into a global variable named `$electro-items`.

```
let $items := $input/Order/Lineitem;
insert $items[Type eq 'electronics']
     into $electro-items
```

Instead of materializing the whole sequence of lineitems, in our concept each lineitem is generated on demand and passed to the insert statement iteratively. Instead of loading a potentially gigabyte sized sequence of lineitems into memory and processing it in the next step, the lineitems are processed one at a time sequentially.
The streamed query processing model based on the iterator pattern is very common in database systems and has been described for example by Graefe 1993 [Gra93]. But iterators as they are used in databases do not fit to our situation and the model has to be adapted as described, later on, in section 11.5.2. Still our iterator model preserves the advantages of the database iterator model:

- **Memory requirements**. Depending on the expression results are not materialized but streamed through the program. Therefore only a small fraction of a potentially big variable has to be kept in main memory.
- **Response time**. Even if the execution time of an expression is not necessarily reduced, partial results can be accessed already before the execution is finished.

- **Dynamic Adaptivity**. Since the evaluation of iterators in databases is demand-driven, only those results are generated which are actually needed by subsequent iterators. The same kind of demand-driven result generation can be used for statements as well.
- **Parallelism**. By adding internal send- and receive-Iterators a parallel execution as in distributed databases is feasible.

11.2 Definition and Use cases

Streamed data processing is a common technique used under different circumstances. In order to structure the very diverse applications of streamed (or pipelined) data processing a brief definition of the concept and a list of use cases is given in the following:
In order to apply a streamed data processing concept data, processing steps, or typically both are divided into smaller chunks which can be processed individually. Examples of streamed data processing are:

- Common processors subdivide complex statements into a sequence of smaller pipeline steps executed in a physical processing pipeline [Men98].
- Signal processing of continuesly collected values (e.g., temperatures or stock prices)
- Database query processing (as mentioned previously)
- Audio and Video processing

Since these examples are picked from different processing domains, concepts and implementations cannot be compared easily. In each use case data is split into smaller elements (or "*tokens*") and flows from a data source to a drain. Typical characteristics of streamed data processing are:

- Compare to the total size of the processed data, the individual token is small.
- Each token represents a piece of information which can be processed individually. It must be possible to process single tokens, or subsequences of tokens without knowing the whole sequence. Obviously, individual processing of tokens is not always possible, ordering a sequence of elements requires to read each element at least once.
- The data processing itself is achieved by letting the data flow through the system. Data is not processed at the drain but by the combination of all processing steps.

11.2.1 Push versus Pull

An important aspect of streamed data processing is the distinction between "*push*" and "*pull*". The verbs push or pull referr to where the processing is initiated. Either the data source pushes the data into the system or the drain, say, the data consumer pulls the data by iteratively requesting the next token. Both concepts are equally expressive (not to be proved here), depending on use case the appropriate streaming method has to be chosen. For example, in the signal processing domain the processing of the data is driven the frequency of new input, therefore a push concept is used. For database query processing the iterator based pull concept proved to be right [Gra93], since the execution is driven by application using the database.
In order to execute a whole sequence of statements, several problems have to be addressed:

- If- and While-clauses and the sequencing of statements have to be expressed
- Side effect have to be considered.
- Optimization considering for example variable scope or variable size has to applicable.

In the context of this thesis, the XL language provides the execution environment the streamed data processing is applied to. The following section provides again a brief abstract of the logical algebra the streamed processing approach is based on.

11.3 Logical Algebra

At compile time each operation is represented by a graph structure defined in the following subsections. In this section only a simplified XL algebra is described necessary to illustrate the use of pipelined processing in XL. A more detailed description of the XL algebra is provided in section 8.1.

11.3.1 Statements

The logical algebra of XL considered in this section contains only a small set of basic statements:

- **Assignment** This statement evaluates an XQuery expression and associates the result with a given variable name. The generated value is stored in a context (see subsection 11.3.3).
- **Send- and Receive-Statement** The send- and receive statements of the logical algebra express the XL service invocation. The send statement evaluates an input expression and sends the result to a given destination. A destination is specified by a URI and an optional operation name to be called. If a reply message is expected, a message identifier is included into the message and stored in a local variable.

  The receive statement waits for return message to arrive. If the program waits for a reply message related to a previously send message, the local variable containing the message identifier has to be checked. The result of the message received could be stored in a variable.
- **Begin- and End-Block** Like in Java, the scope of variables is delimited by curly brackets ($\{\cdots\}$). In the logical algebra, begin and end of a statement block is indicated by these two statements.

Additionally to this most basic statement set, some XML Update statements were included in the logical algebra as well. These statements do not add further expressiveness, since all update could be expressed by assignments as well. Though they provide a more convenient way of updating variables:

- **Insert Statement** The insert statement contains two XQuery expressions. The first expression generates the new content to be inserted, while the second expression specifies the location where to insert.

```
let $order := $input/PurchaseOrder;
if (exits($order//Email)) {
 insert $order//Email
      into $addressbook
};
let $output := count($addressbook)
```

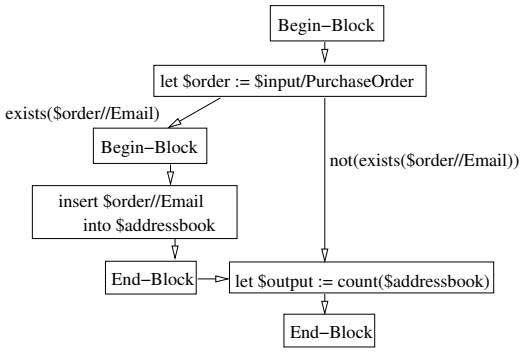


Figure 11.1: *XL listing and the generated statement graph*

- **Delete Statement** The delete statement contains an XQuery expression which specifies the XML element to be deleted.

- **Replace Statement** The replace statement contains two XQuery expression, one specifies the content to be replaced and one the new content.

Additionally further update statements could be added (e.g., move, rename or replace).

11.3.2 Statement Graph

Each XL program is represented internally by a directed and attributed graph. The nodes of this graph are the statements of the XL logical algebra as defined by the previous subsection. This algebra represents the basic command set, which can be executed by an XL virtual machine.
An edges e in the graph connecting two statement s_1 and s_2 expresses a previous-next relationship between these two statements. The graph structure used by XL matches the block graph used by Aho, Sethi and Ullman [ASU86] data flow analysis. Additionally, each edge is attributed by a boolean expression denoted by e_{cond}. An edge e between two statements s_1 and s_2 has the following semantic: if statement s_1 is executed and the boolean expression e_{cond} of the outgoing edge e evaluates to true, then statement s_2 has to be executed as well.
Figure 11.1 gives an example illustrating simple XL program and the corresponding statement graph.

11.3.3 Execution Context

The execution context for the logical algebra can only be specified at a very abstract level. Each implementation of the logical algebra has to provide a context for an operation respectively a statement to be executed. The statement execution context has to provide an environment for the execution of XQuery statements. This includes for example namespace and function definitions and a set of variables which can be accessed by the expression itself. The context for the statement execution in return has to provide means for storing values in main memory or some other storage facility and associating these values to variable names. Additionally the statement context has to provide some sort of lock handling. To avoid conflicts during parallel access the context has to provide a lock handling which enables the virtual machine to set read- and write locks depending on the variables accessed.

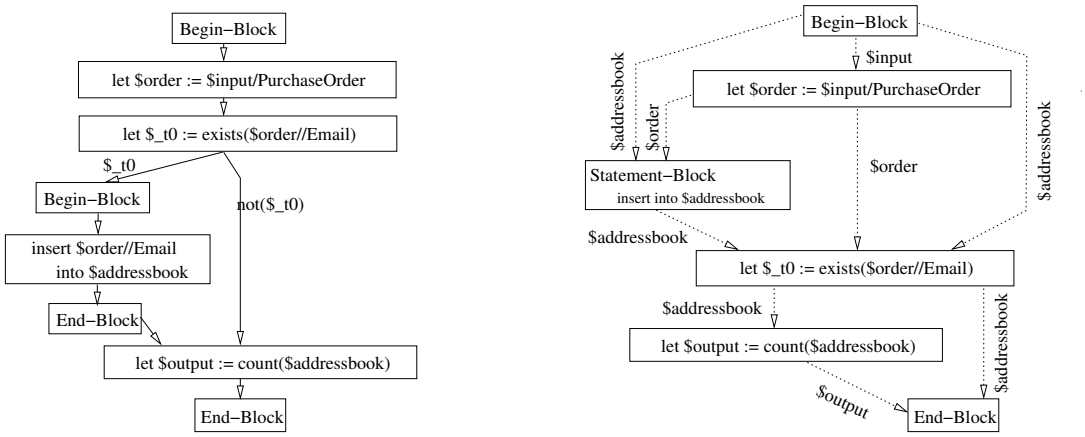


Figure 11.2: *The generated statement graph and the corresponding data flow graph*

11.3.4 XQuery Expressions

The XL runtime system uses the BEA/XQRL Query Processor published at the VLDB 2003 by Florescu et al.[FHK+03], for any kind of expression evaluation. This query processor provides a JDBC like interface of Prepared-Statements called XDBC. At compile time an XDBC Prepared-Statement is created for each expression. The query processor generates an evaluation plan for each query. If the expression is evaluated, this evaluation plan is executed by a set of iterators based on the common iterator model as in most SQL systems [Gra93].
As described by Florescu et al.[FHK+03], the XML values are represented by a sequence of tokens returned by evaluating the query iterator. These tokens represent an XML value by materializing the events generated by an XML Parser.
The XML element `<author> Andreas </author>` for example is represented by the following sequence of tokens:

```
[BEGIN-ELEMENT [author],
      [anyType@http://www.w3.org/2001/XMLSchema]
]
  [TEXT ' Andreas ' ]
[END ELEMENT]
```

11.4 Normalization

The normalization translates the statement graph described in section 11.3.2 at compile time into a new graph expressing dependencies between different statements. The objective of normalization is to identify time or data parallelism. Time parallelism refers to independent statements being executed at the same time concurrently. Data parallelism refers to the parallel generation and use of values by different statements, which is what we are focusing on in this section.
In the motivating example in section 11.1 the first statement is still generating results while the second statement is consuming them. Compared to the initial graph described in section 11.3.2 the edges of the normalized graph are not labeled by expressions but variable names.
The basis for the streamed execution model proposed in this chapter is the data flow analy-

sis described by Aho, Sethi and Ullman [ASU86]. For each operation a definition-use chain (du-chain) is setup. This du-chain specifies for each variable:

- the statement which defines a variable instance
- those statements which use a specific variable instance

This data flow information is used to create a normalized statement graph.

11.4.1 Pipelining

Normalization is optional, and it has to be considered independently from the pipelined processing of statements. If a statement graph is not normalized, pipelined statement processing could still be applied. Either each individual is implemented as a separate pipeline, say, only statement-internal but no inter-statement data parallelism takes place. Alternatively, data could flow through a statement graph providing data parallelism between different statements (inter-statement). Each statement is represented by an iterator which implements the corresponding processing logic. Due to a previous compilation step, each iterator knows its input variables, say, each iterator knows which instance of the input variables has to be used. Since the granularity of the streamed data, in the XL case the individual tokens of an XML stream, is very fine grained, processing overhead connected to the singel token has to be small. Therefore, no additional thread synchronization, or additional buffering should be used for the inter statement communication. Single tokens should be directly passed from one iterator to the next one. In order to achieve such an efficient communication a normalization is essential.

11.4.2 Normalization in XL

By starting at the inner most statement block, statements changing a variable and subsequent statements using it are connected by attributed edges. Each data flow edge is attributed by the name of the variable which is passed along this edge. The non local variables inside a statement block are considered being dependent on the initial Begin-Block statement as they have to be initialized by statements prior to the block.
The side effects of executing a statement block are materialized by changes made to the non local variables and by external dependencies not expressed by variables as for example asynchronously send messages. The normalization process determines for each statement block two different statement sets:

- For each updated non local variable, the statement determining its final value after the block is executed.
- Statements which incorporate external dependencies.

After determining these dependencies for an all inner most statement blocks, in the next step the normalization process addresses the statement block representing the next bigger scope. Each inner block hereby is simply represented by an artificial statement, which exports the side effects of the inner block by naming the variables changed inside.
Figure 11.2 shows on the left the statement graph of the previous example (figure 11.1). The complex expression attached to the edges in figure 11.1 expressing the if-clause are replaced by an assignment to a local variable named `$_t0`. The outgoing edges are therefore only labeled

by the variable name.
The right part of figure 11.2 shows the corresponding normalized graph. The data flow edges are indicated by dotted edges. The statement block inside the if-clause is replaced by an artificial statement, which exports the variables changed inside (`$addressbook` in this case).
The If-clause is expressed by the assignment evaluating the if-condition
`let $_t0 := exists($order//Email)`. This statement requires one incoming edge for the evaluated variable `$order` and two edges labeled `$addressbook` representing the two different variable instances depending on the predicate.
In the example the global variable `$addressbook` and the variable `$output` containing the return value are updated. Hence the side effects of the program are expressed by the assignment `let $_t0 := exists($order//Email)` which determines the variable instance `$addressbook` to be used and the final assignment setting `$output`. The final End-Block statement therefore has incoming edges for these two variables expressing this dependency.
Inside each statement block the sequencing of statements is replaced by dependencies expressed by variables.

11.5 Physical Algebra

In this section the streamed and the traditional concepts for executing XL programs are compared.

11.5.1 Traditional Approach

The traditional implementation of the logical algebra applies a straightforward approach. The execution of XL is very much alike the Java virtual machine (Java VM). The specification of the Java VM [LY99] describes the execution inside the Java VM only by a few lines of pseudocode:

```
do {
  fetch an opcode;
  if (operands) fetch operands;
    execute the action for the opcode;
} while (there is more to do);
```

Except for the operands loaded, the XL strategy is the same. Since XL does not use operands loaded into registers but XML token sequences, the token sequence representing a variable instance used by an expression is loaded on demand as the expression is evaluated.
In order to execute an XL operation a context representation and the initial statement of the statement graph are bound together and added to a scheduling queue of an XL Thread. This thread fetches the initial statement from the queue and executes the Java code associated with the specific statement-type. After executing a statement, all following statements are added to the scheduling queue if the expressions associated with the outgoing edges in the statement graph evaluate to true. The XL Thread continues until the scheduling queue is empty.

11.5.2 Streamed Approach

The key idea of streamed statement execution is streaming values through a program instead of of materializing values after every single statement. In database systems the streamed query

processing is based on the well known iterator pattern [Gra93]. The evaluation of such an iterator is split into three parts:

- *open*: allocate resources and prepare for evaluation.
- *next*: return the next result or *null* if the evaluation is finished.
- *close*: release allocated resources.

SQL or XQuery expressions are commonly represented by nested iterators implementing the logical algebra of the different operations. The modularity of the iterator patterns reduces the complexity of the represented expressions. As for example a join iterator simply requires two input iterators generating the sequences to be joined. In order to achieve this modularity a set of rules has to be set up, which defined how the methods of the iterator pattern are to be implemented and which assumptions can be made based on these iterators. One common assumption made in database systems is, if an iterator is evaluated repeatedly, it returns the same sequence of results. If we now consider representing an assignment like `let $i := $i +1` by iterators, repeated evaluations must not return the same result. Therefore the common iterator pattern has to be adapted.

Iterator Model

As the simple assignment suggests, a rule has to be setup which defines when side effects may occur or not. Since variables are usually initialized and then used by different expressions, several subsequent iterators have to access the same value. Because the initializing statement cannot be evaluated a second time to generate the same value again, registering and de-registering demand for a specific value by subsequent iterator and its actual usage have to be separated.
These considerations lead to a set of rules for the implementation of iterators to be used in our model:

- Calling the methods *open* and *close* without a next in between must not have any side effects.
- Calling the method *next* may have side effects.
- The *open* method of all potentially required input iterators has to be called in the *open* method.
- All opened input iterators have to be closed.

By repeatedly calling the *next* method, each iterator generates a sequence of results which is passed to one subsequent consuming iterator. The execution is therefore demand drive - results are only generated if the subsequent iterator evaluates the corresponding input iterator.

Iterator Types

For expressing any type of XL programs by iterators according to our model, several different iterator types have to be defined:

- **Assignment Iterator** The assignment-iterator evaluates an expression and passes the generated results to a subsequent iterator. The Iterator requires one input-iterator for each variable the expression depends on.

- **Split-Buffer Iterator** If a variable is evaluated by several expressions, a local buffer-queue is used to store the generated value. For each subsequent expression, a buffer-iterator is created which reads the buffer content. The buffer itself is filled by a single input iterator. The evaluation of this input iterator is driven by the fastest advancing buffer iterator. If a buffer iterator tries to fetch the next result from the end of the buffer queue, the *next* method of the input iterator is called. The input iterator is opened as soon as the first buffer iterator is opened and the input iterator is closed if all opened buffer iterators are closed. All buffer iterators which are opened simultaneously return the same result sequence. Since each iterator has open all potentially required input iterators within its open method, the buffer knows whether buffered values can be released or not. Since each opened iterator has to be closed, each buffered variable is released.
  To improve readability the split-iterators are omitted in all given figures.

- **Write- and Read-Iterator** An assignment requires a variable being materialized in the context. Materializing a result sequence in the context is achieved by a write-iterator. If the *next* method is called, the write-iterator evaluates a single input iterator and stores the result sequence in the local context. The *close* method has to ensure that the input iterator has to be evaluated completely if *next* has been called.

  The read-iterator either reads a given variable from the local context or it evaluates an optional input iterator in case the variable does not exists in the context yet. If a variable does not exists and the input iterator is missing an error is raised.

  In order to avoid read-write conflicts a read-iterator returns the value of the variable at the time the *open* method is called. Likewise the *close* method of the write-iterator associates the given token sequence with a variable name in the context.

- **Send- and Receive Iterator** The send-iterator evaluates an input iterator providing the value to be sent and a second iterator specifying the destination of the message. The two iterators are evaluated completely and a message identifier is attached to the message sent. The generated message identifier is returned to the calling iterator by the *next* method.
  The receive-iterator in return uses the message identifier provided by a previous send iterator and waits for this message to arrive. The received message is returned to a subsequent iterator.

- **Update Iterators** Since updates could be expressed by an assignment as well, the associated iterators are very much alike. Each update iterator evaluates an input iterator providing the result sequence to be updated. Furthermore the position of the update has to be specified by an expression. The iterator inserts or deletes content at the specified location and returns the modified result sequence to the subsequent iterator.
  An Update iterator reads the whole updated variable from the corresponding input iterator, changes the specified part and passes the whole variable on to a subsequent iterator.

Additionally a set of iterators is needed which express the structure of an XL program and not the dependencies given by variables:

- **Materialize Iterator** The Materialize iterator evaluates several input iterators completely. When the *next* method is called, it evaluates all input iterators at once and returns null.

- **If-Iterator** The if-iterator requires three input iterators. The first input iterator represents the predicate of the if clause. Depending on the result returned by this predicate iterator the

result sequence provided by one of the two further input iterators is returned to the subsequent iterator.

- **While-Iterator** In the statement graph described in section 11.3, while-loops are described by cycles. Since cycles in an iterator graph would lead to a possibly infinite recursion, an acyclic graph of iterators expressing the loop has to be generated. Mapping a loop on to the iterator model requires a repeated evaluation of all iterators representing the body of the loop until a given predicate evaluates to false. Therefore the While iterator requires two input iterators, one for evaluating the predicate and one representing the loop body. The iterator evaluates these two iterators alternately until the predicate iterator returns false. The execution of the loop-body is finished by closing the corresponding iterator and opening it again for the next iteration of the loop.

  The while-iterator does not return any variable content to the subsequent iterator, but a boolean result indicating whether the loop body was executed at least once.

Since in our iterator modell each iterator can only return a sequence representing a single variable, an if clause might require several if-iterators, one for each variable changed inside the if-clause. The same kind of argument applies to the while-loop as well. The loop-iterator executes the loop as described above. For each variable changed by the loop, an additional if-iterator is added. If the loop-body has been executed, the changed variable is materialized in the context and can be read by an read-iterator. If the loop-body has not been executed, the variable has to be set by a statement prior to the loop. The boolean return value of the while-iterator is used to make this distinction.

Graph Generation

As described in section 11.4, for each block of statements a set of included statements is determined which represent the side effects of executing the block. If an iterator graph for an XL program is generated, the following distinction has to be made:

- **Non Local Variable**: For each non local variable changed inside a statement block, an iterator representing the final statement changing the variable has to be generated. Likewise for each variable required to evaluate this iterator, a new iterator is generated recursively. The previous statement setting this variable has been determined by the data flow analysis described in section 11.4. Still the iterator only has to be evaluated, if the variable is required by an expression in the bigger scope given by the surrounding statement block.

- **External Dependency**: An asynchronously send message could contains a side effects which cannot be addressed by the data flow analysis. But since these statement are not dead code, they cannot be eliminated. Iterators representing these statements have to be generated and evaluated.

For each statement of an XL program an associated iterator structure is created. A statement block is executed by evaluating all iterators representing changes of non local variables and iterators representing the external dependencies of this block.
The XL operation is executed by evaluating all iterators representing changes of non local variables and iterators representing the external dependencies for the outer most statement block of the operation.

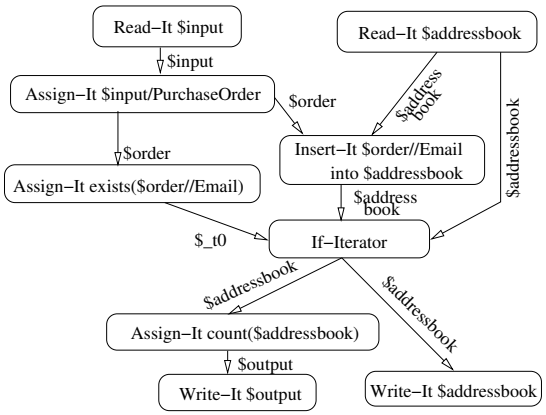


Figure 11.3: *Iterator graph for the XL program shown in figure 11.1*

```
{
 let $order := $input/PurchaseOrder;
 if (exits($order//Email)) {
   insert $order//Email
          into $addressbook;
   <order>OK</order>
       ==> "mailto:"{ $order//Email };
 }
 let $output := count($addressbook);
}
```

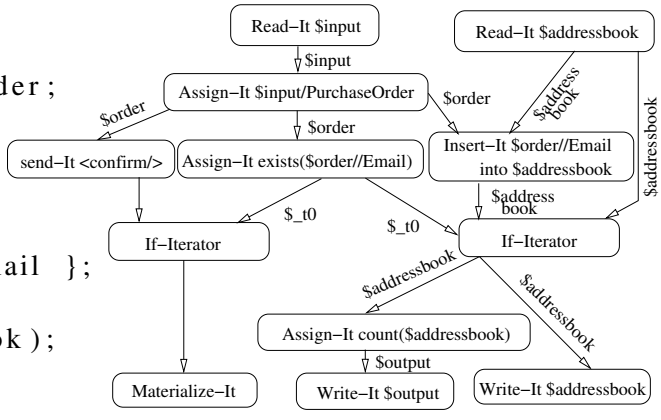


Figure 11.4: *XL if-clause containing two statements and the generated statement graph*

The key assumption made here is, that dependencies between statements are always expressed by the use of variables. The explicit sequencing of independent statements is not possible.
The execution strategy applied here could be called inverted compared to the traditional approach. The execution of an operation is initialized by opening the iterators representing those statements setting the final state of the context. In many cases these are the statements at the end of an operation. The execution of an operation therefore is initialized at the end and by recursively calling the *open* method the initialization is propagated up the iterator graph to the initial read-iterators. The generated values in return stream top down through the graph as the iterators are executed.
The following bullet points illustrate the generated graphs by discussing some examples:

- **Global Variables** The graph representing the previous example including an if-clause is straight forward as shown in figure 11.3.

  The two initial read-iterators read the global variables `$input` and `$addressbook`. The final write-iterators in return materialize the updates of `$addressbook` and `$output` in the context.

- **If-clause** The XL assignment is expressed by the so called assignment-iterator and a subsequent write-iterator materializing the result. An implicit optimization step already shown in figure 11.3 is to drop all write-iterators materializing local variables. Materializing the non local variables is moved to the end of the program. The local variable `$order` for instance is not materialized at all. Blocks of statements which do not include any local variable could be integrated into the surrounding block like the statement block contained in the if-clause in the example shown in figure 11.3.

  As mentioned in section 11.5.2 for each variable changed inside the If-clause an if-iterator has to be generated. A single assignment-iterator evaluating the if-condition is created which in return has to be evaluated by each if-iterator. Depending on the condition-variable (denoted by `$_t0` in figure 11.3), the if-iterator evaluates one of the two further input iterators (denoted by `$addressbook` in figure 11.3).

  If a second statement is added to the If-clause, another if-iterator has to be added. An example illustrating this case is shown in figure 11.4. If an email address exists a confirmation email is sent, which requires a second if-iterator to be added. Since this send-iterator contains a dependency which is not expressed by variables, a materialize-iterator is included which evaluates the new send-iterator.

- **While-clause** As the description of the while-Iterator in section 11.5.2 suggests, iterator graphs become a little more complicated. The example used in this section to illustrate an iterator graph expressing a loop is shown in figure 11.5.

  The small example iterates through the addressbook and sends an email to each included address. The variable `$output` returns the number of emails sent. The while-iterator is preceded by the predicate provided by a usual assign-iterator
  `$_t0 := exits($addressbook/Email[$i])`
  and a materialize-iterator which evaluates the loop-body. The loop-body itself contains the send-iterator and the assignment `$i := $i +1` which is evaluated by a write-Iterator. The loop-body is delimited by a set of read-iterators on one side and a set of write-iterators on the other side. Each further evaluation of the loop body reads the materialized values of the previous iteration.

```
let $i := 0;
while(
  exits($addressbook/Email[$i]))
{

 <bargain>new XML language</bargain>
  ==> "mailto:"{$addressbook/Email[$i]};

   let $i := $i +1;
}
let $output := $i;
```

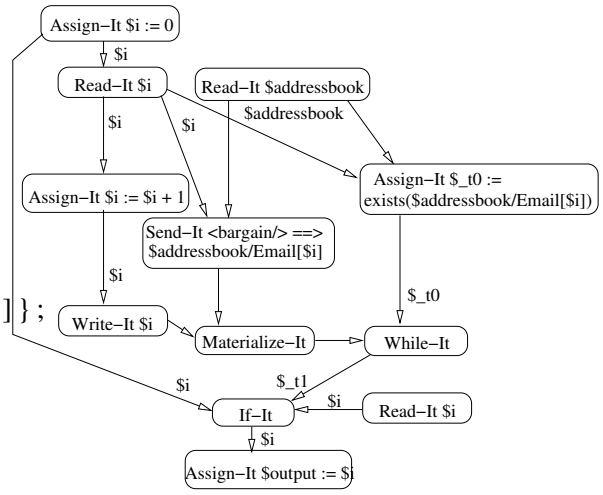


Figure 11.5: *XL while-clause and the generated statement graph*

11.5.3 Discussion

In this section consequences of the described model are discussed.

Lock contention

The persistent variables in the global scope of an XL Web service can be accessed concurrently by different operations. To avoid read-write conflicts a variable based locking mechanism has to be implemented. In the traditional execution approach for each statement a set of read- or write locks has to be set before the statement is executed and can be released afterwards.
For the streamed execution approach on the the other hand, read- and write locks for all variables accessed throughout the whole streamed operation have to be set. Since the iterative execution of a statement could take as long as executing the whole operation, locks have to be held longer compared to the traditional execution approach.
This increased lock contention is due to the implementation of the update iterators. Since each update iterator reads a whole variable, changes the content and passes the whole variable to a subsequent iterator, a lock on the whole variable has to be held. If instead locks are not held for whole variables, the update iterator could possibly read and update only a small fraction of a variable and lock contention would be reduced. The implementation of this kind of sophisticated update iterator is part of future work.

Variable Scope

The optimization distinguishes between local and global variables. Global variables within a Web service are valid throughout several operation calls. As described in section 11.4 the scope of a variable has to be considered during graph generation. An assignment to a local variable not being used anymore is dead code while the same assignment to a global variable is not dead code. Changing the scope of a variable therefore causes significant changes in the generated iterator graph. In order to materialize a changed global variables in the context, a write-Iterator has to be added which evaluates the assignment. As described in section 11.5.2 each block of statements is translated to a set of iterator graphs. Each block propagates the changes made to non local variables to the surrounding scope by providing one iterator for each variable. If

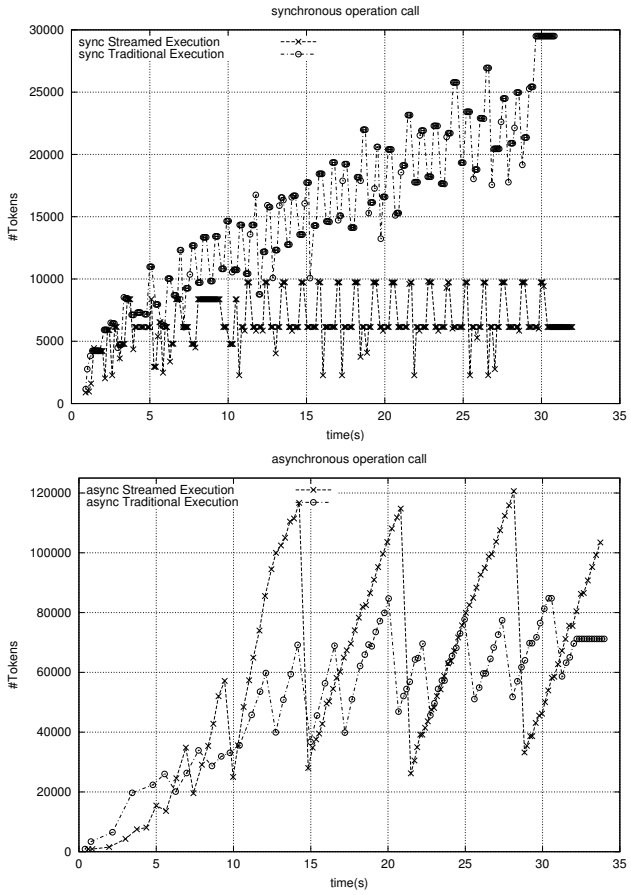


Figure 11.6: *Memory requirements: Number of tokens referenced by the XL runtime system if a global variable is updated. (on the left: synchronous calls and no lock contention, on the right: asynchronous calls and high lock contention)*

an XL operation is executed, the surrounding scope contains the predefined variables `$input` and `$output` and the global variables. The iterator graph expressing an operation therefore provides a set of iterators which execute changes to these variables. An operation is executed by evaluating these iterators. If a further global variable is changed, a new iterator has to be added expressing this dependency.

Streamed Values in Java

A streamed processing of data in hand coded Java is also feasible but could only be achieved if it is explicitly programmed. The evaluation of a cursor inside Java JDBC for instance is simple and implements a data stream as well. But writing a whole program in a streamed fashion is much more difficult. Especially if query results are access by different subsequent expressions, the optimization of query processing becomes significantly more complicated.

11.6 Experiments

The iterator concept described in the previous sections has been implemented as part of a Java based XL runtime system. By taking up the initial motivation, we wrote a small set of simple XL programs illustrating the advantages of the stream execution. In the following subsections experiments exemplifying the memory requirements and the dynamic adaptivity are shown.
For all experiments we used a Intel Pentium 4 processor with 2.80 GHz, 1 GByte main memory running a Linux kernel 2.4.20. The XL runtime system is implemented in Java using the Java VM 1.4.2.

11.6.1 Memory Requirements

In order to illustrate the reduced memory requirements, already a simple program like the one listed in the initial motivation section could be used. The program we chose to illustrate the memory requirements selects certain lineitems from a incoming purchase order and inserts the invoice-code into a global variable named `$e-invoices`:

```
let $items := $input/Order/LineItem;
let $elec := $items/[Type eq 'electro'];
insert $elec/Invoice into $e-invoices;
let $output := count($elec);
```

In order to illustrate the memory requirements two different test setups were performed. The listed XL fragment above was called synchronously. A single XL client program continuously issued operation calls and waited each time for the return message. Figure 11.6 shows the resulting graph on the left. Instead of Bytes allocated by the Java VM, the graph plots the number of tokens not removed by the Java garbage collection.
As shown in the listing above, the global variable is continuously growing each time the operation is called. If the operation is executed traditionally, the variable has to be loaded into the VM each time completely for being updated. In contrast to this the streamed execution does not load a whole variable at once but reads it iteratively. Therefore the number of tokens referenced at a time during the streamed execution remains at the same level for each operation call while the number of tokens required by the traditional execution grows (as shown in figure 11.6 on the left). The right part of figure 11.6 shows the same operation being called, but this time asynchronously – the XL server is flooded by calls for the same operation. This time the lock contention for the global variable dominates the number of tokens referenced significantly. The two graphs on the right of figure 11.6 plot the number of tokens if the system is flooded by calls. As explained in section 11.5.3 variable locks for the streamed execution are held longer compared to the traditional approach. In figure 11.6 this effect is shown by the high peeks of the streamed execution plot. Several issued operation calls add up and wait until locks are released. As the lock contention for the traditional execution model is not as high, concurrent operation calls do not block each other so much.

11.6.2 Response time

If the program executed does not contain a loop, the time required for generating partial results, depends on XQuery expressions used. If the evaluated expressions do not include pipeline-

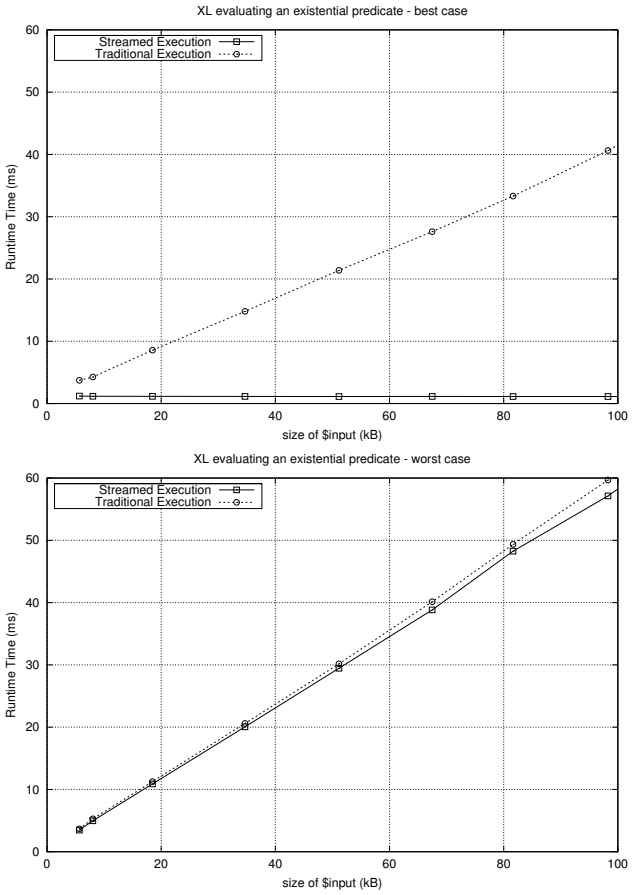


Figure 11.7: *Existential predicate. On the left the predicate is decided after evaluating a small fraction of the input variable, on the right the whole variable is required.*

breaking iterators, the the first result elements can be generated immediately compared to the total runtime of the program (< 1 ms).

11.6.3 Dynamic Adaptivity

The demand driven execution of statements accomplished by the streamed statement execution is advantageous if local variables are set but only a small fraction is used. Subsequent statements using a local variable provide feedback by closing input iterators which are not needed anymore. A frequent example are statements containing an existential predicate as shown in the listing below

```
let $items := $input/Order/Item;
if ($items/Currency = 'EURO') {
   let $output := <answer>
                    Only Dollars accepted
                  </answer>;
   } else {
   let $output := sum($items/Amount);
}
```

The predicate `$items/Currency = 'EURO'` is true if the sequence returned by the path expression `$items/Currency` contains the value `'EURO'`. Therefore this expression can

possibly by evaluated by only considering a single item instead of the whole sequence assigned to `$items`. Figure 11.7 shows the runtime of the program listed above in milliseconds depending on the size of the variable `$input` for the streamed and the traditional execution concept. The left part of figure 11.7 illustrates the potentially unlimited speedup which could be achieved by not executing the initial assignment completely if the following predicate evaluates to true. In fact figure 11.7 shows the best case, as the predicate only needed the first item-element for evaluation in all cases. The XL-program listed above only makes sense if the variable `$items` is further processed in case the predicate turns to false. The graph on the right side of figure 11.7 plots this case as the variable `$items` is consumed completely by a sum expression. By showing these two extrems either consuming only the first item or the whole sequence, the two graphs in figure 11.7 illustrate the best and the worst case for executing this simple XL program.

11.6.4 XL versus Java

In order to give a more meaningful performance impression, we compared an XL program being executed by the streamed and traditional concept and a similar Java Class. Therefore we implemented the small program listed in the previous section as a Java Class using the same XQuery engine as XL.
In Java the same semantic can be programmed as a JDBC program evaluating each expression and materializing the result. As in JDBC, in the XDBC interface [FHK+03] query results can be bound to external variables in other XQuery expressions. If the variable used by an XQuery expressions is directly linked to an iterator representing a previous expression, a hand coded data stream through several expressions is possible as in the streamed processing concept of XL. As mentioned in section 11.5.3, this kind of hand coded data stream is much more error prune and complicated compared to XL. Figure 11.8 shows the same XL graph as in 11.7. The program listed in the previous section is evaluated, whereas the predicate `$items/Currency = 'EURO'` only needs the first item element included in each message to evaluate to true. This use case shows again the best case for the streamed processing concept, but comparing XL and hand coded Java.
Figure 11.8 plots the runtime for executing the program depending on the size of the `$input` variable. In both cases, Java and XL, lazy evaluation is used and only a small fraction of the `$input` has to be read. As figure 11.8 shows the generated XL program almost performs as well as the hand coded Java version.

11.7 Related Work

The streamed processing concept based on the iterator pattern is well known in the database comunity and has been described in great detail by Goetz Graefe 1993 [Gra93]. The iterator concept has been used by numerous projects.
The streamed processing of information in general is applied in various scenarios. The MIMD languages used most prominently for scientific computation incorporate a different data streaming concept. Instead of the demand driven iterator model, data is pushed into the different working processes. The Nested Data-Parallel Language (NESL) of Guy E. Blelloch et al. [BCH+93] includes aspects of data-parallel and control-parallel languages. Another very interesting functional language called *SISAL* [Mo85] was developed by the Lawrence Livermore National Lab-

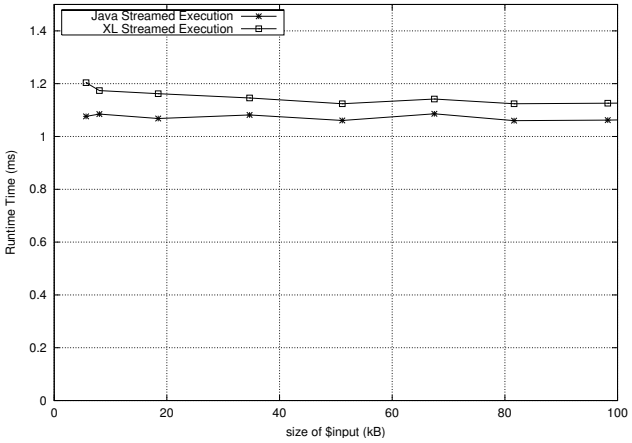


Figure 11.8: *Streaming XL versus Java evaluating an existential predicate.*

oratory for high-performance numerical computing. The Manchester dataflow computer implemented a runtime environment for SISAL programs [GKW85].
Another domain using streamed processing concepts is the signal processing software. The Dataflow Process Networks from E. Lee and T. Parks [LP95] for example treat signal processing networks as a special case of Kahn process network [Kah74].
The performance of Web services is addressed by several different approaches. Both, the *Ninja* project and the *Staged Event-driven Architecture (SEDA)* [WCB01] from Berkeley describe an event driven service architecture. SEDA for instance achieves a high level of concurrency by a sequence of multi-threaded stages processing a request. The *Ninja* Project proposes several different processing patterns like wrapping or pipelining for service processing [GWo01].
The Telegraph project from the Berkeley database group implement adaptive query processing techniques based on data streams [AH00].

Chapter 12

Related Work

The range of subject covered by the concepts used by the XL Web service engine and its implementation is very wide. In the following subsections the related work of different areas are summarized.

12.1 Web Services

Interfaces

The topic *Web services* is subject of numerous publications. For the description of a Web service interface, the Web service Description Language (WSDL [W3C01]) standard is established by now. Since Web services are mainly used integrate existing applications, some effort is put into languages like BPEL [IBM03] or WS-CDL [W3C04e] which describe how different services interact using SOAP messaging format [W3C04c].
Projects like OWL-S [Dav04] or WSMF [FB02b] focus on extending the interface of a service by not only describing the technical interface but also the non functional properties.

Application Server

Another issue related to XL and Web services is the processing of XML data. To process XML one could choose the *Java style* approach. Among other things, the J2EE™framework [Sun05b] and the Java Web Services Developer Pack [Sun04b] contain tools to marshal XML to Java objects and back (JAXB), a Web service registration API to be used for example with UDDI [UDD] (JAXR), and to send and receive XML messages (JAXM). SOAP [W3C04c] has established itself in the XML context as the standard messaging protocol using predominantly HTTP as the underlying network protocol. The Apache Axis framework [Apa06] provides an open source implementation of SOAP, a commercial implementation is provided for example by the Systinet Server For Java [Sys06]. . As for high-level programming languages providing more abstraction within J2EE, the recent development of Enterprise Java beans (as part of the J2EE™framework [Sun05b]) using metadata annotations and the EJB query language (EJB QL) is very interesting.
Since application servers are being build and are commercially successful, the big players should be mentioned as well. The IBM Websphere [IBM; HK04] server implements the J2EE model and provides Java design principles for middleware applications. In competition with

WebSphere, BEA promotes their application server WebLogic [BEA]. Like WebSphere, WebLogic provides additional support for various application scenarios, as for example clustered Web services [Jac03].
Since Web services are mainly used as a means to integrate existing applications the issue of efficient composition of Web services is rarely addressed. The meta programming approach published by C. Pautasso and G. Alonso [PA04] distinguishes between different service types requiring different wrapping.
The design patterns and concepts we used are strongly related to the resource and process management used within operating systems [SGG03].

12.2 Requirements Engineering

The conceptual part of this thesis on how to build an XML processing engine addresses the distinction between functional and non-functional requirements and is based on fundamental software engineering concepts. Several publications based on the work of L. Chung, J. Mylopoulos et. al. [MCN92]) address how to represent NFRs and how to consider NFRs while designing software [CNYM00; GY01]. Further means to achieve NFRs are provided by [ABC+97].

Part III

Conclusion and Outlook

The development of a Web service engine, like XL, is a complex project. Technical and functional aspects are addressed:

- Implementation of the engine itself
- XML storage and the XQuery engine integration.
- Cluster and PDA Web service engines

The non-technical and non-functional aspects of Web services are best described by a set of "how to" questions:

- How to describe and develop a Web service ? Which level of abstraction is necessary ? Which are the required primitives ? How to debug a Web service ?
- How to use XML ? How to process XML efficiently ? How to structure XML well ?

The XL Web service engine uses XML not only as means of communication, but also as a modelling device used to structure data. XL provides an integrated XML processing platform. The term *integrated* implies: XML serves as the general data modell.
The paradigm of a Web Service architecture requires additional means, compared to the common application development. Web services are meant to be the building blocks, used to compose complex services. In order to compose services conveniently, additional features are necessary: a more sophisticated interface description and a Web service engine being able to use this description.
When developing Web services, one typically makes assumptions on how a service is provided. Compared to common applications, Web service rely typically on remote software components. The developer, of course, makes assumptions about the functional as well as non-functional properties of the used components:

- cost for invoking a service is less then €10
- the machine providing a service uses a UPS
- the service is tested according to a certain standard
- 99% operation calls return an answer in < 10 ms.

When using a certain component, the developer usually checks whether it fulfills his requirements or not. But for Web services, this type reliability if not given. In contrast to a common application which is developed configured and deployed, a Web service changes – only the technical interface description as it is provided by a WSDL document remains valid. For Web services becoming more reliable software components a more detailed property description is necessary and it has to be integrated into the service engine. The Web service engine has to guarantee, that the assumption made during developing an application remain valid.
Using a proper XML dialect (e.g., RDF [W3C04b]), it is simple to describe the properties of a service. If Web services are software components, convenient and reliable to use, NFRs have to be checked automatically – by the Web service engine. This next step, the integration of Web service properties into the a Web service engine, is still missing. Future work is therefor to integrate service property descriptions into a Web service engine and to give the Web service developer means at hand to design the property handling. In the following, the basic questions arising form these considerations are described.
Properties could be grouped depending on the part of the computer system they address:

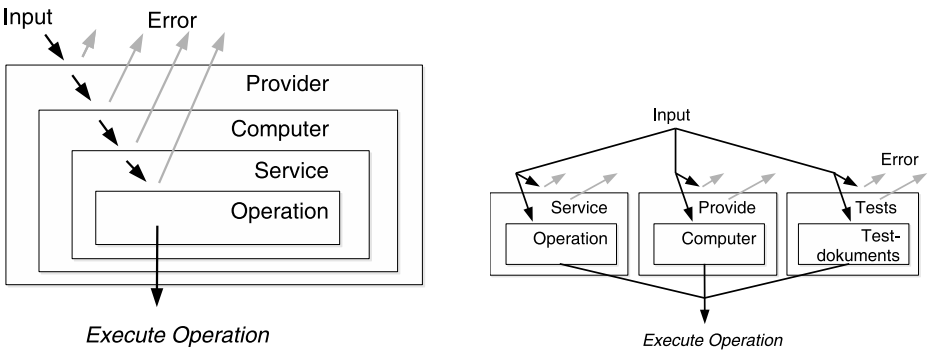


Figure 12.1: *Possible strategies on how to handle service properties in a Web service engine. The arrows in these two figures illustrate in and out going messages. Different properties of a Web service are modelled and evaluated by different layers within the engine: service-provider, computer, service, conversation or operation.*

- the cost for invoking a service could depend on the service provider hosting the service.
- whether the computer is connected to UPS depends on the machine the service is deployed on.
- the level of quality-assurance could depend on the service provider, the service type, or on a individual operation.
- the quality of service could depend on various factors (e.g., current workload, hardware, etc.)

As a consequence, the properties describing a service have to modelled themselves. Figure 12.1 shows two possible approach on how to treat modell properties and correspondingly the how to evaluated conditions set by a client invoking a service. Properties of a service could be modelled by a layered architecture: properties describing the service provider are checked first. If the conditions of the client are fulfilled, the properties describing the computer, and the service itself are checked (left part of figure 12.1). Alternatively, the different properties of service could be checked independently (right part of figure 12.1).
If service properties are to be integrated into the service engine several design questions have to be answered:

Development How should requirement description and the conditions set by a client be integrated into service development and deployment?

- How are conditions specified, which elements can be used to express properties and conditions ? Either XQuery pre- and post conditions like in XL or possibly Java annotations could be used.
- When are conditions checked ? Conditions could be checked at compile time , at time of deployment, at runtime, or randomly.
- How often do conditions change, and how are changes handled ? If the properties of remote service change it could be necessary to select a different service provider or a different service. Changing properties and conditions cannot be treated uniformly, some changes might require an instant response while other are not so urgent.

- How do I integrate conditions into my IDE ? Requirements engineering has to be integrated into the development environment.

Runtime engine How should the runtime engine be designed in order to process service properties and conditions set by clients efficiently:

- Is meta information provided by the Web service itself or by a separate system ?
- How do properties depend on each other ?
- Which conditions can be checked at compile time or at runtime and how is the result represented in the meantime ?
- Which condition is checked by the server, which by the client ?

Costs If the Web service engine not only provides the service itself but provides additional information about the service itself and evaluates requirements, a price has to be paid. The question is therefore: how much performance does it cost ?

Describing service properties is even when using XML difficult (Which properties are important ? Which XML dialect should be used ? Which ontology is the description based on ?), but sooner or later a common service description language will establish. What is left, is engineering problem on how to process these properties.
XL is a XML processing language describing Web services. XL could grow into an integration tool being able to orchestrate XML, Java, or RMI message between different services. Furthermore, XL could turned into a language and service engine intergrating a more sophisticated service description.

Bibliography

[ABC+97] Gregory Abowd, Len Bass, Paul Clements, Rick Kazman, Linda Northrop, and Amy Zaremski. Recommended Best Industrial Practice for Software Architecture Evaluation. Technical report, Software Engineering Institute, Carnegie Mellon University, Jan 1997.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *SIGMOD 2000, Dallas*, pages 261–272, 2000.

[Apa05a] Apache Software Foundation. The Apache HTTP Server Project. http://httpd.apache.org, Jan 2005.

[Apa05b] Apache Software Foundation. The Apache XML Project. http://xml.apache.org, Jan 2005.

[Apa05c] Apache Software Foundation. The Jakarta Project. http://jakarta.apache.org, Sep 2005.

[Apa06] Apache Software Foundation. WebServices – Axis. http://ws.apache.org/axis, Apr 2006.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[BCH+93] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a Portable Nested Data-Parallel Language. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–111, San Diego, May 1993.

[BEA] BEA Systems, Inc. WebLogic Server. http://www.bea.com/products/weblogic/server.

[BL94] T. Berners-Lee. RFC 1630: Universal Resource Identifiers in WWW. `http://www.ietf.org/rfc/rfc1630.txt`, 1994.

[BMS05] Gavin Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in Cω. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference*, pages 287–311. Springer, 2005.

[BPM] BPMI.org. Business Management Initiative. http://www.bpmi.org.

[cas05] castor.org. The Castor Project. `http://www.castor.org/`, Jan 2005.

[CB00] R. G. G. Cattell and Douglas K. Barry. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[CCDG05] Joëlle Coutaz, James L. Crowley, Simon Dobson, and David Garlan. Context is key. *Commun. ACM*, 48(3):49–53, 2005.

[CD99] Luca Cardelli and Rowan Davies. Service combinators for web computing. *IEEE Trans. Software Eng.*, 25(3):309–316, 1999.

[CFR06] Don Chamberlin, Daniela Florescu, and Jonathan Robie. XQuery Update Facility. http://www.w3.org/TR/2006/WD-xqupdate-20060127, Jan 2006. W3C Working Draft.

[CNYM00] L. Chung, B.A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.

[Coa] DAML Service Coalition. DAML-S: Semantic Markup for Web Services. http://www.daml.org/services.

[CRF00] Donald D. Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An xml query language for heterogeneous data sources. In *WebDB (Informal Proceedings)*, pages 53–62, 2000.

[CW03] Sandeep Chatterjee and James Webber. *Developing Enterprise Web Services: An Architecture Guide*. Prentice Hall, Nov 2003.

[CX00] J. Cheng and J. Xu. XML and DB2. In *Data Engineering*, pages 569–573, 2000.

[Dav04] David Martin and others. OWL-S: Semantic Markup for Web Services. http://www.daml.org/services/owl-s, Nov 2004.

[DK05] Cristian Duda and Donald Kossmann. Adaptive XML Storage or The Importance of Being Lazy. *XIME-P Workshop (Poster)*, Jun 2005.

[DOM04] DOM. Document Object Model. http://www.w3.org/DOM, Apr 2004.

[EM05] Andrew Eisenberg and Jim Melton. XQuery 1.0 is Nearing Completion. In *ACM SIGMOD Record*, volume 34, pages 78–84, New York, NY, USA, December 2005. ACM Press.

[FB02a] D. Fensel and C. Bussler. The Web Service Modeling Framework WSMF. http://www.cs.vu.nl/ dieter/wese/wsmf.bis2002.pdf, 2002.

[FB02b] Dieter Fensel and Christoph Bussler. The web service modeling framework wsmf. *Electronic Commerce Research and Applications*, 1(2):113–137, 2002.

[FGK02] D. Florescu, A. Grünhagen, and D. Kossmann. XL: An XML programming language for web service specification and composition. In *WWW2002 Conference Proceedings*, 2002.

[FGK03] D. Florescu, A. Grünhagen, and D. Kossmann. XL: A Platform for Web Services. In *CIDR 2003, Asilomar*, Jan 2003.

[FHK+03] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. Carey, A. Sundararajan, and G. Agrawal. The BEA/XQRL Streaming XQuery Processor. In *Proc. VLDB 2003*, pages 997–1008, 2003.

[FK99] Daniela Florescu and Donald Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.

[FSC+03] Mary Fernández, Jérôme Siméon, Byron Choi, Amélie Marian, and Gargi Sur. Implementing XQuery 1.0: The Galax Experience. *Proc. of the VLDB Conf.*, 2003.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1), 1985.

[GKW85] J. R Gurd, C. C Kirkham, and I. Watson. The manchester prototype dataflow computer. *Commun. ACM*, 28(1):34–52, 1985.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[GWo01] Steven D. Gribble, Matt Welsh, and other. The Ninja architecture for robust Internet-scale systems and services. *Computer Networks*, 35(4):473–497, 2001.

[GY01] Daniel Gross and Eric S. K. Yu. From Non-Functional Requirements to Design through Patterns. *Requirements Engineering*, 6(1):18–36, 2001.

[HH04] Michael P. Haustein and Theo Härder. Fine-Grained Management of Natively Stored XML Documents. submitted, 2004.

[Hib05] Hibernate. `http://www.hibernate.org`, 2005.

[HK04] Robert High and Matthias Kloppmann. WebSphere Programming Model and Architecture. *Datenbank-Spektrum*, 8:18–31, 2004.

[HKM04] Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Evaluating Lock-based Protocols for Cooperation on XML Documents. *SIGMOD Record*, 33(1):58–63, March 2004.

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

[IBM] IBM Corporation. IBM WebSphere Software. http://www.ibm.com/software/websphere.

[IBM03] BEA IBM, Microsoft. Business process execution language for web services. `http://www.ibm.com/developerworks/library/ws-bpel`, May 2003.

[IEE00] IEEE Std 1471-2000. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, Sep 2000.

[ISO86] ISO (International Organization for Standardization). ISO 8879: Standard Generalized Markup Language (SGML)., Oct 1986.

[ISO03a] ISO. Information Technology-Database Language SQL., Dec 2003.

[ISO03b] ISO/IEC-10646. Information technology – Universal Multiple-Octet Coded Character Set (UCS). Technical report, ISO (International Organization for Standardization), Dec 2003.

[Jac03] Dean Jacobs. Distributed Computing with BEA WebLogic Server. In *CIDR*, 2003.

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

[Jos03] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. `http://www.ietf.org/rfc/rfc3548.txt`, 2003.

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

[Kah74] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proceedings of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

[KM00] Carl-Christian Kanne and Guido Moerkotte. Efficient Storage of XML Data. *ICDE, San Diego, USA*, pages 198–, 2000.

[Lon05] Fred Long. Software Vulnerabilities in Java. Technical report, Carnegie Mellon University, 2005. `http://www.sei.cmu.edu/publications/documents/05.reports/05tn044/05tn044.html`.

[LP95] Edward A. Lee and Thomas M. Parks. Dataflow Process Networks. In *Proceedings of the IEEE*, May 1995.

[LY99] T. Lindholm and F. Yellin. The Java Virtual Machine Specification . http://java.sun.com/docs/books/vmspec, 1999.

[MCN92] John Mylopoulos, Lawrence Chung, and Brian A. Nixon. Representing and Using Nonfunctional Requirements: A Process-Oriented Approach. *Software Engineering*, 18(6):483–497, 1992.

[Men98] Matthias Menge. Sprungvorhersage in Fließbandprozessoren. *Informatik Spektrum*, 21(2):73–79, 1998.

[Mic05a] Microsoft Corp. COM: Component Object Model Technologies. `http://www.microsoft.com/com/default.mspx`, 2005.

[Mic05b] Microsoft Research. $C\omega$: Comega. `http://research.microsoft.com/Comega/`, 2005.

[Mic06a] Microsoft Corp. Biztalk initiative. http://www.microsoft.com/BizTalk, 2006.

[Mic06b] Microsoft Corp. .net. http://www.microsoft.com/net, 2006.

[Mo85] J.R. McGraw and other. Streams and Iteration in a Single-Assignment Language. Language reference manual, Lawrence Livermore National Laboratory, January 1985.

[Net03] Network Working Group. Blocks Extensible Exchange Protocol . http://www.beepcore.org/beepspecs.html, Oct 2003.

[Obj05] Object Management Group. Common object request broker architecture (corba/iiop). `http://www.omg.org/technology/documents/corba_spec_catalog.htm`, Mar 2005.

[OOP+04] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: insert-friendly XML node labels. In *ACM SIGMOD*, pages 903–908, New York, NY, USA, 2004. ACM Press.

[PA04] Cesare Pautasso and Gustavo Alonso. From Web Service Composition to Megaprogramming. *Proceedings of VLDB Workshop on Technologies for E-Services*, (5), Aug 2004.

[Par05] Terence Parr. ANother Tool for Language Recognition (ANTLR). `http://www.antlr.org`, Jan 2005.

[Pat05] David A. Patterson. 20th century vs. 21st century C&C: the SPUR manifesto. *Commun. ACM*, 48(3):15–16, 2005.

[Pos82] Jonathan B. Postel. RFC 821: Simple Mail Transfer Protocol. `http://www.ietf.org/rfc/rfc821.txt`, Aug 1982.

[RF99] et al. R. Fielding. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1. `http://www.ietf.org/rfc/rfc2616.txt`, 1999.

[RLS98] Jonathan Robie, Joe Lapp, and David Schach. XML Query Language (XQL). In *QL*, 1998.

[SAX04] SAX. Simple API for XMLl. http://www.saxproject.org, Apr 2004.

[Sax06] Saxonica Corp. Saxon. http://www.saxonica.com, 2006.

[SGG03] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts, 6th Edition*. John Wiley & Sons, Inc., 2003.

[Sun] Sun Microsystems, Inc. Sunone. http://www.sun.com/software/sunone.

[Sun01] Sun Microsystems, Inc. JDBC™3.0 Specification. http://java.sun.com/products/jdbc, Dec 2001. Final Release.

[Sun04a] Sun Microsystems, Inc. Java Architecture for XML Binding (JAXB). http://java.sun.com/xml/jaxb, June 2004.

[Sun04b] Sun Microsystems, Inc. Java Web Services Developer Pack. http://java.sun.com/webservices/jwsdp, Oct 2004.

[Sun05a] Sun Microsystems, Inc. Java 2 Platform, Standard Edition (J2SE). http://java.sun.com/j2se/index.jsp, Mar 2005.

[Sun05b] Sun Microsystems, Inc. Java Platform, Enterprise Edition (Java EE). http://java.sun.com/j2ee/index.jsp, May 2005.

[Sys06] Systinet Corporation. Systinet Server For Java. http://www.systinet.com/products/ssj, May 2006.

[Tha01] S. Thatte. Xlang overview. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, 2001.

[Tra02] Transaction Processing Performance Council. TPC Benchmark™-W. (Web Commerce). http://www.tpc.org/tpcw, Feb 2002.

[TVB+02] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 204–215, New York, NY, USA, 2002. ACM Press.

[UDD] UDDI.org. Universal Description, Discovery and Integration of Businesses for the Web. http://www.uddi.org.

[VS05] Sreeni Viswanadha and Sriram Sankar. Java Compiler Compiler (JavaCC). `https://javacc.dev.java.net/`, Jan 2005.

[W3C98] W3C. Extensible Markup Language (XML) 1.0. http://www.w3.org/TR/1998/REC-xml-19980210, Feb 1998.

[W3C99] W3C. XML Path Language (XPath). http://www.w3.org/TR/xpath, Nov 1999.

[W3C01] W3C. Web Services Description Language (WSDL). http://www.w3.org/TR/wsdl, Mar 2001.

[W3C03] W3C. XML Protocol Abstract Model. `http://www.w3.org/TR/xmlp-am/`, Feb 2003.

[W3C04a] W3C. Extensible Markup Language (XML) 1.1. http://www.w3.org/TR/2004/REC-xml11-20040204, Apr 2004.

[W3C04b] W3C. Resource Description Framework (RDF). http://www.w3.org/RDF, Oct 2004.

[W3C04c] W3C. Simple Object Access Protocol (SOAP). http://www.w3.org/2000/xp/Group, Jun 2004.

[W3C04d] W3C. Web Services Architecture. http://www.w3.org/TR/ws-arch, Feb 2004.

[W3C04e] W3C. Web Services Choreography Description Language (WS-CDL) . http://www.w3.org/2002/ws/chor, Oct 2004.

[W3C04f] W3C. XQuery 1.0 and XPath 2.0 Data Model. http://www.w3.org/TR/xpath-datamodel, Oct 2004.

[W3C04g] W3C. XQuery 1.0 and XPath 2.0 Formal Semantics. http://www.w3.org/TR/query-semantics, Feb 2004.

[W3C04h] W3C. XQuery 1.0 and XPath 2.0 Functions and Operations. `http://www.w3.org/TR/xquery-operators`, July 2004.

[W3C05a] W3C. Extensible Stylesheet Language (XSL). http://www.w3.org/Style/XSL, Jan 2005.

[W3C05b] W3C. Scalable Vector Graphics (SVG) 1.2. http://www.w3.org/Graphics/SVG, Apr 2005.

[W3C05c] W3C. Voice Extensible Markup Language (VoiceXML) . http://www.w3.org/TR/voicexml21, Jun 2005.

[W3C05d] W3C. XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery`, Feb 2005.

[W3C06a] W3C. HyperText Markup Language (HTML). http://www.w3.org/MarkUp, Jan 2006.

[W3C06b] W3C. XML Schema 1.1. http://www.w3.org/XML/Schema, Feb 2006.

[Wal91] Gregory K. Wallace. The JPEG still picture compression standard. In *Communications of the ACM*, volume 34, pages 30–44, Apr 1991.

[WCB01] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, Banff, Canada, October 2001.

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

[Wit03] Elmar Witte. Optimierung von xl web-services. Diploma Thesis (in German), TU München, Feb 2003.

List of Figures

List of Tables