



**XML-BASED  
FRAMEWORKS AND STANDARDS  
FOR B2B ECOMMERCE**

**Frederik Willaert**

Verhandeling aangeboden tot  
het behalen van de graad van  
Handelsingenieur in de Beleidsinformatica

Promotor : Prof. Dr. J. Vandenbulcke



Frederik Willaert

XML-based  
Frameworks and Standards  
for B2B eCommerce

Korte Inhoud Verhandeling:

The central topic of this thesis is the application of XML (eXtensible Markup Language) for electronic business between companies – the so-called “*Business-to-Business eCommerce*”. It mainly deals with two bodies of specifications: the W3C XML standards, and the ebXML framework proposed by UN/CEFACT and OASIS. First, the family of XML standards is discussed, paying attention to their interrelationships and relative importance. The second part concentrates on ebXML, which provides a complete framework for setting up electronic business-to-business transactions. The combined application of the specifications discussed in this thesis, is likely to be the foundation of most B2B eCommerce activities that are about to flourish in the upcoming years.

Promotor : Prof. Dr. J. Vandenbulcke

## Word of Thanks

Many people have played an important role during the accomplishment of this thesis. Without them, this study would have been a lot harder or even impossible to complete, and therefore I would like to express my sincere gratitude.

Particular thanks are due

*to my promotor, Prof. Dr. J. Vandenbulcke, for giving me full support and providing guidance throughout the work,*  
*to Mr. G. Van Humbeeck of Application Engineers, for his good advice and useful hints, and for keeping my attention focused,*  
*to Dr. W. Lemahieu, for his help and assistance.*

Though they may not have provided direct input, several other people have meant a lot to me during the past year. Special mention goes

*to my parents, for giving me solid backing and for all those small but valuable things,*  
*to Anne and Evelien, for their ready help,*  
*to Peggy, for being there for me,*  
*to my housemates at “Blink 36”, and especially to Tom, for being a good listener and a helpful hand, and*  
*to many other friends, whose presence has been of great importance to me.*

Frederik Willaert

May 14, 2001

---

*Make everything as simple as possible, but no simpler.*

*(Albert Einstein)*

## **Overview Table of Contents:**

<b><i>Part 1: General Overview of XML</i></b> .....	<b>3</b>
<b>1.1 Introduction</b> .....	<b>4</b>
<b>1.2 The XML Language : A First Outline</b> .....	<b>7</b>
<b>1.3 Know what you're talking about: An Example</b> .....	<b>10</b>
<b>1.4 Family Matters</b> .....	<b>14</b>
<b>1.5 Conclusion</b> .....	<b>18</b>
<b><i>Part 2: The Family of XML Standards</i></b> .....	<b>19</b>
<b>2.1 Introduction</b> .....	<b>20</b>
<b>2.2 The XML Specification</b> .....	<b>21</b>
<b>2.3 XML and Design: Setting things up</b> .....	<b>22</b>
2.3.1 Basic Modelling Requirements .....	22
2.3.2 DTDs: A preliminary answer .....	25
2.3.3 Advanced Modelling Requirements .....	28
2.3.4 XML Namespaces: Know where you belong.....	29
2.3.5 XML Schemas: A Typical structure.....	32
<b>2.4 Accessing XML: Climb the tree or just strip the leaves ?</b> .....	<b>39</b>
2.4.1 DOM (Document Object Model) .....	40
2.4.2 SAX (Simple API for XML – Non-W3C) .....	41
2.4.3 Conclusion: DOM vs. SAX.....	42
<b>2.5 Manipulating and Querying XML : Get what you need, and put it where you want it</b> .....	<b>43</b>
2.5.1 XPath (XML Path Language).....	43
2.5.2 XSL(T) (eXtensible Style Sheet Transformations) .....	45
2.5.3 XQuery (XML Query Language).....	56
2.5.4 XQuery: Conclusion.....	61
<b>2.6 Connecting XML: XLink</b> .....	<b>62</b>
<b>2.7 XML Standards: Conclusion</b> .....	<b>65</b>

<b>Part 3: <i>Electronic Business and ebXML</i></b> .....	<b>67</b>
<b>3.1 Introduction</b> .....	<b>68</b>
<b>3.2 XML Application Areas</b> .....	<b>70</b>
3.2.1 A double dichotomy for classification .....	70
3.2.2 The resulting Four-Quadrant Scheme .....	71
3.2.3 Some further observations about the scheme .....	74
<b>3.3 Why (not) EDI ?</b> .....	<b>77</b>
3.3.1 What is EDI ? .....	77
3.3.2 What's wrong with EDI ?.....	78
3.3.3 Lessons to be learned from EDI.....	81
3.3.4 Down to earth: a well-balanced attitude.....	82
<b>3.4 A Generic B2B Framework</b> .....	<b>84</b>
3.4.1 Introduction .....	84
3.4.2 An example: The eCo Framework .....	85
3.4.3 Building blocks.....	87
3.4.4 An additional view on electronic business .....	89
<b>3.5 ebXML: A “Complete” Framework</b> .....	<b>92</b>
3.5.1 Introduction .....	92
3.5.2 ebXML Requirements and Problem Domain.....	95
3.5.3 ebXML Architecture .....	98
3.5.4 The ebXML Specifications .....	110
3.5.5 Business Process and Core Components.....	112
3.5.6 Transport, Routing and Packaging .....	127
3.5.7 Collaboration Protocol Profile and Agreement (CPP/CPA) .....	150
3.5.8 Registry/Repository .....	158
3.5.9 Marketing/Awareness and Education.....	183
3.5.10 ebXML: Conclusion .....	183

## **Detailed Table of Contents:**

<b><i>Part 1: General Overview of XML</i></b> .....	<b>3</b>
<b>1.1 Introduction</b> .....	<b>4</b>
<b>1.2 The XML Language : A First Outline</b> .....	<b>7</b>
<b>1.3 Know what you're talking about: An Example</b> .....	<b>10</b>
<b>1.4 Family Matters</b> .....	<b>14</b>
<b>1.5 Conclusion</b> .....	<b>18</b>
<b><i>Part 2: The Family of XML Standards</i></b> .....	<b>19</b>
<b>2.1 Introduction</b> .....	<b>20</b>
<b>2.2 The XML Specification</b> .....	<b>21</b>
<b>2.3 XML and Design: Setting things up</b> .....	<b>22</b>
2.3.1 Basic Modelling Requirements .....	22
2.3.1.1 Documentation and Explanation .....	23
2.3.1.2 Definition of Constraints .....	23
2.3.1.3 Validation .....	24
2.3.2 DTDs: A preliminary answer .....	25
2.3.2.1 Documentation and Explanation .....	25
2.3.2.2 Definition of Constraints .....	26
2.3.2.3 Validation .....	27
2.3.2.4 DTDs: Conclusion .....	27
2.3.3 Advanced Modelling Requirements .....	28
2.3.3.1 Mixing Vocabularies .....	28
2.3.3.2 Modularization and Reuse .....	29
2.3.3.3 Inheritance .....	29
2.3.4 XML Namespaces: Know where you belong.....	29
2.3.5 XML Schemas: A Typical structure.....	32
2.3.5.1 Characteristics of the XML Schema Language.....	33
2.3.5.2 Documentation and Explanation .....	35
2.3.5.3 Definition of Constraints .....	36
2.3.5.4 Validation .....	36

2.3.5.5	Advanced Requirements.....	37
2.3.5.6	XML Schemas: Conclusion.....	37
<b>2.4</b>	<b>Accessing XML: Climb the tree or just strip the leaves ? .....</b>	<b>39</b>
2.4.1	DOM (Document Object Model) .....	40
2.4.2	SAX (Simple API for XML – Non-W3C) .....	41
2.4.3	Conclusion: DOM vs. SAX.....	42
<b>2.5</b>	<b>Manipulating and Querying XML : Get what you need, and put it where you want it.....</b>	<b>43</b>
2.5.1	XPath (XML Path Language).....	43
2.5.2	XSL(T) (eXtensible Style Sheet Transformations) .....	45
2.5.2.1	Introduction .....	45
2.5.2.2	The XSLT Mechanism .....	47
2.5.2.3	Why XSLT? – Push/Pull Approach to Stylesheet Design.....	50
2.5.2.4	A small digression : Transforming into SQL Insert Instructions .....	52
2.5.2.5	Conclusion.....	54
2.5.3	XQuery (XML Query Language).....	56
2.5.3.1	Introduction .....	56
2.5.3.2	The XML Query Language .....	56
2.5.3.3	The XPath-XSLT Combination as a Query Language.....	59
2.5.4	XQuery: Conclusion.....	61
<b>2.6</b>	<b>Connecting XML: XLink.....</b>	<b>62</b>
<b>2.7</b>	<b>XML Standards: Conclusion.....</b>	<b>65</b>
<b>Part 3:</b>	<b><i>Electronic Business and ebXML.....</i></b>	<b>67</b>
<b>3.1</b>	<b>Introduction .....</b>	<b>68</b>
<b>3.2</b>	<b>XML Application Areas.....</b>	<b>70</b>
3.2.1	A double dichotomy for classification .....	70
3.2.2	The resulting Four-Quadrant Scheme .....	71
3.2.3	Some further observations about the scheme .....	74
<b>3.3</b>	<b>Why (not) EDI ? .....</b>	<b>77</b>
3.3.1	What is EDI ? .....	77
3.3.2	What’s wrong with EDI ?.....	78

3.3.3	Lessons to be learned from EDI.....	81
3.3.4	Down to earth: a well-balanced attitude.....	82
<b>3.4</b>	<b>A Generic B2B Framework.....</b>	<b>84</b>
3.4.1	Introduction .....	84
3.4.2	An example: The eCo Framework .....	85
3.4.3	Building blocks.....	87
3.4.3.1	Transport .....	87
3.4.3.2	Registry .....	88
3.4.3.3	Profile Description .....	88
3.4.3.4	Document Definition.....	89
3.4.3.5	Additional Functions and/or Blocks.....	89
3.4.4	An additional view on electronic business .....	89
<b>3.5</b>	<b>ebXML: A “Complete” Framework.....</b>	<b>92</b>
3.5.1	Introduction .....	92
3.5.2	ebXML Requirements and Problem Domain.....	95
3.5.2.1	Requirements.....	95
3.5.2.2	Problem Domain: Business-to-Business Collaboration .....	96
3.5.2.3	B2C and EAI .....	97
3.5.3	ebXML Architecture .....	98
3.5.3.1	Background: Open-edi and Unified Process .....	98
3.5.3.2	The Open-edi Reference Model .....	100
3.5.3.3	UN/CEFACT Modelling Methodology (UMM).....	101
3.5.3.4	ebXML Business Operational View (BOV) .....	103
3.5.3.5	ebXML Functional Service View (FSV).....	106
3.5.3.6	ebXML Architecture: Conclusion .....	109
3.5.4	The ebXML Specifications .....	110
3.5.5	Business Process and Core Components.....	112
3.5.5.1	Introduction .....	112
3.5.5.2	Business Process Specification Schema.....	114
3.5.5.2.1	Introduction and Overview.....	115
3.5.5.2.2	Business Collaborations .....	116
3.5.5.2.3	Choreography .....	117

3.5.5.2.4	Business Transactions .....	117
3.5.5.2.5	Business Document Flows .....	118
3.5.5.2.6	Business Signals .....	118
3.5.5.2.7	Patterns .....	120
3.5.5.2.8	Structure Example .....	120
3.5.5.3	Naming Conventions for Core Components and Business Processes 121	
3.5.5.4	ebXML Methodology for the Discovery and Analysis of Core Components.....	122
3.5.5.5	The role of context in the reusability of Core Components and Business Processes .....	124
3.5.5.6	Specification for the Application of XML-based Assembly and Context Rules .....	125
3.5.6	Transport, Routing and Packaging .....	127
3.5.6.1	Introduction .....	127
3.5.6.2	Overview of the Message Service .....	128
3.5.6.3	Simple Object Access Protocol (SOAP) .....	130
3.5.6.3.1	Introduction .....	130
3.5.6.3.2	Limitations of Traditional RPC.....	132
3.5.6.3.3	The SOAP Mechanism.....	134
3.5.6.3.3.1	Parts of the SOAP Specification .....	134
3.5.6.3.3.2	Message Structure: SOAP Envelope Contained in a HTTP message	135
3.5.6.3.3.3	Serialization Mechanism: Encoding Rules.....	137
3.5.6.3.3.4	SOAP Transport Architecture and Mechanism.....	138
3.5.6.3.4	SOAP Messages with Attachments .....	139
3.5.6.3.5	SOAP: Summary and Perspectives .....	141
3.5.6.4	Packaging and SOAP Extensions.....	142
3.5.6.4.1	ebXML Packaging.....	142
3.5.6.4.2	ebXML SOAP Header Extensions .....	144
3.5.6.4.3	ebXML SOAP Body Extensions .....	146
3.5.6.5	Reliable Messaging .....	147
3.5.6.6	Security.....	148

3.5.6.7	Error Handling.....	149
3.5.7	Collaboration Protocol Profile and Agreement (CPP/CPA) .....	150
3.5.7.1	Introduction .....	150
3.5.7.2	Description of the Collaboration Protocol Profile.....	152
3.5.7.2.1	Overview .....	152
3.5.7.2.2	ProcessSpecification.....	154
3.5.7.2.3	DeliveryChannel.....	154
3.5.7.2.4	DocExchange.....	155
3.5.7.2.5	Transport .....	155
3.5.7.3	Composing a Collaboration Profile Agreement (CPA).....	155
3.5.8	Registry/Repository.....	158
3.5.8.1	Introduction .....	158
3.5.8.2	UDDI, A Complement to the ebXML Registry .....	159
3.5.8.2.1	What is UDDI?.....	159
3.5.8.2.2	Registration of Service Types, Identification and Taxonomy Systems	161
3.5.8.2.3	Registration of Businesses and the Services they provide .....	164
3.5.8.2.4	Discovery of Businesses and Services .....	167
3.5.8.2.5	Summary overview and Perspectives.....	168
3.5.8.3	The ebXML Registry/Repository .....	170
3.5.8.4	Registry Information Model (RIM).....	172
3.5.8.4.1	RegistryEntry.....	174
3.5.8.4.2	Association, Classification and ClassificationNode.....	175
3.5.8.4.3	AuditableEvent (and Use, Organization and PostalAddress).....	176
3.5.8.4.4	Other Objects.....	176
3.5.8.5	Registry Services Specification.....	177
3.5.8.5.1	Browse and Drill Down Query.....	179
3.5.8.5.2	Filtered Query.....	179
3.5.8.5.3	Content Request Query .....	181
3.5.8.5.4	SQL Query .....	182
3.5.8.6	Further Issues .....	182
3.5.9	Marketing/Awareness and Education.....	183
3.5.10	ebXML: Conclusion .....	183



## **General Introduction**

Conceptually, this thesis is built around two bodies of specifications: on one hand, the family of *XML standards*, published by the W3C, and on the other hand, the *ebXML specifications*, which provide a complete framework for setting up business-to-business collaborations.

Discussing B2B eCommerce in general and the ebXML framework in particular is the ultimate goal of this thesis. Although for the most part this discussion requires no detailed knowledge of XML, we will assume that the reader is familiar with its concepts. For this reason, in **Part 1** we start off with a “**General Overview of XML**”, either as a basic introduction or to refresh the reader’s understanding of the XML concepts. It does not intend to discuss the *whole* family of XML standards, but should serve as a warm-up for people who are primary interested in ebXML. Some background and an outline of the XML 1.0 specification are given, immediately followed by an example of an XML document. A glance at the possibilities of the other XML standards is also included, with a small *XSL* example.

To develop a full understanding of XML and its use in the ebXML framework, **Part 2: The Family of XML Standards** is nonetheless highly recommended, covering the various specifications of the “fundamental” XML standards. Much attention is devoted to the **design** of schemas for XML documents: the *requirements* for a modelling language, and the extent to which *DTDs* and *XML Schemas* meet these requirements. Next, we consider two alternative methods for gaining **access** to XML documents, *DOM* and *SAX*. The other main point of attention in Part 2 is **manipulation and querying** of XML data. After a short discussion of *XPath*, which provides us with an expression syntax to address specific parts of an XML document, we will focus on the *XSLT* transformation language. We will see that XSLT offers an extremely versatile method to extract the information from a document and use it for a wide range of purposes. Next, there is *XQuery*, which is more specifically aimed at *querying* XML data. It is however still very young, and therefore will be discussed quite succinctly. A look at *XLink*, a proposed standard for **linking** XML and non-XML sources together, concludes the second part.

The central purpose of this thesis, i.e. considering XML as a basis for business-to-business eCommerce frameworks, is addressed in *Part 3: Electronic Business and ebXML*. It is intended to provide the reader not only with background and overview information, but also with a more thorough discussion of the emerging ebXML framework. We first ask ourselves which **application areas** XML can play a role in, which is related to the question what kind of information can be marked up in XML syntax. One application area, namely *message exchange*, will appear to be our point of interest. Since avoiding past mistakes should always be a goal to keep in mind, we then turn to **EDI** for a minute, mainly because it provides several lessons to be learned. With these caveats in mind, the following step will be an attempt to express the characteristics and requirements for a **generic B2B framework**. Starting from the *eCo Architecture*, an existing reference framework, several “building blocks” will be identified. All this information will ultimately serve as a basis for understanding the brand new, end-to-end B2B eCommerce framework called **ebXML**. For this framework, we will not only address its *requirements* and *problem domain*, but also try to expose some of its origins, lying within the *Open-edi Reference Framework*. After this, the *ebXML Architecture* is discussed, which should give you a good overview of the constituent parts of ebXML. For the reader who is only interested in knowing what ebXML is all about, this may suffice. However, since *each* member of the ebXML assortment of specifications proves itself a worthy opponent to other standards or even frameworks, we believe a more comprehensive discussion of these documents is richly deserved. Therefore, we will address all ebXML specifications separately, per ebXML Project Team: *Business Process* and *Core Components*, *Transport, Routing and Packaging*, *Trading Partners* and *Registry/Repository*. Along the road, we will also give attention to two initiatives that are both related to ebXML and to the emerging *web services*, namely SOAP and UDDI.

Note: During the writing of this thesis, many specifications have kept evolving, with new versions being published on a regular basis. Therefore, we have chosen to devote a separate section in the list of sources to these specifications, where the most recent versions are mentioned, and not to include explicit references to these specifications in the text.

## Part 1: GENERAL OVERVIEW OF XML

## 1.1 Introduction

When one is writing a book or a dissertation concerning an Internet-related matter, trying to describe some of its characteristics, there's only one thing that is free of doubt: chances are small that it is still up-to-date by the time the intended reader gets his or her hands on it. "*The times they are a-changing*", Bob Dylan sang in the sixties and that is ever more true today, considering the speed at which the Internet is evolving. Technologies seem to be either innovative or outdated. Then again, why should one ever bother reading documents like this, or even worse, why should they ever be written?

However, there are a few additional observations to make. Firstly, though technology is changing at light speed and is exerting great influence on the way we do business, the latter tends to level off some of the former's extravagances. The frequently heard question whether XML is "*Hype or Hot*" is only one indication – rational IT decision makers do not adapt their strategy every time a new buzzword turns up. Yet it also means that attentively monitoring innovations is indispensable to see the right opportunities approaching, and to be ready for them when they arrive.

Related to the previous consideration, there is the good practice not to put your nose right on top of the stream of novelties. Keeping a certain distance when examining them gives you the opportunity to discover mutual relationships and the direction in which they are developing. It helps you to filter out the hypes, and even to *predict* certain changes. This will be the main approach for this thesis: instead of losing track in numerous technical aspects, it will give you a higher-level overview of XML-related standards and applications, whilst indicating certain practical issues where applicable.

So, although books and publications in general can hardly keep pace with the evolutions they describe, they still have substantial value in offering a broader context, together with some well-considered interpretations. Moreover, there is also a much more imperative reason why internet technology innovations need your attention : conducting business over the Web presents an enormous potential for cutting costs and yielding profits! Stated reversely, if your company's rivals can take advantage of a successful innovation but you are (too) late because you weren't prepared for it, you are at great risk. Time is not –no longer– on your side... *Reacting* to your opponents' moves always leaves you that one step behind – a crucial step, perhaps. Reacting is no longer an option if you want to stay in business: “*Act before your competitors do*”, is a catchphrase to keep in mind in your struggle to win. Indeed, “to win”, because in the e-Business world, there will be no mere survivors, only winners and losers. Victory or defeat, take over or be overtaken. Your competitive edge will be the essential key to your success, and the best way to gain it is through the use of innovative technologies.

Enter XML. Today, many IT executives feel rather uncomfortable when these letters are being pronounced. Either they don't have a clue what it is and are wondering what the big fuss is all about, or they already know that this “eXtensible Markup Language” is an emerging new standard, but are anxious to know what *they* can do with it. If I would have written this thesis one year ago, the “What is XML?” section would have been a lot more elaborate. In the upcoming pages this question will be addressed in more detail, but first let us try to give a rather intuitive definition.

First of all, XML is not a product, a software tool or some off-the-shelf component: you cannot walk into a store and buy yourself a box of XML. When developing applications, XML is more than just an extra feature. It is often said to be “an enabling technology”. Although that description holds some truth – it's indeed a technology – it is not entirely correct, as for most applications there *is* an alternative. Instead, **XML is the main catalyst of e-Business**, because it basically makes many “e-facets” easier and faster to develop. And it looks like it is here to stay...

One could expect that in a couple of years, after a lot of *talking about XML*, many of us will be *talking XML* when using the web – though it might be hidden from us. The

Internet is changing the world, and in the same way as the “World Wide Web” is now simply “the web”, today’s “e-Business” will be just “business” tomorrow. Don’t miss the transition!

## 1.2 The XML Language : A First Outline

Most people who start reading this thesis, probably already know *something* about XML, the *eXtensible Markup Language*. You may have heard about the *World Wide Web Consortium (W3C)* for instance, the standards body that has developed this new technology. Furthermore, considering all the media attention, it should not surprise you to hear that XML is a standard supported by most leading software vendors.

Although it is not the purpose here to describe the whole history of markup languages, the least one should know, is that the origin of XML is in the world of SGML, the *Standard Generalized Markup Language*. SGML is an ISO standard that defines an extremely powerful markup language, and for many years has been used in the publishing industry and large manufacturing companies. However, as we will discuss further on, its power comes at the price of complexity: the large efforts required to implement this 500 page specification mean that the benefits must be accordingly big, so despite of its many qualities, this standard has never enjoyed a wide-spread use and acceptance. Being a *meta language*, SGML is used to create other markup languages such as HTML.

This good old HTML, the *HyperText Markup Language*, can be seen as the other side of the spectrum : used by thousands of people all over the world, not in the least because of its simplicity. As mentioned above, the *tagset* of the HTML language is specified using SGML, which makes it an *application* of SGML. Nevertheless, the ease of use that characterizes this markup language, has a similar but opposite drawback as is the case with SGML, namely limited functionality. Despite numerous nifty web developers' inventions, HTML is in fact nothing but an *information access and display layer*: its *tags*, the identifiers put before and after text blocks, can only be used to describe the "look and feel" of the document. This way, a web browser can figure out how the presentation of the document should look like, but nothing more. HTML was never designed for tasks other than presenting information on a screen.

Now that it is clear what is good and bad about SGML and HTML, it would be nice to have a standard that provides “80% of the benefit of SGML with 20% of the effort”. It exists, and it’s called XML.

With XML, it is finally recognized that the information published on a website is usually more than text – it is *data*. Using a presentation-only markup language as HTML, it is quite obvious that much of the structure and meaning of this data is lost. To overcome this and other limitations, the W3C (World Wide Web Consortium) has developed XML. The goal was to create a language that was easy to learn and use, compatible with SGML, powerful enough to have a wide variety of applications and legible to both humans and computers. Adding up all these features, we can build the following definition :

*XML (eXtensible Markup Language) is a meta-markup language defined by the W3C, that provides a way to create extensible formats for describing structured data in electronic documents and to express rules about those data. It is a subset of SGML, optimised for delivery over the web, and its current version is 1.0 Second Edition.*

An important detail in this definition is “meta”, which means that it is a “language to create other languages”. Unlike HTML, which has a *fixed tagset*, XML has no tagset of its own, but is used to define new tagsets and data structures, called *XML applications*<sup>1</sup>. In a certain sense, there’s no such thing as an ‘XML document’ – all the documents that use XML-compliant syntax are really using applications of XML. Indeed, this also means that HTML can be described as an XML-application, which *has* been done in the **XHTML** specification, a reformulation of HTML 4.0 in XML 1.0.

It should be noted that (X)HTML is only one *type* of application. So far, only matters related to web publishing have been mentioned, where the fact that XML separates content from style is essential. However, when the XML-designers had a “wide variety

---

<sup>1</sup> The word “application” has two totally different meanings in SGML- and XML-related matters: 1) a tagset, a markup language defined with XML, 2) the general word for a computer program. To distinguish, “application” or “XML application” shall be used for the first meaning, while “application program” and “computer application” shall be used for the second.

of applications” in mind, they were not only thinking of humans interacting with a web server through a browser. At least as important and exciting is communication between computers, possibly located in different companies and organisations. To give an equally famous counterpart of HTML in this *inter-application context*, there’s *EDI* (Electronic Data Interchange), a standard format for exchanging business data. EDI, which will be discussed more in detail below, has aimed to be a universal language for computer-to-computer communication. Pretty much like SGML however, it has also failed the complexity test, and therefore it is soon to be replaced by XML-based business-to-business (B2B) standards.

The other features of the XML language will be dealt with shortly, but at this point one more aspect of the definition is worth mentioning, namely “electronic documents”. While browsing the web can still be seen as reading documents, in the context of communication between computers it is more appropriate to talk about “data structures”. Since the ability of something like “common sense” is still rare to computers, a document like this, even nicely divided into paragraphs, has little meaning. To have application programs talk to each other, they need a way to know how the data they receive is structured, and what the vocabulary means. As we will see, this is where XML can be of assistance.

## 1.3 Know what you're talking about:

### An Example

The question to be addressed here is: “What does an XML document look like?”. Notwithstanding the fact that this thesis is aimed at a non-developer public, it is often of great value to be able to visualize some of the concepts that are discussed. Besides, you will certainly notice that there is not that much to an “XML document”.

This will be done through a “mini-case-study”, based on a simple scenario of the XML-adventures of the imaginary Charlie, a store-owner. The case-study contains some example code and data, of which the purpose is *not* that you achieve full understanding of how they work – which also means that not everything will be explained here. Rather, you should get an intuitive insight into the matter we are discussing. These examples will then be gradually extended throughout the upcoming chapters.

Let's say Charlie is the owner of a store where books and postcards are sold. Unfortunately business has been slow lately and Charlie, very eager to read and learn, is probably the most regular customer of his own store. What strikes him time and again, is the beautiful, concise and powerful way some writers express their thoughts and opinions. Over time, Charlie has collected a huge number of these quotes and proverbs, in different languages, mostly accompanied by detailed book and author information.

Every once in a while, Charlie also has a look at the non-fiction section, and a tiny divine intervention is enough to let him get his hands on a book titled “XML – Everything you need to know”. He starts reading, and bit by bit the idea of an online version of his quotes collection, a “Famous Quotes Website” is formed.

One of the nice things about XML is that its level of entry is very low: it is not necessary to have a complete understanding of the whole technology to produce your first XML document. So, after a week or so, Charlie already has a preliminary version of his XML-marked-up quotes collection:

```

1. <?xml version="1.0"?>
2. <FamousQuotes>
3. <Quote Type="Quote" Category="Life" UsedLanguage="English"
   OriginalLanguage="English">
4.   <Content>Minds are like parachutes. They only function when
      they are open.</Content>
5.   <Author>
6.     <Name>Sir James Dewar</Name>
7.     <YearOfBirth>1877</YearOfBirth>
8.     <YearOfDeath>1925</YearOfDeath>
9.   </Author>
10. </Quote>
11. <Quote Type="Quote" Category="Life" UsedLanguage="English"
    OriginalLanguage="English?">
12.   <Content>Those are my principles. If you don't like them I have
      others.</Content>
13.   <Author>
14.     <Name>Groucho Marx</Name>
15.     <YearOfBirth>1890</YearOfBirth>
16.     <YearOfDeath>1977</YearOfDeath>
17.   </Author>
18. </Quote>
19. </FamousQuotes>

```

**Example 1 : Preliminary version of a FamousQuotes XML Document**

Note: For this and all further examples, the line numbers are *not* part of the document. Also, the indentation is only for “pretty-printing”, emphasizing the hierarchical structure.

What you see here, is a fully legitimate instance of an XML document. Simply by marking up the quotes collection in this way, we have created our own “Quotes Markup Language”, but more about this later. As you can see, if you’re backed by a little bit of common sense, it’s even easier to read this example than it has been to write it. On the other hand, some things might benefit from a short explanation, and meanwhile we can identify some basic XML “rules”.

Line 1 is probably the most obscure, although there is not much to it: it simply identifies your document as being marked up in XML, version 1.0.

The basic building blocks of an XML document are *elements*, often called the “nouns” of XML. You could compare them to the column headers of a table: they label, and thereby describe, the information they contain. Elements are delimited by *start- and end-tags*: start-tags begin with < and close with >. End-tags begin with </ and close with >. On line 2, you see the <FamousQuotes> start-tag, bringing about the first rule: there

is always exactly one *document element*, encompassing *all* other document content – apart from the first line and some special items, that is. This means that this element begins at line 2 and ends at line 19, delimited by `<FamousQuotes>` and `</FamousQuotes>`.

Looking at lines 3 till 18, there’s a hierarchy of elements, each named and marked by tags. A different, perhaps more appealing way of presenting this hierarchy is the familiar tree form:



**Figure 1: Tree structure of the preliminary FamousQuotes XML document.**

This brings us to the next important issue: all elements must be *properly nested*. Each element *must* have a start- and end-tag, and the start- and end-tag of a parent element at every level of the hierarchy must contain both the start- and end-tag of all its child elements. This may seem a trivial issue, but violation of this rule is in fact one of the major trouble-makers in the HTML world. If these two conditions are fulfilled (a single root element and proper nesting), the document is said to be *well-formed*. This is the *conditio sine qua non* for an XML document: if you’re not talking in a well-formed manner, you might as well say nothing.

So far, you should be able to understand the better part of the example, except for the two `<Quote ...>` tags: these contain several *name = "value"* pairs, which designate *attributes*. If elements can be thought of as the “nouns” of XML, attributes would be the “adjectives” [Birbeck et al., 2000, p43]. They are meant to contain “additional information” about properties of the element’s content (between the tags). The (heated) discussion about what that “additional information” is supposed to be, i.e. when to use attributes versus elements would lead us too far in this introductory chapter.

The above example contains only “normal” elements, which are allowed and supposed to have a certain content. In addition, there exists a special kind called *empty elements*, which are *not* allowed to have any content. Because there is little use to having two separate tags for these elements, they have a different markup of the form `<element-name />`. Attributes, if any, have to be placed between the element-name and the `/`. Empty elements can be seen as *boolean values* (either they are present or not), or their attributes can contain information.

The set of rules described above is obviously not complete. It allows you to create simple XML documents, without paying attention to issues like character encoding, references and relationships between elements, validation, data typing,... Some of these issues are strictly technical, and their use depends on the type of application on hand. But for now, Charlie’s goal of making a website can be easily reached with the topics covered so far... Can it ?

## 1.4 Family Matters

As the XML-picture is getting a bit clearer, a reasonable consideration at this point would be: “Alright, now the data is all marked up, what can be done with it?!” Indeed, the XML specification on its own isn’t much of a revolution. Charlie could just as easily have put everything in a nice and simple MS Access database – though this comparison wouldn’t really hold for more complex data. The only good reason for storing data in a non-trivial format is the ability to get it back out easily the way(s) you want, possibly after selecting a qualifying subset. In this case, the reader might argue that there isn’t much point in sending the XML document as-is to a web browser.

But then we have **XSL, the *eXtensible Stylesheet Language***. While in the marketing pep talk, it’s all about big mighty XML solving all problems, in reality it is XSL doing most of the work. As mentioned before, one of the great things about XML (compared to HTML) is that it separates data from presentation, allowing different output formats to be generated from one instance of XML. This generation is a job for XSL: by applying different *stylesheets* to your marked-up data, the result can vary greatly: differently sorted or structured XML, HTML, WML (*Wireless Markup Language*<sup>2</sup>) are only a few possibilities. For most applications, these two specifications are inseparable.

Again, an example is in place here. Suppose Charlie wants to display his collection on a webpage, as HTML, but unfortunately he suddenly realizes that it would be nice to have the quotes alphabetically listed *per author*. The following code sample shows how to do this in XSL.

---

<sup>2</sup> WML is used in WAP, the *Wireless Application Protocol* for mobile communication.

```

<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/TR/WD-
xsl">

<xsl:template match="/">
  <html><body>
    <xsl:apply-templates select="FamousQuotes"/>
  </body></html>
</xsl:template>

<xsl:template match="FamousQuotes">
  <table border="1">
    <tr><th>Author</th><th>Quote</th></tr>
    <xsl:for-each select="Quote">
      <xsl:sort select="./Name"/>
      <tr>
        <td><b><xsl:value-of select="Author/Name"/></b></td>
        <td><xsl:value-of select="Content"/></td>
      </tr>
    </xsl:for-each>
  </table>
</xsl:template>

</xsl:stylesheet>

```

**Example 2 : A sample XSL Stylesheet**

An explanation of each tag would lead us too far for now, of importance here are the two bold-printed lines: the first one makes sure that every <Quote> is processed, the second that they are sorted per author. Together, they re-organize the data the way we want it, enabling us to “see” the XML file not as a collection of quotes with associated authors, but as a series of authors, each with one or more citations. As will become clear throughout this thesis, these sorting and restructuring capabilities provided by XSL will play an important role when connecting heterogeneous environments in a B2B context.

When taking a look at the XML file through the eyes of this stylesheet, an output similar to the following would appear in an XML-capable web browser:

Author	Quote
<b>Groucho Marx</b>	Those are my principles. If you don't like them I have others.
<b>Sir James Dewar</b>	Minds are like parachutes. They only function when they are open.

**Figure 2: The XML example after applying the stylesheet**

An important feature of XSL stylesheets is that they are marked up in XML themselves – in contrast to, for instance, Cascaded Style Sheets. This will come in quite handy when we get to manipulating the stylesheet at runtime, because it can be accessed using the same techniques as for ordinary XML documents. For a matter of fact, as you might have spotted, there are several examples of *empty elements* here, such as `<xsl:apply-templates .../>`.

The XSL example introduces another seemingly natural, but important extension to the initial XML Specification: *Namespaces*. A *namespace* can be described as web address that identifies the names of particular data elements or attributes used within an XML file, by this means indicating that it belongs to a particular domain or application. The attentive reader will no doubt have noticed that there are two kinds of tags in the XSL example, i.e. the “normal” tags (which will appear in the output result), and those starting with “xsl:...” (which can be seen as instructions for how to process the XML file). Quite obviously, the “xsl:” prefix makes it possible to draw the distinction between both types. Elements with that prefix are said to *belong to* or *be in the XSL namespace*, which is identified by the address specified in the xsl:stylesheet tag: `<... xmlns:xsl="http://www.w3.org/TR/WD-xsl">`.

As it has been stated above, we will not spin the XSL specification here any further here. The above examples should have given you an impression of what XML technology is all about, emphasizing the fact that it is the cooperation of several standards that makes XML so powerful. Some of these standards that have remained unmentioned so far are:

- *DTDs (Document Type Definition)* and *XML Schemas*, for defining data structures and types;
- *DOM (Document Object Model)* and *SAX (Simple API for XML)*, for providing ways to access and process XML documents;
- *XPath* and *XPointer* for traversing documents and pointing to specific parts of them.

As each of these standards is doing its own job, together they provide the key ingredients for almost any XML application. Next time you hear a product vendor saying his product supports XML because it can accept or produce tagged up data, you should wonder if the vendor is welcoming XML as a newborn son or stepmotherly treating it as a foster child...

## 1.5 Conclusion

The purpose of this introductory part was to give the reader a look at the XML Specification, and at some of the surrounding core technologies. While some of these technologies will be examined more closely in the next part, it is important to have in mind the big picture of several specifications working together. The XML 1.0 Specification on its own was an important first step, but it provides only the basis for what is generally known as “XML”.

While many qualities of the XML language –and its relatives– have not been mentioned yet, one aspect deserves special attention: XML enables us to express our data in a structured fashion, oriented around our perspective of the nature of the information itself rather than the nature of an application’s choice of how to represent our information. This does *not* mean that XML provides a simple answer to all our problems, but as you will notice, it gives us the freedom and flexibility we will need to be a pro-active, dynamic player in the information economy.

## Part 2: THE FAMILY OF XML STANDARDS

## 2.1 Introduction

The introductory section has given a first impression of what XML looks like. In this part, an overview of several XML-based or -related standards will be given. You will not be overwhelmed with technical information on *how* exactly everything works. Rather, we will focus on *what* role each standard is playing, together with its importance and relevance for B2B communication.

Some comment on the way the standards to be discussed have come into the world might be useful. As has already been briefly mentioned, the XML Specification is written by members of the W3C (World Wide Web Consortium), which has important implications with respect to the spreading and adoption of XML as a standard. This organisation describes itself as follows:

The W3C is an industry consortium which seeks to promote standards for the evolution of the Web and interoperability between WWW products by producing specifications and reference software. Although W3C is funded by industrial members, it is vendor-neutral, and its products are freely available to all.

After HTML, the major achievement of the W3C can be considered to be XML, as a core standard that drives the better part of the W3C activity today.

The way the W3C operates is through *Working Groups* that produce specification documents in accordance with a *Recommendation Track*, from *Working Draft* to *Recommendation*. (See the W3C website for descriptions of these “levels of maturity”.) While the XML 1.0 Specification can be considered as “finished” (i.e. in Recommendation phase), some other standards like XML Query and XLink are still under development. Although implementation is encouraged for the developers community to gain experience, investments and developments grounding on these standards should be done with caution.

## 2.2 The XML Specification

What has been said about the XML Specification so far can be roughly recapitulated in one word, “well-formedness”: a document must have one root element and the elements must be properly nested. That’s about the deepest we will go here. Since other aspects like design, access and transformation of XML documents are far more interesting, further digging into the details of the XML Specification seems unnecessary.

One side issue that might be worth noting here is *Internationalisation*. As the web is living up to its “world-wide” reputation, taking care of problems like support for non-European languages becomes a necessity. If we start dreaming of arbitrary applications all over the world communicating with each other in a non-predetermined manner, a basic prerequisite is that the format is capable of adequately representing all possible characters. As the W3C website [www.w3.org] states, “*The general goal of W3C's work on internationalization is to ensure that W3C's formats and protocols are usable worldwide in all languages and in all writing systems.*”. For this reason, XML documents are by default assumed to use *Unicode* encoding, either the 8- or the 16-bits version. In addition, as it is often not enough to correctly *encode* characters but also to *render* them (e.g. to apply the correct direction of writing), it is possible to specify the language used on a per-element basis.

Note: About XML’s simplicity, we want to add a small remark. Originally, XML was *not* designed for information exchange. As we have seen, it was meant as a “light-weight SGML for the web”, to be used as a presentation mechanism. However, you might ask yourself if this is more than just a “curiosity”, used by people who want to show off their “deep knowledge” of XML: If the purpose would have been to create a “universal data format”, then would the result *ever* have been so flexible and easy to implement as it is now? If your answer would be “no”, some serious questions arise concerning the apparent *complexity* of recent standards such as XML Schema...

## 2.3 XML and Design: Setting things up

As from now, it is important to see an XML Document not in the first place as a marked-up text, but rather as a tree-structured collection of data – though the second viewpoint does not exclude the first, of course. As soon as you are thinking of data structures, you will quickly recognize the need of information modelling in order to formalize the syntactic and, as far as possible, the semantic rules applicable to those structures. Without a model, there is no information, only data. In terms of XML, this means that we will have to specify a *vocabulary* that describes the elements and their internal structure, including attributes. In fact, every time you mark up a new kind of data in XML (as we have done for the Famous Quotes), you are implicitly defining a new vocabulary, but now we need a way to cast this in a formal mould.

Note: As you will notice, these design issues will be covered quite detailed – a lot more thorough than the other specifications, that is. The main reason for this is that, in all the hype surrounding XML, some people tend to “forget” they are working with data. XML’s simplicity does *not* mean that careful modelling has become redundant in avoiding the many pitfalls!

### 2.3.1 Basic Modelling Requirements

As in most modelling situations, the extent to which an XML vocabulary is representative of the problem at hand greatly determines the ease at which applications based on this vocabulary can be built and maintained. We want to stress the importance of the modelling phase, by departing from its primary goals and asking the question what gains we expect from building a formal model. As you will notice, this will immediately bring up several expectations for the *schema language* used to formalize the model.

Note: Again, there is a potential source of confusion here: A *schema* (with the first letter in lower-case), is a general name for a document describing data structures and their contents, usually in a database context. An *XML Schema* (with the first letter capitalized) is, as we will see, a particular kind of schema to describe an XML document type.

Before we start digging into the particularities of XML modelling itself, we should take a closer look at why we want to do information modelling in the first place. Although this design step is usually the toughest one, in the end it should make our life easier.

### **2.3.1.1 Documentation and Explanation**

From the database and programming world we know that *documenting* your code or data structures is imperative. Many things resulting from the problem analysis, while obvious at modelling time, later turn into big questions for someone who was not involved from the beginning. The enormous number of legacy systems unfortunately only confirms the underestimation of this activity. Another benefit that arises from proper modelling and documenting, is that it forces designers to thoroughly think everything through, reducing the probability of errors and points being overlooked. In short, a good schema should provide a precise and unambiguous documentation of the business domain model.

Not only developers can benefit from the meta-information contained in the model. A nice but very seldom seen feature could be that this meta-information is made available as *explanations of data fields* to end-users, which would de facto turn our schema into a *data dictionary* [Birbeck et al., 2000, p133]. However, since end-users are not supposed to understand the schema language itself, this brings up the requirement on the language syntax to be easily application-readable!

In a B2B context, another important purpose of a schema is being an instrument to communicate the structure of our documents to business partners. This puts even bigger pressure on their readability, both to humans (other companies' developers) and computers. The ideal of an information model being truly self-explanatory should be attained as close as possible.

### **2.3.1.2 Definition of Constraints**

An obvious expectation from a modelling language is that it allows us to put *constraints* on things like structures and data types. From the XML perspective, this means in the first place that we need control over the way elements are *nested* within each other.

When a particular element occurrence is different from the next, there should be a way to tell if this is a mistake or just an element with different characteristics.

Secondly, we want to be able to specify the data type of *element content*. Despite the fact that an XML document is text-based, we would like to indicate the difference between a numeric value, a date and a string of text.

Thirdly, keeping in mind the duality of attributes and elements, the same data typing need exists for *attribute values*. Although these conditions seem elementary for a technology that claims to be a “Universal Data Format”, we will soon discover that they have only recently been fulfilled.

### 2.3.1.3 Validation

So far, we have only considered well-formed documents. As mentioned before, well-formedness is a necessary but not sufficient condition for document integrity. Consequently, a requirement that is interrelated with the previous one, is the ability to *validate* document instances against the model they are based on. There are two aspects to this: when data is first keyed into a system, the data entry program can force to obey the rules described in the schema. Likewise, when a document comes from an external source, like a business partner, a *parser*<sup>3</sup> can compare it to the schema upon arrival, and reject it if not conformant. In short, document *producers* validate to ensure they are providing what they have promised, and document *consumers* do this to “protect” themselves from producers’ mistakes. As many structural and content errors can be detected by this parser instead of by your application logic, this can greatly leverage the efforts you have put in your schema. Moreover, it is a good habit to check for errors as close to their origin as possible.

In theory, not all parsers *have* to provide for validation. In practice however, especially in an open and unpredictable environment as the Internet, data coming from external sources (e.g. trading partners) will *always* pass through a *validating parser*. Since you

---

<sup>3</sup> An XML *parser* or *processor* is a program that is used to access XML documents, using the DOM or SAX methods described below.

are not in control of the document creation process, you cannot assume that incoming data will comply to any standard! On the other hand, validation can be confined to the test phase in case the document source is internal to your company, especially since the impact on parsing time is far from negligible.

## 2.3.2 DTDs: A preliminary answer

Rome wasn't built in one day, so when the W3C XML Working Group brought out the XML 1.0 Specification, they spared themselves the trouble of creating a new schema language. The modelling mechanism provided as part of the specification was simply borrowed from SGML: the *Document Type Definition (DTD)*. Now that we have a pretty good idea of what we are looking for in a basic schema modelling language, the question is whether DTDs meet these requirements. As you might already suspect, they don't do a very good job. Therefore, we will only take a brief look at these issues.

### 2.3.2.1 Documentation and Explanation

When XML is said to be a meta-language, one is actually referring to the DTDs! DTDs are a means for developers to define the content of an XML document, so they sure provide us with a fair amount of meta-information. However, things are not always what they seem to be, as you can see from the following example DTD for the Quotes Collection:

```
<!ELEMENT FamousQuotes (Quote*)>
<!ELEMENT Quote (Author, Content)>
<!ATTLIST Quote
  Type (quote | proverb) #REQUIRED
  Category CDATA #REQUIRED
>
<!ELEMENT Author (Name, YearOfBirth, YearOfDeath?)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT YearOfBirth (#PCDATA)>
<!ELEMENT YearOfDeath (#PCDATA)>
<!ELEMENT Content (Specimen+)>
<!ELEMENT Specimen (#PCDATA)>
<!ATTLIST Specimen
  Language CDATA #REQUIRED
  Original (True | False | Unknown) #REQUIRED
>
```

**Example 3 : The DTD for the FamousQuotes Collection**

Now who said that XML was easy? Indeed, while the sample XML document instance was fairly intuitive to understand, this (simple!) DTD is a lot more cryptic. When you put the DTD and its instance side by side, you can probably figure out most of what the DTD says, but still... This brings us to a first drawback: not only are DTDs hard to write, but also they are far from easy to understand. This makes work more difficult for their designers, increasing their error-proneness. Moreover, it promotes the formation of *legacy schemas*: already, there are organisations where no-one dares to change certain (XML or SGML) DTDs any more, because they have become too complicated to understand!

Part of the DTD's lack of comprehensibility is actually caused by its syntax (namely *Extended Backus Naur Form, EBNF*). As *application-readability* is also an issue, this means that XML tools have to be capable of reading this syntax and presenting the DTD contents to a user or program. In theory this is possible, but in practice few or no tools have this option. Things would be a lot easier if an XML-based syntax was used, because then a standard XML parser would be enough to investigate the structure of the schema (using the *DOM*, the *Document Object Model*, see further). In a sense, we are striving to make the data model a similar vehicle to what the *system tables* are for an SQL database!

### 2.3.2.2 Definition of Constraints

The DTD syntax being somewhat complicated doesn't imply that we should simply do away with it: complexity might be counterbalanced by functionality. DTDs do a reasonably good job on defining *nesting constraints*, that's one thing. Next, what options do we have to specify the *data* type of element or attribute content? Well, ultimately, all of our content turns up as text (*PCDATA* or *CDATA*, for (*Parsible*) *Character Data*)... For example, both `Name` and `YearOfBirth` in the above DTD are denoted as `#PCDATA`. With DTDs, we have *no possibility at all* to differentiate between different primary data types (text strings, real or integer numbers,...). This can certainly be seen as one of its biggest shortcomings!

### 2.3.2.3 Validation

Obviously, if the answer of DTDs to the previous requirement is unsatisfactory, problems will turn up with validation as well. If it is impossible to pose stringent constraints on your model, you don't even have to consider validating them. It is not surprising that [Birbeck et al., 2000, p134] states: “...*I have often found that DTD validation adds so little value, it isn't worth bothering with.*”

Of course, it is still possible to apply the structural constraints, such as the presence of required elements, their nesting,... But then again, there are a couple of snakes in the grass. First, the document decides whether it is going to reference a DTD or not, and *which* DTD it is going to submit itself to. Secondly, a document is allowed to have an *internal DTD subset*, that *overrides* any external DTD. This means, indeed, that the actual constraints are decided by the document, and not by the application that wants to use the data!

### 2.3.2.4 DTDs: Conclusion

The picture we have sketched of DTDs might have raised the question why we even bother to look at them. It is quite clear that they do not fulfil the requirements for B2B communication *at all*. The reason for not leaving them out, is simple: at the time of writing, they are still the only official standard! Replacement is on the way with the *XML Schema Specification* (which has become a W3C *Candidate Recommendation* on October 24<sup>th</sup>, 2000), but at the moment DTDs are the only sure ground we have.

However, because none of the newer standards still rely on DTDs, they are expected to die a gentle death... Therefore, the technical details of DTDs have been barely covered here. Instead, we will see that the more recent XML Schema language greatly outperforms this outdated DTD standard.

Note: Things have been somewhat simplified in this section. In fact, there are some special language elements (such as *ID*, *IDREF*, *NMTOKEN* and *NOTATION*) and constructs (like *parameter entities*, *include/import*) that have special functions. Nevertheless, they do not solve the basic problem of data typing and/or are limited

and inflexible to use. Consequently, for the sake of clarity and brevity, it was chosen not to discuss these topics.

### **2.3.3 Advanced Modelling Requirements**

Having a look at the modelling aids we have found so far, the picture doesn't look very good. Being a fundamental part of the XML 1.0 Specification, DTDs served well as a first start, but they even lack some basic features like data typing. Despite of its undeniable value to early XML developments, there are many things you *can't* do with a DTD, and even more that could be done more *flexibly*. Fortunately, in a continuously evolving world as the Internet, it is not surprising that new standards are on their way to make up for these shortcomings and allow us to have a look at more advanced requirements.

#### **2.3.3.1 Mixing Vocabularies**

In this thesis we are looking at XML from a business point of view: how can it facilitate or enable B2B communication. But B2B means trading partners, and trading partners mean interdependencies and common standards. In some cases, we will have to combine parts of XML documents, based on both internal and external schemas, into one compound document. For example, you want to adhere to an industry standard schema (see further) for external use, while you prefer to use a superset of this standard internally. Or, a particular application may draw definitions from several related but distinct vocabularies, requiring these definitions to be mixed. A third possibility is that you are confronted with an existing body of schemas, and changing them could affect applications using it.

A good schema language should offer a mechanism to use definitions stemming from different schemas in one document. In DTDs, there is something called *parameter entities* that are used for referencing and/or modifying commonly used constructs. Unfortunately, they are far from flexible to use, and therefore we will not discuss them.

### 2.3.3.2 Modularization and Reuse

Beside merging several existing schemas, you may decide to *split up* a schema when it would become too large or complex. This *modular* approach (minimizing external coherence while maximizing it internally), is familiar from the programming world: it is usually a good idea to break up a large body of code or definitions into logical units, rendering your model into a better representation of the problem domain. Not surprisingly, we do not have this option either when working with DTDs.

Breaking up your schema into parts has another benefit: with strongly coherent chunks of schema definitions, they can often be *reused* as building blocks for your models. It is not unusual to find common constructs appearing during the modelling process, and putting them into isolated schema “libraries” can save a lot of work. Ideally, you end up with a number of schemas describing your business domain, which can be stitched together by reference to meet the needs of actual applications [Birbeck et al., 2000, p 92]. This is an important topic that will cross our path still more than once.

### 2.3.3.3 Inheritance

In the spirit of the object-oriented paradigm, no technology concerned with data structures can call itself mature without the concept of *inheritance* being addressed. Indeed, the power and flexibility offered by a subclassing mechanism that enables us to, say, defining a *Quote* and a *Proverb* as special kinds of a *Statement*, is substantial. While we didn’t dare to even think of this requirement with DTDs, we will see that it is successfully dealt with in the XML Schema Language.

## 2.3.4 XML Namespaces: Know where you belong

One might argue that the *parameter entities* available with DTDs roughly fulfil several advanced requirements, but their use is far from convenient. Moreover, they leave one element totally untouched: *name collision*. As soon as we start using element definitions from different, possibly external sources, we risk having semantically different elements having the same name. For example, a *Quote* can be a type of statement, but also a figure indicating the price of a stock. Fortunately, this problem has been solved through the concept of *namespaces*.

In general, namespaces are a mechanism for uniquely qualifying names for objects, by adding either pre- or postfixes. The most famous use of namespaces is probably the *Internet Domain Name system*: the .com, .net, .org,... suffixes in fact designate distinct namespaces. As an example, *soap.com*, *soap.org*, and *soap.net* each point to different websites – though none of them is about SOAP as an XML-based standard, that’s the drawback of choosing a “popular” name for a standard...

The same mechanism can be applied to XML. As an abstract of the W3C *XML specification* states:

XML namespaces provide a simple method for qualifying element and attribute names used in Extensible Markup Language documents by associating them with namespaces identified by URI references.

(...)

[Definition:] An **XML namespace** is a collection of names, identified by a URI reference [RFC2396], which are used in XML documents as element types and attribute names.

Most often, the reference will point to some kind of schema, although this is not essential: the key aspect here is defining *prefix aliases* for these namespaces, and employing them to discriminate between elements belonging to different problem areas. As the prefix is an alias for the namespace, the namespace is identified by a URI, and URIs are *unique* by definition, so will be all element names that contain such a prefix. Thus, namespaces allow you to use element or attribute definitions from several information models in one and the same document. At one stroke, all the element mixing, reuse and modularisation problems are solved in an elegant fashion.

Note: Simply stated, a *URI (Universal Resource Identifier)* is just a unique name for *some* resource. It *can* be a *URL (Locator)*, which locates the resource in terms of an access protocol and network location. It can also be a *URN (Names)*, a universal name that has to be mapped to the resource’s location in a separate processing step.  
[Maler, 2001b]

A quick (partial) example will be enlightening here and in the upcoming sections, so let's add an extra dimension to the "Charlie mini-case study".

Let's say Charlie has developed a schema for his quotes collection, placed online under `http://www.charlie.com/schemas/quoteschema.xsd`. But suppose he suddenly realizes that instead of simply undergoing the drop in sales caused by online bookstores and alike, he could turn his quotes collection into a valuable asset. By putting the quotes online, he might attract the attention to his cosy little store again. Moreover, another idea comes to his mind: why not allow people to order custom-made postcards? Charlie knows that his postcard supplier offers him a service to have custom cards printed, but so far it has been of little use since few people come to the shop to *order* a postcard. As an extra assumption, let's say the supplier also has got a schema for how custom postcards should be ordered, available online at `http://www.thepostcardcompany.com/schemas/customcardschema.xsd`.

For now, let's stick to the question how Charlie can use a namespace to differentiate between the elements and attributes defined in his own schema (`quoteschema.xsd`) and those that come from the supplier's schema (`customcardschema.xsd`), within a single XML document. Two things must be done: first, the namespace must be *declared*, while at the same time an *alias* is associated with it. To do this, a special attribute `xmlns` exists, that can be attached to any element – often the root element:

```
xmlns:quotes="http://www.charlie.com/schemas/quoteschema.xsd"  
xmlns:custcard="http://www.thepostcardcompany.com/schemas/custom  
cardschema.xsd"
```

As you can see, `xmlns` is followed by a colon and the namespace alias (e.g. `quotes`), and the attribute value is a reference to the location of the schema, although as said before, this is not strictly required.

Secondly, elements and attributes belonging to a namespace must *qualify* as such. This is pretty straightforward: just use the name alias as a prefix, followed by a colon. Example:

```

<quotes:Author>
  <Name>Sir James Dewar</Name>
  <YearOfBirth>1877</YearOfBirth>
  <YearOfDeath>1925</YearOfDeath>
</Author>

```

Here, `Author` is explicitly mentioned as being part of the “quote namespace”. Automatically, the namespace for `Author` is also valid for all elements (`Name`,...) occurring inside it. We will see more examples of the use of namespaces in the next section.

Note: The namespace alias may be omitted in the declaration, and then all the elements in the subtree of the element where the `xmlns` attribute is used are assumed to belong to the referenced namespace by *default* (so, unless prefixed by some different namespace alias).

## 2.3.5 XML Schemas: A Typical structure

One of the most exciting upcoming W3C specifications is the XML Schema language. Some years ago, people started realizing that DTDs are insufficient to support the features required for B2B e-commerce, as we have elaborately demonstrated. To use the words of Dave Hollander, a co-chair of the W3C XML Schema Working Group, XML Schema being a Candidate Recommendation means that “...developers of XML e-commerce systems can test XML Schema’s ability to define XML applications that are far more sophisticated in how they describe, create, manage and validate the information that fuels B2B e-commerce.” [W3C Press Release, 2000].

The XML Schema specification consists of three parts:

- **Part 1: Structures**, provides language mechanisms for describing the structure and constraining the contents of XML documents, and also talks about schema-validation rules.
- **Part 2: Datatypes**, defines a set of simple datatypes for elements and attributes (dates, numbers,...).
- An extra **Part 0: Primer**, provides an easily readable description of the language features in Part 1 and 2, illustrated by many examples.

XML Schemas are represented in XML 1.0 (making them accessible to a standard parser) and allow the use of the previously discussed *Namespaces*, while preserving all of the capabilities of DTDs. Consequently, this specification finally lives up to the expectations created by the XML 1.0 (like some decent validation), and fills the gaps in DTD's capabilities we have found above.

Before proceeding to the “evaluation” of the XML Schema language – as opposed to DTDs – we will discuss some of its general characteristics.

### **2.3.5.1 Characteristics of the XML Schema Language**

A detailed description of all the language constructs and features is unfortunately out of scope for this thesis. For the reader who wants to have a more thorough coverage, the *Primer* (Part 0) is highly recommended. Here, the most relevant language features will be briefly discussed.

Let's start off with an example, containing the XML Schema for our FamousQuotes collection:

```

<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
  <xsd:element name="FamousQuotes">
    <xsd:complexType content="elementOnly">
      <xsd:sequence>
        <xsd:element name="Quote" type="QuoteType" minOccurs="0"
          maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="QuoteType" content="elementOnly">
    <xsd:sequence>
      <xsd:element name="Author" type="AuthorType"/>
      <xsd:element name="Content" type="ContentType"/>
    </xsd:sequence>
    <xsd:attribute name="Type" use="required">
      <xsd:simpleType base="xsd:string">
        <xsd:enumeration value="quote"/>
        <xsd:enumeration value="proverb"/>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="Category" type="xsd:string"
      use="required"/>
  </xsd:complexType>

  <xsd:complexType name="AuthorType" content="elementOnly">
    <xsd:sequence>
      <xsd:element name="Name" type="xsd:string"/>
      <xsd:element name="YearOfBirth" type="xsd:short"/>
      <xsd:element name="YearOfDeath" type="xsd:short"
        minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="ContentType" content="elementOnly">
    <xsd:sequence>
      <xsd:element name="Specimen" type="SpecimenType"
        maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="SpecimenType" base="xsd:string">
    <xsd:attribute name="Language" type="xsd:string"
      use="required"/>
    <xsd:attribute name="Original" use="required">
      <xsd:simpleType base="xsd:NMTOKEN">
        <xsd:enumeration value="Yes"/>
        <xsd:enumeration value="No"/>
        <xsd:enumeration value="Unknown"/>
      </xsd:simpleType>
    </xsd:attribute>
  </xsd:complexType>
</xsd:schema>

```

**Example 4: The XML Schema for the FamousQuotes Collection**

Note: It is really worth the effort to compare this example with the previous schema in DTD format!

One thing immediately appears from the example, as compared to the DTD: XML Schemas are expressed in XML syntax, which allows for easy access and manipulation, but is also extremely *verbose*. However, this verbosity allows for increased functionality and makes the whole a lot more readable than was the case with DTDs. In contrast with DTDs, XML Schemas allow the use of *Namespaces*. In fact, the language itself makes extensive use of namespaces, witness the numerous “xsd:...” tags in the example Schema used for language element names (e.g. `<xsd:element ... >`), data types (e.g. `xsd:string`), etc.<sup>4</sup>

Those *data types* are a second important feature. There are *primitive, simple data types*, but also mechanisms for defining new data types by combination, restriction or enumeration. *Complex types*, with the data type definition separated from the element definition, among other things replace the use of parameter entities in DTDs. *Global attributes* can be defined for assigning properties to all elements.

As for the more advanced features, *schema inheritance* allows borrowing from a schema, but overriding it where new features are needed – derivation is possible *by Extension* or *by Restriction*. Support for relationships includes specifying their *cardinality*, and elements which must be *unique* or may have *null values* (familiar from relational databases) can be used. Equivalence between elements can be expressed using *aliases*, and finally, *authoring information* may provide improved documentation for schema designers.

### 2.3.5.2 Documentation and Explanation

Recalling the requirements section, there was a need for powerful, expressive Schemas, easy to read and write by both humans and computers. The Schemas being XML documents themselves means that applications will have easy access to them, through the same methods (see below) that are used for normal document instances. As developers do not need to learn a new syntax and XML Schemas are a lot easier to

---

<sup>4</sup> As we will see in the third part of this thesis, this is a very popular and convenient use for namespaces.

handle than DTDs, they can concentrate on exploring the language functionality and building better models.

The specific documentation features go beyond using “comment” constructs to add some extra information. The `<documentation>` and `<appinfo>` elements are meant to provide descriptions of schema fragments to both human readers and applications. This paves the path for the desired *data dictionary* functionality, although this does not necessarily mean that this mechanism will be actually used, of course.

The availability of namespaces and inheritance not only touches upon the advanced requirements, but also has a positive effect on comprehensibility: a Schema hierarchy is a better representation of the problem domain and enables us to subdivide it into subdomains, which are then handled one at a time.

### **2.3.5.3 Definition of Constraints**

Since this important requirement was one of DTD’s big problems, it is not surprising that a lot of work was done to provide more powerful tools for constraint definition. An extensive set of built-in types, user-defined types, either simple or complex, together with *domain constraints* on allowed values, makes it possible to place more fine-grained restrictions on document content.

Also, XML Schemas provide much better constraints for relationships. There is a simple means to indicate the *maximum cardinality* of a part-of relationship (i.e. the maximum number of occurrences for an element within another element), which was impossible with DTDs. It is also possible to specify if an element or attribute must be *unique* or is a *primary* or *foreign key*.

### **2.3.5.4 Validation**

As the constraints on structures and data type can be expressed more precisely, we can finally do some useful validation. As an example, because of the availability of data types and default values, it can not only be checked if the constraints are satisfied, but this extra information can also be added to the parsed representation of the model.

Here too, Namespaces play an important role. They make it possible to discriminate between multiple “kinds” of tags in one document by validating them with different Schemas. This also allows for *partial validation*, i.e. marking only selected parts of a document to be validated. Moreover, connecting an element name inseparably to a namespace (with a Schema), also removes the problem of unauthorized changes to validation rules!

### 2.3.5.5 Advanced Requirements

As far as the advanced requirements are concerned, most of the good news comes from the *Namespaces* again. In short, the straight-forward prefixing mechanism proves to be the perfect way to avoid any model merging problems, and to allow a modular approach to XML design.

Namespaces also have opened up the road for *schema inheritance*, making it possible to explicitly define *kind-of* relationships. This provides even greater and more flexible expressivity, and also makes the maintenance task a lot easier.

### 2.3.5.6 XML Schemas: Conclusion

In many ways, XML will finally reach a substantial level of maturity thanks to the XML Schema specification. Using XML Schemas, with their added inheritance capabilities, it becomes possible to apply an “*Adopt and Adapt*” strategy<sup>5</sup>: the developer’s job can largely consist of combining (an) existing schema(s) – possibly industry standards – with some building blocks containing “common constructs”, leaving only problem-specific additions or modifications to be made. The extra functionality also helps breaching the gap between XML and the database world by providing common data types, unique keys, etc.

Especially for B2B e-commerce, DTDs were simply insufficient. As we will see in the following sections, XML Schemas combined with other recent XML standards will enable us to explore a whole new range of exciting applications.

---

<sup>5</sup> As a side note, this should *not* be understood as “Copy and Corrupt”.

Note: Apart from the two modelling languages we have discussed above, there are several other standards available or in development, such as: *XDR (XML Data Reduced)* is the most famous one, as it is currently used in Microsoft's BizTalk Framework and SOAP (see below), in anticipation of XML Schemas reaching Recommendation Status. *SOX (Schema for Object-oriented XML)* is an initiative by Commerce One, focusing on extensible data types and inheritance. Nevertheless, assuming Microsoft keeps its promise to switch to the W3C standard once it is finished, these alternatives have little chance of achieving wide-spread use.

## 2.4 Accessing XML: Climb the tree or just strip the leaves ?

Once we have finished the design of our XML document structure by building a DTD or Schema, we want our software to have access to the document instances. As two application programs communicate, one may build a “request XML document”, the other will receive and parse that request, and may want to compose a response, all in XML. If anything, we do not want to burden each application with the low-level problems of treating XML documents as pure *text files*. As [Bos, 1999] states, “If the software developers do their jobs right, then in the future more and more of the nuts and bolts will live invisibly under the hood of XML applications, and content creators won’t have to think about them unless they really want to.” So to start with, we need a standard set of programmatic calls to handle XML document structures in application code, in other words, we need an *API (Application Program Interface)*. This API will then be implemented by an *XML Processor* or *Parser*.

We have already marked the distinction between a non-validating parser, that only checks XML’s well-formedness rules, and a validating parser, which makes life easier for application programs by forcing conformance to certain schema rules. A different way to classify parsers is based on the kind of API-model they use to give program code access to the parsed files. One approach is to build an internal *hierarchical tree* representing the document structure, of which the nodes can then be randomly traversed by the application code. The W3C standard API for these *tree-based parsers* is the *Document Object Model (DOM)*. A second approach is based on an *event-driven* model, that offers a “one-way walk-through” of the document structure: one by one, the occurrences for each class of XML data (elements, attributes,...) are signalled to the code. This approach is represented by the *Simple API for XML (SAX)*, an industry *de facto*-standard.

As parsers are part of the “lower tier” of XML architecture, we will discuss the two approaches only briefly – from a conceptual point of view – and leave out any implementation aspects.

Note: As we have mentioned before, an important use for these APIs is the dynamic processing they provide for XML Schemas and XSL Stylesheets accompanying our documents. For example, the DOM may be used to change styling or transformation rules on the fly by altering the in-memory copy of the referenced stylesheet!

### 2.4.1 DOM (Document Object Model)

The DOM is a specification published by the W3C, which describes it in general as follows:

"(The DOM) Defines a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of HTML and XML documents."

As we have seen in the introductory chapter, any well-formed XML document can be represented by a tree (cf. Figure 1): each element is either a node with an attached subtree containing other elements, or else it is a leaf. Attributes and other language constructs are attached in a similar way. This is exactly what the DOM does for us: when provided with an XML document, it builds an internal tree-structured representation (a *document object model*) from it, if necessary validating each leaf or subtree before it is attached. Once this is done, the entire document can be traversed in a very flexible way, using the calls available in the DOM interface.

The good thing about the DOM is that it allows *random access* thanks to the in-memory tree representation, while *context information* –about parent elements and their attributes– is always readily available. However, there's also a down side: for large XML files, the memory requirements become a limiting factor as they can amount to several times the size of the file. [Birbeck et al., 2000, p187]

Like most standards, the DOM is still evolving in order to keep up with recent changes in related specifications. The second version, called *DOM Level 2*, has reached the status of W3C Recommendation on November 13, 2000. Important features are added support for *namespaces* and *stylesheet manipulation*.

## 2.4.2 SAX (Simple API for XML – Non-W3C)

The second type of API we can use to disclose the information in an XML file is the *event-driven model*. The *de facto* standard here is the popular *Simple API for XML* (SAX), originally put together by a heterogeneous group of XML developers (see <http://www.megginson.com/SAX/history.html> for the full story). As it works similar to the sequential reading of a file, SAX is indeed *simple*: in contrast with the DOM, it does not offer an in-memory view of the parsed file, but just produces a series of *events* as it is working its way through the document. Each event represents a node or a leaf of the document tree (which is imaginary this time).

A simple example might help to understand this:

XML File	Events occurring (method calls)
<?xml version="1.0">	startDocument()
<FamousQuotes>	startElement("FamousQuotes")
<Quote ..... >	startElement("Quote",.....)
(...)	(...)
<content>	startElement("content")
blablabla	characters("blablabla",...)
</content>	endElement("content")
(...)	(...)
</Quote>	endElement("Quote")
</FamousQuotes>	endElement("FamousQuotes")
(EOF)	endDocument()

**Example 5: The (simplified) "output" of a SAX parser.**

Note: As before, indentation is only for better understanding. "..." means that some details have been omitted. *EOF* means *End-Of-File*.

The correspondance between the XML mark-up and the events, represented by method calls, is quite obvious. The work remaining for the programmer is to provide an implementation for each of the methods (such as `startElement()`) in the SAX interface.

The most common reason to use a SAX parser is because it can *parse files of any size*. Since there is no need to load the complete document in memory at any time, the memory requirements of a SAX parser are typically much lower compared to the DOM. Also, it is very useful and fast when you just need *some* of the information contained in

a document, or when the data actually is available as a "stream of XML"<sup>6</sup>. However, these benefits come at a price: SAX makes the programmer's life a lot harder. No random access nor context information is offered here, making more complex matters like searching and sorting a lot harder to program. Moreover, SAX is read-only (at least in the current version), so it is not designed for writing XML documents.

### 2.4.3 Conclusion: DOM vs. SAX

It would have been nice if we could just conclude which is the better: DOM or SAX. However, things are not that simple, since the right choice depends on the situation. We have mentioned some things to take into account (memory footprint, programmatic flexibility, write-capability,...), which should enable you to make up your mind.

On the other hand, we are usually not confronted with a dilemma here. Your XML software is likely to give you the choice, or perhaps make the best choice for you. Parser-related choices can be seen as *low-level optimization matters*, which are not the things *we should* worry about. After all, both APIs provide access to XML documents, and that is our primary interest.

---

<sup>6</sup> To keep things simple, we normally suppose we are working with *tangible* XML documents. However, there is no reason why a "document" could not be presented as a stream of data to the parser.

## 2.5 Manipulating and Querying XML : Get what you need, and put it where you want it...<sup>7</sup>

With the DOM and SAX, we now have a means to gain access to the content of our documents from within application programs. Although this is a valuable and much-needed feature, it still means that the code we write –the implementation of the API’s interface– must handle a lot of XML’s particularities. In this section, we will discuss *XSL(T)*, offering an alternative way of reading and processing XML documents which is better tailored to our needs, because it is *specifically designed* for working with marked-up data.

As we will see, XSLT (with XPath) also has some functionality which makes it applicable for *querying* XML documents. However, some of the query options offered by present-day query languages like SQL are not available within XSLT. For that reason, the W3C has called the *XML Query* working group to life, which is currently working on a new specification to “provide flexible querying facilities to extract data from real and virtual documents on the Web.” [<http://www.w3.org/XML/Query>] In the third subsection, we will have a look at recent evolutions in this field.

Before proceeding to XSLT and the XML Query Language, we will first take a quick look at an auxiliary specification called *XPath*, a language for addressing parts of an XML document, which is used by XSLT, XPointer and XQuery.

Note: Whenever the text refers to “the DOM” in this section, it usually means either “the DOM or SAX”, as these APIs can be used interchangeable.

### 2.5.1 XPath (XML Path Language)

As we have discussed in the previous section, the DOM allows you to *traverse* the XML structures, making it possible to reach every node in your document. However,

---

<sup>7</sup> Remark: I do not say anything about XPath’s *axes* in this section, but refer to the Recommendation document for this.

this can become quite tedious as you have to move your way step by step through the hierarchy to get the item(s) you want. We need a way to *directly* point to the desired piece of information, much like a full directory path (e.g. “d:\My Documents\Projects\Project\_1\Project\_1\_Description.doc”) works. This is exactly what *XPath*, the *XML Path Language*, has been made for.

XPath provides us with an expression syntax to address parts of the node-tree of an XML document. In practice, a node is almost any distinguishable part of the document, such as an element, attribute, comment... The addressing can be either absolute, or relative to the *context node*, i.e. the current node being processed when an XPath expression is used from within XSLT, for example. In addition, there are also some basic functions for manipulation of strings, numbers and booleans.

Instead of digging into all the specialities of this language, a few examples are more useful to get an idea of how this *path notation* mechanism actually works:

```
1. /FamousQuotes/Quote[3]/Author/Name
2. /FamousQuotes/Quote[@Category='Life']
3. /FamousQuotes/Quote[Author/Name='Sir James Dewar']
4. /FamousQuotes/Quote[string-length(Content/Specimen[1])<144]
```

#### **Example 6: XPath Expression Examples**

The first example is pretty straightforward: it selects the `Name` of the author of the third `Quote` in the document. The second gives us all `Quotes` where the `Category` attribute equals “Life” – attribute names are preceded by “@” to distinguish them from element names. The third example selects all `Quotes` of the `Author` “Sir James Dewar”. Finally, the last expression uses the built-in function `string-length()` to pick only those `Quotes` having at most 143 characters in their first content “`Specimen`”. An obvious but important remark is that the selection criteria shown here can be combined to build more complex expressions.

Note: 143 characters is not an arbitrary number. Charlie could use this expression to select only those quotes suitable for sending as an SMS message to the mobile phones of his customers: the maximum length for such messages is 160 characters, leaving 17 places for “- www.charlies.com”!

As you can see, the XPath syntax is fairly easy to understand, and offers an excellent way to select precisely the document part(s) we need. In a way, we can draw a parallel between the way Namespaces support XML Schemas on one hand, and the services provided by XPath to XSLT on the other hand: this powerful technique for locating specific elements in a document will be of great help for describing transformation rules, a topic covered in the next section.

Note: *XPointer*, the *XML Pointer Language*, is another auxiliary specification to address into XML documents. It mostly relies on XPath, but adds functionality such as getting any *arbitrary* region in the document – not only complete, well-bounded XML elements, that is. As for now, XPointer is a Proposed Recommendation, and its primary use is in XLink (see further).

## **2.5.2 XSL(T) (eXtensible Style Sheet Transformations)**

### **2.5.2.1 Introduction**

In the introductory section of this thesis, we already had a first look at *XSL*, the *eXtensible Stylesheet Language*, a specification that is of high importance to XML-based information exchange. There, your attention has been drawn to XSL's capabilities to sort and restructure XML documents to apply a desired “view” on the data. Later on, we have stressed the importance of associating a schema with your XML documents. In the upcoming sections, we will discuss industry initiatives which try to define sector-specific schema standards. Of course, it would be nice if everyone, even within a certain sector, would use the same standard. Unfortunately, it is quite hard to reach agreement among all players, and consequently, different types of schemas are emerging for the same kind of data. Also, a company may choose to use its own data format internally, while it still wants to be capable of communicating with external parties, using the industry-standard format. Therefore we will now further explore XSL(T), as a way to convert a document from one schema to another.

By now, you will have noticed that there is some ambiguity in the use of the acronyms XSL and XSLT. In fact, the original styling initiative was called XSL (eXtensible Stylesheet Language), and its general purpose is to define a “presentation layer” on top of the XML data. This means, indeed, that XSL makes the much-desired promise of “Separating data from its presentation” true. However, soon it became clear that this topic was too broad, and that its constitutive parts were developing at different speeds. For these reasons, XSL was divided into two parts<sup>8</sup>: a *Transformations* part, *XSLT*, to convert one XML document into another XML document, and complementary, a vocabulary to specify formatting semantics, *XSL-FO (Formatting Objects)*.

Note: Often, XSL-FO is simply referred to as “XSL”, but for reasons of clarity of the text, we want to clearly distinguish between the transformation and rendering part here.

This subdivision in fact tells us something about the styling process itself, being a two-step course of action: first, we must *transform* our information from the structure which was applicable for its creation to an organization suitable for its consumption. This includes restructuring, filtering and resorting, but especially expressing the data in a syntax that is understood by the processor in the second step, the *rendering agent*. So, within the W3C Styling concepts, XSLT is supposed to produce presentation-oriented markup, which is however still *independent from the target medium*. Consequently, the work done by XSL-FO in the second, device-dependent step consists of applying the correct *formatting* (either visual, aural or whatever the medium) to the data. So, XSL-FO (which is still a Candidate Recommendation) is purely aimed at rendition of the data. Since in electronic information exchange between companies, most often our computer will be talking to another *computer* on the other side of the line, the Formatting Objects part is considered to be out of scope for this thesis and will not be dealt with any further. For the same reason, we will not cover another XML-suitable rendering technique called *CSS*, the *Cascading Stylesheet Language*.

It should be noted however, that although styling is only indirectly connected with the matters under discussion, the general idea behind separating the styling process into two

---

<sup>8</sup> Strictly speaking, XPath was a third part of the original XSL initiative. However, it has become a separate Recommendation, and it also forms the basis of XPointer – as has been mentioned.

distinct parts is quite interesting. Instead of merely separating data from its presentation, XSLT brings the more general objective of “Separating data and data model *creation* from its intended *consumption*” within reach. As in many software problems, adding one or more extra levels of indirection increases flexibility and brings us closer to a solution.

### 2.5.2.2 The XSLT Mechanism

In short, XSLT provides us with a way to *transform* (i.e. *to reorganize the structure of*) a document instance into a different format. Reasons for this can be, for example, that we want to prepare an incoming document for processing in our system, or adapt an outgoing set of XML data to conform to a certain standard we have agreed on. In other words, we want to describe a set of rules to change the structure and syntax of a given instance, in order to make it viable under a schema that is different from the one that was used to create it. Note that, if required (especially during testing), sending the result to a *validating parser* gives us a means to check whether the transformation is successful or not.

In the context of XSLT, it might be a source of confusion to talk about *stylesheets*, since our objectives have little to do with styling. Therefore, XSLT stylesheets are often called *transformation scripts*, a name better representing their function. We will now take a closer look at how these scripts are built up, in order to obtain a clearer view on the transformation process itself.

To have a place to start from, let’s recall the XSLT example we have given in the very first section of this thesis, but this time stressing other features:

```

1. <?xml version="1.0"?>
2. <xsl:stylesheet
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      version="1.0">
3. <xsl:template match="/">
4.   <html><body>
5.     <xsl:apply-templates select="FamousQuotes"/>
6.   </body></html>
7. </xsl:template>
8. <xsl:template match="FamousQuotes">
9.   <table border="1">
10.    <tr><th>Author</th><th>Quote</th></tr>
11.    <xsl:for-each select="Quote">
12.      <xsl:sort select="./Name"/>
13.      <tr>
14.        <td><b><xsl:value-of select="Author/Name"/></b></td>
15.        <td><xsl:value-of select="Content"/></td>
16.      </tr>
17.    </xsl:for-each>
18.  </table>
19. </xsl:template>
20. </xsl:stylesheet>

```

**Example 7: The XSLT example revisited**

Applying what we have seen before to this example, we notice that this “transformation script” first identifies itself as an XML document. Then the *document element* tells us that we are dealing with an `xsl:stylesheet`<sup>9</sup>, associates the “xsl:” prefix with the XSLT namespace URI “http://www.w3.org/1999/XSL/Transform”, and that the XSLT version we are using is 1.0, all as required<sup>10</sup> by the specification. So far, nothing spectacular.

In the subsequent elements however, we discover the real nature of an XSLT script: the “set of rules” that determines how the transformation is effected, is presented as a collection of *templates* (`<xsl:template...>` elements). Each template consists of a *match pattern* in the form of an XPath expression, that determines where this template applies, one or more *XSL(T) instructions* (prefixed with “xsl:”), and usually some *literal result elements* coming from non-XSL namespaces – in casu, HTML content, which would of course be unlikely for an inter-application message, but other examples will be given

---

<sup>9</sup> As the XSLT 1.0 specification states: “A stylesheet is represented by an `xsl:stylesheet` element in an XML document. `xsl:transform` is allowed as a synonym for `xsl:stylesheet`”, so there is no inherent difference between the two, and therefore we have chosen to avoid confusing the reader.

<sup>10</sup> Strictly speaking, there is no obligation to use “xsl:” for the prefix, but it is common practice to do so.

below. For *each* node in the source document, the XSLT processor examines *each* template to determine whether the node satisfies the match pattern in the template. Here we see the importance of XPath expressions: as the match pattern specifies which nodes of the input a certain template applies to, it forms the connection between the newly created, copied or modified elements of the result document and their origin's element tree.

It should be pointed out that XPath's built-in functions are a substantial asset in constructing powerful match patterns. For instance, we can adapt line 11 as follows (in correspondence with the last XPath example)<sup>11</sup>:

```
11.      <xsl:for-each select="Quote[string-
          length(Content/Specimen[1]) &lt; 144]">
```

By simply changing the match pattern<sup>12</sup> used in this `for-each` instruction (which will be explained shortly), only “SMS-suitable” quotes would appear in the output.

Inside the `<template...>` element, that contains the match pattern, we find the reason for the name of this element: its content is a *template* for content that is to be inserted into the transformation result! On one hand, there are *instructions* that control the output creation, on the other hand we have *literal result elements*, of which an unmodified copy is inserted into the result. In some respect, XSLT can be seen as a complete *programming language*, so giving an overview of its complete instruction syntax would lead us too far, although some parts of it will turn up in the examples further down. Broadly categorized, it includes constructs for:

- Repetition and Sorting (e.g. the `for-each` element shown above)
- Creation of elements and attributes (modified from input or new)
- Conditional processing (if, choose,...)

---

<sup>11</sup> “<” is written as “&lt;” here. As you might suspect, the left angle bracket cannot appear in an XML document when it is not a markup delimiter, because a parser would have no means to distinguish it from the “opening bracket” of a tag. Therefore it must be *escaped* using this “&lt; entity reference”.

<sup>12</sup> Strictly speaking, this is not a match pattern but a *select pattern*. Although both use exactly the same syntax, the first only checks whether a given node can be identified by the pattern, while the latter is used to *retrieve* all nodes that meet the criteria.

- Creation and manipulation of numeric, boolean and string variables
- Stylesheet combination (Importing & Including)

Again, XPath expressions play an important role for many of these language elements. One remark about the last category, *Stylesheet combination*, may be in place here because it provides some very interesting options for practical use of XSLT. A stylesheet doesn't have to be a monolithic unit. In some cases, for instance when dealing with a large, complex stylesheet, it may be easier to break it into modules and recombine these afterwards using the `<xsl:include...>` instruction. This also makes it possible to share “code snippets” among stylesheets. Moreover, while `<xsl:include...>` simply inserts the contents of another stylesheet, `<xsl:import...>` allows us to customize an existing stylesheet by importing its contents and then *override* (i.e. adapt) only those template rules that do not fit our needs. In brief, the modularization and reuse facilities offered by the *include* and *import* instructions lighten the design and maintenance of transformation scripts.

There is little to say about the literal elements, except that they are usually specific to the type of output the transformation produces. We will however come across more examples in the following paragraphs.

### 2.5.2.3 Why XSLT? – Push/Pull Approach to Stylesheet Design

The attentive reader could now correctly remark that we *already had* a mechanism to do the transformation work. We could use a DOM or SAX parser to programmatically extract the information from the source document, adapt it as desired, and then write out the result in the desired form. In a way, this is true, but at the same time there are both practical and conceptual considerations to bear in mind.

*Practical* issues mainly concentrate on making the programming task easier. A first minus of the DOM specification (Level 1 and 2), is that it does not support XPath expressions to reach a particular node in the input tree, which makes it a laborious job to navigate through the document tree. When using XSLT, several of these basic functions, such as memory management and node traversal, are dealt with by the XSLT processor. Also, high-level constructs for sorting, counting and restructuring elements are available as simple XSLT instructions – although coding more complex algorithms

can become quite verbose. Finally, if we make use of the *stylesheet combination* features mentioned above, it is very easy to create a collection of stylesheets that can be easily put together, in a similar way as we have seen with XML Schemas. Here again, the ease at which modularization can be achieved greatly increases the opportunities for *reuse* of code.

On the *conceptual* side however, there is a big difference between using the DOM and XSLT, in that the former prescribes a purely *procedural* approach, while as we have seen, the latter is rather a *declarative* language. This means that the *templates* describe the state of the transformed document in relation to the original document, without necessarily specifying *how* this transformation should take place. [Birbeck et al., 2000, p417] In principle, the process is driven by whatever the processor encounters in the document, not by the code itself. This approach is sometimes called “programming by example”, because for each kind of input data that you want to use, you specify an according “wrapper” for how it should appear in the output.

However, using only templates doesn’t give you the whole picture, since we have also seen explicit loop instructions and conditional constructs. In fact, XSLT offers two at first sight distinct ways of programming:

- “*Pushing*” the input data, also called the *data-driven* approach: here, the elements in the input XML-document trigger the appropriate templates by “presenting” their data to the stylesheet, which plays a “passive” role.
- “*Pulling*” the input data, also called the *stylesheet driven* approach: in this case, the stylesheet has an “active” role, “instructing” the parser to fetch the required elements from the input document and processing them according to these instructions. Indeed, this approach adds a procedural dimension to XSLT programming.

Note: For the reader who has further interest in the concepts behind the XSLT language, it is good to know that it is related to the world of *Functional Programming*. Under this paradigm, a program is built by combining functions (our *templates*) into more powerful functions, until the final function produces the required result. On <http://www.xml.com/pub/a/2001/02/14/functional.html>, you find an interesting

introduction to this topic under the title “Functional Programming and XML”, by Bijan Parsia [Parsia, 2001].

Now, what does the availability of *procedural* constructs have to do with the comparison between DOM- and XSLT-based transformations? Well, basically it extends our freedom of choice, or stated differently, it draws our attention to the trade-off between the power offered by DOM-based programming and the flexibility and simplicity when using XSLT. In practice, you will “never” encounter a stylesheet that is purely based on either the push or the pull approach, rather you would see a coarse-grained template structure, with each template containing more fine-grained pull instructions. The pull instructions make it easier to program some of the harder problems in XSLT. In the end, only the really complex transformations require you to use the DOM, when an XSLT script would sink away in over-verbosity. In theory, the rule is simple: use XSLT when you *can*, and the DOM when you *must*.

Note: Does this mean that the DOM is useless whenever we can use XSLT? No, because a stylesheet is itself an XML document, so it can be modified (or even *created*) with the DOM API *at runtime*. An example is changing one or more of the match pattern so that the “filtering” or the sorting of the output is different.

#### **2.5.2.4 A small digression : Transforming into SQL Insert Instructions**

In many XSLT stylesheets, the *literal result elements* are part of some standard XML vocabulary, so the transformation result will be a well-formed XML document. However, to stress the versatility of XSLT transformations, we want to give a completely different, but relevant example here.

Let’s get back to Charlie for a minute. Our shop owner is still eager to learn more about XML, and so far, he believes he’s pretty much getting the picture. However, Charlie is a little displeased and even worried about the tremendously fast evolution of the XML world. To him, it appears that although new promises are made with every new version of some XML-related specification, there isn’t much certainty nor stability in all those standards. So he visits his old friend Bill, a database administrator, and what does Bill tell him? “If you want to be on stable

ground, SQL is still what you need.” Now, Charlie would not feel so comfortable with simply doing away with the XML matters he has learned, and decides to bet on two horses. The question that arises is: is it possible to convert the XML Quotes Collection to SQL, using XSLT?

We are not going to cover the design of an SQL database here, so we’ll simply assume that Charlie wants to put the quote type, category, authorname, and content of the first “specimen” into a single, “denormalized” table called “famousquotes”. What he needs is an XSLT transformation script, that generates a text file containing *SQL Insert instructions* from his XML database, something like this:

```
INSERT INTO famousquotes (type, category, authorname, content)
VALUES ("quote", "Life", "Marx, Groucho", "Those are my principles.
If you don't like them I have others.")

INSERT INTO famousquotes (type, category, authorname, content)
VALUES ("quote", "Life", "Dewar, Sir James", "Minds are like
parachutes. They only function when they are open.")
```

As you will see, the stylesheet to do this is quite simple. First of all, in XSLT, there’s a special element to specify the desired kind of output, namely `<xsl:output method="...">`, where the value of the `method` attribute can be either “xml”, “html”, or “text”. In the examples, we have seen so far, the output was assumed to be in HTML format, because the first element in the result document was “`<html>`”. Otherwise, the default value is “xml”, so usually only the “text” output method has to be explicitly indicated, and this is what we need here.

This is what the stylesheet looks like:

```

1. <?xml version="1.0"?>
2. <xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
3. <xsl:output method="text"/>
4. <xsl:template match="/">
5. <xsl:for-each select="FamousQuotes/Quote">
6. INSERT INTO famousquotes (type, category, authername,
  content)
7. VALUES ("<xsl:value-of select="@Type"/>", "<xsl:value-of
  select="@Category"/>", "<xsl:value-of
  select="Author/Name"/>", "<xsl:value-of
  select="Content/Specimen[1]"/>")
8.
9. </xsl:for-each>
10. </xsl:template>
11. </xsl:stylesheet>

```

**Example 8: An XSLT Stylesheet transforming XML data into SQL entries.**

As you can see, nothing really spectacular here. The HTML markup has been removed, and the SQL syntax has taken its place. Regarding XSLT instructions, the sorting was redundant here, and this time not only element content but also some attribute values are selected. When this stylesheet is applied to our sample XML document, it produces the desired output you have seen above, that's how simple it is!

### 2.5.2.5 Conclusion

The volume and the level of detail displayed above have probably made it clear to you: together with the schema-related matters, the W3C specifications dealing with XML document transformation are ranking high on the ladder of B2B-relevant standards.

In the modeling section, we were looking at DTDs and XML Schemas to investigate if their possibilities are sufficient to express the semantics of the information in our business model. In other words, it was strongly advocated to concentrate on *business reasons* when making design decisions, and *not* on how you plan to *use* your data.. At that time, it was implicitly assumed that a proper method for producing the desired output would come up. In this section, we have seen that XSLT excellently fulfills this task.

XSLT's biggest merit can be summarized as allowing us to concentrate on the semantics of our information during the modeling phase, so we can start from the underlying business model. This also means that even if we have to communicate with a system or standard that is beyond our control, we can still exchange document instances based on a different schema type by transforming them to our own schema and vice versa. As *interoperability* with systems belonging to independent parties becomes a necessity, and the XML data format will become the *lingua franca* for this integration, XSLT makes us at the same time independent of and compatible with the standard vocabularies in our industry.

Although the language can hardly be called “easy-to-learn”, the examples have shown us that as soon as one is accustomed to its basic *declarative* nature, XSLT has a very good power/complexity trade-off. It opens up any XML marked-up document to an abundance of applications, making it both easier and worthwhile to build an XML wrapper around legacy systems. Moreover, in some beliefs, in the future *any* system that cannot be approached through an XML-based interface, will be labeled as “legacy”<sup>13</sup>. In this regard, XSLT can be seen as the big *leveraging technology* for XML efforts. XSLT makes XML *work*.

In the next section, we will shortly revisit XSLT in a different context, that of querying XML data.

Note: The version of the standard that we have discussed here, is XSLT 1.0, which has become a Recommendation on November 16, 1999. This specification is very useful and powerful, all the more since it includes a built-in *extension mechanism*. This has allowed several XSLT processors to incorporate extra functionality, but has created a problem of *portability* because these extensions are proprietary. Therefore, a new working draft for *XSLT 1.1* has already been issued, of which the primary goal is to address this problem by standardizing several extension-related aspects.

---

<sup>13</sup> In this context, “legacy” has the meaning of “lacking interoperability mechanisms, needing special-purpose programming to communicate with the external world”.

## 2.5.3 XQuery (XML Query Language)

### 2.5.3.1 Introduction

As XML is rapidly gaining popularity as a *universal data format*, the expectations towards the provided functionality are also maturing. Increasingly, companies are thinking of storing and exchanging their data in an XML-marked up format, whether this is in a file-system based manner, a native XML database, or some type of traditional database system that has been “*XML-enabled*”. Consequently, there is a growing need for fine-tuned access to large volumes of XML data, by means of a *query language*.

Although talks about this matter have started several years ago, the development of a generally agreed-upon query language has proven to be a hard nut to crack. Numerous proposals have been submitted to the W3C, and only very recently a first working draft for XQuery, the XML Query Language has been published. As such a working draft is likely to undergo several changes before reaching Recommendation status, we will mainly focus on the *XML Query Requirements*, in other words, what we can *expect* XQuery to become. Considering the early status of the working draft<sup>14</sup>, only a simple, illustrative example will be given. After that, we will pay another short visit to XSLT, as document *translation* is argued to be closely related to document *querying*.

### 2.5.3.2 The XML Query Language

As we have mentioned, the main starting point to describe XQuery will be the *XML Query Requirements* document. The "Goals" specified at the start of this document give us an excellent idea of what this language is meant for:

"The goal of the XML Query Working Group is to produce a **data model** for XML documents, a set of **query operators** on that data model, and a **query**

---

<sup>14</sup> As is discussed in the “XSLT as a query language” paragraph, the XML Query syntax is far from final – several experts have serious concerns about it and *really* hope to see some changes. Moreover, as the current working draft states: "The Query Working Group has identified a requirement for both a human-readable query syntax and an XML-based query syntax. XQuery is designed to meet the first of these requirements. An alternative, XML-based syntax for the XQuery semantics will be defined separately."

**language** based on these query operators. The data model will be based on the *W3C XML Infoset*, and will include support for Namespaces."<sup>15</sup>

From the bold-printed lines of this first part of the *Goals* statement; it already appears that there is much more to developing a new query language than just defining its syntax, and this would be hard to fit into one specification. Indeed, apart from the Requirements, the W3C website shows several working drafts. According to the abstracts of the respective documents:

- **XML Query Use Cases**, specifies usage scenarios for the W3C XML Query data model, algebra, and query language.
- **XML Query Data Model**, defines the W3C XML Query Data Model, which is the foundation of the W3C XML Query Algebra. Together, these two documents (will) provide a precise semantics for the XML Query Language.
- **The XML Query Algebra**, introduces the XML Query Algebra ("the Algebra") as a formal basis for an XML query language.
- **XQuery: A Query Language for XML**, describes a new query language called XQuery, which is designed to be broadly applicable across all types of XML data sources.

The *Goals* also tell us something about what the language should do:

Queries operate on single documents or fixed collections of documents. They can **select** whole documents or subtrees of documents that **match conditions** defined on document content and structure, and can **construct new documents** based on what is selected."

This draws our attention to an important subdivision in the definition of a query, and correspondingly, in the tasks it has to perform:

1. Identifying and selecting values to retrieve or manipulate
2. Constructing the query result

---

<sup>15</sup> *XML Info*, or the *XML Infoset* defines an abstract data set which contains the useful information available from an XML document. [55] This specification was considered too "low-level" to discuss in this thesis.

The need for the first part is obvious: the query mechanism has to know which information we are interested in. The current working draft uses a language *very similar*<sup>16</sup> to XPath to do this. The importance of the second part however, might seem less evident, especially to people who have experience with relational databases and SQL – where this is a rather trivial matter, since the output of a query is essentially always a two-dimensional table. In XML however, we are dealing with *hierarchical* data, both on the input and the output side: as section 3.4.20 of the *Requirements* demands, a query must be *closed* with respect to the *XML Query Data Model*<sup>17</sup>. This means that we have to express quite precisely how we want the output to appear, such as the ordering and the hierarchy. For this task, the working group has defined a new syntax, of which you can see an example below. Considering the current status of the working draft, this example is meant as a syntax illustration only. Instead of applying the specification to our *Famous Quotes* collection, the example comes from the XQuery working draft itself – we wouldn't want to disappoint Charlie by making him learn a syntax that is still subject to change...

```
FOR $b IN document("bib.xml")//book
WHERE $b/publisher = "Morgan Kaufmann" AND $b/year = "1998"
RETURN $b/title
```

**Example 9: A simple XQuery example – taken from example "Q9" in the XQuery Working Draft.**

The example should be pretty self-explanatory. It lists the titles of the books published by Morgan Kaufman in 1998, from some book database. The "For-In-Where-Return" sequence of clauses is very similar to SQL's "Select-From-Where", although XQuery's "Return" often needs no SQL counterpart, as explained above.

The *Requirements* document discusses quite a few other topics. The *Usage Scenarios* express the need for both document- and data-oriented queries, while the input may be

<sup>16</sup> Evan Lenz calls XQuery's path expression syntax an "extended subset" of XPath [65] – more on these matters in the next subsection.

<sup>17</sup> This also means that non-native-XML resources (e.g. relational databases) have to offer a representation in this Data Model to make them "queryable".

in the form of a byte-stream. Several requirements are common to query languages in general, such as performing *joins*<sup>18</sup>, sorting query results, handling error conditions, handling NULL values, etc. Others are rather XML-related, such as *namespace* awareness support *XML Schemas*, although the absence of a schema may not prevent querying. This means that the data model a query operates on must be “enriched” with schema information, if available. In general, the XML Query Working Group is required to cooperate with other working groups, to ensure that any dependencies between XQuery and other specifications (XML Schema, DOM, XSL(T), XPath,...) remain valid.

If you have any experience with query languages, you may now remark that we have covered only part of the story, corresponding to SQL’s *select* statement. You are right: the XQuery designers have started off in modesty. As *Issue 15* of the first working draft somewhat euphemistically says: “We believe that a syntax for update would be extremely useful, allowing inserts, updates, and deletion. This might best be added as a non-normative appendix to the syntax proposal, since the algebra is not designed for defining this portion of the language.” XQuery is still very young, and there are many problems to overcome.

### 2.5.3.3 The XPath-XSLT Combination as a Query Language

After the previous extensive discussion of XSLT, the reader might frown at another paragraph being devoted to this specification. However, you already have some hints that the XPath-XSLT combination might serve as a query language, if you recall the “subdivision in the tasks a query has to perform” from the previous subsection. About the first part, identifying and selecting the right values, it has already been mentioned that XQuery essentially uses XPath to do this. The differences are too much in detail for us to go into, and moreover, the XML Query and XSLT Working Groups are now coordinating on the development of XPath 2.0 [Dodds, 2001b].

---

<sup>18</sup> A *join* is a query that combine data from multiple sources into a single query result, mostly used in the context of relational databases, but also applicable to XML data.

For the second task, constructing the query result, paragraph 3.4.11 of the *XQuery Requirements* is quite telling: “Queries must be able to transform XML structures and must be able to create new structures.” Indeed, this is pretty much what XSLT is doing for us, and besides, both are *declarative, functional languages*. Immediately after the publication of the first XQuery working draft on February 15, 2001, this overlap in functionality has been remarked by software engineer Evan Lenz, in a paper named “*XQuery: Reinventing the Wheel?*” [Lenz, 2001]. He argues that the overlap is too great to recommend XSLT and XQuery as separate languages, and that “XSLT as it currently stands may function well as an XML Query Language.” Furthermore, the paper shows how each of the XQuery examples in the working draft can be expressed in XSLT. Again, we give an illustrative example of the XSLT version of the example “Q9”, mentioned in the previous paragraph:

```
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:for-each select="document('bib.xml')//book">
      <xsl:if test="publisher='Morgan Kaufmann' and year='1998'">
        <xsl:copy-of select="title"/>
      </xsl:if>
    </xsl:for-each>
  </xsl:template>
</xsl:transform>
```

**Example 10: XSLT version of example "Q9" in the XQuery Working Draft.**

From the findings in the paper, it seems that XSLT is indeed capable of doing roughly the same job as XQuery. Expected XSLT problems in regard to *performance* and *scalability* are largely a matter of internal parser optimization, and several other problems could be dealt with in a next version of XSLT. However, as the example already lets you suspect, there is a big difference in *conciseness* and *readability*, which are both non-negligible factors in the context of application development and maintenance.

Lenz’ opinion has met both sympathizers and opponents on the influential *XML-Deviant* mailing list on the XML.com website [Dodds, 2001b]. The supporting side questions the utility of additional complexity caused by a completely new syntax, instead of taking a more “evolutionary” approach. The opponents however, argue that the two

languages were simply designed for different purposes. Moreover, XQuery is said to lend itself better to optimization.

### **2.5.4 XQuery: Conclusion**

Querying XML marked-up information is an important matter in many contexts, and software vendors are probably closely monitoring the developments in this area. However, the XQuery-related specifications are still very young, so it is early to draw conclusions. Its intuitive and easily readable syntax (especially when compared to XSLT “queries”) is a valuable advantage, and it is unquestionably a good thing that the underlying query algebra and data model are now being formally defined. The functionality overlap however, is a potential source of confusion as to which language should be used for a specific application. Whether XQuery has the right to coexist with XSLT, will ultimately become clear when the first implementations become available.

## 2.6 Connecting XML: XLink

For those of you familiar with HTML, you might feel like something important has been missing so far, that the picture is not complete. If XML is “redefining” everything in the way we publish and communicate information over the internet, then where is the functionality to *link* all the information resources together? W3C’s answer to this question is *XLink*, the *XML Linking language*, currently a Proposed Recommendation, which we will discuss briefly. [Maler, 2001a]

As usual, it is good to ask ourselves what is good and bad in the way web resources are currently linked. *HTML Linking* is performing well as a general-purpose instrument to associate arbitrary resources in one direction. To make a link, you just need write access to the “starting” point, not to the resource you are linking to. This introduces some robustness through tolerance of broken links – lost resources do not jeopardize the system – but at the same time makes it hard to keep links up to date. Also, being able to edit the starting point may not be so obvious, especially for non-HTML (or non-XML) documents. Next, there is no notion of the *granularity* of links: a person who has no access to the referenced resource, in general can only link to a *complete* entity. And even if you could point to an *anchor ID* inside the resource, they merely tell a browser to “move the cursor” to a certain position. You can only specify a destination *spot*, not a paragraph, table or *section*, which also means that it is impossible to embed a part of a linked document in your own document!

In short, the weaknesses of HTML Linking arise mostly from the obligation to insert the link into the starting point of the referencing document, from the absence of any relationship in the opposite direction, and from the “pointedness” of the links. *XLink* solves these problems in two ways: first, the definition of the link can be separated from its starting point and put into a *link element*, which also contains the reference to the destination point. For instance, this makes it possible to store groups of links together in *linkbases*. Secondly, the *XPointer* language is used to *address into* XML resources and to specify a destination *region*. In addition, some new options are provided for the *link traversal behaviour*, such as *when* to traverse the link and *what exactly happens* then.

A special point to remark is that XLink consists *only* of attributes, making it an “enabling vocabulary”, meant to be incorporated in your own vocabulary. It’s namespace is “http://www.w3.org/1999/xlink”.

In XLink, there are two types of *link elements*: HTML-style *simple links* can be defined, but there are also (*extended*) *link elements* that contain several kinds of *participants*, and one or more *arcs*. The *participants* can be resources of any type, either XML or non-XML, and are either contained inside the link element (*local resources*) or be referenced by a URI (*locators* or *remote resources*). In case the resource is an XML document, *XPointer* expressions can be used. An important remark is that one extended link element can have *more than two* participants, which removes an important restriction of HTML links. The actual “connection” between the participants is defined by the *arcs*, which indicate the starting and ending point(s) of the respective resources. In addition, metadata can be attached to these arcs and participants, containing human-readable *titles* and/or machine-processable *roles*.

For specifying the behavior of the link (what *happens* when the link is traversed<sup>19</sup>), we have again more options than with HTML. The referenced resource may either establish a *new* context separate from the current document, or can *replace* it, but a third possibility is that the referenced resource contents are *embedded* into the referencing document. In addition, it may be indicated that *other* behavior is specified elsewhere, or that it is not specified at all (*none*). Concerning *when* the link is to be followed (*actuated*), this can be *onRequest* of the user or application, *onLoad* of the referencing document, or again *none* or *other*.

The above should give you an idea of what XLink is about and what advantages it provides over HTML-style linking. We will not explain this specification in detail, but

---

<sup>19</sup> Opposite to HTML links, it would be wrong to say “when the link is *clicked*” here, since XLinks have a much more wide-spread use: they will often be intended for traversal by applications rather than humans.

will only provide an illustrative example here. It is excerpted from [Maler, 2001a] and shows a basic extended link with some of the described aspects<sup>20</sup>.

```
<extendedlink xlink:type="extended">
  <loc xlink:type="locator"
    xlink:label="prolog-ref"
    xlink:href="#xpointer(string-range(//text(),'prolog'))" />
  <loc xlink:type="locator"
    xlink:label="prolog"
    xlink:href="#/1/2/5" />
  <arc xlink:type="arc"
    xlink:from="prolog-ref"
    xlink:to="prolog"
    xlink:show="new"
    xlink:actuate="onRequest" />
</extendedlink>
```

**Example 11: Illustration of XLink**

---

<sup>20</sup> This example is an illustration only, since it cannot be fully understood without its original context.

## 2.7 XML Standards: Conclusion

By now, you should have a fairly good insight in what the members of the XML Family are dealing with, and how they are cooperating to offer a complete framework to support XML-based application programs. The following diagram gives a good overview of the place each standard takes in the XML genealogical tree:

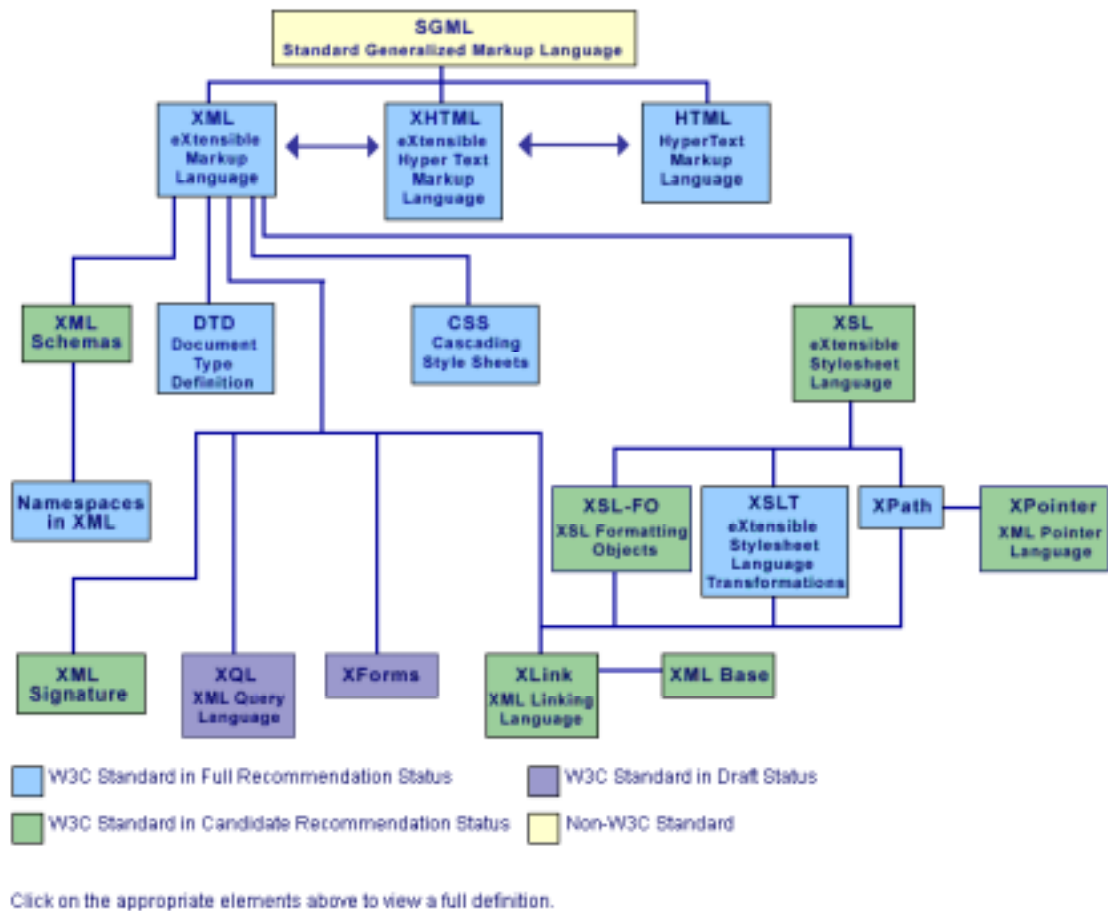


Figure 3: Snapshot overview of the XML Family - adapted from [http://www.reims.net/Resource\\_Zones/xml/xml\\_overview.html](http://www.reims.net/Resource_Zones/xml/xml_overview.html)

Some of these standards, like XML Schema and XSLT, have received more than average attention, as they are key to developing interoperable applications. On the other hand, related standards like XML Fragment, XHTML, RDF,... are not covered in this text, or have only briefly been mentioned. They are considered to be less important or too technical in our point of view.

To conclude, a word of caution is in place here. In the XML Community, more specifically among the contributors to the *XML-Deviant* mailing list (Simon St. Laurent, Jonathan Robie,...), concerns have recently been expressed about the mutual interdependencies between several W3C standards. In the document “Time to Refactor XML?”, Leigh Dodds gives some examples of these “intertwined specifications” [Dodds, 2001a]:

- XSLT 2.0 must support XML Schema datatypes
- XPath 2.0 must support the regular expressions defined in XML Schema datatypes, as well as the XML Schema datatypes
- XML Query and XPath 2.0 will share a common data model
- XML Query may itself use XML Fragments<sup>21</sup>
- XML Query must support XML Schema datatypes
- Both XPath and XML Query must be modeled around the Infoset, and particularly the "Post Schema Validation Infoset"
- XML Schema itself depends on XPath to define constraints

As each standard is both required to support new versions of other standards, *and* to be updated itself for extra functionality, there is a danger of both software vendors and end-users losing their grip on the whole evolution. “Dead-lock” situations, especially when products from different vendors have to cooperate, become a potential problem. Again, additional functionality comes at the price of complexity. A careful conclusion for anyone who is using XML in software development might be, that a careful check for what version of each standard is supported by the tool(s) in use, is certainly not to be overlooked. However, the question arises in how far this affects the foundations on which XML was built?

---

<sup>21</sup> **XML Fragment:** Defines a mechanism for providing to a recipient of a fragmentary XML document the appropriate information about the context from which the fragment came.”

## Part 3: ELECTRONIC BUSINESS AND EBXML

## 3.1 Introduction

By now, it should be clear to you: XML is *hot*. Sometimes it seems hard to find a new product name that doesn't have an "X" in it, and every single self-respecting computer company is eager to announce that "their product is XML-enabled too" – whatever that may mean. For some young, innovative companies, it seems that XML is especially a key enabler for *themselves*. As XML is mushrooming in front of our very eyes, the growth of companies bringing some innovative "X-technology" is at least as spectacular. So the trend is clear: *everyone* is jumping on the XML-train.

This *unanimity* of choice among all parties is in fact one of XML's biggest strengths. At least partially, it is attributable to XML's *simplicity* – an issue that has been stressed several times in the foregoing. In the past, only technologies that are adequately positioned on the power-complexity axes, have had a chance to become widely accepted, or even unanimously agreed-upon *de facto* standards.

Moreover, given that a technology is reasonably easy to implement, it is arguable that for some application areas, the mere fact that a common standard can be agreed upon is a blessing. One of these areas is *information exchange between companies*, where a unanimous choice for the supporting technologies is of capital importance.

In this third part of the thesis, we will mainly discuss the ebXML framework, which offers a (supposedly) complete set of guidelines to implement electronic business systems. We will consider how XML, and its related standards, can fit into an XML-based framework like ebXML. Such a framework enables or facilitates *interoperability* in general, and *integration* between disparate computer systems, at separate locations and belonging to different companies, which are – in principle – independent, at least from an IT point of view.

Before diving into the turbulent waters of emerging B2B standards and frameworks, we will first sit back and try to gain focus on the problems these proposals are claiming to solve. First, it is good to have a look at the different application areas where XML may play a role. As you will see, this will help us to identify some aspects of the problem

domain. Secondly, we should wonder why the recent evolutions are so important, and why established standards such as EDI are unsuitable to address the problems of general, multi-party, world-wide B2B eCommerce. Therefore we will pay a short visit to EDI-related matters, which will most importantly help us to avoid past mistakes. Thirdly, in order to prevent ourselves from being overwhelmed by beautiful promises and “marketing pep talk”, we should ask ourselves what is actually required to construct an eCommerce system. In other words, we will try to get an idea of the essential building blocks for B2B eCommerce architectures. With these building blocks in mind, the next step will be to look at the ebXML standardization effort, and see where they fit in.

## 3.2 XML Application Areas

### 3.2.1 A double dichotomy for classification

As XML is being hyped by virtually every software vendor, it is not surprising that it has many applications. In fact, we could be really short about this: XML can be used for “almost anything”. Of course, that doesn’t help you very much, so we will have a look at some of the application areas in which XML can be of significant value.

The question we are asking here is: "What can XML do for us?" or "Where can XML be of assistance?" In most publications, the answer to this question is formulated as an enumeration, but unfortunately, the answer differs substantially depending on the author's perspective. Therefore, it might be more valuable to depart from a double dichotomy, where we concentrate on two general characteristics: the kind of information are we dealing with, and the "life span" of the marked-up information (which is closely linked to the purpose of its existence). The first characteristic results in two options, Structured and Semi-Structured information, the second in Permanent and Volatile. This frame of reference provides substantial value to the upcoming analysis, since it is important to be aware of the nature of the information we are dealing with, and to be aware of the application areas we will *not* be covering.

In this context, we speak about *Structured* information, when *each* meaningful piece of data is clearly identified by some label. In other words, we are actually talking about an internally structured collection of elements. A good example is a purchase order, for which all fields can be tagged with a name. *Semi-Structured* information on the other hand, should rather be seen as a document in which *some* parts are "marked" to add meaning. Think of a text book for instance, where some words (such as names of persons, places,...) could be given a label. Of course, this distinction is not always perfectly clear, all the more because sometimes semi-structured information is a merger of structured information and descriptive text.

The second dichotomy, *Permanent* or *Volatile*, is easier to describe. The distinction that is made here, is whether we want to use XML to *store*<sup>22</sup>, or to *exchange* information (usually over the web). Again, there is a form of interaction here, since permanent information will often be the source and/or the destination of volatile information. The latter could also be described as "data in transit", either to a user for display, or to another application. Notice that "Volatile" does not imply that the information is *never* stored, rather that storing is not the purpose. The following four-quadrant scheme indicates some of the typical applications. It is not meant to be exhaustive, but should make the frame of reference more clear.

### 3.2.2 The resulting Four-Quadrant Scheme

	Permanent	Volatile
Semi-Structured	<ul style="list-style-type: none"> <li>- <b>Document Management</b></li> <li>- <b>Web Content:</b> XML as "SGML Light", for later publishing by applying a specific stylesheet.</li> </ul>	<ul style="list-style-type: none"> <li>- <b>Online Publishing</b>, integrating different types of media: render to HTML, WML... afterwards</li> <li>- <b>"One-time" usage</b> of information between server and client (human).</li> </ul>
Structured	<ul style="list-style-type: none"> <li>- <b>Data archival</b> – in an unproprietary format (e.g. Registry/Repository)</li> </ul>	<ul style="list-style-type: none"> <li>- <b>Data Transfer:</b> "pure" data (e.g. data-extraction from different sources to feed a data warehouse)</li> <li>- <b>Message Exchange</b> : data + <i>what to do with it</i> (e.g. EDI, RPC)</li> </ul>

---

<sup>22</sup> There are various ways in which XML can be stored, e.g. as files in the file system, within relational databases, or in "native" XML databases. Because this is considered to be out of scope, we will not go deeper into it.

As you may expect, for B2B Commerce we will be primarily interested in structured, volatile information, since this is obviously the category where *electronic business transaction documents* will pertain to. The upper two quadrants – dealing with semi-structured information – are less important for us, as they are mostly related to publishing, and perhaps B2C Commerce. One detail worth mentioning, is that two imaginary arrows could have been placed between the upper and lower half of this schema: one arrow directed up from structured to semi-structured, titled "Styling and Transformation – XSL(T)", the other directed down, titled "Data-extraction" or even "Data/Text mining". However, these arrows would not always be applicable, so they have been left out.

The lower-left quadrant, about the permanent storage of structured data, is more interesting. After all, as we have already mentioned, the information contained in transactional documents is likely to originate from and finally end up in some sort of "database", although this will not always be an "XML-database". This quadrant deserves your attention because of the example, "Registry/Repository". It will soon become clear what we mean by this, moreover, Registry/Repository systems will turn out to be an essential part of the frameworks to be discussed below.

When we finally have a closer look at the lower-right quadrant, we see there has been made a distinction between **Data Transfer** and **Message Exchange**. When we talk about *Data Transfer*, we assume that both the sending and receiving application know the characteristics of the ongoing process they are participating in. The sending side knows what data has to be sent, the receiving side knows what it can expect and, so there is no need to specify "what to do" with the data. A slightly different, but more plausible case is when we have an application on one side, which exchanges data with a *database system* on the other side. An interesting example, that is given in the schema, is the collection of data in order to "feed" a data warehouse. Since the numerous sources of the data – mostly incompatible, legacy database systems – will usually be different in nature and geographically dispersed, we are gladly taking advantage of XML as the "universal data standard" here. If it can be achieved that every data source "delivers" its data in XML syntax, one big step of the data-extraction process has already been taken.

Note: It should be obvious that we have made abstraction here of all extra information accompanying the transmitted data or message to handle things like routing, references to schemas etc.

Our interest will however mainly concentrate on the second item in the Structured-Volatile Information quadrant, *Message Exchange*. The better part of the following sections will deal with this topic: message exchange means electronic communication, and electronic communication between companies is a general way of saying “B2B eCommerce”. Yet, it might have attracted your attention that two extreme examples are given: RPC (*Remote Procedure Call*) and EDI (*Electronic Data Interchange*). They are each others opposite in the way in which they express “what to do” (we could refer to this as the *call*) with the data. Therefore, we could add another distinction here: data can be accompanied by an *explicit call*, as in RPC, or by an *implicit call*, inherent to the type of data (the *document*, conforming to a specific schema) that is being sent. The first, RPC-style communication with explicit calls is usually referred to as *messaging*, or more recently as *web services*, while the second “EDI-style” could be called *document exchange*.<sup>23</sup>

In the following sections, we will mainly discuss ebXML. In essence, this framework handles the interchange of messages (business documents) between applications in different companies: how the underlying transport mechanism works, how to make sure their content is understood, how to discover what types of messages can be sent... In most cases an application located within another company is asked to do something, i.e. to produce a certain output when provided with a request. *How* this functionality is implemented is none of our interest, we just send a message and expect some appropriate response. For the most part, ebXML deals with documents of which the conformance to a certain schema denotes an implicit call. However, it is a very general framework, and could be used for almost any type of structured data exchange. Along the road, we will meet several aspects of the emerging *web services* framework, mainly formed by SOAP and UDDI. These and related matters will be discussed in detail in the upcoming chapters.

---

<sup>23</sup> We realize that these terms are used interchangeable in practice, but they have been chosen with the mere intention to clarify the ideas expressed in this paragraph.

Note: “Web services” are one of those new buzz words. It almost seems that every application that is able to react to some request arriving over the Internet suddenly turns into a web service... The reader might be a bit confused to see the relationship with documents and “document exchange”. However, from a data-exchange point of view, a web service can simply be described by the kinds of documents an application is willing to accept, and what it will return in response. Below, in the context of ebXML, we will digress two times into the world of web services: once with SOAP, and once with UDDI.

### 3.2.3 Some further observations about the scheme

Two more aspects of the classification scheme are worth considering. First, although the purpose of this scheme is to discern between various types of information and application areas, a somehow *opposite* observation can also be made: XML brings the four quadrants together! For instance, we see a convergence between database systems (structured information) and document-centric tools (mostly semi-structured information), opening up new possibilities.

Secondly, as might be apparent from the classification, XML *could* be considered as a replacement candidate for almost every message and storage mechanism currently in use. It is however to be noted, that the choice for XML as a data format (as opposed to “some” ad hoc syntax or structure) may be quite arbitrary. When we look beyond the hype, in some cases XML is nothing more than a – convenient – way to structure information which would otherwise be in some proprietary format. In general, the merits of using XML markup as a way to structure data stem purely from the availability of its associated standards – such as DOM or SAX for easy access, a DTD or Schema to provide a formal model, and XSL(T) as the “gateway to the world”. As we have indicated before, the added value of using XML is minimal when using the “stand-alone” XML 1.0 Specification, without its supporting relatives.

Note: We find two interesting examples in the new Microsoft .Net vision. With .Net, programmers are no longer confronted with the hassle of putting every single program setting in the Windows Registry, but instead they can store these settings

(mostly name-value pairs) in XML-marked up “Configuration Files”. However, this approach is not so new as it seems, because it can be seen as a return to the Windows 3.x philosophy, where everything was put into “.ini” files<sup>24</sup>. Basically, the difference is just the way these files are marked up. The second example is XLANG, an XML language for describing processes which will be discussed in the context of BizTalk. As David Wascha, BizTalk Product Manager, mentions in an interview: “The fact that XLANG is an XML-based language is really arbitrary – it could be anything.” More about XLANG in the BizTalk section.

Discussing the latest visions of major software developers such as Microsoft (.Net) and Sun (ONE) would lead us too far here, but we could say that XML is moving down the overall system architecture ladder. For many applications, accessing XML files will become almost as natural as handling simple text files is today. The only difference is the need for a parser, which will then again be assumed to be readily available, either from a runtime library or embedded into the operating system itself. As we will see below, XML’s claims to become “ubiquitous” are really not so unrealistic as they may seem.

Nowadays, a typical programming exercise may include reading and writing data from and to a file, either in pure text format or in a self-defined record format. In a few years from now, this exercise might be different: the two “options” (text or record file) might converge into one, namely reading and writing XML marked-up data, by calling a parser through an API (of course, an XML file could be seen as a text file structured into “record” elements). It seems likely that in most cases, the extension “.xml” will be replaced by something more meaningful, pretty much as the “.txt” extension today covers only a fraction of the “text” files. “So what?”, you might think, since you are probably not a programmer yourself. Well, for two reasons: first, this would mean that XML has effectively become *the* universal data format, and that you are not wasting valuable time now by reading this. But second, and most important, because it means

---

<sup>24</sup> The interested reader may have a look at his Windows directory, where he will likely find the files “win.ini” and “system.ini”, among others. Nowadays, these files normally serve little purpose, they just have to be there for reasons of compatibility – be careful not to make any changes. However, it’s an interesting exercise and should take little effort to design your own “XML version” of them.

that the data format used by the program, no matter how simple, is implicitly available from the file, or is made explicitly available in a separate schema. Especially in the latter case, *the data model is removed from the logic*, is made independent from it, so the program's data are very easily accessible by *any* other program. And *that* unleashes the power of XML.

## 3.3 Why (not) EDI ?

Note: For ease of reading, in this section the notions “EDI-” and “XML-based eCommerce systems” have been replaced by the words “EDI” or “XML”, respectively.

### 3.3.1 What is EDI ?

Contrary to what you might expect, these paragraphs are *not* only here “for completeness”. It would not be surprising if you have the intention to just skip or scan-read this section. Yet, that would be a pity, because it is always interesting to see what we can learn from the past. On the other hand, what you will find below is by no means a full description of EDI (Electronic Data Interchange). The intention is to give you an idea of what EDI is, what it is good at, and what its problems are. To decide where you want to go, it helps to know where you come from.

In general, EDI can be described as the electronic communication of business transactions between organizations. There are two major standards for EDI: ANSI X.12<sup>25</sup> in the United States and UN/EDIFACT<sup>26</sup> in Europe and the rest of the world. In addition, these standards have been the basis for various industry-specific EDI standards. UN/EDIFACT defines EDI as “*the computer-to-computer transmission of (business) data in a standard format.*” [UN/EDIFACT, 1995]. Even though it is very short, this definition captures the basic principles:

- *computer-to-computer*: no human intervention or rekeying should be required
- *business data*: the data to be exchanged, are electronic business documents (which in many cases closely resemble their conventionally printed counterpart)
- *standard format*: the format of the business documents is supposed to conform to the specifications of the standards organisation (either X12 or EDIFACT).

---

<sup>25</sup> X.12 was originally developed by ANSI, the *American National Standards Institute*, but is currently maintained by the not-for-profit organisation DISA, the *Data Interchange Standards Association*.

<sup>26</sup> UN/EDIFACT (*United Nations Electronic Data Interchange for Administration, Commerce and Transport*) is managed by CEFACT (*Centre for Facilitation of Administration, Commerce and Transport*) and UNECE (*United Nations Economic Commission for Europe*).

EDI is based on the concept of *transactions*, that comprise *messages* (business documents) in predefined formats. Broadly described, a message consists of *data segments*, which themselves are a collection of *data elements* – interchangeable, basic units of information. The main purpose of standards like X12 and EDIFACT is to specify message structures (in terms of their constituent parts) for the cooperation of different types of business processes between two companies. Typical examples are invoices and purchase orders.

The typical benefits of using EDI can be summarized as follows:

- Less paperwork and less human intervention, which makes processing and communication quicker and less error-prone.
- Automated inventory management makes it possible to hold reduced inventories, possibly vendor-managed.
- Tracking of individual units, which would be impossible by hand, is achievable.
- Automation of regular transactions enables a “management-by-exception” approach, increasing focus on “real” problems.

In general, this boils down to streamlined business processes, cost savings and improved productivity.

After freshening up of what EDI is and what it is good for, we will now have a look at some of its weaknesses, to see where XML might be able to help. Again, it is not the purpose to provide a complete comparison between XML and EDI. We will rather concentrate on facts that tell us why EDI is *not* suitable for global eCommerce, but at the same time show us that the experience from almost 30 years of EDI is too valuable to ignore.

### **3.3.2 What’s wrong with EDI ?**

Although we will not advocate the seemingly popular “away with EDI” point of view here, the fact that it has never reached wide-spread deployment indicates that it must have certain shortcomings.

Summing up *all* EDI problems would be too tedious, but some of the difficulties that are often mentioned are:

- Since the Internet was not yet commercially available when these EDI standards were developed, they define not only message formats, but also communication protocols and even certain hardware requirements. This renders implementation more complex, and requires expensive Value Added Networks (VANs). In turn, it entails the use of a compact syntax which is difficult to understand.
- EDI is based on detailed agreements between all parties, both on business and technical aspects. This makes the agreement process lengthy and thus more expensive, while preventing short-term relationships from being established. Moreover, it means that the number of partners must be relatively small.
- The message specifications themselves are very rigid, with business rules embedded into them. They are meant to be applicable to nearly all industries and businesses, taking into account the particularities of *each* of them. This results in elaborate, complex specifications, difficult to understand and taking very long to implement. At the same time however, each industry has its own implementation guidelines. This especially forms a problem for companies interacting with several industries.
- The mere fact that there are *two* major standards, X12 and EDIFACT, causes problems for companies that do business with both American and non-American companies.

The problem of the rigid transaction sets requires a bit more explanation. While this rigidity renders the structure of the incoming data predictable, which enables automated processing, it also makes implementation harder. Business processes have to be re-engineered and applications have to be modified in order to produce the data content exactly as the standard prescribes. In EDI theory, there is one standard for all industries, but in practice every industry has its own set of business rules and practices, often called a “subset” of the standard. Next, individual organisations that want to use (the subset of) the standard must agree on the *exact* format and code values to be used, so they will often “stretch” the industry guidelines to fit their particular needs – and so reduce re-engineering efforts. The belief is that “as long as two partners understand each

other, there is no problem”. However, this means that an official standard is used in an unofficial way, and every interchange becomes an individual customisation of the general standard. Interoperability towards companies that weren’t part of the original arrangement is then a lot harder to accomplish.

Note: XML doesn’t solve this problem all at once, but the big difference is that we are now aware that setting one standard for such a diversity of ways to conduct business across industries is unfeasible. As we will see, this problem can partially be solved by defining “core” data elements common to “every” business, but XML also offers an inherent advantage: it is so flexible as to allow XML documents to contain elements which the partner does not understand, so parties communicating via XML only have to agree on *common* elements.

As a consequence, the potential benefits for smaller companies do not make up for the high costs. They have no incentive to implement an EDI system unless they are “forced” to do so, for instance because one or more big customers demand it. This is often referred to as a “hub-and-spoke” model: for a big company (the hub), EDI can provide substantial savings, so it forces its suppliers (often smaller companies, the spokes) to choose between using EDI and terminating the contract. Even for the hub though, the long implementation time is a barrier. For the spokes, in the light of the previous considerations it is clear that the problem gets even bigger when *several* customers want to use EDI, as the better part of the implementation work has to be redone for *each* new partnership.

It appears that many of the problems and limitations especially become stringent in a market with many players, both large and small, where dynamic interactions take place. In today’s rapidly changing business environment, classic EDI systems lose more and more of their value.

### 3.3.3 Lessons to be learned from EDI

After reading the previous section, in which EDI seems to have been torn down to the bottom, you may find it a bit strange to pay any further attention to it. Yet, we don't want to jump to hasty conclusions. EDI has been out there for more than 25 years now, and it's not going to disappear overnight. Many companies have EDI systems up and running just fine, that's one thing, but it doesn't explain why we should bother about EDI when *designing* XML-based architectures. So what else is there to remember about EDI?

First of all, the benefits we see in existing EDI systems, are a good guideline for setting design goals for XML-based systems. If anything, we want to retain the goals reached with EDI, and in certain cases improve upon them.

However, there are many more reasons for keeping EDI in mind. Following is a (non-exhaustive) list of recommendations that build on the vast experience built up in the past 25 years.

Note: This list is partially based on the excellent XML.com article "XML and EDI: Lessons learned and Baggage to leave behind" by Alan Kotok. [Kotok, 1999]

- *Make data exchanges predictable.* This is even more important in a world where "any" other company may want to do business with you, with minimal prior negotiation. Therefore, you should identify any possible constraints and put them into the schema. Indeed, this is one of the reasons why we have paid special attention to modelling.
- *Provide unique identifications* wherever possible, or you could literally lose track of your business transactions. The availability of namespaces will be a great help here.
- The idea of *common, interchangeable data elements* (even as simple as a date or an address) is an important step towards interoperability among different industries. We will see that ebXML's *Core Components* are very similar to this.
- *Acknowledge everything*, so every party, at any time is *sure* about the state a transaction is in. For instance, as we will see when we discuss ebXML's transport

mechanisms, there is a difference between sending a receipt acknowledgement and confirming that the received document is understood!

- *Consider redesigning business processes.* Although the need for re-engineering is usually seen as an obstacle, an interesting lesson to be learned from EDI history is that significant productivity gains are not achieved by simply pulling the automation trigger. For substantial benefits to occur you have to do more than using electronic versions of your paper documents: you should ask yourself if and how you can improve them.
- *Avoid rip-and-read processing.* Related to the previous matter is that the adjective “electronic” should not drop out once a transaction has entered company boundaries. In too many companies, incoming EDI data is printed and rekeyed into other systems – which has undoubtedly something to do with EDI being imposed by business partners, of course. Here, XML offers an advantage in that it may be used for internal systems as well.
- We will see that the *use of repositories*, where the “schemas” for message formats are stored for common access, is an important aspect of XML-based frameworks.
- *Stable standards are needed.* Industry standards are nearly always a compromise. Therefore, no matter how excited organisations are about the prospects, they should take their time to get consensus or they will end up with rubbish. This puts the current standards rush in a different light, of course.
- *Take care of security*, if you want it to take care of you. This is much more important in an open Internet-environment than it was in closed VAN-based systems. Security is an issue that should be included in the foundations, not something to add at the end.

### **3.3.4 Down to earth: a well-balanced attitude**

In this discussion about EDI, we have tried to escape the cloud of dust surrounding XML. It should be clear now that an attitude like “EDI is dead, long live XML” is wrong. EDI offers important insights for the development of new eCommerce frameworks, not in the least because it *still* proves to be valuable – in situations that correspond to the context in which it was developed. This also explains why EDI is *not*

expected to “die” within the next couple of years: there is little or no reason why an XML-based solution should replace an existing, properly functioning EDI solution – and this is *not* only because the large investment in EDI first has to be recovered. Even a real XML advocate has to be aware that EDI *does* have its strengths. By seeing XML as a substitute for EDI, one is simply doing away with the experience built up in the past 25 years and will end up in making many unnecessary mistakes. However, as summed up above, there are also disadvantages, mainly resulting in EDI not being suitable for general deployment. It seems that it is a better attitude to see XML as “the *new* EDI”. The underlying idea is: even if you start building from scratch on the *technology* side, this still doesn’t mean that the underlying *business processes* change completely, so the experience built up in this area should be taken into account.

When we discuss the ebXML framework, we will discover that many of its principles are actually based on more recent work from within the EDI community, namely the *Open-edi Reference Model*. We will also see that ebXML has special attention for small companies, which currently cannot afford to build an EDI solution. However, “backwards” integration with existing EDI systems is made as easy as possible. The aim is to offer a system that has the same or more to offer as EDI, but without its drawbacks.

## 3.4 A Generic B2B Framework

### 3.4.1 Introduction

In this section, we will try to identify separate building blocks, which could together constitute a complete framework. For each block, we will indicate its function, and certain requirements for it. We will try to do this in very general terms.

A general requirement on all blocks is that they should be *extensible*, meaning that they are flexible enough to go along with any future developments. This also means that they should be as *independent* from each other as possible: for instance, a transport mechanism should not affect the content of the documents it carries, nor should it be affected by these documents of course.

The problem with describing a *generic* framework of building blocks for electronic business is that for a *given* framework, the blocks may be respectably independent, while this still doesn't mean that they can be simply replaced by a corresponding block from a different framework. A function that is fulfilled by one block in framework A, may be fulfilled by a different block in framework B. Although this does not *have* to be a problem in practice – especially *overlaps* in functionality can be quite easily solved in principle – it makes it hard to describe the functionality of a “generic” framework. It appears that much depends on the premises on which a framework is built.

For these reasons, we will keep this section quite short. In the next subsection, we describe the *eCo Framework*, an initiative that has tried to develop a conceptual framework that provides a very general overview of an eCommerce system, in which different, principally independent layers are defined. Following, we will try to build our own, more minimal set of building blocks. In the last subsection, a different approach for a general description of electronic business is provided, which actually stems from a reference document provided on the ebXML.org website.

### 3.4.2 An example: The eCo Framework

Certain framework initiatives have tried to build such a *generic* framework. A good example is the *eCo Framework*, which has been created by CommerceNet (see [eco.commerce.net](http://eco.commerce.net) for detailed information). It provides a method for companies to negotiate how they can conduct business. This negotiation takes place in the context of a conceptual framework, which defines seven layers, each covering a different aspect of the interaction. In addition, the eCo Framework provides a type registry system for each layer, which can be queried to discover the element types in the layer. eCo's main concept is that transactions consist of a series of *exchanged messages* or *documents*: the last four layers of the framework focus on these documents, while the top three deal with discovering products or services. Figure 4 shows the layers of the eCo Framework:

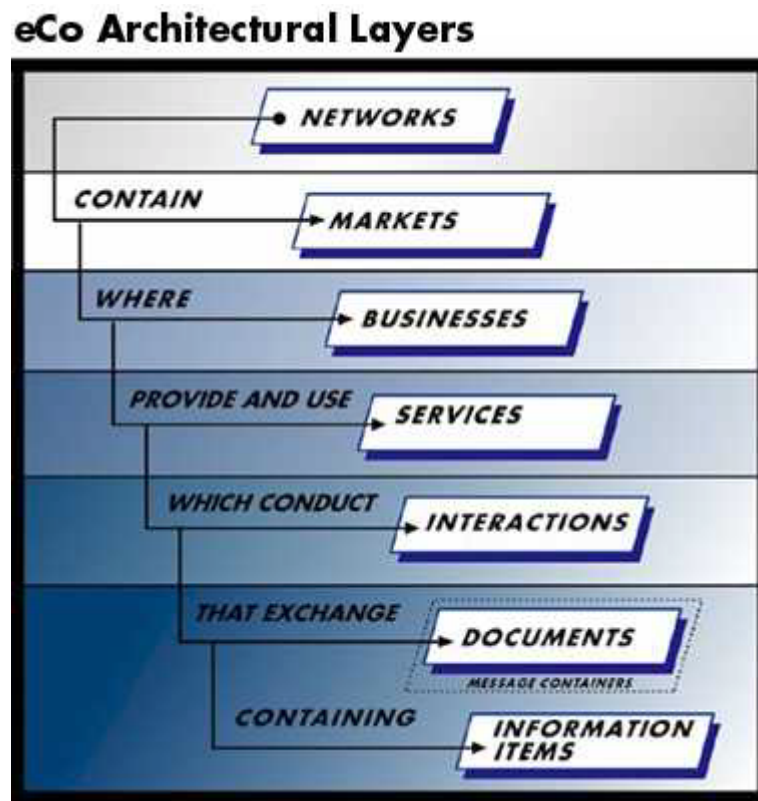


Figure 4: The eCo Architecture

(Source: [eco.commerce.net](http://eco.commerce.net))

The 7-layer architectural model can be described as follows (the term “list” should be considered very generally):

1. The **Network** layer represents physical networks for eCommerce systems, such as the Internet. Within a network, a list of *Markets* may be available – based on a products classification scheme, for instance.
2. The **Markets** layer contains online marketplaces where one or more businesses provide goods and/or services. On a marketplace, a list may be available of the participating businesses.
3. The **Business** layer gives information on a particular business, such as the products or services it offers and where it is located. This is the level on which general business transactions are defined.
4. The **Services** layer is the first level for describing these interactions. Here, a business gives information about all the eCommerce interactions it may take part in.
5. The **Interactions** layer details on the interactions, by describing the different steps in it in terms of document exchanges.
6. The **Documents** layer provides information on the type of documents that are exchanged, which are either standardized or company-specific.
7. The **Information Items** layer, finally describes the data elements that documents are composed of.

Apart from the central role that document exchanges are playing, each layer is backed by a *registry* that contains the type of information used in that layer. It should be pointed out that although the framework seems to define a “separate” registry for each layer, there is no reason why this could not be implemented by a single registry system. Although this framework is very general, we will see that most of these layers will be implemented by the ebXML architecture.<sup>27</sup>

---

<sup>27</sup> As a matter of fact, after having described this architecture, we have found a document on the ebxml.org website, describing the “mapping” between the eCo framework and ebXML. From this document, it appears that most of the eCo concepts actually *have* an ebXML equivalent. We may refer to the *Core Components* for instance, which

### 3.4.3 Building blocks

#### 3.4.3.1 Transport

Document exchanges are the key aspect of the eCo framework, and in fact of every electronic business system, so the need for a transport mechanism is obvious: after all the exchange of documents needs a “wire” to run through. However, several aspects of this transport mechanism need more clarification.

On the W3C website, there is a very interesting comparison of XML transport protocols, to be found on <http://www.w3.org/2000/03/29-XML-protocol-matrix>.<sup>28</sup> It provides a good overview of the functions (called *facets*) we may expect a transport system to have. These *facets* can thus be seen as putting requirements on our transport layer. A summary of the definitions of the facets is given below:

- **packaging** (or **enveloping**): construction of multi-document messages including non-XML documents as attachments.
- **routing**: message forwarding from agent to agent depending on attributes of the message
- **remote procedure**: *explicit* support for sending requests to a remote system to execute a designated function, method, or procedure defined by an application using the protocol rather than functions defined in or by the protocol itself.
- **serialization** (or **marshalling**): format for storing primitive data types (int, char, string), structures, and matrices
- **protocol**: definition of requests and responses, implying state transitions and defined error messages when state transitions fail
- **extensibility**: allowing features and functionality to be added beyond what is predictable (generally, the use of namespaces and specification of what to do with unknown namespaces ensures extensibility).
- **security**: rules on how data should be digitally signed to prove authenticity and authorization of requested actions

---

<sup>28</sup> One has to be careful using the information contained herein, as the comparison is a bit outdated. For instance, it doesn't take the SOAP with Attachments specification into account.

The following facets are also defined, but we might as well put them into a different block (see below) – this illustrates the “problem of the different framework premises” mentioned above.

- **transactions (ACIDity)**: atomicity and state clarity that allows an action to be partially completed and then undone. (ACID == atomicity, consistency, isolation and durability)
- **business process**: modelling of business workflow conventions

Additionally, we may add a (possible) requirement for **reliable messaging** (assuring that a message reaches its destination, normally using acknowledgements), and of course, **platform independency**.

### 3.4.3.2 Registry

The transport and registry system together seem to be the most essential elements of a framework. Again referring to the eCo architecture, a framework will always have to define some *registry* system where any information that needs to be shared is placed. This can be done per (group of) levels, or there may be one overarching registry. It is hard to define general requirements on the registry, since the specific framework in which it is used has serious implications on the required functionality (interfaces), and the types of information it has to contain.

A few requirements are: the registry should of course be **always online**, may need to have to **interact with other registry systems** (possibly defined in a different framework), and therefore should be able to **use the transport mechanism(s)** as defined by the framework. Furthermore, **flexible querying functionality** should be provided (the importance of this requirement depends on the types of information in the registry).

### 3.4.3.3 Profile Description

For this block, we might refer to eCo’s *services* layer, which describes the interactions that a business may participate in. In other words, it should be defined what **kind of electronic business** a company is capable of – both from a **business and a technical**

point of view (services offered, transport mechanism supported,...). If a framework deals with this aspect, it should also provide a way for interested parties to **discover the these descriptions**, and to make sure they can **understand and use** them. Additionally, it might be interesting for a business to define how it can participate in a **multiparty interaction**.

#### 3.4.3.4 Document Definition

Documents may be mostly assembled from **common items**. If the framework also addresses the definition of documents, it should ensure their **interoperability** and **reuse**. Therefore, documents should be **industry** or **cross-industry standards** as much as possible (or based on such elements), and should be **extensible** (which also means that support for **versioning** is important). An additional point of attention might be the backwards **compatibility** with legacy **EDI** systems.

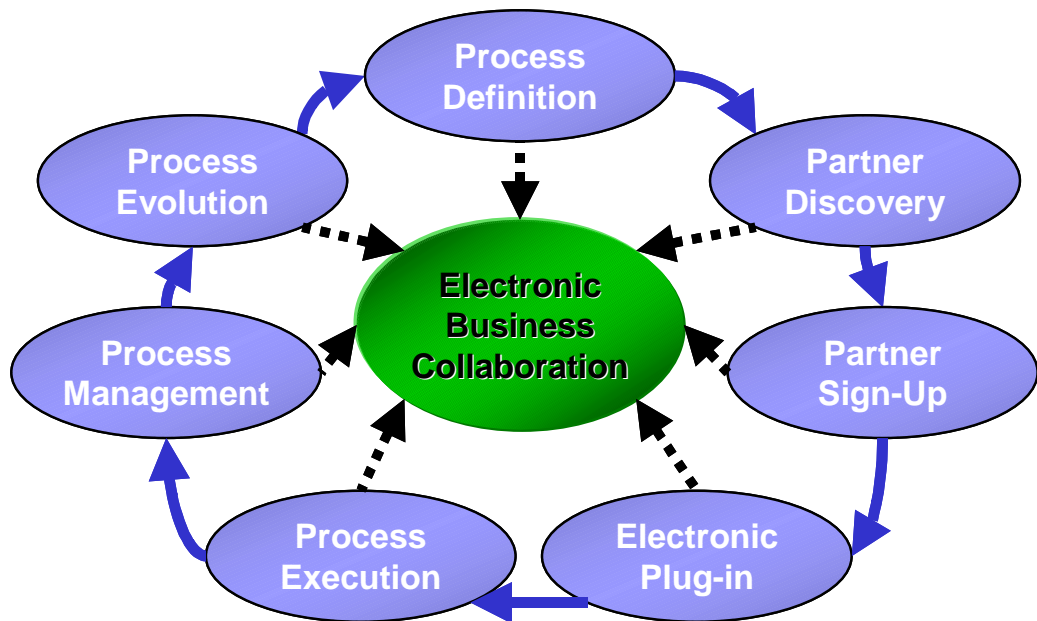
Note: As we will see, ebXML normally assumes that document definition (by using generic data elements) goes together with **business process modelling**. The document definitions may however be provided by other (vertical) frameworks (see below).

#### 3.4.3.5 Additional Functions and/or Blocks

Additional functions which may be provided by the above or other blocks, include **transaction semantics** (we refer to the ACIDity already indicated above), **security** (authentication, authorization,...), and also **legal aspects**.

### 3.4.4 An additional view on electronic business

A reference document on the ebXML.org website contains another view on electronic collaboration, which is depicted in Figure 5. Many of the above-mentioned aspects return in this figure, and it also serves well as a transition to the following section, where we will meet this picture again, but then to introduce the ebXML specifications.



**Figure 5: Circular view of Electronic Business Collaboration**

(Source: ebXML Business Process and Business Information Analysis Overview v0.7)

This circular view of Electronic Business Collaboration stresses the fact that it is a multi-step process, of which every stage should be passed through by companies engaging in a collaboration project. Although a circle has per definition no start nor end, the most logical place to depart from is “Process Definition”. We will briefly explain the successive steps:

1. **Process Definition:** Using *Business Process* and *Business Document Analysis*, the necessary eCommerce processes are selected, modelled (using some modelling method), and recorded in a standard format.
2. **Partner Discovery:** This concerns the search for potential trading partners and the examination of their “profile descriptions” – the services they offer.
3. **Partner Sign-up:** When a trading partner has been found, the details of the interaction (business and technical) should be negotiated, and after agreement each partner signs up for a specific role.
4. **Electronic Plug-in:** Using the definitions of the interactions of the previous phase, the parties set up their applications for the intended communication.
5. **Process Execution:** The interaction between the systems of the two parties actually starts, according to the configuration of the previous step. This means

that the business documents are exchanged in order to execute the agreed interaction.

6. **Process Management:** This phase is applicable *during* the period for which the interaction agreement is valid, to monitor it and ensure that execution takes place as agreed.
7. **Process Evolution:** Interactions are usually agreed for a certain period or number of transactions. When the interaction reaches its end, processes may be redefined and agreements be renegotiated.

Note: Contrary to the previous subsection, the above description of eBusiness Collaboration already contains certain typical ebXML aspects. This may be seen as a preliminary introduction.

## 3.5 ebXML: A “Complete” Framework

### 3.5.1 Introduction

“ebXML” stands for “electronic business XML”, and is meant to become a global standard for electronic business. Its goal is to enable anyone, anywhere to do business with anyone else over the Internet, which is expressed by its mission statement :  
*“Creating a single global electronic market™”*<sup>29</sup>

To many people, it might be quite unclear who exactly is behind the ebXML initiative, so this question is more important than with other standards. After all, history provides abundant proof that the “quality” of a product or service is not the only success factor, but even more, the way it is marketed – we can merely hope that these two aspects coincide. There are at least three possible answers to this question, which each shed a different light on the importance and viability of ebXML:

- UN/CEFACT and OASIS<sup>30</sup>, two major non-profit, international standards organisations, of which the first covers worldwide policy and technical development concerning electronic trade facilitation, has developed the EDIFACT standard, and is one of only four organisations in the world that can set *de jure* standards.
- The same two organisations can also be seen as an enormous group of vertical and horizontal industry consortia and standards bodies (IETF, OMG, DISA, CommerceNet,...), government agencies, companies and individuals from all over the world. In principle, they all support the standards their respective organisation sets, and in this way form an enormous user base. Among these companies are both large organisations with experience in EDI, and smaller companies interested in a cost-effective way to do electronic business.

---

<sup>29</sup> *“creating a single global electronic market”* is a trademark of the ebXML Working Group (!)

<sup>30</sup> UN/CEFACT: United Nations Center for Trade Facilitation and Electronic Business, [www.uncefact.org](http://www.uncefact.org) ; OASIS: Organization for the Advancement of Structured Information Standards, [www.oasis-open.org](http://www.oasis-open.org)

- High-tech companies such as IBM, Sun Microsystems, Commerce One and Oracle, whose engineers are actively working on the technical side of the ebXML standard.

A fundamental characteristic of ebXML in its pursuit of interoperability is to move beyond the mere exchange of “electronified” business documents by adopting a *transaction-* and *business process-oriented* approach.

ebXML can be seen as a “unified global EDI standard”, for companies of *all* sizes, both large international companies and SMEs (Small and Medium Enterprises), in *every* industry sector. Since we have seen that EDI leaves a gap to fill in, one of ebXML’s specific design goals is to lower the entry barrier to electronic business for SMEs. Of course, this special attention is not simply an act of “generosity”: Large businesses, who have been active in EDI for years, see an opportunity in linking e-business collaborations to a wider base of trading partners. One of the important aims following from this, is to provide “out-of-the-box” interoperability, in the form of “shrink-wrapped” applications that smaller companies can plug in to their ICT infrastructure. However, this also puts serious requirements on the modelling aspects of ebXML, in that the business processes must be detailed and specific enough for immediate application. Consequently document specifications from other (vertical) standards organisations have to be usable within the ebXML framework.

Cooperation with other frameworks is therefore vital, and is the reason why ebXML sees itself as *complementary* (not competitive) to other B2B initiatives such as the SOAP and UDDI standards and the RosettaNet and OAGIS frameworks. SOAP and UDDI and their relationship to ebXML will be covered in quite some detail “along the road”. As compared to other frameworks, ebXML is neither a *vertical* (industry-specific) standard, nor is it merely *horizontal* (cross-industry): it wants to be a *horizontal* standard that can be developed in depth for use within vertical industries. The purpose is that other frameworks (such as *RosettaNet*) can “plug into” ebXML.

The ebXML initiative was launched in November 1999 with an 18-month development timeframe, so the final specifications will be finished in May 2001 – which is, at the

time of writing, within a month. With the exception of the *Technical Architecture Specification*, which is final, the specifications are still in review. Although this means these writings will be mostly based on draft specifications, we believe –hope– that they have sufficiently matured<sup>31</sup> and that changes in the final versions will be mostly editorial or technical details. Detailed information and specifications can be found on the ebXML website, [www.ebxml.org](http://www.ebxml.org).

In the following sections, we will first have a look at some of the requirements and problem domain for ebXML. In a next step, we will consider some of the background of ebXML, and use that information to divide the various specifications into two groups: one dealing with business related aspects, the other handling the technical side. From this, the role each of the specifications is playing should become clear. Following, there is a more thorough discussion of the specification documents.

Note: While an overview of the Project Teams which have made up the various specifications would certainly be useful to people wishing to contribute to the initiative, it has limited value for companies who want to start an implementation. On the other hand, information about how the ebXML specifications have come into existence can provide a better insight into the relationships between the specifications. Therefore, we will not discuss the tasks of the Project Teams separately.

It should also be clear that the ebXML specifications form a huge amount of information, which we cannot all discuss here. Therefore, we will try to use schematic representations as much as possible: while we will not explain all basics that are immediately clear from the figures, this gives us the opportunity to go a little deeper into the most interesting matters.

---

<sup>31</sup> To be specific, the Business Process, Collaboration-Protocol Profile and Agreement, Message Service, Registry Information Model, Registry Services specifications are currently in Public Review, while the Core Components and Requirements specifications are still in the hands of the Quality Review Team

## 3.5.2 ebXML Requirements and Problem Domain

### 3.5.2.1 Requirements

After the ebXML initiative was officially established by UN/CEFACT and OASIS, the first task was of course to develop a coherent set of requirements, which could serve as the foundation for the specifications to be developed by the other project teams. Version 1.0 of the Requirements document was approved in May 2000, but in the meantime, there is an updated version 1.04 which is now in review with the Quality Review Team.

The Requirements Specification first gives a general introduction on ebXML, which roughly corresponds to what we have discussed above. The vision of ebXML is expressed as follows:

The ebXML vision is to deliver:

"A single set of internationally agreed upon technical specifications that consist of common XML semantics and related document structures to facilitate global trade."

ebXML clearly intends to be fully compliant to the W3C XML standards, which have been covered in the previous part of this thesis. Essentially, they form the basis on which ebXML builds its models. In addition, the interoperability goals and the focus on facilitating the transition from traditional EDI are mentioned. There is also an intention to submit ebXML to a standards organisation upon completion.

The Requirements Specification, which is the work of the Requirements Project Team, mainly sets the stage for the other project teams<sup>32</sup>, which each cover a different aspect:

1. Technical Architecture (TA)
2. Business Process (BP)
3. Core Components (CC)
4. Registry and Repository (RR)

---

<sup>32</sup> This will also be the order in which the specifications of these project teams will be discussed, later it will become clear why.

5. Trading Partner (TP)
6. Transport/Routing and Packaging (TRP)
7. Security<sup>33</sup>
8. Proof of Concept (POC)

Note: Instead of mentioning the full name of each team every time, in the following the abbreviations (between brackets) that are indicated here may be used – most of which are common in ebXML “parlance”.

Additionally, there are three support teams, for Quality Review, Marketing Awareness and Project Management.

The rest of the document mainly contains *Business* or *Technical Requirements* for many aspects of ebXML, which we will come across in the other specifications, so we will not cover them here.

### **3.5.2.2 Problem Domain: Business-to-Business Collaboration**

Enabling every enterprise to start doing electronic business with any other enterprise involves a lot more than connecting order-entry and purchasing systems. Before companies can start exchanging business documents (orders,...), they have to make sure they understand what the other party is saying (syntax, semantics,...) so they can find out how the other party’s business processes are composed. It is clear that there are many things to agree on. Before all this however, companies first need a way to find each other electronically: they need to get hold of descriptions of businesses and search them for potential trading partners. But for descriptions to be searched, their syntax and semantics have to be understood...

While the above only sketches part of the problem domain, it gives an idea of the huge scope of a project like ebXML, that has set itself the goal of providing a complete “end-to-end” solution. We will see that the ebXML specifications will try to facilitate the task by leveraging established standards such as HTTP, TCP/IP, MIME, SMTP, FTP, UML

---

<sup>33</sup> The Security Team is not referenced in the Requirements Specification v1.04, but is mentioned on the website and has published a “Technical Architecture Risk Assessment” document on April 20, 2001

and of course, XML. Together, the ebXML specifications will cover almost the entire B2B collaboration process we have discussed above, as is clearly indicated in the following figure.

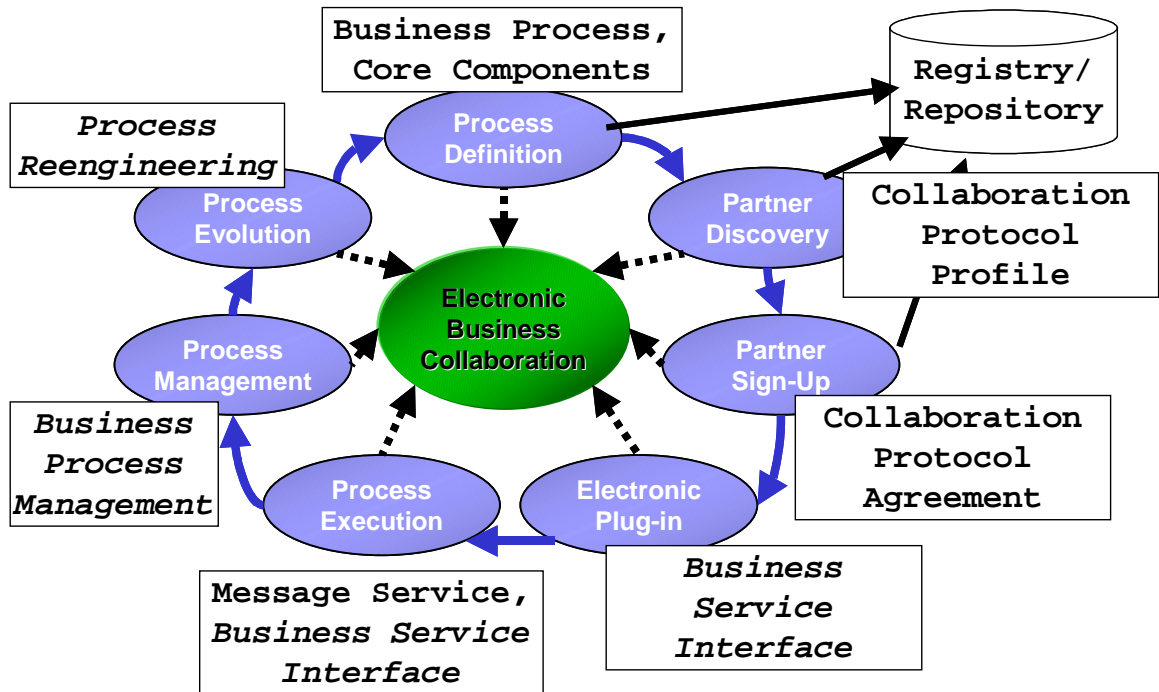


Figure 6: B2B Collaboration Process, with ebXML Specifications fitting in (Source: Presentation on "ebXML Architecture" by Anne Thomas Manes, [www.ebxml.org](http://www.ebxml.org))

The illustration is also valuable in that it clearly indicates the function of the various specifications and certain mutual dependencies. Only the last two phases (*Process Management* and *Process Evolution*) are out of scope for ebXML. It is arguable that ebXML's most important advantage (as compared to many other initiatives) is that it takes this activity into account. Also, as we will see, ebXML's Registry/Repository fulfils an essential role in the entire course of action. The discussion of these specifications will be the centre of our attention in the subsequent sections.

### 3.5.2.3 B2C and EAI

Although this thesis focuses on B2B eCommerce, you may ask yourself if ebXML only addresses the needs of collaboration *between businesses*, and not business-to-consumer

(B2C) interaction or Enterprise Application Integration (EAI). First, it is important to realize that B2C is only the last (but vital) link in the supply chain, so B2B can be seen as enabling the “pull” of goods through that chain. Second, the increasing number of mergers and acquisitions, together with the rise of *virtual enterprises*, requires companies to adopt a long-term integration strategy: a competing company today can be your partner tomorrow. This means that the distinction between company-internal application integration and B2B eCommerce becomes more vague.

In this context, the scope of ebXML is clarified by the Requirements specification (v1.0):

“The scope of the ebXML business requirements is to meet the needs for the business side of both business to business (B2B) and business to consumer (B2C) activities. Consumer requirements of the B2C model are beyond the scope of the ebXML technical specifications. Application-to-application (A2A) exchanges within an enterprise may also be able to use the ebXML technical specifications, however ebXML A2A solutions will not be developed at the expense of simplified B2B and B2C solutions.”

So, in short, ebXML concentrates on B2B collaboration, although B2C applications and EAI may also benefit from the initiative.

### **3.5.3 ebXML Architecture**

#### **3.5.3.1 Background: Open-edi and Unified Process**

We have mentioned above that one of the principles of ebXML is to build on existing standards. One of these standards is UML, the *Unified Modelling Language*, an object-oriented analysis and design language from the Object Management Group (OMG). As we will see, ebXML deals also with *Business Process and Information Modelling*, which makes use of UML. However, there is more to a modelling methodology than just a “diagramming language” like UML.

Therefore, ebXML recommends<sup>34</sup> to use the *UN/CEFACT Modelling Methodology* (UMM), which is the result of work done by UN/CEFACT itself, and Rational Rose. Since the second part of the 1980s, within UN/CEFACT efforts have been done to lower the barriers of traditional EDI, which resulted in the *Open-edi Reference Model* (ISO 14662). The developers of this model believe that EDI's problems can be solved by introducing standard business scenarios and support services. This means that electronic interaction between organizations can be done without prior agreement by separating business and technology aspects of business transactions (see below). Rational Rose, on the other hand, has developed a UML-based modelling methodology called *Unified Process*. In general, ebXML uses UML because of its versatility, rich modelling expressiveness and visualization facilities, which make modelling both powerful and relatively easy. Moreover, UML provides an extension mechanism so that domain specific, object-oriented metamodels can be defined, and the models can be converted to XML syntax using XMI<sup>35</sup>. Finally, it might be worth noting that another OMG standard, the Object Constraint Language (OCL), is advised by UMM for a formal description of the business rules – the alternative is to use natural language.

Now, as it already appears from the name, the model developed by UN/CEFACT serves only as a *reference*, so it is not applicable for direct use. However, it has had substantial influence on the development of eCommerce specifications, among which is the ebXML initiative. Hence, UMM should be seen as the customisation of the Unified Process methodology to meet the *business process* modelling requirements defined by the Open-edi Reference Model.

---

<sup>34</sup> ebXML does not *require* implementers to do Business Process and Information Modeling. However, *if* modeling is done, then UMM must be used.

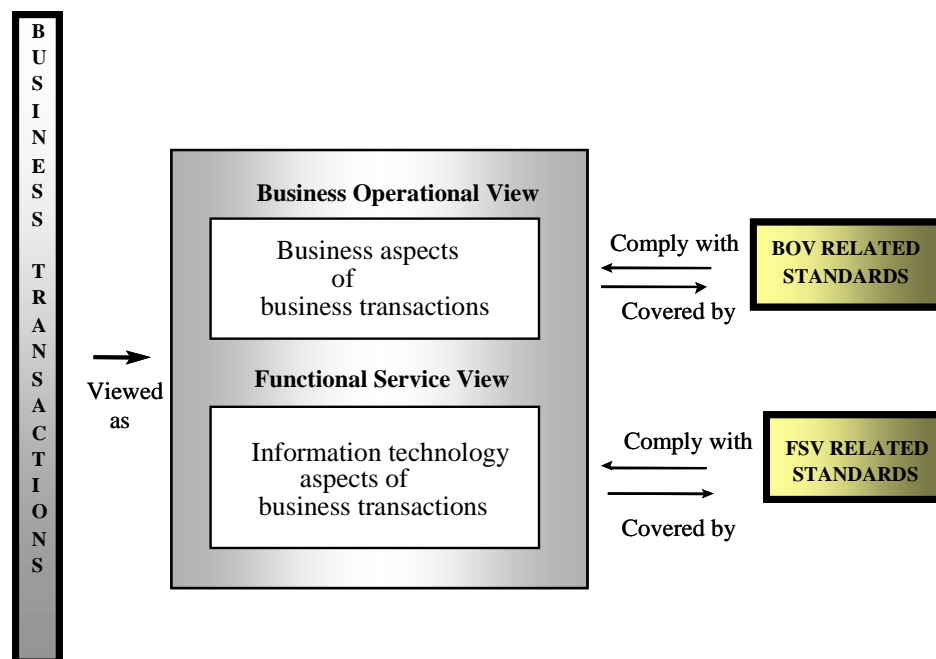
<sup>35</sup> XMI (XML Metadata Interchange) is an XML application to “enable easy interchange of metadata between modeling tools (based on the OMG-UML) and metadata repositories (OMG-MOF based) in distributed heterogeneous environments”. For more information, see [www.omg.org](http://www.omg.org).

### 3.5.3.2 The Open-edi Reference Model

In the next section we will give an overview of ebXML's architecture, but here we will first describe the basic aspects of the Open-edi Reference Model, which is discussed in the ebXML Technical Architecture Specification to explain UMM.

The Open-edi model tries to pin down the relevant aspects of business transactions by clearly distinguishing between an operational and a functional view on them:

- The *Business Operational View (BOV)* addresses the business data semantics, with the associated rules (conventions and agreements) for business transactions. These semantics are often referred to as “business practices”. The BOV describes transactions in terms of “role players” and “scenarios” (sets of operations).
- The *Functional Service View (FSV)* deals with the supporting IT services for these transactions (service capabilities, service interfaces, protocols and messaging service<sup>36</sup>). It can also be called the “technical” view.



**Figure 7: Business Transactions as seen by the Open-edi Reference Model**  
(Source: ebXML Technical Architecture Specification)

<sup>36</sup> These terms are not explained here, as they will be thoroughly discussed in the following sections.

In fact, the importance of the distinction between Business-related and Technical-related information is the very point Open-edi wants to make. UMM uses this distinction for purposes of interoperability: if two partners (role players) both use the same BOV scenario, and if their systems conform to the standards defined in the FSV, they should be able to engage in a transaction. In practice, technical interoperability will be easier to achieve than business practice interoperability.

Of course, the BOV and FSV standards have to be defined first, and for us, this is where ebXML enters the story: although the developers almost seem to keep it quiet,<sup>37</sup> **ebXML should be seen as a realization of the Open-edi Reference Model.** While there is *absolutely no reason* to view it as “EDI using XML”, being aware of the Open-edi roots is of great help in understanding the overall system. *This* is why ebXML can be said to “build on the experience and strengths of existing EDI knowledge, *and* avoid its weaknesses”.

### 3.5.3.3 UN/CEFACT Modelling Methodology (UMM)

Note: Although it would lead us too far to describe the full UMM methodology here, we will try to give a short overview. The purpose is to clarify the relation to ebXML’s BOV, which is described in the following section.

UMM prescribes a software and standards development methodology that iteratively runs through four phases. These phases are *Inception*, *Elaboration*, *Construction* and *Transition*. However, UMM itself only encompasses the first two phases, since Construction and Transition are performed by software vendors and end-users, respectively.

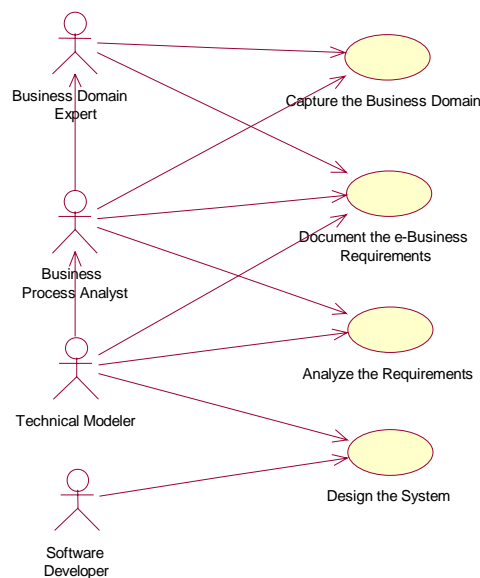
---

<sup>37</sup> Reference to Open-edi is barely made in the Technical Architecture specification (v1.04). We might mention a discussion on the ebxml-architecture mailing list, starting with: “I recommend that we remove Section 9.0, Open-edi. This is an ebXML arch doc, it just seems out of place to devote so much upfront presentation to Open-edi.”(<http://lists.ebxml.org/archives/ebxml-architecture>). As a personal note, I must admit that I strongly disagree with this vision – many people trying to understand ebXML will lose valuable time because of this.

During these phases, there is an iteration of several *workflows*, which produce a set of deliverables. The Inception and Elaboration phases focus on understanding the business needs to produce *business scenarios*, *business objects* and *areas of business collaboration*, and involve four workflows:

1. **Business Domain Modelling**, where a generally agreed-upon understanding of the business domain should be achieved, and some high-level business requirements be derived.
2. **eBusiness Requirements**, where a subclass of the work done by the previous workflow is elaborated, to provide more detailed requirements.
3. **Analysis**, with the purpose of translating the requirements into a specification ready for implementation by software developers and message designers.
4. **Design**, where sets of *design patterns* are used to construct the business scenarios and eBusiness collaboration model specifications.

The following *use case diagram*<sup>38</sup> gives a visual representation, which includes the *actors* involved:



**Figure 8: Use Case for UN/ CEFACT Modelling Methodology**  
(Source: UN/CEFACT Modelling Methodology specification)

<sup>38</sup> For more information about UML-specific terminology, we refer the [omg.org](http://omg.org) website. Here, and in the subsequent section, we will assume the reader is familiar with basic UML terms.

The deliverables of each phase include many UML diagrams (such as *use case diagrams*, *activity diagrams*, *class diagrams*,...), but these will not be discussed here as we will meet them in the BOV. UMM also supposes that the deliverables will be held in a *repository*, which will prove to be an important aspect of ebXML.

#### **3.5.3.4 ebXML Business Operational View (BOV)**

Contrary to what its name would let you suspect, the *ebXML Technical Architecture* specification includes a brief section about the Business Operational View. This is the area covered by the (recommended) UMM methodology. The whole idea is that even though organizations conduct their *business transactions* in a highly variable manner, their business practices can be described as a composition of more generic *business processes*. The result of applying the UMM methodology will be the ebXML *Business Process and Information Models*, which allow partners to detail a business scenario<sup>39</sup> in a consistent way. These meta models will then be candidates for standardization.

Figure 9 zooms in on the BOV part of the Open-edi model to show how the UMM methodology is applied:

---

<sup>39</sup> As defined by the ebXML Glossary v0.95, a *scenario* is “A formal specification of a class of business activities having the same business goal.”

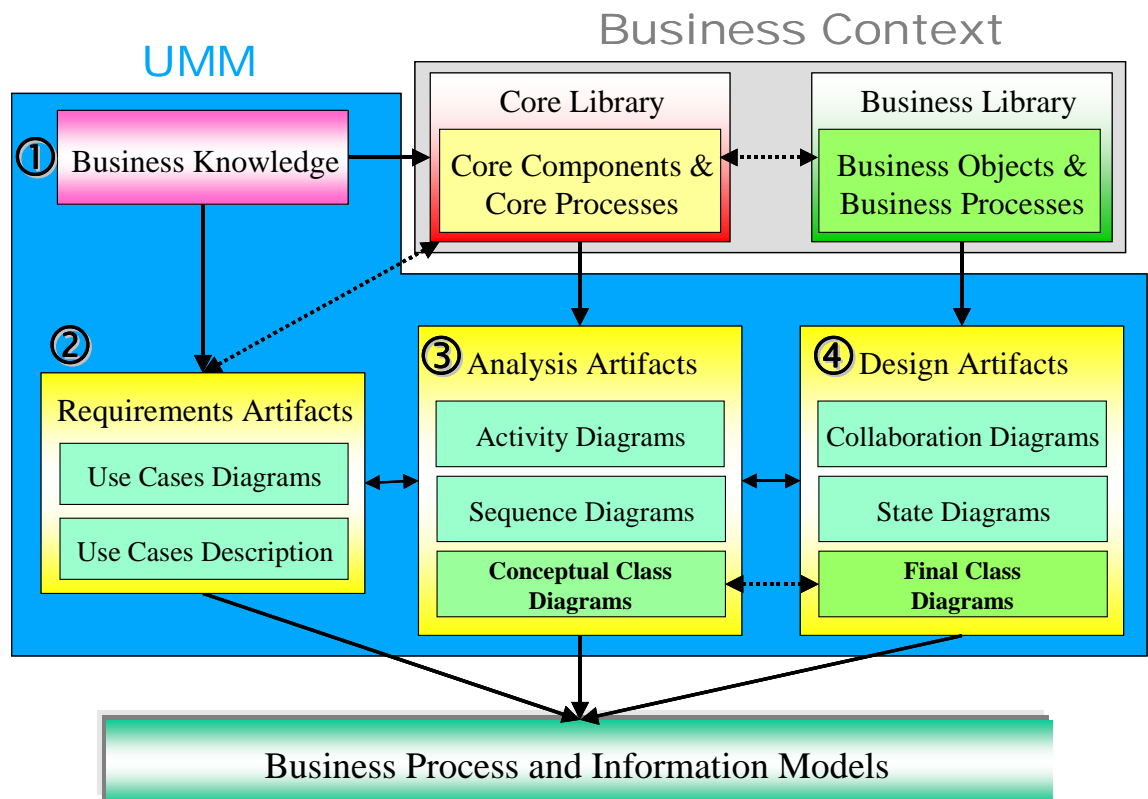


Figure 9: ebXML Business Operational View

(Source: Adapted from a presentation by Klaus-Dieter Naujok on ebXML, for the W3C Web Services WorkShop, w3c.org)

First, the gray area shows the “business context”. Its left part, the **Core Library**, contains generalized information captured from the **Business Knowledge**. These are data and process definitions at an elementary level, expressed in business (non-technical) terminology. The Core Library is said to be the “bridge” between the *business- or industry-specific language* and the (intended) *neutral* knowledge in the meta models. The other part of the business context area, the **Business Library**, contains both industry-specific and cross-industry common *Business Objects* and *Processes*. In general, the Core Components can be considered as semantically neutral data elements, for which the context is provided by the Business Processes in which they are used, in order to build the eventual business messages. More details will be given when we discuss the specifications provided by the *Business Process* and *Core Components* Project Teams.

In the blue-backgrounded part of the BOV, we recognize the four workflows<sup>40</sup> from the Inception and Elaboration phase of UMM, albeit under a slightly different name. As we have already discussed what they are about, we will now focus on the deliverables (the “artifacts”) that spring from each phase.

1. **Business Knowledge**, is the information gathered about the business problem domain. The overview shows no artifacts, but according to the UMM Specification, high-level *package*, *class* and *use case diagram* can already be derived here.
2. **Requirements Artifacts** are *Use Case Diagrams* and *Descriptions* of the (sub)problem. The dotted double arrow means that Core Components should be used when available, and new components can be created otherwise.
3. In the third phase, **Analysis Artifacts** are created, which comprise *Activity* and *Sequence Diagrams* for the selected business processes. The *Conceptual Class Diagram* mentioned here, is a free structured diagram illustrating how business documents (messages) related to these processes are exchanged. Again, the arrow means that common Core Processes may be referenced here.
4. Ultimately, in the **Design Artifacts** the *Collaboration Diagrams* are provided, which describe the interactions between all objects, i.e. the messages that are sent. *State Diagrams* may also be created. An important step here is the use of the Business Context information to harmonize the description of concepts, types and classes in the *Final Class Diagram*. As indicated by the arrow, this is done using the Core Business Objects and Processes.

It should be emphasized that the arrows between the blue and grey part are important: UMM, and therefore ebXML, achieves interoperability by using the components from the Core and Business Libraries in the diagrams and models wherever possible. The resulting artifacts form the ebXML compliant **Business Process and Information Models**, and together with the core components, they will be put into a *Registry*, which brings us to the Functional Service View.

---

<sup>40</sup> The blue background and the numbers (1 to 4) on the BOV overview have been added for clarification.

### 3.5.3.5 ebXML Functional Service View (FSV)

So far, all the models and specifications we have seen above in the Business Operational View, were completely implementation-independent and protocol-neutral. However, the *Functional Service View* shows that the business process information contained in those models will be used in the actual implementation. This FSV in fact gives an overview of the remaining specifications, from the *Registry and Repository*, *Trading Partner* and *Transport, Routing and Packaging* Project Teams.

The description of the FSV will be quite short. As we have not discussed the internals of each specification yet, we will consider the FSV blocks as “black boxes” for now, only explaining the basic function of each specification in relationship to the other ebXML parts. Once more, we will start from a visual representation, this time zooming in on the FSV part of the Open-edi Reference Model:

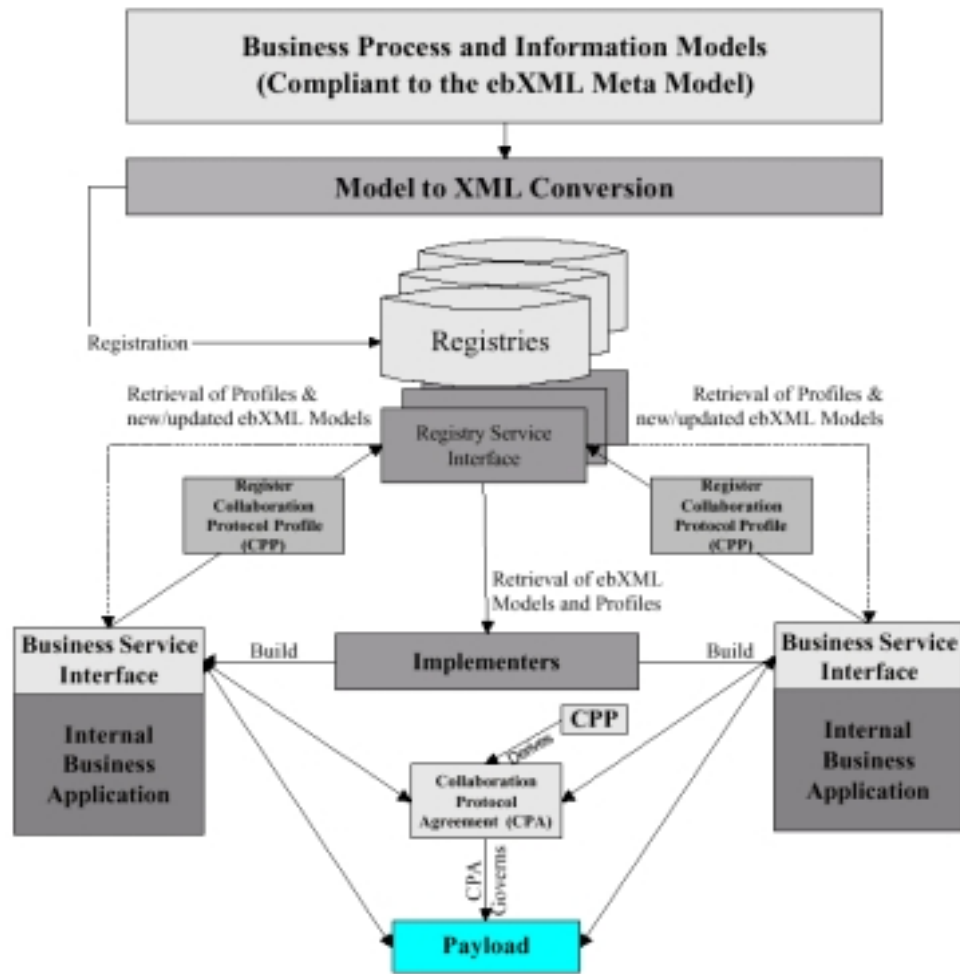


Figure 10: ebXML Functional Service View  
 (Source: ebXML Technical Architecture Specification v1.04)

Note: This FSV overview is sometimes referred to as the “Technical Architecture” itself.

The top box of the FSV figure, with the **Business Process and Information Models**, forms the connection to the business side, the BOV. This information will be in some modelling syntax (preferably UML), and may be converted to XML syntax – using the previously-mentioned XMI, for instance.

Next, there is the registry, or actually a set of **Registries**, as an access mechanism to a distributed set of *repositories*. Here, not only the modelling information (Core and Business Libraries, artifacts, meta models) is stored, but also information about trading partners and the agreements between them (see below) and even the ebXML

Specifications themselves. Basically, *every* type of information that has to be accessible to several parties should be available here. Therefore, the registry mechanism is sometimes called the “heart” of ebXML, as it links the business and functional parts together.

The lower part of the figure can be conceptually divided into three phases, as described in the Technical Architecture specification:

1. In the **Implementation Phase**, an enterprise seeks to turn itself into an “ebXML enabled” *trading partner*<sup>41</sup>, either by purchasing a so-called “shrink-wrapped application” from a third party, or by constructing an ebXML compliant application. In the latter case, the ebXML Specifications and the Core and Business Libraries are retrieved from the Registry. After the application has been built, the company makes up a description of its IT capabilities and supported business processes, and registers it as a *Collaboration Protocol Profile* (CPP<sup>42</sup>) in the Registry.
2. Second, in the **Discovery and Retrieval Phase**, there is still only interaction with the Registry, but this time for a different purpose, i.e. to search potential trading partners by requesting their CPPs. This is done through the *Business Service Interface*<sup>43</sup> component of the ebXML application.
3. Finally, in the **Run Time Phase**, two companies can execute an ebXML scenario (consisting of a number of related transactions) by defining a *Collaboration Protocol Agreement* (CPA), which describes a set of business Message exchanges linked together by a choreography<sup>44</sup>. This CPA is a kind of “greatest common denominator” of the CPPs of the respective partners. As can be seen on the figure, it stipulates the actual business content (*Payload*) of the

---

<sup>41</sup> A *Trading Partner* or *Business Partner* is defined as “An entity that engages in business transactions with another business partner(s).” (ebXML Glossary v0.95)

<sup>42</sup> Many sources still mention TPP (Trading Partner Profile) and TPA (Trading Partner Agreement), instead of CPP and CPA.

<sup>43</sup> A *Business Service Interface* is defined as “A software component that exposes an interface for one or more roles in an ebXML collaboration.” (ebXML Glossary v0.95)

<sup>44</sup> A *Choreography* is defined as “A declaration of the activities within a collaboration, and the sequencing rules and dependencies between these activities.” (ebXML Glossary v0.95)

messages, which are transported according to the ebXML *Messaging Service* specification.

There is one element that we have not mentioned yet: the **Business Service Interface (BSI)**. This is the runtime system (purchased or self-built) that executes the ebXML business transactions by sending and receiving messages, at the service of the **Internal Business Application**. The BSI is configured using information from the CPPs and CPAs that have been created, and from the Business Process and Information Model. However, when we discuss the work of the Business Process team, we will see that a complete Business Process and Information Model is not actually required since a more direct way to configure the BSI is provided.

The remainder of the Technical Architecture Specification contains further information and certain requirements on the other specifications, and in the appendix several examples of ebXML scenarios are given. One other issue worth mentioning here is the definition of ebXML Conformance, with the purpose of increasing the probability of successful interoperability between implementations:

“ebXML Conformance is defined as conformance to an ebXML system that is comprised of all the architectural components of the ebXML infrastructure and satisfies at least the minimum conformance requirements for each of the ebXML technical specifications, including the functional and *Interface* requirements in this Technical Architecture specification.”

Related to this, the use of publicly available test suites is recommended to verify this conformance.<sup>45</sup>

### 3.5.3.6 ebXML Architecture: Conclusion

The above discussion of the ebXML mechanism has been mainly based on the the Technical Architecture Specification (Version 1.0.4). This specification gives a high-level overview of all ebXML specifications: the roles they play, and the interactions and interfaces between them. Consequently, the entire ebXML system and the business

---

<sup>45</sup> Since ebXML is conceived as a *modular* system, the specification also mentions the conformance verification of “ebXML *Implementations, Applications and Components*”.

processes involved are seen as a whole, both focusing on their independence and alignment.

Since we have already covered the first part of the specification document above (the BOV and FSV views), and the specifications and examples in other part will be detailed below, we consider the discussion of this specification as finished.

### 3.5.4 The ebXML Specifications

After examining the architectural overview, the picture of ebXML you probably have in mind is one of a closely integrated set of specifications with many essential interdependencies. Of course, the framework provides most value when *all* of its constituent parts are used together as seamlessly cooperating pieces of the big ebXML puzzle. On the other hand, ebXML is intended to be *modular* in nature, meaning that businesses do not have to implement all specifications at once but can follow an incremental path [ebXML, 2000a]. For this reason, as we zoom in on each specification separately in the following sections, we will not stress their mutual relationships too much.

All ebXML specifications taken together represent a huge body of information,<sup>46</sup> which is moreover still not complete at the time of writing. Of course, it is not the purpose to simply summarize all that information here. The discussion of the specifications should in no way be regarded as a guide for implementation – that’s what the specifications are there for. Rather, the central objective is to make the reader aware of the capabilities and possible limitations of each of the specifications. As before, we will concentrate on *what* the ebXML components do, not on *how* their functionality is implemented. This also means that some parts will be considered at length, while others will be only briefly outlined, according to their relative importance.

---

<sup>46</sup> As a matter of fact, we are covering almost 600 pages of ebXML specifications here, *without* the SOAP, UDDI specifications and any reference documents, that is.

Yet, even when we take the above into consideration, there is still very much material to present. Therefore, we will focus on two ebXML components in particular<sup>47</sup>:

- the *message service*, as the exchange of messages is of course the ultimate goal,
- and the *registry service*, since we have already seen that this is where all the information comes together and is redistributed.

These two components could be called the “nerves” and the “nerve centre” of ebXML, respectively. The fact that much of the overall standardization effort is concentrating on frameworks for these two activities, is another sign on the wall. Therefore, we will not only discuss the mentioned services more thoroughly, but also go more deeply into two related specifications: SOAP for messaging, which has once been a “competitor” but is now incorporated into ebXML, and UDDI, for the registry mechanism, of which the future relationship with ebXML is still vague, but there are encouraging signs that convergence is within reach in the mid-term.

As an “overview” of the following sections, we have included a “pyramid” view on the ebXML system, which points out some of the mutual dependencies between the specifications – but not all... it should mostly be seen as a nice way to introduce the specifications.

---

<sup>47</sup> It should be noticed that the emerging “web services framework” in some respect *only* deals with these two parts, leaving the rest (e.g. specification of the message payload itself) to other specifications or individual companies.

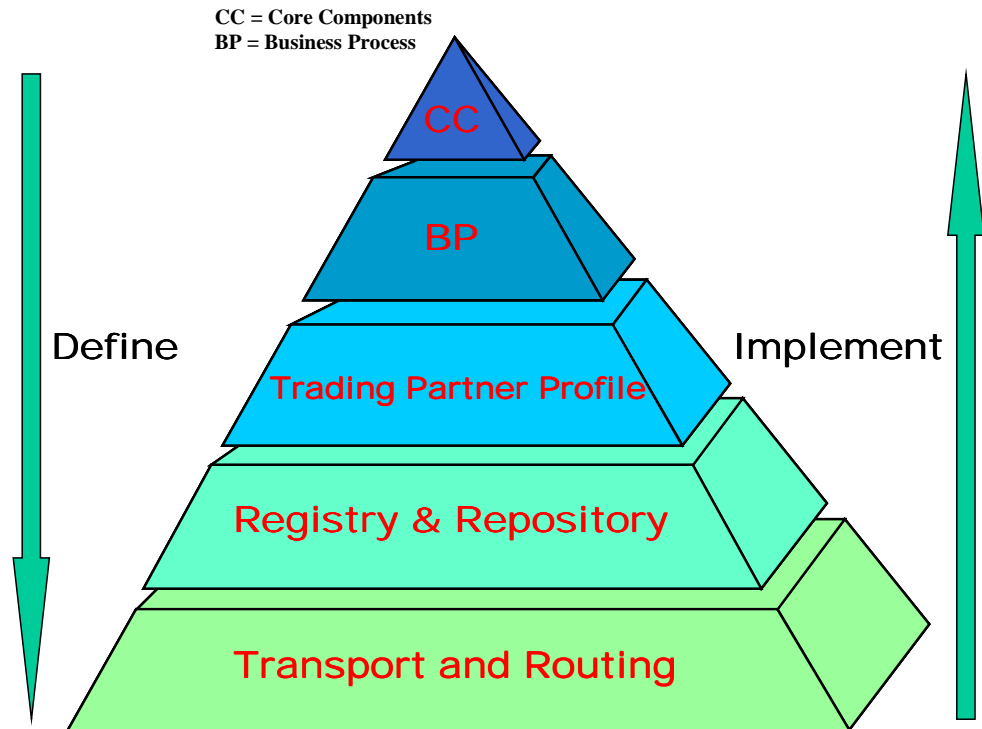


Figure 11: "Pyramid" layer view of the ebXML specifications

Note: To keep a good overview of the document, the numbering of the Specifications has been "lifted" to the third level.

## 3.5.5 Business Process and Core Components

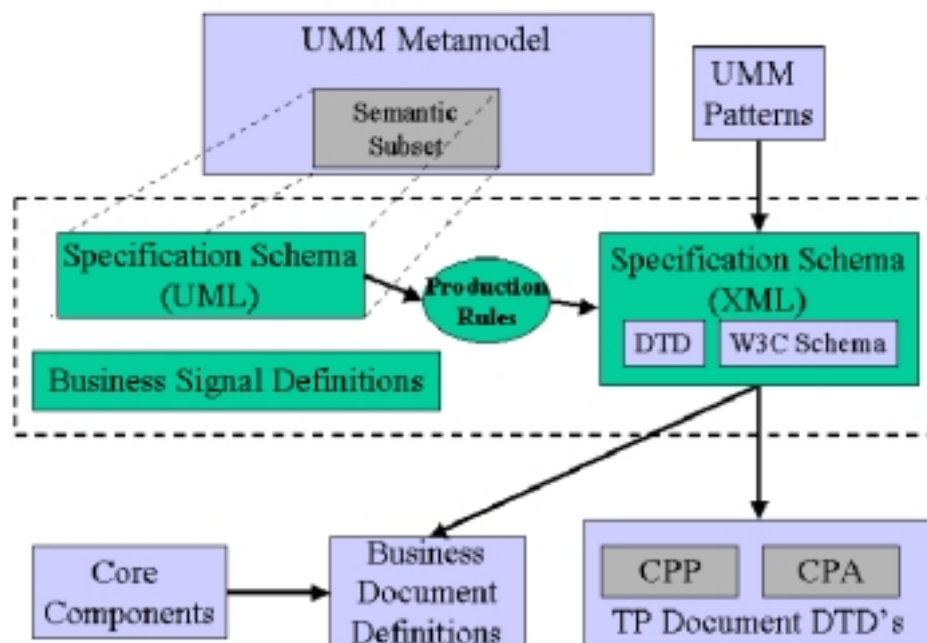
### 3.5.5.1 Introduction

The specifications provided by the Business Process and Core Components Project Teams largely correspond with the Business Operational View described above, and consequently, with the UMM methodology. Since the work of these two teams is largely correlated, we discuss them together in this section.

The UMM analysis methodology prescribes the *overall* process for defining Business Processes. This results in a *meta model*, which specifies *all* the information to be captured during this process. In ebXML, this is the *Business Process and Information Model*. As we have already mentioned in the discussion of the ebXML's *Functional Service View*, the information in the meta model will be used to configure ebXML compliant software such as the *Business Service Interface* (BSI). However, we have

already mentioned that it is only *recommended* to use UMM, and that it is even not *required* to do modelling and analysis. Furthermore, to configure the BSI, we do not need the *complete* meta model, i.e. *all* the artifacts we have seen in the BOV.

Therefore, the ebXML Business Process team has developed the *Business Process Specification Schema*, which supports the *direct* specification of the runtime aspects of a BSI.<sup>48</sup> This *semantic subset* of UMM that is contained in this schema is at the moment the only part of UMM that is mandatory in ebXML. The relationship of the Business Process Specification Schema (in the dotted box) with UMM on one hand, and with the CPP/CPA specifications on the other hand, is depicted in Figure 12.



**Figure 12: Relationship of Business Process Specification with UMM and CPP/CPA**  
(Source: ebXML Business Process Specification Schema Version 1.0)

Here, the schema is considered as an additional view of the UMM Meta Model, drawing those elements from it that are necessary for the BSI configuration. In the lower part, we

<sup>48</sup> In addition, the team is also working on the *Business Process Analysis Worksheets & Guidelines*, which “contains several worksheets that guide analysts towards UMM compliant specifications of their common business processes”, which does not require analysis and modeling expertise but uses UMM only as a reference manual.

see that the Business Process Specification is then used as input for the formation of *Collaboration Protocol Profiles and Agreements* (CPP and CPA), which will be discussed later. In fact, it will be these CPPs and CPAs that are the “configuration files” for the BSI, to execute a set of ebXML business transactions.

In fact, this figure provides an overview of most of this section, in which we will discuss the work of the Business Process and Core Component teams. The parts of the Business Process Specification are shown inside the dotted box, and will be discussed in the following subsection.<sup>49</sup> To the lower right, we see the *Core Components*, for which we will briefly discuss the four specifications:

- *Naming Conventions for Core Components and Business Processes*
- *Methodology for the Discovery and Analysis of Core Components*
- *The role of context in the reusability of Core Components and Business Processes*
- *Specification for the Application of XML-based Assembly and Context Rules*

Note: During the study of this area of the ebXML specifications, we have experienced that information was hard to find, and not always up to date.<sup>50</sup> Many terms seem to have synonyms, which are however not always ambiguously defined, or clarification was only found in non-normative documents. We have tried to filter out a clear description of these specifications, and believe any ambiguity is inherent to the available information.

### 3.5.5.2 Business Process Specification Schema

Note: The *ebXML Business Process Specification Schema Version 1.0*, dated April 27, 2001 was used.

---

<sup>49</sup> Although the “UMM Patterns” are not part of ebXML, we will discuss them together with the Business Process Specification.

<sup>50</sup> For instance, there should be a version 1.02 for the four Core Component specifications, but we were unable to find it – even after a *complete* download of the website. The same is true for several reference documents, such as the *ebXML Glossary*.

### 3.5.5.2.1 Introduction and Overview

The semantic subset of the UMM meta model in this Schema, enables the specification of a Business Process in terms of related Business Transactions. For this purpose, it provides a UML Model, an XML Model (with both a DTD and an XML Schema), and a set of production rules to define the mapping between these two, from UML to XML. Additionally, it defines a set of *Business Signals* (see below). We will not discuss these models in detail, but rather describe the concepts behind the specification schema, and give an structure example of the XML version of the specification at the end.

In the discussion of the BOV and FSV views used by Open-edi Reference Model and by UMM, we have seen that they concentrate on business versus technical aspects of *Business Transactions*, and how they can be combined into *Business Processes*. However, there are more concepts in between, as the following definition of *Business Process Specification* indicates.

[Business Process Specification:] A declaration of the partners, roles, collaborations, choreography and business document exchanges that make up a business process. [ebXML, 2001b]

Figure 13 graphically illustrates this definition, and also provides a general overview of the Business Process Specification Schema:

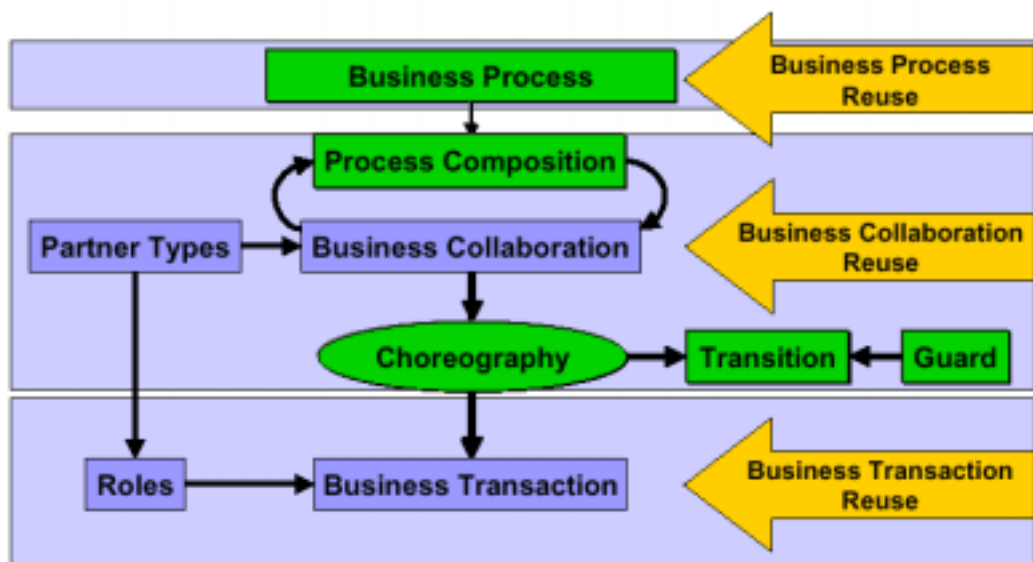


Figure 13: Reuse in the Business Process Model  
(Source: Catalog of Common Business Processes, ebxml.org)

Although this figure is too general to show all the concepts we will discuss in the following paragraphs, the reason why we have chosen it is that it concentrates on *reuse*. This appears to be a central idea of UMM and the Business Process and Core Component specifications: maximal interoperability is achieved through maximal reuse. Reusability can occur at various levels. In the subsequent specifications we will see that at the lowest level we find the *Core Components*, which are used to assemble *Business Documents*. Here, we see the higher three levels, being *Business Processes*, *Binary Collaborations*, and *Business Transactions*. Throughout the following paragraphs subsections, it will become clear that the **key to reuse lies in the separation of context-specific and generic information**.

The discussion of this specification will depart from its key concepts, which are discussed in the following subsections. We will pay special attention to reusability. Additionally, a simplified example will be given.

Note: It should be emphasized that a *Business Process Specification* and its *Business Collaborations* are expressed *in general*, that is, they do *not* apply to particular parties that will play the specified *roles*. It is only when an enterprise develops a *Collaboration Protocol Profile* (CPP) that it references such a process specification and places itself into one or more roles. In the formation of a *Collaboration Protocol Agreement* (CPA), the “other” roles of the collaboration will be filled in. Furthermore, it should be clear that *Business Process Specifications* will be placed in a repository, where they are accessible to enterprises through a registry interface. We refer to the sections describing the CPP/CPA and Registry/Repository for more details.

### 3.5.5.2.2 Business Collaborations

Business partners interact with each other by taking on one or more roles<sup>51</sup>. As such, a *Business Collaboration* consists of a set of roles that interact through Business Transactions by exchanging *Business Documents*. A collaboration can either be a

---

<sup>51</sup> We have not found a specific definition of a *role*. The ebXML Glossary only mentions the UML definition, being “The named specific behaviour of an entity participating in a particular context. A role may be static (e.g., an association end) or dynamic (e.g., a collaboration role).”

Multiparty or a Binary Collaboration. A *Multiparty Collaboration* is a collaboration among more than two roles, but this is not directly supported by the specification: a Multiparty Collaboration will always be decomposed into its respective *Binary Collaborations*, which are between two roles and describe the actual interaction.

Collaborations are expressed as a set of *Business Activities*, which can each consist of a *Business Transaction* (e.g. Request Catalog), or of another complete *Business Collaboration*. In the latter case we have a *nested Business Collaboration*, which enables the recursive composition of collaborations, and thereby facilitates their *reuse*.

### 3.5.5.2.3 Choreography

Within a Binary Collaboration, the activities have to be ordered, which is done by defining a *Choreography*.<sup>52</sup> This results in a set of *transitions* between the Business Activities, which can be either single transactions or nested collaborations. Transitions can be controlled by *Guards*, which are expressions in the Business Documents to put constraints on them. The Choreography can also be seen as defining the flow of messages (either Business Documents or Business Signals) in the collaboration.

### 3.5.5.2.4 Business Transactions

*Business Transactions* are the central concept in the interaction between the opposite roles played by two business partners. One party always plays the *requesting* role, the other the *responding* role, which results in one or two *Business Document Flows*. It may also be supported by one or more *Business Signals*. Both Business Documents and Business Signals are sent according to the rules of the *Message Service Specification*.

A Business Transaction is said to be the “atomic unit of work” in a trading arrangement, which means two things. First, we should refer to the *ACIDity*<sup>53</sup> properties that transactions should guarantee. Transactions can have two outcomes in ebXML: *Success*

---

<sup>52</sup> The specification mentions that this could be done by using UML *activity diagrams*.

<sup>53</sup> ACID stands for Atomicity, Consistency, Isolation, and Durability. As mentioned in the text, the BSI is responsible for this, and should take appropriate measures (like using a *Two-Phase-Commit* protocol). We will however not go deeper into these matters, as they are considered to be out of scope.

or *Failure*. In case of success, it may be used as a “legal binding”. In case of failure, the end state of the two systems should be as if the transaction never happened, so the transaction is *rolled back* – this is the task of the BSI software. Second, transactions are an important unit of *reuse*, which happens by referencing them from the collaboration level. However, contrary to collaborations, a transaction is *atomic*, in the sense that it cannot be decomposed into other transactions (i.e. there is no *nesting* here).

### 3.5.5.2.5 Business Document Flows

Aside from any *Business Signals*, a business transaction is realized as *Business Document Flows* between the requesting and responding roles. However, there will not always be a *responding* business document flow, in the case of a so-called *one-way notification*. Below, we will see that one of the Core Component specifications addresses the assembly of business documents.

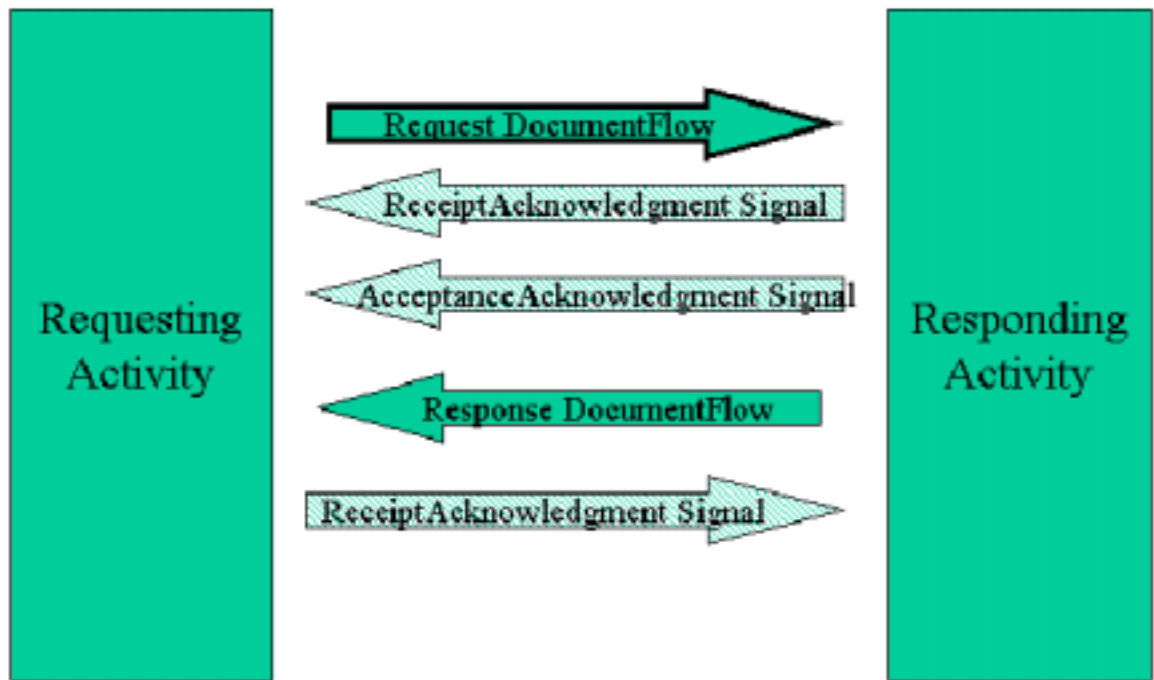
### 3.5.5.2.6 Business Signals

*Business Signals* can be considered as a special kind of message between two parties. In addition to the flow of Business Documents, Business Signals can be sent, as a way to acknowledge the associated document flow. These signals are optional, in that the transaction definition should specify whether or not these signals must be sent. However, their structure and Choreography does *not* depend on transaction parameters, but is defined as part of the Business Process Specification Schema.

The specification defines three types of acknowledgements (as a DTD)<sup>54</sup>, which in fact “signal” a different possible state of a business transaction each: ReceiptAcknowledgement, AcceptanceAcknowledgement and Failure. The Choreography for the first two signals is displayed in Figure 14.

---

<sup>54</sup> According to the specification, these DTDs have been specified by the RosettaNet framework.



**Figure 14: Choreography of Business Signals**  
 (Source: ebXML Business Process Specification Schema Version 1.0 )

Using these signals, three “levels of acknowledgement” can be achieved:

- If the requesting party wants to know that a document is received, it requests a *ReceiptAcknowledgment*”.
- The parties may agree that that the document *structure* should be validated (i.e. it conforms to the schema that is used) before processing, which is done by setting a special attribute (*isIntelligibleCheckRequired*) to true. Only when the structure is valid, the *ReceiptAcknowledgment* will then be sent.
- Furthermore, it can be requested to validate the *content* of the document against the business rules that apply to the transaction. This is done by using an *AcceptanceAcknowledgement* signal.

Additionally, it can be requested to send a *Failure* signal (by specifying a timeout) if the conditions for sending the acknowledgement are not met.

### 3.5.5.2.7 Patterns<sup>55</sup>

The overview diagram in the introduction shows a box called “*UMM Patterns*” in the upper right corner. This is a set of standard transaction interaction patterns, which can be used to implement the transactions. A pattern determines how the exchange of Business Documents takes place in a business interaction.

While the Patterns are not part of the ebXML specifications, they are recommended to be used in the Business Process Specification. The (re)use of standard patterns should improve both flexibility and consistency, which facilitates design and implementation.

### 3.5.5.2.8 Structure Example

Although we do not have the space here to give any example that makes sense, we will give part of the structure of the example that is included in the specification below (“...” means that the previous type of element is repeated several more times).

---

<sup>55</sup> “ebXML e-Commerce Patterns”, a supporting document to the ebXML Business Process Specification Schema to address common pattern implementation issues and provide examples, will also be provided by the Business Process and Core Component Project Teams. Since its status is “DRAFT – NOT FOR IMPLEMENTATION”, we will not discuss it here.

```

<!DOCTYPE ProcessSpecification SYSTEM "ebXMLProcessSpecification-
v1.00.dtd">
<ProcessSpecification name="Simple" version="1.1"
    uuid="[1234-5678-901234]">
<!-- Business Documents -->
  <DocumentSpecification name="EBXML">
    <BusinessDocument name="Catalog Request"/>
    <BusinessDocument name="Catalog"/>
    ...
  </DocumentSpecification>
  <Package name="Ordering">
    <!-- First the overall MultiParty Collaboration -->
    <MultiPartyCollaboration>
      <BusinessPartnerRole>
        <Performs authorizedRole="requestor"/>
        <Performs authorizedRole="buyer"/>
        <Transition fromBusinessState="Catalog Request"
            toBusinessState="Create Order"/>
      </BusinessPartnerRole>
      ...
    </MultiPartyCollaboration>
    <!-- Now the Binary Collaborations -->
    <BinaryCollaboration name="Request Catalog">
      <AuthorizedRole name="requestor"/>
      <AuthorizedRole name="provider"/>
      <BusinessTransactionActivity name="Catalog Request"
          businessTransaction="Catalog Request"
          fromAuthorizedRole="requestor"
          toAuthorizedRole="provider"/>
    </BinaryCollaboration>
    ...
    <!-- Here are all the Business Transactions needed -->
    <BusinessTransaction name="Catalog Request">
      <RequestingBusinessActivity name="">
        <DocumentEnvelope isPositiveResponse="true"
            businessDocument="Catalog Request"/>
      </RequestingBusinessActivity>
      <RespondingBusinessActivity name="">
        <DocumentEnvelope isPositiveResponse="true"
            businessDocument="Catalog"/>
      </RespondingBusinessActivity>
    </BusinessTransaction>
    ...
  </Package>
</ProcessSpecification>

```

**Example 12: Structure of a sample Business Process Specification**

### 3.5.5.3 Naming Conventions for Core Components and Business Processes

Note: The *ebXML CC Dictionary Entry Naming Conventions* v1.0, dated February 16, 2001 was used.

The discussion of this first of the four Core Component specifications will be kept very short, as we cover it mostly for completeness. It provides naming conventions for the Core Components and Business Processes. Although this issue is quite important in practice, it is merely a convention, far from interesting to discuss.

The specification discerns between *Basic Information Entities*, and *Aggregate Information Entities*, which are composed of the former. Naming rules are provided that lead to a unique *Dictionary Entry Name*, which is composed of an *Object Class*, a *Property Term* and a *Representation Type*. Respectively, these represent the activity or object in a certain context, its distinguishing characteristics, and the form of a set of valid values (for which a list is provided). For the aggregate level, this third part is omitted.

An example is “*Goods. Delivery. Date*”. Possible values for the *Property Term* include *Amount, Code, Date, Time,...*

### **3.5.5.4 ebXML Methodology for the Discovery and Analysis of Core Components**

Note: The *ebXML Methodology for the Discovery and Analysis of Core Components* v1.0, dated February 16, 2001 was used.

Since the use of *Core Components* is so essential to ebXML analysis and design, it is important to have a method to discover them. Therefore, this specification describes a method for reviewing Business Processes to extract their Core Components. Since the actual meaning of a Core Component depends on the context in which it is used (see the relevant specification), information about the business domain and process in which such a component is found should also be recorded. The discovery method is described in the following paragraphs.

In the discovery of Core Components, the starting point will always be certain business processes that involve the interchange of business information. Hence, the first step in the method, is to **identify these business processes**, and to document their

characteristics and requirements. This includes identifying possible Core Components used in the processes.

Together with the *use* of Core Components, it is also important that their *re-use* is stimulated as much as possible. Therefore, the next step will always be to **search the Core Library**, that contains not only Core Components, but also Common Business Processes<sup>56</sup>. The process specifications of these Common Business Processes mention the Core Components that are specific to it. So if one of these processes matches the requirements of the process being reviewed, it should be used, together with its Core Components. Additionally, the fact that these components are used in a new context should be recorded in the library.

In many cases however, there will be no exact match with any of the processes in the library, because certain requirements are different. In this case, some of the separate Core Components may be re-used. If there is no match at all, new Core Components have been found, and should be added to the Core Library as a non-core, domain-specific component<sup>57</sup> together with the context in which they are used.

In general, the result of this Discovery and Analysis process is a set of Core Components that can be used as building blocks for deriving business documents within a given context.

Note: The specification mentions that the discovery method that it describes is expected to be replaced by an automated process, and since it should be used by a domain-neutral team, we will not go into more detail about this specification.

---

<sup>56</sup> A Catalog of Common Business Processes will also be provided by the Business Process Project Team. It puts together an initial list of common business process names, generic in nature that can be used across various industries. Version 0.99 of this document is available from the website, but since its status is “work in progress- not for implementation”, we will not discuss it.

<sup>57</sup> A domain-specific component may become a Core Component, depending on the degree of reuse. A process for this is not yet defined.

### 3.5.5.5 The role of context in the reusability of Core Components and Business Processes

Note: The *ebXML The role of context in the re-usability of Core Components and Business Processes v1.0*, dated February 15, 2001 was used.

The *context* in which the objects collected in the *Core* and *Business Libraries* (Business Processes, Core Components,...) are used, is very important. The reason for this importance can again be found in the objective of “maximal interoperability through maximal reuse”. Thus, every time an object is (re)used, its context should be specified.

The specification gives a list<sup>58</sup> of *Context Classifications* or *Taxonomies*, including *Region*, Product, Industry, Role and Business Process – the latter being especially relevant for Core Components. A classification for an object is specified by assigning a value to a *context descriptor*. Together with these classifications, sources are provided where possible values for these descriptors can be found. Examples are the United Nations Standard Product and Service Code (UNSPSC) or the ISO 3166 Geographic Taxonomy. In addition, it is expected that other classification schemes and sources for descriptor values will be usable. As we will see, classifications are supported by the *Registry* (where the Libraries are stored), which allows an arbitrary number of *Classification Nodes* to be attached to any entry.

The specification explains how *Business Documents* may be built by *drawing* components from the Library in the Registry and selecting appropriate values for their context descriptors. As we will see in the next specification, documents can be created by applying a set of *Context Rules* (also called *Constraint Rules*) to a set of Core

---

<sup>58</sup> In fact, this is also an implicit definition of what is meant by “Context”. The specification defines Context as “a method of taking a set of components, that are needed to enact Business Processes, and refining them to take into the ‘environment’ in which the processes will be used”, which is rather unclear. For an explanation of the classifications, we refer to the specification. In fact, only those classifications that are further explained in the specification are mentioned here.

Components<sup>59</sup>. These rules define modifications (extensions or reductions) to the definition of a component, to fit it into the actual context in which it is used. By applying Context Rules to a Core Component, a *Context Constrained Information Entity* is created, which is also registered in the Library if not yet available.

### 3.5.5.6 Specification for the Application of XML-based Assembly and Context Rules

Note: The *ebXML specification for the application of XML based assembly and context rules* v1.0, dated February 15, 2001 was used.

This specification describes a mechanism for assembling documents from Core Components and using *Context Rules* to adapt the components to the context in which they are used. It provides two syntaxes,<sup>60</sup> one for the assembly rules and another for the context rules. In addition, a *Semantic Interoperability Document* is specified, which presents a syntax-neutral output format for the assembly process so that the output of one processor can be automatically compared to that of another. Since this specification is not prescriptive – different rules may be provided by other mechanisms – we will only provide a general overview and a small example, and will not discuss the syntaxes in detail.

*Document Assembly* is defined as “the rules-based process whereby various Core Component schemas are extracted from the repository and put together in a schema for a full business document, in a specified order, with the appropriate occurrence indicators and with the right choice of element content (sequence, choice, etc.)”

This “rules-based process” has already been mentioned, but needs some more explanation. To create a specific document, the appropriate values for the *Context Descriptors* are determined. The possible “situations” in which a *Context Rule* should

---

<sup>59</sup> For further details, the reader may consult the article *Agent-Oriented Technology for Telecommunications* on page 30ff. of the April 2001 edition of *Communications of the ACM*. It deals with the “intelligent components and agents”, and provides extra clarification for this section.

<sup>60</sup> The specification provides XML DTDs for these syntaxes. However, In order to avoid them to a specific schema language, the Assembly and Context Rules are also presented in tabular form.

be applied is represented as an XPath expression, which serves the same use as a *match pattern* in XSLT<sup>61</sup>: if the condition is matched, then the rule is applied. A match means that the combination of specific context values satisfies the condition. Through the application of the rule, an *Action* is performed involving a modification of the component, such as including an additional field in the definition. This process can be seen as the selection of an appropriate *Context Restrained Information Entity*, as mentioned above.

Although the underlying concept is quite simple, problems start when the result of the assembly process is no longer unambiguous: several conflicting rules match the context values and/or the order in which rules are applied influences the result. Also, a hierarchically lower value may be present when the condition expects a value that is higher in a hierarchical classification – this the question if a rule for “Europe” should also be applied to “Belgium”. The first issue is solved by using an attribute (**Order**), that specifies the order: rules with higher values for this attribute must be processed first<sup>62</sup>. The other problem is solved by another attribute (**Apply**), which can have two values: “exact” or “hierarchical”, the latter meaning that there is also a match if the value provided is the same or a child of that specified in the rule.

Below, a simplified example of a Context Rules Document is provided, which should clarify the above (important elements and attributes are printed in bold).

---

<sup>61</sup> In fact, there seems to be a similarity between XSLT templates and Context Rules documents (although the specification does not mention this).

<sup>62</sup> Alternatively, document order may be used.

```

<ContextRules>
  <Rule apply="hierarchical" order="10">
    <Taxonomy context="Region"
      ref="http://ebxml.org/classification/ISO3166"/>
    <Condition test="Region='United States' ">
      <Action applyTo="Buyer/Address">
        <Occurs>
          <Field name="State"/>
        </Occurs>
        <Add after="@id='fred' ">
          <Field name="Floor" type="string"/>
        </Add>
      </Action>
    </Condition>
  </Rule>
  <!-- other rules ... -->
</ContextRules>

```

**Example 13: Context Rules Example**

## 3.5.6 Transport, Routing and Packaging

Note: This discussion is based on version 0.99 of the Message Service Specification, dated April 20, 2001.

### 3.5.6.1 Introduction

Contrary to other teams, the Transport, Routing & Packaging Project Team has produced but a single document: the *Message Service Specification*. However, this is in fact our primary objective: exchanging electronic documents between trading partners. The final justification for all efforts spent on developing a complete, modular framework like ebXML, is to be able to construct well-designed messages, both from a business and technical point of view. For this reason, we will provide a more thorough discussion of this topic.

This specification defines a method for exchanging of electronic business messages that is communication-protocol neutral, reliable and secure. The *message envelope* can contain *payloads* of arbitrary data format, so that any related document can be transported. Moreover, this flexibility ensures that integration with systems using non-XML syntaxes (e.g. UN/EDIFACT) is possible, and enables encrypted information to be exchanged to provide for better, more fine-grained security. Since the system has to

be suitable both for SMEs and large corporations, a low cost solution can be achieved by leaving out several optional, non-basic features, which are however available when robustness and performance are needed.

In the following subsection, an overview of the *Message Service* will be outlined to get an idea of its components, and a short description of the interface to the business application is given. Before moving on to other ebXML aspects, we will first take a look at an industry standard and former competitor of the Message Service specification: SOAP, the *Simple Object Access Protocol*, and an extension to it, SOAP with Attachments. Yet, we will see that ebXML provides much more than SOAP with Attachments, so some of ebXML's advanced *packaging*, *reliability* and *security* features are discussed, with a short description of *error handling* to conclude.

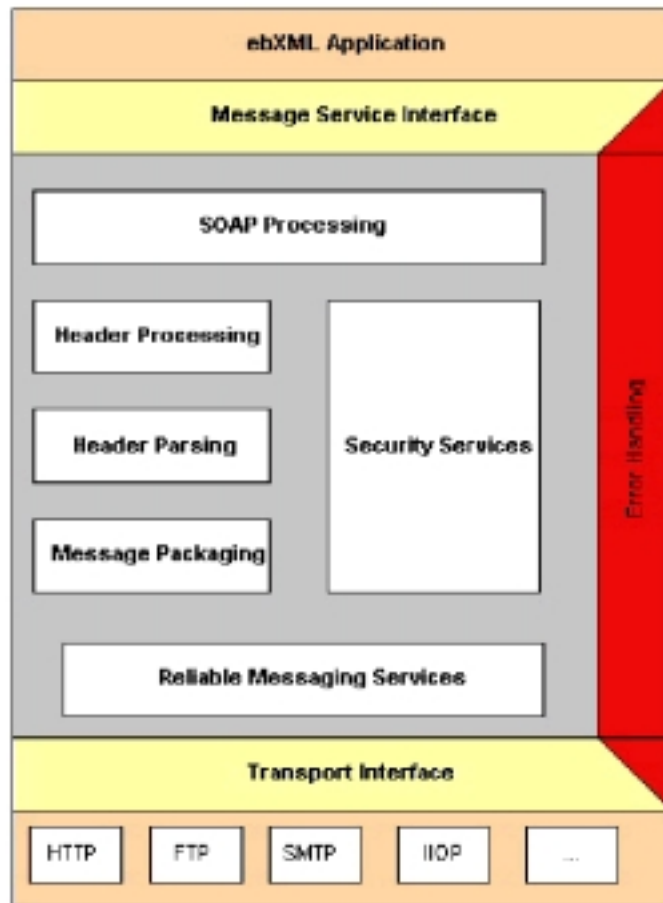
### 3.5.6.2 Overview of the Message Service

From the outside, a *Message Service Handler* (MSH, the physical implementation of the service), is made up of three parts:

- an abstract **Message Service Interface**, which connects the Message Service to the ebXML application, and from there to a company's Information System.
- a **Transport Interface** on the "opposite" side, connecting the MSH to the Internet by means of a mapping to one or more transport protocols (HTTP, SMTP,...)
- the **functional modules** of the MSH itself.

Note: It is possible that the terms mentioned here are not all clear, but they will be explained throughout the section.

We will concentrate on the last part, the internal components of the MSH. The specification provides a modular view of these components:



**Figure 15: Components of the Message Service Handler**  
 (Source: ebXML Message Service Specification v0.99)

The modules are more or less arranged in function of their interdependencies:

- **SOAP Processing** is not really considered as part of the Message Service<sup>63</sup>, as it provides the functionality described in the SOAP specification. Basically, it contains an XML Parser to handle incoming and outgoing messages according to SOAP rules.
- **Header Processing** deals with the outgoing messages: it generates a *SOAP Header* (see below) for an ebXML message, based on the application input that is passed through the service interface and the rules prescribed by the CPA, and adds some extra, message-specific information (s.a. digital signatures).

---

<sup>63</sup> In fact, the specification says little or nothing about the interaction with the SOAP Processor, except that Reliable Messaging is layered on top of it, as we will see.

- **Header Parsing** takes care of the incoming messages, so it extracts all useful information from the SOAP header to make it available to the other MSH modules.
- In the **Message Packaging** component, the final ebXML message is constructed as a SOAP with Attachments envelope wrapped around the SOAP envelope itself and the ebXML payload.
- The **Reliable Messaging Services** will all issues regarding reliable delivery, such as persistence, acknowledgement, retry and error notification (if necessary)
- **Security Services** may create digital signatures, and control authentication and authorization when requested by other components.
- **Error Handling** responds to errors that may be reported by several other components.

Note: In general, an outgoing message can either be initiated by the party in question, or be an *acknowledgement* for another message, or both. The same applies for incoming messages, of course.

In addition to this, there are two optional **Message Service Handler Services**:

- the **MSH Ping Service**, which enables one MSH to determine if another MSH is active or not,
- and the **Message Status Request Service**, that uses a **StatusRequest** and a **StatusResponse** Message to make inquiries about the status of a message that has been sent, especially when reliable messaging is used.

Since these two services are of little interest to us, they will not be treated any further.

### 3.5.6.3 Simple Object Access Protocol (SOAP)

#### 3.5.6.3.1 Introduction

The ebXML Message Service is defined as a set of layered extensions to the Simple Object Access Protocol (SOAP) and SOAP Messages with Attachments (SwA) – which is itself an extension of SOAP. We could go a long way explaining how the incorporation of SOAP into ebXML took place, but that would arguably be interesting. There's only the facts: SOAP is here to stay, and moreover, it is not only the basis for

ebXML's messaging system: SOAP is also the foundation of *web services*, so we believe an in-depth analysis is justified. After this, we will see how ebXML's security and reliability features, necessary to support international electronic business, are fitted into SOAP's minimal framework.

SOAP (*Simple Object Access Protocol*) is an initiative of Microsoft, DevelopMentor, Userland Software, IBM and Lotus. In general, it can be described as a lightweight mechanism for exchanging XML-marked up data over the Internet, a very heterogeneous, distributed environment. On the one hand, this information can consist of a request or response, with appropriate parameters, for some application logic on the receiving side. Therefore, SOAP's Request/Response model is often said to be an RPC (*Remote Procedure Call*) protocol<sup>64</sup>. On the other hand though, this standard is also applicable for more general, "EDI-style" document-exchange – see the four-quadrant scheme above. The full specification can be found on <http://www.w3.org/TR/SOAP>.

As Don Box, one of the founders of the specification, expresses in well-chosen words, "the guiding principle behind SOAP was to 'first invent no new technology'." As a matter of fact, SOAP can be seen as the combination of HTTP, a de facto internet standard protocol, and XML – together with some of its relatives such as Namespaces and Schemas. SOAP simply puts XML data into the payload of a HTTP message. The initial version 1.0, published in 1999, was not received with open arms because it was seen as a Microsoft-only initiative – although its roots lie with Userland Software, and DevelopMentor was also involved from the beginning. Curiously, the most important "change" in version 1.1, published in April 2000, was the support by IBM and Lotus, and several other vendors followed later. The specification itself was extended by supporting new transfer protocols (SMTP, FTP<sup>65</sup>), and introducing the concept of intermediary "stops" between the sending and receiving party. Next, in October 2000, the "*SOAP Messages with Attachments*" specification was published, as an extension to SOAP 1.1. It defines how a SOAP message can be carried within a MIME

---

<sup>64</sup> SOAP's request/response model is specifically suited for *RPC systems*, although more general *messaging* systems can be built on top of it.

<sup>65</sup> *Simple Mail Transfer Protocol* and *File Transfer Protocol*. In the discussion below however, we will assume that HTTP is used, although essentially this makes little difference.

multipart/related message, together with other entities of arbitrary nature – such as binary attachments. This overcomes an important limitation of SOAP 1.1, which only accounts for transporting data in XML format, and was a major step in the convergence with the ebXML Message Service specification.

### **3.5.6.3.2 Limitations of Traditional RPC**

As we have said, RPC-style communication basically means that Requests and Responses are exchanged in pairs between two applications. Although, this is not really the focus of this thesis, SOAP is widely used for RPC. Therefore, we will have a brief look at why SOAP provides a solution in this area.

With RPC, one party (the “calling” application) sends a request to a remote application, where it is processed, and the result is sent back in a response message. The two main RPC protocols currently in use are DCOM (*Distributed Component Object Model*) from Microsoft and CORBA IIOP (Internet Inter-ORB Protocol) defined by OMG, the Object Management Group, and implemented by several vendors. In general, the ORB (*Object Request Broker*) software that implements these protocols tries to make low-level communication aspects as transparent as possible to the programmer.

We will not go into detail about these “traditional” RPC protocols, but rather we will have a look at why they are not suited for Internet-based communication. As the following table shows, the assumptions made by DCOM and Corba IIOP about the situation in which they are used, are simply not valid for the actual environment in which web services are to be deployed.

Intended situation for a typical DCOM/Corba IIOP implementation.	Actual situation for deployment of web services
No form of hindrance between sender and receiver. Port numbers are assigned at will.	Firewalls, blocking nearly all incoming traffic except for “standard web access”, i.e. HTTP on port 80.
Centrally administrable, closed environment, allowing consistent configuration and security.	Geographically dispersed, open environment, with sender and receiver under different control.
Relatively small number of well-known systems, mostly server-to-server communication.	Large number of systems, with a few servers and many clients with unknown and/or unpredictable characteristics.
Homogeneous group of systems, in practice only from a single-vendor and on one platform.	Heterogeneous group of systems, probably from different vendors (Microsoft and several Corba implementations) and implemented on various platforms.

**Table 1: Intended situation for DCOM/CORBA vs. Actual Situation for Web Services**

While there are solutions for some of these problems (such as *tunnelling* to pass firewalls, *bridges* to link cross-vendor implementations), they are usually complex to implement and cause loss of performance and functionality, especially for higher-level services such as transaction management and security.

In general, traditional RPC mechanisms are very powerful and useful, as long as they involve a centrally-controlled group of servers that are housed in one location, run the same operating system and use an ORB product from a single vendor. However, in an open internet environment, there are many problems, and for the most part the advantages of CORBA and DCOM are lost. Complexity and flexibility issues are the main obstacles for solutions.

SOAP will handle this by utilizing standard internet protocols, presuming *nothing* about the implementation of the communicating systems and taking a “minimalist approach”. The HTTP protocol used by SOAP is suitable for use over the Internet. Below, we will see that HTTP’s basic request/response model neatly fits the needs of an RPC system,

and the use of the HTTP POST method allows any kind of data to be sent. The SOAP mechanism is platform independent, language-neutral and non-intrusive: it has minimal impact on existing applications, so these normally don't have to be rewritten – and investments aren't lost.

SOAP is far more limited in scope than distributed object protocols as DCOM and CORBA IIOP. As the SOAP 1.1 specification mentions in the design goals, several features (such as distributed garbage collection) are clearly designated as *not* being part of SOAP. On the contrary, major goals were simplicity and extensibility. Especially the latter will prove to be very important in the context of ebXML: a lot of functionality has to be provided by other protocols or systems that are layered on top of SOAP. Another essential remark is that SOAP's ability to pass through firewalls also means that the work for these firewalls becomes more complex. To fulfil their security task, they will have to examine the message content. However, SOAP's star keeps rising, as several major CORBA vendors have committed to support the SOAP protocol in their ORB products, and Microsoft has committed to support SOAP in future versions of COM.

### 3.5.6.3.3 The SOAP Mechanism

#### 3.5.6.3.3.1 Parts of the SOAP Specification

According to the specification, the SOAP 1.1 protocol consists of three parts:

- the *SOAP envelope*, describing the general structure of the message (header and body), intermediaries, processing party and optional or mandatory nature;
- a set of *encoding rules* to define a serialization mechanism using typed XML data;
- a convention to represent *remote procedure calls and responses* – possibly, but not necessarily, over HTTP.

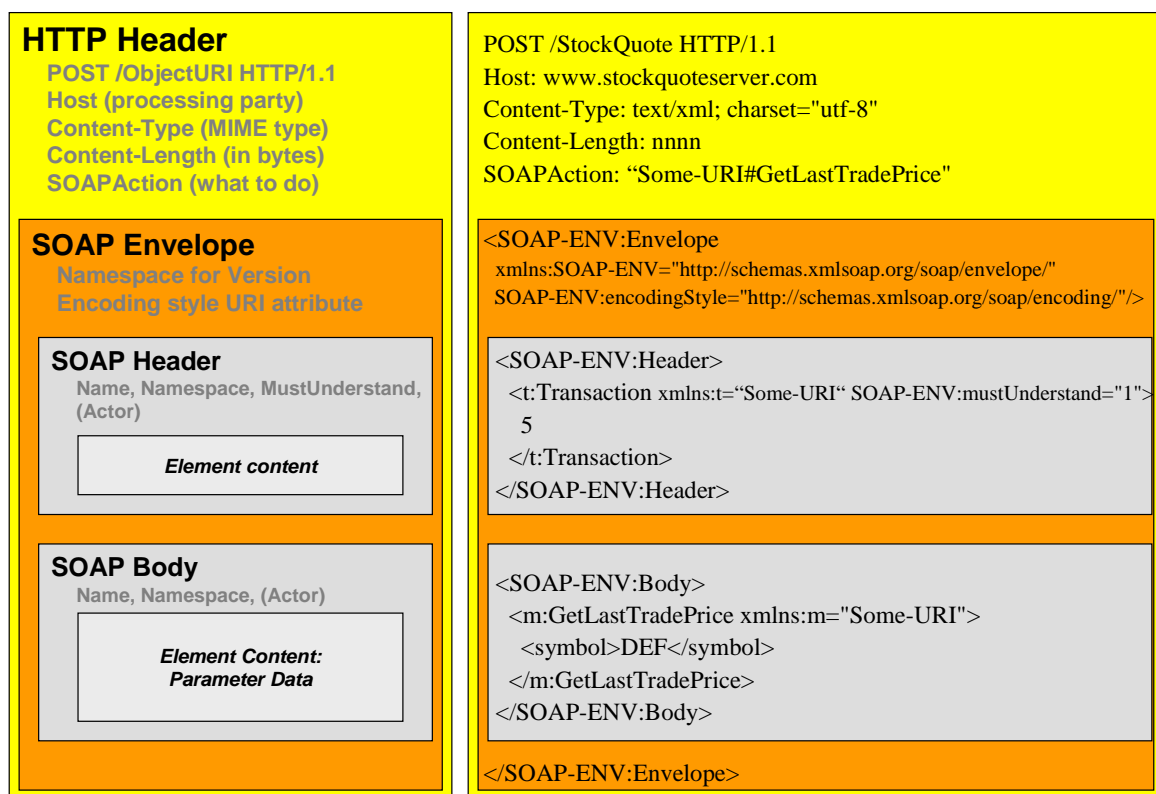
If we compare this to the *facets* as from our generic *Transport* building block, we easily find the *routing*, *serialization*, *interface discovery*, *remote procedure*, and *protocol* facets. As we will see, *extensibility* (one of SOAP's design goals) will be achieved by using *extension headers*. The *packaging* requirement is met by the additional *SOAP Messages with Attachments* specification, which is discussed below. SOAP says nothing

about the *transactions*, *business process* and *security* facets, so these will have to be provided by higher levels.

We will now discuss some aspects of the SOAP specification. First we will have a look at the core of SOAP, the structure of a request or response message. Next, the basic architecture for SOAP interaction will be examined.

### 3.5.6.3.3.2 Message Structure: SOAP Envelope Contained in a HTTP message

The easiest way to describe the SOAP message structure is by looking at a schematic visualisation, as you can see in the following figure. The right-hand side also shows an example request message. In the message structure description we will give below, the example values will be indicated between brackets.



**Figure 16: Basic SOAP Message Structure with Example**  
 (Adapted from [Bordes&Dumser, 2000] and SOAP v1.1 Specification)

At the very beginning of the message, there is the **HTTP Header**. The first line contains the send method (POST)<sup>66</sup>, the ObjectURI (StockQuote) which is used by the HTTP server software to identify the target of the request, and the HTTP version (1.1). The next line indicates the target host for the message (www.stockquoteserver.com). Following, we see the Content-Type (must be text/xml) of the payload<sup>67</sup>, i.e. the SOAP message, the character set (utf-8) used, and the Content-Length of the payload, in bytes (nnnn, for “some” number). Finally, and most important, there is the SoapAction, which tells the receiving application what to do, composed of a URI, a # sign, and an identifier that must equal the first element of the SOAP Body.

The XML part of the message is also known as the Payload, or the **SOAP Envelope** (SOAP-ENV: Envelope). As you can see, the XML structure contains several specific tags and attributes, which are identified by the *SOAP-ENV* namespace prefix. This namespace is being defined in the first attribute of the SOAP Envelope itself, which also functions as a kind of version indicator. Next comes the encodingStyle URI, which points to a link where the encoding rules are defined – more about this in the next subsection. The SOAP Envelope consists of an optional SOAP Header, and the SOAP Body.

The **SOAP Header** (SOAP-ENV:Envelope) is an optional element in the SOAP specification, but it is very important in ebXML. Here, we see part of the *extensibility* design goal being accomplished: SOAP Headers offer a flexible way to extend a message with certain characteristics (e.g. for security). Again a namespace prefix (t) is assigned, since in general, all elements and attributes used in a SOAP message must be qualified by a namespace – this holds also for the SOAP body. By default, SOAP Headers are “optional”, meaning that an application can ignore such a header if it is not recognized. However, an important attribute here is *mustUnderstand*. If this attribute has a value of “1”, the header is said to be “mandatory”, so the receiving application must return an error message if it does not recognize it. This way, it can be assured that

---

<sup>66</sup> HTTP methods are defined by IETF (the *Internet Engineering Task Force*). In general, GET is used to surf the Web, while POST is used for building applications, because it can transmit arbitrary data.

<sup>67</sup> It should be remarked that SOAP’s *payload* (part of the *SOAP Envelope*), is not to be confused with the ebXML Payload: The complete *SOAP Envelope* is contained in the ebXML Header !

any modification to the message semantics that is essential for correct processing will not go unnoticed – which would probably produce erroneous results. The SOAP Header may contain any user-defined elements (in this case it just contains an identification value, “5”). As mentioned above, SOAP 1.1 also introduced the concept of *intermediaries*, applications where a SOAP message “passes through” before going to its final destination. In the message, these intermediaries are represented by “actor” attributes. This way, it can be indicated if certain information is intended for a particular intermediary.

Finally, the mandatory **SOAP Body** (SOAP-ENV:Body) contains the data that have to be passed to the service. The first element (GetLastTradePrice) is the name of the method being called, and it contains the XML data as input for the method.

It should be clear by now that the SOAP Request/Response model cleanly maps onto HTTP Requests and Responses. The response to this request has a similar structure, so we will not discuss it. Again, there is a HTTP Header and a SOAP Envelope with a Header (in case there was a recognized Header in the request) and a mandatory Body, containing the processing result.

#### 3.5.6.3.3.3 Serialization Mechanism: Encoding Rules

As we have said, SOAP can be seen as the sum of HTTP and XML. In this way, a SOAP message is simply a HTTP request or response of which the payload data are in XML format. Since RPC parameter data is usually in some object structure, it has to be converted (*serialized*) to XML format, and this is what SOAP’s Encoding Rules are for. What is actually happening in the process of “Serialization” or “Marshalling”, is that an XML schema is generated that represents the structure of the object data to be passed. Obviously, this can be done in different ways, and therefore the Encoding Rules describe a standard method to do this. One of the basic rules is to use the “Element Normal Form”, which means that all values are represented as elements<sup>68</sup>. We will not go any deeper into this matter, since it is intended for use in RPC, and there would be a

---

<sup>68</sup> We might refer to the previously mentioned “Element versus Attribute” discussion here.

large overlap with the discussion of XML Schemas. In short, SOAP uses many of the XML Schema constructs, and adds a few more (like arrays).

Note: It should be stressed that, although SOAP defines a serialization mechanism for the body of a SOAP message, it does not make any assumptions about the contents of the data it is transferring, nor does it impose any limitations on them.

### 3.5.6.3.3.4 SOAP Transport Architecture and Mechanism

Finally, we will have a closer look at the SOAP transport architecture and mechanism. A good overview of a possible SOAP Architecture and the interaction between a client and a web service is given in the next figure. This overview has a web services application in mind, although it is also applicable for document-exchange – you should recognize several parts of the Message Service Handler described above, such as the SOAP Processor.

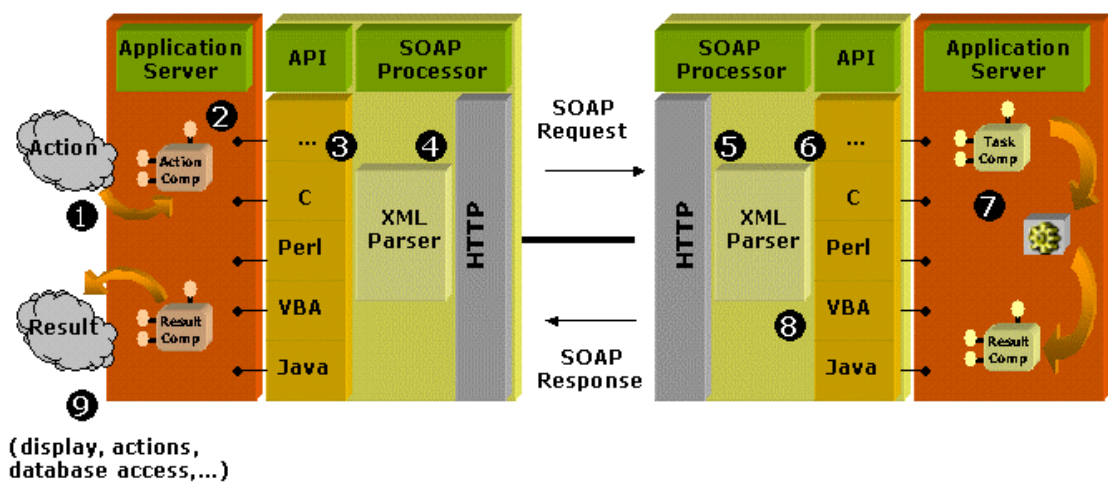


Figure 17: An example SOAP Architecture (Source: Techmetrix.com)

The figure should be pretty clear on its own, but we will give a short description. Starting at step 1, suppose that a client, on the left, wants to use a purchase order web service that is exposed on the **application server** of another company. It issues a command, which is passed to the **API** (*Application Program Interface*) that provides access to the **SOAP Processor**. This **API** will serialize the call using a schema provided by the other party, and send the result as an XML document to the **XML Parser** in the **SOAP Processor**. After doing the obligate well-formedness check (which *should*

normally be unnecessary *in this case*, after the testing phase), the “order document” is packaged as a **SOAP Request** and sent over HTTP to the other party. After smoothly passing any firewalls between the two companies, the request arrives at the receiving party’s SOAP Processor, where it is parsed and checked for well-formedness and validity. Next, the SOAP application does extra verifications, among which identifying the message parts that are addressed to it. Of course, we suppose the message has arrived at its intended destination, so the application must check if all mandatory parts (with `mustUnderstand=“1”`) are supported, or it must respond with a fault message. After that, the application processes the order, and generates a result (probably an acknowledgement). Then, a similar process is started, this time transferring the results back to the calling party in a response message (supposing the processing was successful). Finally, the application that invoked the service, receives the result.

To conclude, three points about this interaction are worth noting separately:

- the platform or programming language of the receiving application’s implementation are irrelevant, since the XML data can be parsed by any system
- what exactly happens on the receiving side, i.e. which program(s) execute(s) the request, is also irrelevant to the initiating application - in fact, there could be an entire middleware system behind the SOAP Processor
- as a final point, we might add that the SOAP Processors can be stand-alone, “special-purpose” applications, or they might be integrated into another system, such as a web server, or the ebXML Message Service Handler we have seen.

#### **3.5.6.3.4 SOAP Messages with Attachments**

A major disadvantage of SOAP 1.1 is its inability to transfer non-XML data. It often happens that related “attachments” have to accompany a message, a scanned image of a physical document for instance. These attachments can be of various types, such as the binary format, the most basic of all.

*SOAP Messages with Attachments* (SwA) is a short specification submitted as a W3C Note on December 11, 2000 by Hewlett Packard and Microsoft. It describes a standard

way to associate one or more attachments in any format with a SOAP message. The resulting **SOAP message package** is a *multipart MIME structure*<sup>69</sup>, which consists of:

- A root body part containing one **primary SOAP 1.1 message**, with a MIME Content-Type equal to “text/xml”
- One or more referenced parts containing the **attachment(s)**, for which the Content-Type can be any media type.

These root and referenced parts are separated by MIME boundaries. The SwA specification describes a method to refer to the MIME attachments from the primary SOAP 1.1 message by using URI references in a *href* attribute, and how the *resolution process* for these absolute or relative URIs.<sup>70</sup> These references contain the *Content-ID* or *Content-Location* header of the referenced MIME part, although the specification recommends Content-ID for more robust error detection.

It is worth noting that the specification authors explicitly avoid inventing anything new, but rather describe how existing technologies (SOAP, MIME, HTTP/SMTP<sup>71</sup>) can be used in a standard way to associate attachments with a SOAP message. To the SOAP processor, it is as if the MIME structure is part of the transport protocol layer, so it must treat the primary SOAP part as the message itself. There is no indication in the SOAP message that it is encapsulated – except for the references to the attachments of course, but these are in fact general URIs. Therefore, a SOAP Processor can simply process the primary SOAP message according to SOAP 1.1. The URI resolution isn’t even explicitly required – the processor may determine whether or not this is necessary based on the message processing semantics.

---

<sup>69</sup> For more information on issues related to MIME, consult <http://www.ietf.org/rfc> for RFC2387 (MIME Multipart/Related Content-type), RFC2045 (MIME Part one: Format of Internet Message Bodies) and related documents.

<sup>70</sup> Indeed, absolute URIs may be used, so in fact the URI may refer to any remote source on the Internet !

<sup>71</sup> The SwA specification is not restricted to a specific protocol binding, but it explicitly describes the HTTP binding, while it refers to an SMTP binding as being straightforwardly implementable.

Note: As we have seen a SOAP 1.1 example above, and will encounter SOAP with Attachments as part of the ebXML framework, we do not give an example of a SOAP message package here.

### 3.5.6.3.5 SOAP: Summary and Perspectives

One of the reasons why SOAP has been adopted so quickly, may be that it offers a trivial, but almost unique advantage as compared to many other initiatives: It has been submitted to the W3C *as a Note*. This may seem a bit strange as an explanation for its wide adoption, but the clue is not that it is “only” a Note<sup>72</sup>, rather that this submission has two consequences. First, no further revisions of SOAP will be issued – which means, from a developer’s point of view, that SOAP-based implementations can build on a “stable” specification. Second, SOAP is generally expected to be the basis for the W3C *XML Protocol Activity* (usually abbreviated as *XP*), which should encompass all similar initiatives, and become the final standard for XML-based messaging systems. As the XML Protocol Charter [<http://www.w3.org/2000/09/XML-Protocol-Charter>] states:

The Working Group shall start by developing a requirements document, and then evaluate the technical solutions proposed in the SOAP/1.1 submission against these requirements. If in this process the Working Group finds solutions that are agreed to be improvements over solutions suggested by SOAP 1.1, those improved solutions should be used.

We have extensively covered the advantages of SOAP as opposed to traditional RPC (e.g. ability to pass through firewalls) and in general (simple, extensible). In the remainder of the discussion of the ebXML Message Service, we will see how it takes advantage of the extensible transport foundation formed by SOAP’s minimal framework.

---

<sup>72</sup> This means that it is not even part of W3C’s Recommendation Track. Notes are a bit of an outsider in the W3C publications. As their website states: “A W3C Note is a dated, public record of an idea, comment, or document. [<http://www.w3.org/Consortium/Process-20010208/tr>].”

### 3.5.6.4 Packaging and SOAP Extensions

#### 3.5.6.4.1 ebXML Packaging

Now that we know how the basic transport functionality of ebXML works, we will see how it is used by concentrating on the *Message Packaging* component of the Message Service Handler. Here, an ebXML Message is placed into its “SOAP with Attachments MIME envelope”, which is then ready to hand over to the transport layer, as described in the following paragraphs. We will also see that ebXML defines several SOAP *element extensions* to provide the reliability, security and error handling functions which will be discussed in the appropriate sections.

Note: Since we have discussed SOAP separately, we will generally not mention those requirements stated by the ebXML Message Service Specification which are already required for SOAP or SwA conformance.

As with SOAP, we will start from a schematic representation of the message structure, shown in the figure below:

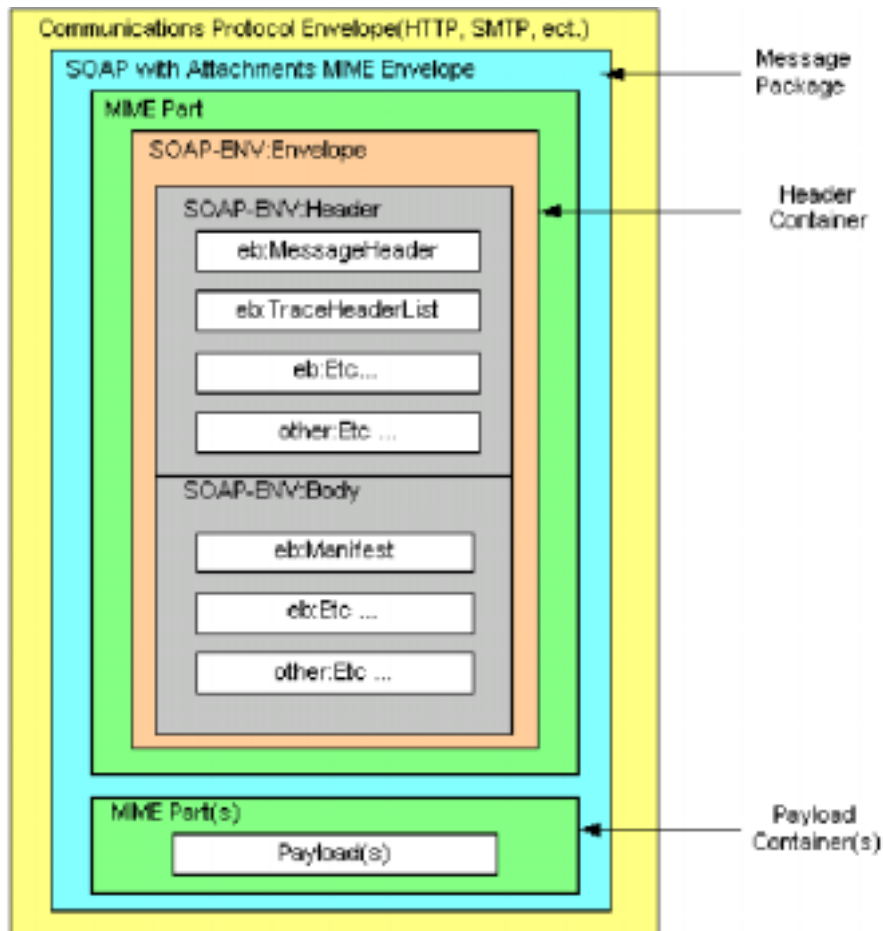


Figure 18: ebXML Message Structure

(Source: ebXML Message Service Specification v0.99)

We will discuss the message structure from the outside inward. The **Communications Protocol Envelope** represents the format used for physical transport over the network. Two possible bindings are mentioned, HTTP and SMTP, which are discussed in an appendix to the specification. These will however not be discussed here.

Next, there is the **SOAP with Attachments MIME envelope**, which is also called the **Message Package**. It is divided into two logical sections: first the **SOAP-Env:Envelope** or **Header Container** (SwA's primary SOAP message), and then the **Payload Container(s)**, physically represented by zero or more MIME Parts. Any application-specific payloads must be enclosed herein. This level is governed by the SOAP Messages with Attachments specification.

Since the Payload Containers contain attachments of arbitrary format which are in no way restricted by this specification, we will not discuss them any further. On the contrary, the Header Container is far more interesting, because here we find the so-called *ebXML SOAP Extensions*. This is a set of *SOAP element extensions*, which are the elements bearing an “eb:” namespace prefix<sup>73</sup> (e.g. eb:MessageHeader) to discern them from SOAP-specific or other elements. We will discuss these element extensions for the SOAP Header and Body separately.<sup>74</sup>

### 3.5.6.4.2 ebXML SOAP Header Extensions

In the overview of SOAP, we have seen that Header element offers a way to extend a message with special characteristics, this way providing for extensibility. ebXML uses this by defining several following extension elements to be put into the SOAP header, which are discussed in the following paragraphs.

Before discussing the extension elements, we should recall the **SOAP *mustUnderstand* attribute**. Remember that the SOAP specification allows the receiving process to ignore any header information unless they are mandatory. It is quite obvious that the specification want the defined headers to be understood, if present. As such, for all header elements the *mustUnderstand* attribute must always be present and have the value of “1”.

- a) The **MessageHeader** element – the only extension element that is *always* required – contains routing and context information. Its most important child elements are:
- **From** and **To** (required), identify the origin and intended recipient of the message.
  - **CPAId** (required), identifies the CPA that governs the message. This may be used for determining *reliable messaging* parameters, as we discussed in the CPP/CPA section.

---

<sup>73</sup> The “<http://www.ebxml.org/namespaces/messageHeader>” namespace must be used for the ebXML SOAP Envelope extensions.

<sup>74</sup> As the number of extension elements is very large, we will only mention the most important ones.

- **ConversationId** (required), as the name implies, identifies a set of related messages which form a “conversation” – how this is done is up to the implementation.
- The **Service**, **Action**, and **MessageData** elements respectively identify the service that acts on a message (normally this refers to the *ServiceBinding* defined in the CPA), a process within that service, and the message itself. This last element can contain a *Timestamp* that indicates the message creation time.

The **TimeToLive**, **QualityOfServiceInfo**, and **SequenceNumber** elements will be discussed under Reliable Messaging.

- b) The **TraceHeaderList** element identifies the MSH has sent the message, and the intermediary and/or ultimate recipient MSH(s). It is optional if there are no intermediaries *and* the message is not sent reliably. As you may expect, this element uses the SOAP **actor** attribute (required).

The receiving application can use this element to trace the route a message has followed. When the message arrives at its final destination, the TraceHeaderList will contain one or more **TraceHeader** elements, depending on the number of intermediaries. Therefore, at each intermediary, a TraceHeader element is added to the TraceHeaderList, containing the **Sender** and **Receiver** for the next “hop”, and a *TimeStamp*.

- c) The **Acknowledgement** element is used to acknowledge the receipt of a message. It contains a **TimeStamp** and a **From** element, which identifies the party that has generated the Acknowledgement. There is also an optional **Reference** attribute, which uses the XML Signature specification<sup>75</sup> to enable non-repudiation of receipt. It is used in *Reliable Messaging* and *Security*.
- d) The **Via** element is used to convey information to the next MSH in row during transport (either an intermediary or the final recipient). It may be used for *Reliable Messaging*.

---

<sup>75</sup> XML Signature (XMLDSIG) is a joint W3C/IETF XML-Signature Syntax and Processing specification, see <http://www.w3.org/TR/2000/CR-xmlsig-core-20001031>

- e) A message may contain an **ErrorList** of **Error** elements reported against the previously received message, if any. The **severity** attribute indicates whether it is only a *warning* (non-fatal for the conversation) or an *error* (the conversation is ended). See *Error Handling*.
- f) Finally, the optional **Signature** element can be used to digitally sign the message according to the XML Signature specification.

### 3.5.6.4.3 ebXML SOAP Body Extensions

The SOAP Body element (not to be confused with the ebXML Payloads) is a container for control information used by the MSH, and more importantly, for including references to payload parts.

- a) The **Manifest** element contains the references pointing to the Payload Containers, or possibly to some other resource. This element is optional, but if any payloads are present, they *must* be referenced here. The Manifest is comprised of one or more **Reference** elements, containing a textual **Description** and some **Schema** (not necessarily an XML Schema) that defines the content of the payload. This is especially useful for an application to find out if it can handle the payload, without first having to parse it. A Reference element is actually a simple XLink (see the previous part of this thesis), with a **href** attribute containing the URI of the payload – as described in the SwA specification.
- b) The optional **DeliveryReceipt** element can be used to let another party know that its previous message has arrived.
- c) The **StatusRequest** and **StatusResponse** elements have already been mentioned above, under the Message Service Handler Services.

### 3.5.6.5 Reliable Messaging

An important feature of the ebXML Message Service is Reliable Messaging. What is understood by “reliable” is that a message will *arrive* at the receiving party’s application “once-and-only-once”.

This puts serious requirements on the MSH implementations of both the sending, intermediary and receiving parties. They must keep the messages that are *sent or received* in reliable storage, properly protected against system interruption or failure. In case of such an interruption or failure, the messages must be processed exactly as if nothing had gone wrong, and any duplicate messages (which have been resent in the meanwhile) must be filtered out. How this is done however, is left as an implementation issue. As far as this specification is concerned, reliability is achieved through by specifying information in the MessageHeader (or in the CPA), and in the Via element (mostly for requesting Acknowledgment messages).

Most importantly, the **QualityOfServiceInfo** element in the MessageHeader has a **DeliverySemantics** attribute that can have two values: *OnceAndOnlyOnce*, which means that reliable messaging is to be used, or *BestEffort* otherwise. Only in the first case, the **reliableMessagingMethod** in the Via element is considered to see how reliable messaging should be achieved. The default value is ebXML<sup>76</sup>. Another interesting element under the QualityOfServiceInfo is **TimeToLive**, which indicates the time by which a message should be delivered and processed by the receiving party.

**Acknowledgments** can be requested by using the **ackRequested** attribute of the Via element, which can be done between every two parties (sending, receiving or intermediary). It can also be specified how many times an unacknowledged message should be retried and how long to wait between retries (using the **retries** and **retryinterval** parameters<sup>77</sup>).

---

<sup>76</sup> The other possible value is “Transport”, but the specification (v0.99) seems to be incomplete on this point: It does not specify what this alternative value means.

<sup>77</sup> Unfortunately, this parameter is not described in the specification (v0.99).

The specification gives further details on the reliable messaging protocol that is used, using the elements described above in many different situations, but we believe this is out of scope for this description.

### 3.5.6.6 Security

The Message Service specification also mentions several mechanisms that can be used to address security problems. These risks include unauthorized access, data integrity and/or confidentiality attacks, Denial-Of-Service (DOS) attacks and spoofing, and are addressed in detail in a separate document called the *ebXML Technical Architecture Security Specification*<sup>78</sup>. A distinction has to be made between *persistent security*, which applies to stored documents, and *non-persistent security*, during transfer of messages.

Some of the possible solutions are:

- *digital signatures*, both for “normal” messages and acknowledgments, can be used for *authentication*. The already mentioned *XML Signature* specification can be used to selectively sign portions of the SOAP Header and Body to bind them to the payloads, and the payload itself can also be signed. This *selective* signing (using the **Signature** extension element) provides flexibility for adding new elements to the message while not invalidating the signature. The specification also describes a detailed procedure for signature generation. As a side note, *non-repudiation* may be achieved by sending an acknowledgement with a digitally signed reference to the previous message.
- *secure network protocols* provided by the communications channel, such as RFC2246 and IPSEC, can provide for *authentication*, *non-persistent confidentiality* (during transfer) and *non-persistent integrity*, and *non-persistent authorization*.
- *encryption* can be used for *persistent confidentiality* (for archival of documents) by using general cryptographic processes such as S/MIME and PGP/MIME. However, the only solution for *selectively* encrypting message parts is said to be the W3C *XML Encryption* activity, which is however not yet available.

---

<sup>78</sup> Since this document seems to be in a very early stage (version 0.35) and it is not mentioned as a core specification but as a reference document aimed at security architects, we will not discuss it here.

These security measures should be configured in the Collaboration Protocol Agreement (CPA). In addition to this, the specification mentions that parties receiving certain messages (e.g. a `StatusRequest`) may ignore them if they consider that the sender is unauthorized or the message is part of some attack (e.g. Denial-Of-Service).

The security measures are summarized in a table, and a number of *profiles* are defined as combinations of measures. However, as it may already appear from the above list of measures, the biggest problem ebXML (and electronic business in general) is facing is that many security specifications are either not yet available or not flexible enough for general, interoperable usage. Therefore, at the moment only the first two profiles (no security or only support for *XML Signature*) must be supported by MSHs.

### 3.5.6.7 Error Handling

Finally, the Error Handling component deals with reporting errors detected in a message to another MSH. This component can be considered as an “application-level handler” on top of the SOAP processor layer, which is necessary because errors can occur anywhere in the MSH (SOAP-related, security, reliability) or in the application. We have already mentioned that the **ErrorList** element is used to send information about the error(s) to the other MSH. Further details on this are of little importance to us.

Note: It might be worth noting that contrary to *SOAP Faults*, any errors in the *communication protocol* layer are *not* dealt with by the Error Handling component.

## 3.5.7 Collaboration Protocol Profile and Agreement (CPP/CPA)

Note: The *Collaboration-Protocol Profile and Agreement Specification Version 0.95* was used for this discussion.

### 3.5.7.1 Introduction

This specification represents the work of the ebXML Trading-Partners team. Since the status of this document is described as “work in progress” (it is said to be a “draft standard for trial implementation”) we will not go into much detail in this discussion. Furthermore, we will see that several aspects correspond to what we have discussed in the Message Service Specification and the Business Process Specification.

We already know that the Business Process Specification provides input for the formation of CPPs and CPAs. More precisely, an enterprise that builds a *Collaboration Protocol Profile* will reference such a “general” process specification, and provide additional information and certain constraints, such as the roles it is capable of “playing”. In general, a CPP can be considered as the expression of the *capabilities* (both in business and in technical sense) of an enterprise to participate in the collaboration(s) contained in the process specification.

When two parties describe a CPP, based on the same process specification (essentially, the same *Business Collaboration*), they can decide to engage in a business interaction by negotiating on any parameters not yet specified in their respective CPPs. Usually, they will take on opposite roles, like “buyer” and “seller”, as described in the discussion of the Business Process Specification. Ideally, this negotiation consists of taking the intersection of the two CPPs, which results in a “greatest common denominator” called the *Collaboration Protocol Agreement*. It is the intention of the specification to enable the automation of this process, so that the intersection can be *computed*, but for now there is only a non-normative section describing certain issues to be handled.

After a CPA has been agreed, the two parties use it to configure their ebXML compliant runtime system, the *Business Service Interface*, which will execute the transactions as described in the CPA.

The specification provides both a DTD and an XML Schema for the complete CPP and CPA. Similar to the *Business Process Specification*, at the moment this specification only supports Binary Collaborations – of course, there is no need to support Multiparty Collaborations as long as this is not required by the process specification. In principle, this means that the specification supports only *direct* collaboration between two parties. Nevertheless, it could also serve to interact with *intermediary* parties such as portals or brokers. This can be done by defining CPAs between each Party and the intermediary, *and* between the parties themselves.

Unlike the *Business Process Specification*, it would make little sense to start from the concepts of a CPP, as it is mostly a mixture of information coming from different sources. Therefore, this time we will *first* give its global structure, including part of the substructure of the first element (*PartyInfo*) which is the most important. After that, we will see that a CPA mostly contains the same information as a CPP – of course – and will discuss some of the aspects pertaining to its formation from two CPPs.

Note: The summary of the specification mentions that “Each Partner's capabilities [...] MAY be described by a document called a *Trading-Partner Profile (TPP)*. The agreed interactions between two Partners MAY be documented in a document called a *Trading-Partner Agreement (TPA)*.” In addition, the CPPs and CPAs are said to deal with the Message-exchange aspects of these TPPs and TPAs, so they are said to be *part of them*. However, it is not specified what extra information a TPP or TPA would contain, and these terms are barely mentioned in recent versions of other specifications – contrary to CPP and CPA, that is. Therefore, it is unclear what the distinction is between TPP/TPA and CPP/CPA, and it might be that they are to be considered as synonyms, with CPP/CPA being used more recently.

## 3.5.7.2 Description of the Collaboration Protocol Profile

### 3.5.7.2.1 Overview

The following example gives an overview of the structure of the CPP, which is adapted from the overview given in the specification. We have added substructure for the *PartyInfo* element, and the most important elements are printed in bold.

```

<CollaborationProtocolProfile
  xmlns="http://www.ebxml.org/namespaces/tradePartner"
  xmlns:ds="http://www.w3.org/2000/09/xmlsig#"
  xmlns:xlink="http://www.w3.org/1999/xlink" version="1.1">
  <PartyInfo> <!--one or more-->
    <PartyId type="..."> <!--one or more-->
      ...
    </PartyId>
    <PartyRef xlink:type="...", xlink:href="..."/>
    <CollaborationRole> <!--one or more-->
      <ProcessSpecification name="BuySell" version="1.0">
        ...
      </ProcessSpecification>
      ...
    </CollaborationRole>
    <Certificate> <!--one or more-->
      ...
    </Certificate>
    <DeliveryChannel> <!--one or more-->
      ...
    </DeliveryChannel>
    <Transport> <!--one or more-->
      ...
    </Transport>
    <DocExchange> <!--one or more-->
      ...
    </DocExchange>
  </PartyInfo>
  <Packaging id="ID"> <!--one or more-->
    ...
  <Packaging>
    <ds:Signature> <!--zero or one-->
      ...
    </ds:Signature>
    <Comment>text</Comment> <!--zero or more-->
  </CollaborationProtocolProfile>

```

**Example 14: Structure of the Collaboration Protocol Profile (CPP)**

(Source: Adapted from the Collaboration-Protocol Profile and Agreement Specification Version 0.95)

First, the *CollaborationProtocolProfile* element itself has several namespace attributes: one for the elements of the CPP itself; another referring to the *XML Digital Signature*

namespace; and another for the XLink namespace, because (simple) XLinks are used for several references.

Apart from the bold-printed elements, the first sublevel of the CPP show three elements:

- *Packaging*, containing information about the way messages should be packaged by the Message Service before transmission, especially about security features, MIME-related aspects and processing capabilities for message parsing and generation.
- *ds:Signature*, since a CPP may be digitally signed in conformance with the XML Digital Signature specification (which we have already met in the Message Service)
- *Comment*, containing text in natural language.

The *PartyInfo* element is however far more interesting. Its *PartyId* element identifies the enterprise or a subdivision of it – since an enterprise may represent itself as several subdivisions having different characteristics. *PartyRef* contains an XLink pointer to extra information about the party. This will usually be a reference to some registry, such as the ebXML Registry itself, or also the UDDI Registry (both will be described when we discuss the Registry/Repository specification). *Certificates* can be included here if the party uses digital certificates (again, based on the XML Signature specification). Once more however, the elements that are left to discuss are the ones that are interesting to us.

As we have mentioned, a CPP describes both the business and technical capabilities of an enterprise, in relation to the role it takes within a process specification. Therefore, we can discern a sort of *layered structure* formed by four elements, which have additionally been marked in italics above: *ProcessSpecification*, *DeliveryChannel*, *DocExchange* and *Transport*. In the following subsections, we will use these four layers as a guideline to discuss the main aspects of a CPP, and related elements (not shown in the overview) will be mentioned where relevant.

### 3.5.7.2.2 ProcessSpecification

The *ProcessSpecification* links the CPP to a Business Process Specification, through a simple XLink, possibly verifiable by a digital signature. This process specification describes the Business Collaboration and Transactions in which the party will participate, so this element can be considered to be the very heart of the CPP.

The *ProcessSpecification* is contained in the *CollaborationRole* element. As the name implies, this also associates the party with a role in the Business Collaboration(s) that it is capable of fulfilling. As such, the *CollaborationRole* describes the *business capabilities* of the party. As a party can participate in more than one collaboration, or can play several roles in one collaboration, there may be more than one *CollaborationRole* element.

Below, we will see that certain characteristics of the party's role in the collaboration are contained in *DeliveryChannel* elements. Therefore, *ServiceBindings* will also be defined here, which "bind" the role to a *DeliveryChannel*. While the first *ServiceBinding* is considered to refer to the default, preferred *DeliveryChannel*, alternatives (with lower preference) may be specified, which increases the chance of success when combining the CPPs of two parties into a CPA.

### 3.5.7.2.3 DeliveryChannel

The second "layer" is formed by one or more *DeliveryChannels*, which is defined by a possible combination of a *Transport* and a *DocExchange* element (see below), which respectively define the transport and document-exchange characteristics for the exchange of messages. Again, there will probably be more than one *DeliveryChannel* in a *PartyInfo* element.

In addition to the characteristics defined by the *Transport* and *DocExchange* level, the *DeliveryChannel* contains a *Characteristics* element which imposes several requirements on the document exchange. This concerns some of the issues we have met in the Message Service description, such as non-repudiation, and security aspects like confidentiality (encryption), authentication and authorization. It should be recalled that

security can be achieved either through the communication channel (the transport layer) or by using the message service itself (the DocExchange Layer).

#### **3.5.7.2.4 DocExchange**

This third conceptual layer in fact provides configuration information for the Message Service. It must satisfy the requirements of the DeliveryChannel: a business document that is provided may have to be encrypted and/or digitally signed, and then be transferred, and the opposite for received messages. However, requirements that cannot be met by the Message Service may be compensated by the underlying Transport layer. In addition, it contains attribute values for the aspects of *reliable messaging*, for instance.

#### **3.5.7.2.5 Transport**

Finally, the *Transport* layer defines capabilities of the communication protocol together with encoding and *security* aspects. It is worth noting that the *Protocol* can be specified for *Sending* and *Receiving* messages separately. The supported protocols are HTTP, SMTP and FTP. Furthermore, the *Endpoint* element specifies the address the “incoming” messages should be sent to.

#### **3.5.7.3 Composing a Collaboration Profile Agreement (CPA)**

When two parties want to engage in a business interaction – usually after one party has discovered the other’s CPP in a registry – their CPPs have to be aligned so that a CPA can be formed. Whereas we have seen that a CPP describes the way in which a party *is capable* to participate in a collaboration, the CPA will have to describe how the *Conversation*<sup>79</sup> *will actually* take place. As we have said, the CPA can be seen as the intersection of the two CPPs. It should be stressed that the CPA purely describes the “interface” that a party exposes, not the internal details of the processes which actually use the CPA. Consequently, it is also completely independent from them.

---

<sup>79</sup> Executing a Business Collaboration, consisting of one or more Business Transactions, is called a *Conversation* in this specification.

The specification contains a non-normative section of how this process may take place, which is summarized in the following figure and will be briefly described:

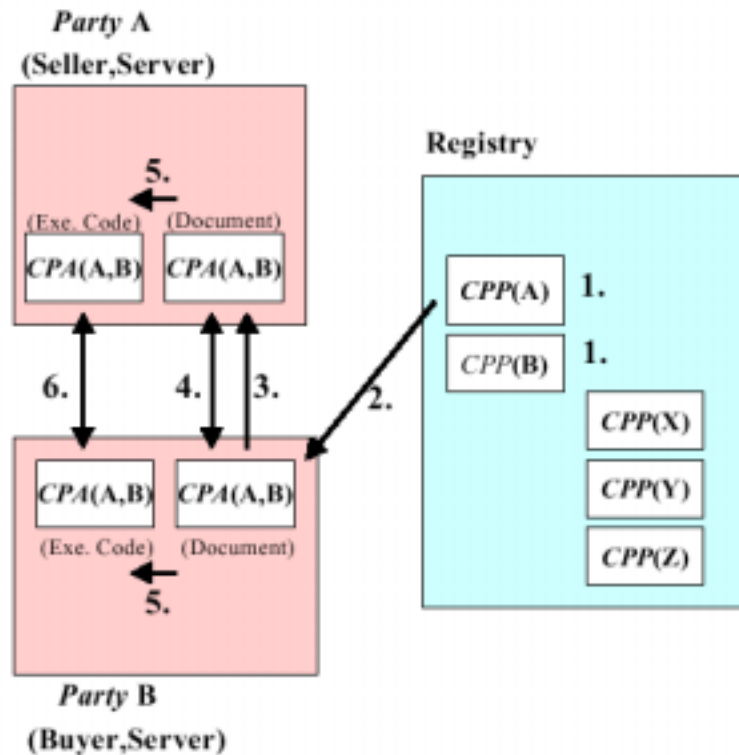


Figure 19: Forming a CPA from Two CPPs

(Source: Collaboration-Protocol Profile and Agreement Specification Version 0.95)

In a first step, parties may upload their CPPs to an ebXML-compliant Registry (1). Suppose a Party B is looking for a supplier for a certain kind of goods. It has selected a Business Process Specification, and has prepared a CPP containing the description of its capabilities and the role (Buyer) it takes in the collaboration. Next, Party B may search the registry, and find Party A's CPP which references the desired Process Specification in the role of Seller (2). Using both CPPs, Party B creates a proposal for a CPA (possibly in an automatic way), and sends it to Party A (3). After any additionally required negotiation, the CPA is agreed and stored locally in the server of both parties (4). Next, they configure their Business Service Interface by inserting the CPA (5). Finally, both parties can start the Conversation (6).

Of course, most of the elements in a CPA correspond to elements in a CPP, so we will not provide an example of this. However, there are two aspects of a CPA that are worth

noting. The first is that it has a *Status* element, which designates state of the composition or negotiation, especially during the creation of the CPA. A CPA that is still under negotiation, has the status of *Proposed*, while for an completed CPA this value can either be “agreed”, or “signed”, depending on whether or not a digital signature has been applied.

Secondly, the lifetime of a CPA can be strictly defined by a *Start* date and time, and either by an *End* date and time, or by specifying an *invocationLimit*, the maximum number of Conversations that may be processed, whichever condition is reached first. When the lifetime has “passed”, the CPA is terminated and must be renegotiated. In case of termination, any transactions in progress are finished first.

Note: Since we have covered SOAP and will cover UDDI quite extensively, it would be strange not to say a few words about WSDL (*Web Services Description Language*), a possible “web services counterpart” for the ebXML Trading Partner specifications. WSDL intends to provide a standard XML-based language to describe web services. As such, it is a natural complement to SOAP: before companies can start exchanging messages over the internet, they need to know how to interact with the web services they are accessing. In other words, WSDL is a way to describe the characteristics of this communication, in terms of transport protocols, message formats and the like. Apart from defining how these characteristics can be described in XML format, it also contains transport mappings onto SOAP 1.1, HTTP and MIME. However, as we will see, there seems to be an overlap in functionality with UDDI, since the WSDL descriptions contain partly the same information as can be recorded in the UDDI Registry. Contrary to what we find in ebXML, WSDL seems to assume that there is no real “negotiation”, but one party describes how its services must be approached, and other “client” parties have to see whether they can meet those requirements.

## 3.5.8 Registry/Repository

### 3.5.8.1 Introduction

We have described the *Registry/Repository service* as the *nerve centre* of the ebXML system. Together with the Message Service, they form the “indispensable” parts of the architecture: one is to discover trading partners, the other to communicate with them.

The Registry/Repository service (RR) is related to many other specifications: basically, it stores *all* information that has to be shared with possible trading partners. Yet, this also means that we already know much about its function, and therefore we will mostly provide a general overview of the Registry/Repository. This is useful to displace our centre of attention in part to another initiative called UDDI, *Universal Description, Discovery and Integration*.<sup>80</sup> Similar to SOAP, this initiative plays a central role in the “alternative”, emerging *web services* framework. Also similar to SOAP, there is no reason why they could not coexist, and there is even likely to be a convergence between UDDI and the ebXML Registry/Repository standard. In fact, the embrace of SOAP by ebXML has been an important step in paving the road to convergence.

*Unlike* with SOAP and the ebXML message service, there is no obvious “dependency” between ebXML RR and UDDI that we could use to determine the order in which we discuss them – it would be stretching the truth to say that one “supports” the other. We might refer to a presentation on the possible convergence of UDDI and the ebXML RR on the ebXML website.<sup>81</sup> There ebXML RR and UDDI are said to be complementary, and the following forecasts are made:

---

<sup>80</sup> Of course, another good reason for having a look at UDDI is that it already has running (beta) implementations, contrary to the ebXML RR.

<sup>81</sup> [[http://www.ebxml.org/documents/ibm\\_ebxmlanduiddiinitialdiscussions\\_02-14-2001.pdf](http://www.ebxml.org/documents/ibm_ebxmlanduiddiinitialdiscussions_02-14-2001.pdf)] This presentation also mentions the OASIS *xml.org* initiative, “The XML Industry Portal”, that (among other things) intended to provide a registry and repository for the access and management of XML schemas and other public resources, a “central clearinghouse” for any XML-related specification. However, it seems likely that it will merge into ebXML, at least as far as the functionality overlaps.

- *UDDI* being used to help businesses find
  - o other businesses *on a global scale*, and more specifically,
  - o other *ebXML-enabled businesses* on a global scale, and
  - o ebXML Registry providers
- *ebXML Registry* “only” being used to find *ebXML-enabled businesses*

From these forecasts, it seems that UDDI will fulfil a more general function. From the ebXML perspective, any interaction with the UDDI registry is likely to come first. For this reason, we will begin with a quite thorough discussion of UDDI, and then continue to describe the ebXML Registry/Repository specifications.

Note: In general, the term *registry* designates the way (the interface) in which the objects in the repository can be accessed, while the term *repository* refers to the underlying storage mechanism itself. Since this distinction is only important when considering internal aspects of the system and the implementation aspects of the repository are of no importance to the users, we will use the term *registry* for the complete mechanism, unless explicitly stated otherwise.

### **3.5.8.2 UDDI, A Complement to the ebXML Registry**

#### **3.5.8.2.1 What is UDDI?**

UDDI (*Universal Description, Discovery and Integration*) is an initiative that has evolved from a collaboration between Microsoft, IBM and Ariba on several standards initiatives such as XML, SOAP, cXML and BizTalk. It claims to accelerate the growth of B2B eCommerce by enabling businesses to discover each other, and define how they interact over the internet and share information using web services.

The idea of UDDI is quite simple. Assuming that there are many companies providing (*web*) services, the problem you are facing is how to discover a service that fits one or more of our needs – the company offering this service is of secondary importance, at least in principle. One option would be to place a file describing the service on each company’s web site, but then consistency among the description files is unsure. The solution UDDI has to offer, is using a distributed registry as a common mechanism to

publish web service descriptions. To smooth the industry acceptance process, UDDI makes use of established standards (HTTP, XML, SOAP), to which companies offering and using web services will usually already be acquainted.

Note: The idea of putting service description files on websites is not abandoned completely. As a matter of fact, an initiative complementary to UDDI, called ADS (*Advertisement and Discovery of Services*), wants to facilitate the task of “filling” a UDDI registry, by defining a standard for a “UDDI advertisement file” to be placed on a website. This way, the information would not have to be “pushed” into the registry by the company, but could then be “pulled” by a *UDDI crawler*.

Conceptually, UDDI is mainly dealing with RPC-style messaging, for access to application functionality that is exposed over the Internet. So even though its function as a Registry is more limited in nature than the registry systems that are part of the frameworks, it certainly has its relevance. Moreover, an important remark is that the term “web service” should be considered in general, so not necessarily implying RPC-style communication. The essential function of UDDI is to *discover* other companies, and *how* these companies choose to expose their services is of little importance to UDDI – it could be an ebXML-compliant runtime system, for instance.

Rather than starting from a conceptual description of the UDDI registry system (which you can easily find on the [uddi.org](http://uddi.org) website), it might provide more insight to look at the various steps that have to be taken by several parties involved to make UDDI work. Several parties are involved: different kinds of standards organisations, businesses offering web services and companies using these services, each in turn they play a role in the subsequent steps. This always involves an interaction with the UDDI registry, and every time we will answer two questions:

- What has to be done, and what information is being exchanged?
- Which registry components are involved, and how do the “actors” interact with them?

Along the way, we will meet all constituent parts of UDDI, and at the end a summary overview will given.

### 3.5.8.2.2 Registration of Service Types, Identification and Taxonomy Systems

We will pay special attention to this first step, because it is critical in making the UDDI registry more than a searchable collection of businesses and services. The final aim is not to let companies *discover* a certain web service, but facilitating the subsequent *interaction*. Thus, the registered service descriptions have to be useful enough to learn about how a discovered web service should be approached. Moreover, it would be useless to do this registration without having a notion of the types of web services that exist.

The UDDI concept we address here, is the *tModel* structure. Although it is often barely mentioned in articles reviewing UDDI, tModel is more than a technical detail: it is a core element of the system, without which UDDI would be nothing more than an arbitrary classification of businesses offering services, albeit accessible in electronic form. And that is *not* what we are looking for.

In principle, a tModel can define almost anything: it is a group of metadata about some concept, *uniquely* defined by a key – the tModelKey. But this idea should remind you of another simple, yet very useful mechanism described in Part 2: Namespaces. Indeed, the use of tModels can be seen as an abstract namespace mechanism. The main difference is that here, uniqueness is not achieved by linking a qualifier to a URI, but by *generating* a UUID (*Universally Unique Identifier*) when the tModel is submitted to the registry.<sup>82</sup> Apart from a tModelKey, a tModel has a *name*, and two optional elements: *description* and *overviewDoc*, the latter being a URL-reference to a more extensive description or usage instructions. It may also be appropriately classified or have alternative identifications.

Now, what would we need this namespace-like mechanism for, and what do the metadata describe? In the current revision (version 1.0) of UDDI, the uniqueness serves

---

<sup>82</sup> As footnote 10 of the UDDI Data Reference V1.0 [110] mentions, this might change in future revisions of UDDI: “*Subsequent work will focus on defining the tModel keys to be more useful by using URN/URI values that are supplied by the data owners.*”

two uses: as a “fingerprint” for a technical specification and as an unambiguous reference to organisations that issue identification or classification systems.

a) **Defining a technical fingerprint**

In the next step we will discuss that businesses have to provide an XML document describing themselves and their services. For describing a service, saying *where* it can be found – usually the URL where requests have to be sent to – is not enough, you also need to say *how* it must be approached: things like the message format, both for request and response, the protocol and security aspects. An example given in the *UDDI Data Structure Reference*, is expressing the compliance of a service to a specification “that outlines wire protocols, interchange formats and interchange sequencing rules”, as we find it in the ebXML Message Service specification. This compliance can be expressed in the service registration by including a tModelKey that uniquely identifies a tModel with information *about* a certain specification. If necessary, more than one tModel can be referenced, each covering a different aspect of the communication. Although this information itself is not enough to start using the service (it only implies compatibility with the referenced specification), it allows a client application used by an interested party to *find out* whether it is capable of interacting with it.

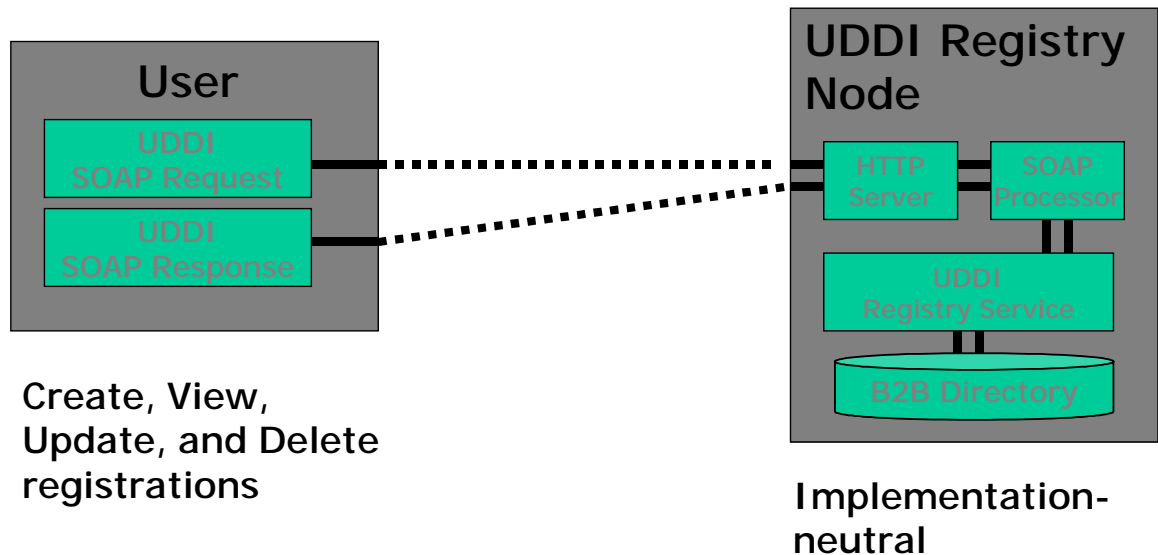
The importance of the mechanism just described should be obvious: in principle, if the application finds out it is compatible with the specification, this provides a fully automatic way of starting interaction with the service that has just been discovered. Perhaps, the value is easier to apprehend when you realize that the tModelKey (the specification’s technical fingerprint) can be used as one of the search criteria during the discovery phase. In a further stage, it might be possible to dynamically extend the client’s capabilities by downloading details about new specifications from a location provided in the tModel. Although the basic means for this extra functionality are available, it is out of scope for the UDDI specification.

b) **Defining an abstract namespace reference**

This second role tModelKeys are playing is easier to grasp, because it better resembles the way XML Namespaces are used. Here, you should compare the key to

a namespace prefix: it is used for qualifying an identification or classification code so that it becomes unique. For instance, it may be used to reference an organisation which has issued a certain coding scheme. An example is found in UDDI's classification mechanism, which will be covered shortly: one of the taxonomies uses the NAICS system (*North American Industrial Classification System*), for which a tModel has been created. To express a company's NAICS classification, the actual code value is qualified by the tModelKey for this system.

The tModels actually form one of the two main logical components of a UDDI Business Registry, comprising service types together with identification and classification schemes. This information is to be registered by the respective organisations, or on their behalf, so an access mechanism must be available. If you realize that a UDDI registry is essentially a web service on its own, which has to interact with all kinds of systems on different platforms, it is not surprising that SOAP is the mechanism of choice. A schematic representation of the SOAP-based communication between companies (in general) and a UDDI Business Registry is shown in Figure 20.



**Figure 20: The use of SOAP in UDDI (Source: UDDI Overview Presentation on [uddi.org](http://uddi.org))**

The registry is accessed through an API which is divided into two logical sections: the Inquiry API and the Publishing API. Here, we only need two methods from the

Publishing API: *save\_tModel* to add or change a tModel, and *delete\_tModel* to remove it. We will meet the other API methods in the next two steps.

### 3.5.8.2.3 Registration of Businesses and the Services they provide

After the extensive explanation about tModels – which is however essential in appreciating the true value of UDDI – the rest of the system will turn out to be a lot more straight-forward. In this second step, the businesses who are offering web services enter the picture: they have to populate the registry with descriptions of themselves and their services, referencing the previously-described technical fingerprints and namespace-qualified references where applicable.

The information passed by the companies to the registry, is contained in a *businessEntity* XML document. Its structure and the elements it contains are shown in the Figure 21.

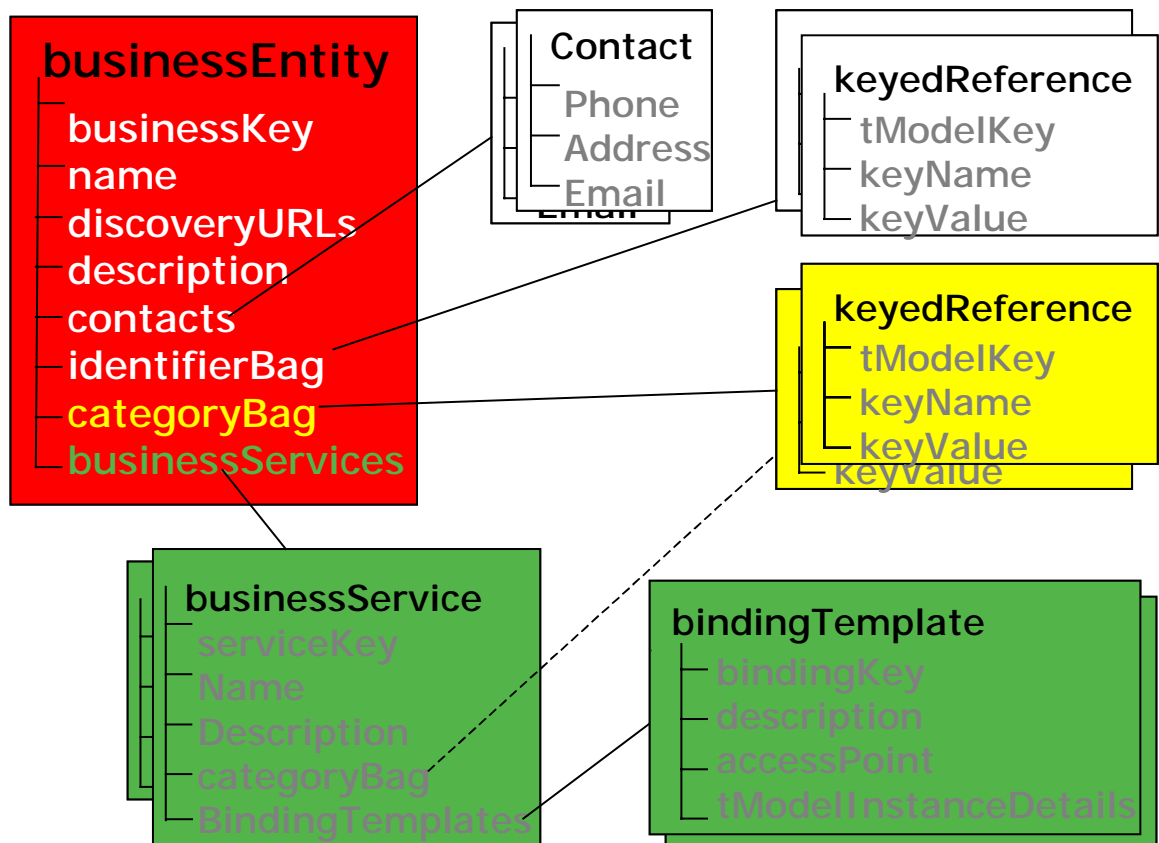


Figure 21: A Business Registration, contained in a *businessEntity* element  
(Source: Adapted from the UDDI Overview Presentation, [uddi.org](http://uddi.org))

In the main *businessEntity* element, we first see a *businessKey* (a generated UUID), the *name* of the business and a *description* it, and a field for *discoveryURLs* where additional information about the business may be found. The other elements are more interesting:

- the *contacts* structure contains address, email and phone number for contacting people within the business. No matter how sophisticated B2B eCommerce grows to be, this information is likely to remain essential in the foreseeable future;
- the *identifierBag* and *categoryBag* contain identification and classification information in a list of *keyedReferences* – name-value pairs that use *tModelKeys* as a namespace qualifier, as described above;
- finally, the *businessService* structures contain information about the services a business exposes.

This last item needs some more explanation, because of its multi-level substructure. A *businessService* usually represents a group of web services which are related by the high-level business process they participate in (e.g. “shipping services”). It is identified by a generated *serviceKey*, has a *name* and an optional *description*, while extra classification information may again be included in a *categoryBag*. Through the *bindingTemplates* however, one *or more* technical web service descriptions can be associated with the service, so here we really get in touch with the *tModels*.

Contrary to the *businessService* element, a *bindingTemplate* must be seen as a technical description of an *individual* service. Aside from the obligate unique *bindingKey* and optional *description*, the other elements are most interesting:

- the *accessPoint* mentions *where* a service can be invoked, in principle with no assumptions being made about the type of address (http, email, ftp, phone, fax,...)
- as you can expect, in the *tModelInstanceDetails*, *tModelKeys* are used to refer to fingerprints of technical specifications describing *how* the invocation should take place. As explained above, the presence of these *tModelKeys* implies that the

implementation details of the service are specified by the specifications associated with their corresponding tModels.<sup>83</sup>

Here, it becomes even more clear that thanks to the use of tModels, many companies can provide web services that are compatible with the same specifications – independent of the platform and the implementation details of these services!

Now, when we look at the way this information is logically represented from the viewpoint of a UDDI Business Registry, we discover the famous White, Yellow and Green Pages.

- **White Pages** include company descriptions and contact information, and company identifiers such as DUNS<sup>84</sup>.
- **Yellow Pages**, outlining the categories of business a company takes part in. In UDDI v1, there are 3 standard taxonomy types:
  - o NAICS (*North American Industry Classification System*)
  - o UNSPSC (*Universal Standard Products and Services Codes*)
  - o ISO 3166 Geographic Taxonomy

Additionally, there may be other taxonomies such as the GeoWeb geographic classification or the SIC (*Standard Industrial Classification*). It is worth noting that two of these classification (UNSPSC and ISO 3166) are also explicitly mentioned in an ebXML specification, *The role of context in the reusability of Core Components and Business Processes* (see above).
- **Green Pages**, providing the actual service descriptions, or “how to do eCommerce with a company”.

Note: It is important to be aware that *several* company identifiers and/or taxonomy codes can be assigned to a company, since the “bags” can contain any number of name-value pairs (the keyedReference structures).

---

<sup>83</sup> Alternatively, the *hostingRedirector* element can be used: this designates that the bindingTemplate entry is a pointer to a different bindingTemplate entry, e.g. as a proxy for a remotely hosted service, or an alias for a service with multiple possible names.

<sup>84</sup> Dun&Bradstreet’s *Data Universal Numbering System*. “The D&B D-U-N-S Number is an internationally recognized common company identifier in EDI and global electronic commerce transactions.” ([www.dnb.com](http://www.dnb.com))

It is barely necessary to mention that the colors used in the figure for the `businessEntity` substructures correspond with the three types of Pages, which should supply some extra clarification.

For the SOAP messages to be sent between the businesses and the registry service, the Publishing API offers methods such as *save\_business* and *delete\_business*, and analogous for the *service* and *binding* information.

#### **3.5.8.2.4 Discovery of Businesses and Services**

Understanding the previous two steps, this last step uncovers nothing really new, although it is of course the reason of existence for the UDDI system. What happens here, is that all the information that has been put into the registry in the first two steps, is searched and used by companies interested in using services, or perhaps, composing higher-level services.

Here, the question might arise how the registry system itself functions, i.e. how the different nodes cooperate – this matter may have been addressed in the previous steps, but here it becomes important. At this moment, the three initial movers (Microsoft, IBM and Ariba) each operate a registry server that interoperates with the other server nodes, and in the future new partners may add extra nodes.<sup>85</sup> The cooperation takes place in quite a different way than in the ebXML system, for instance. Instead of employing a “truly” distributed method of cooperation, the information is replicated among the servers on a daily basis, pretty much in the same way as the notorious DNS (*Domain Name System*) system works. This means, of course, that a company only has to register its information with one UBR (UDDI Business Registry), after which the data is automatically shared with other UBR nodes.

In this step, the interaction with the UDDI Business Registry is prominent. As a matter of fact, there are two ways a registry can be approached.

---

<sup>85</sup> The existing UDDI Business Registries can be accessed through [uddi.microsoft.com](http://uddi.microsoft.com), [ibm.com/services/uddi](http://ibm.com/services/uddi) and [uddi.ariba.com](http://uddi.ariba.com), respectively.

- Each of the websites where a registry is hosted offers HTML forms which are suitable for the rather simple queries on the registry.
- More complex queries can use the second logical group of SOAP messages, contained in the Inquiry API. This opens up a variety of ways to construct powerful searches.

The Inquiry API offers methods to *find* elements in the registry (e.g. *find\_business*) based on names, identifications, classifications and “technical fingerprints”, with the options to be expected (sorting, case sensitivity,...). Next, further *details* (e.g. *get\_businessDetail*) can be requested about an item by providing its unique key value. Again, this is analogous for *service*, *binding*, and *tModel*.

### 3.5.8.2.5 Summary overview and Perspectives

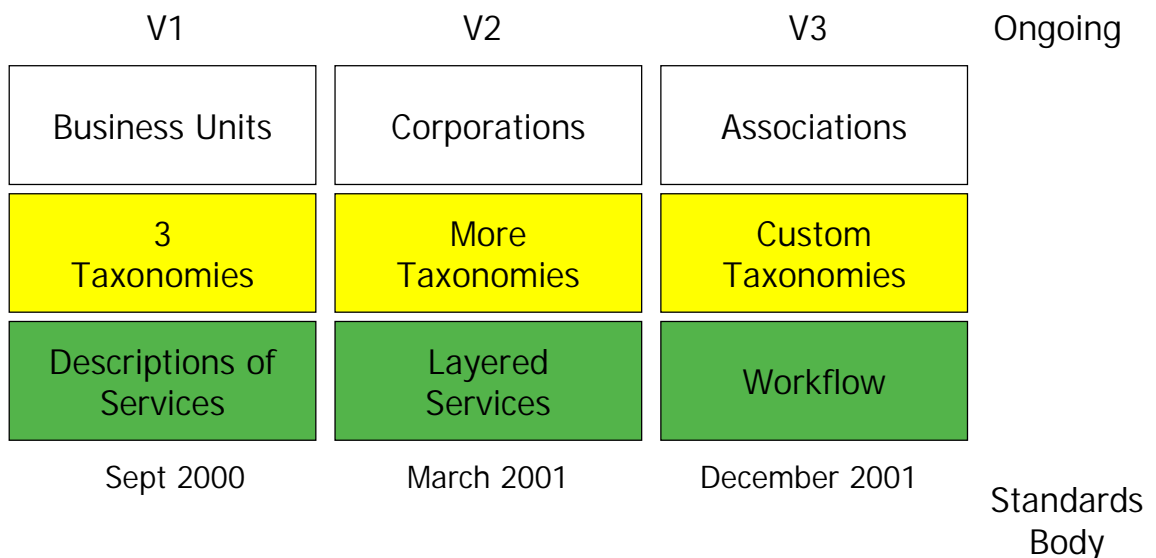
We have seen that UDDI wants to construct an online business registry containing registrations of companies and the services they offer. Its aim is to facilitate building B2B connections, speeding interoperability and web services adoption, building on existing standards such as HTTP, XML and SOAP. The registry contains two logical parts. The first part contains service types, identifier mechanisms and classification schemes. These are represented by *tModels*, a very general concept that offers a namespace-like mechanism. Second, there are the business and service registrations, containing information that is logically grouped into the White, Yellow and Green Pages. Interaction between users and a registry node goes through a SOAP API, that is comprised of a Publishing and an Inquiry part. Between different nodes, replication of their contents takes place on a daily basis.

At this moment, the initiative appears to gain momentum, as more than 200 parties already have signed up, such as high-tech companies like Hewlet-Packard, Oracle and Sun Microsystems. Not surprisingly, the power of UDDI again seems to be its simplicity. UDDI has very limited scope, a broad usability and few limitations on *what* can be described or discovered:

- It only provides several mechanisms to *facilitate* B2B cooperation, but is hardly involved in the actual interaction.

- It does not define schemas, but makes it possible to use any industry standard in conjunction with it.
- It does not make any assumptions about the *kind* of services that are advertised, or the way they are to be invoked, so it is not limited to web services only.

It is clear that the above characteristics make integration with frameworks like ebXML very easy. We must however keep in mind that UDDI is still very young, and is not quite there yet. Although the first UDDI Registries are now online, at this moment their main function is beta testing<sup>86</sup>. Not only it takes time for companies to get familiar with this new technology, the practical utility is greatly dependent on related standards and initiatives (authentication, transactions, payments), and on the further development of the registry itself. Whereas the base technology is established, the functionality of the White, Green and Yellow Pages is meant to be extended in three steps, as shown in the roadmap below.



**Figure 22: The roadmap for UDDI (Source: uddi.org)**

---

<sup>86</sup> Microsoft, IBM and Ariba have announced that a full, publicly available version of the registry will be launched by early May. [<http://www.techweb.com/wire/story/TWB20010413S0010>]

An important detail is the mention of “Standards Body” in the lower-right corner. UDDI claims to develop through an “open” process<sup>87</sup>, with the purpose of transferring the technology to a standards organization and turning license control and intellectual property to a third party by year-end 2001. Meanwhile, as can be seen on the roadmap, the three types of pages should be extended with a registration system for multi-unit businesses, a more flexible taxonomy system and compound services that add up to business process workflows.

Finally, we might re-address the question of how UDDI is related to ebXML. We have seen that the use of protocols like the one provided by the ebXML Message Service is possible by registering a tModel for it. Apart from the fact that the ebXML registry is far more extensive, there seems to be a difference in focus: UDDI is mostly concerned with the whole “web services” buzz, thereby primarily aiming at the integration of businesses around Net Marketplaces. These Marketplaces are supposed to query the registry to discover the services companies have on offer. ebXML on the other hand, is addressing the use of XML in general B2B integration, so intermediary Marketplaces are less important.[Webber&Dutton, 2001] As a side note, it is said that the major reason for starting the UDDI initiative was that the three initial movers were not prepared to wait for the ebXML project to finish. More recently, both initiatives seem to be interested in working together to create interoperability, which could lead to a seamless cooperation between ebXML and UDDI registries – as we have seen in the introduction. The upcoming months, starting with the end of ebXML’s 18-month development period in May 2001, will teach us what the outcome will be.

### **3.5.8.3 The ebXML Registry/Repository**

Note: For this discussion, the *ebXML Registry Information Model v0.90*, dated April 20, 2001 and the *ebXML Registry Services v0.90*, dated April 23, 2001.

After this quite thorough overview of UDDI, we will now see how ebXML deals with this essential piece of eBusiness functionality. There are two ebXML specifications dealing with the Registry:

---

<sup>87</sup> However, the three initial movers retain certain veto rights on newly submitted technologies.

- the *Registry Information Model*, which describes *what* objects can be put into the registry, the metadata that is kept about those objects, and *how* the Registry is organized.
- the *Registry Services Specification*, which provides a detailed view of the interfaces an client can use, and the functionality of the registry services behind these interfaces.

Although both UDDI and ebXML RR are “registries”, we already know that the latter contains far more than business and services descriptions and a couple of taxonomies. Therefore, both the information and interfaces model are much more versatile. While UDDI is organized around the *businessEntity* element, the central concept in the Registry Information Model is the much more general *RegistryEntry*, which provides “metadata” (classifications, identifiers,... but also arbitrary attributes) for *any* repository item.

Note: This also means that repository items are left intact, so no information is “inserted” in them. Since the distinction between these two terms is quite essential for a good understanding of the registry, want to add the literal definitions given in the Registry Information Model specification here:

The term “**repository item**” is used to refer to *actual Registry content* (e.g. a *DTD*, as opposed to metadata about the *DTD*). It is important to note that the *information model is not modelling actual repository items*.

The term “**RegistryEntry**” is used to refer to *an object that provides metadata about content Instance (repository item)*.

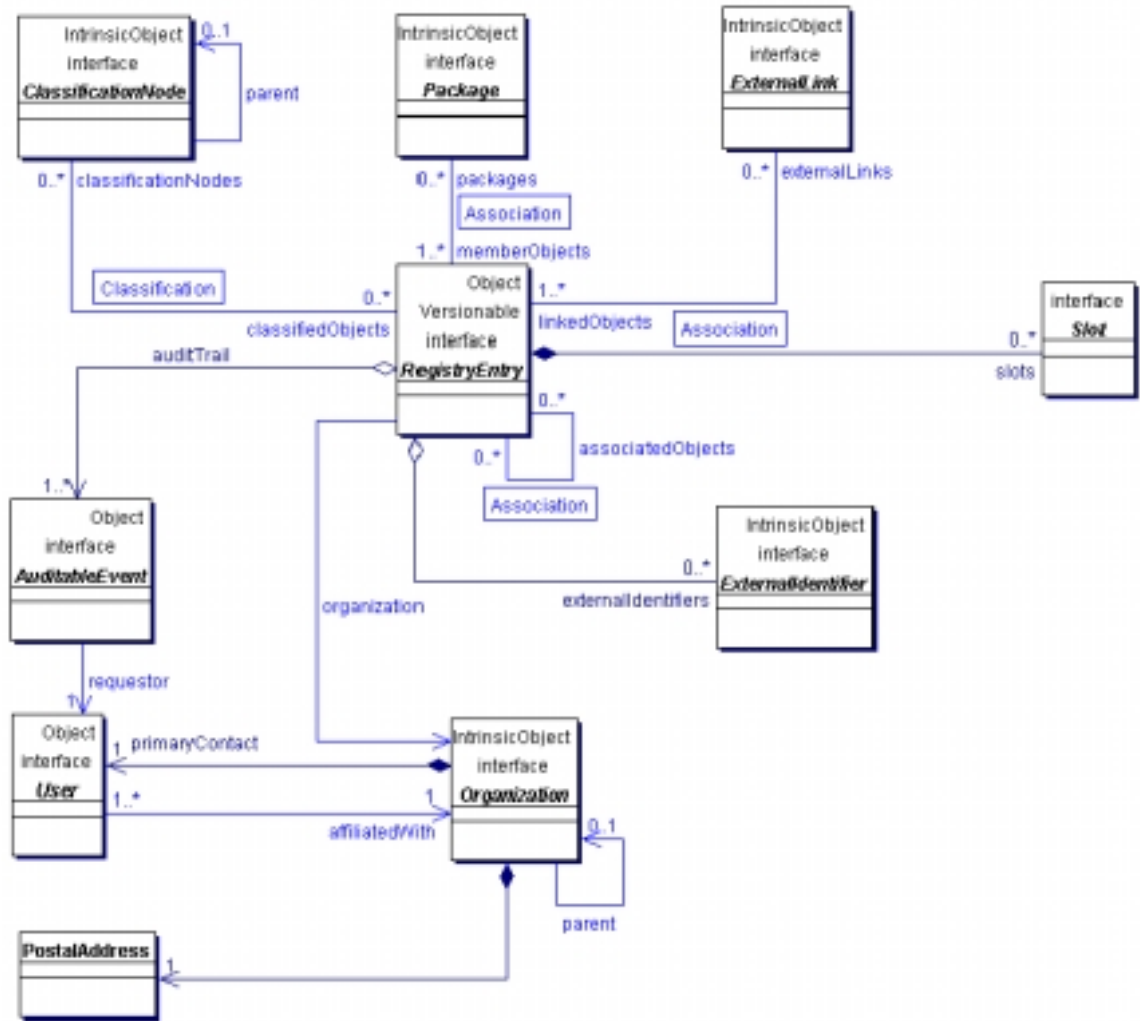
As a consequence of this generality, the simple three-step interaction pattern that we have used to discuss UDDI would be too restrictive in this section. For that reason, we will discuss the two specifications separately: first the *Registry Information Model*, to see what information it contains, and then the *Registry Services Specification*, which will show us how to put that information into the registry, and how to get it back out.

For general usage examples, we refer to other sections, such as the architectural overview or the CPA formation. We will however provide a few examples of the query functionality of the registry services.

#### **3.5.8.4 Registry Information Model (RIM)**

The question we are addressing here is, “What is stored in the Registry and how is it organized?” The specification provides a “high level public view” of the information model, on which we will base this discussion.

Note: This view is depicted as a *UML Class Diagram*. It is mentioned that UML is only used as a way to describe concepts, without implementation or methodology implications. Also, it does not certain aspects like *inheritance* between the classes. Furthermore, we believe the distinction between the terms *class* and *object* is non-essential to us so we will use them interchangeable, and as before, we refer to appropriate sources for more information on UML concepts.



**Figure 23: Registry Information Model - High level public view**  
 (Source: ebXML Registry Information Model v0.90)

For each class shown here, an interface is defined in the specification. We will use this diagram as a guideline to discuss several important aspects of the information model. At the centre of this figure, we see the *RegistryEntity* class<sup>88</sup>, which will be our starting point. The other objects will be discussed afterwards.

<sup>88</sup> A more detailed view of the information model would show that *RegistryEntry* is inherited from *RegistryObject*, which is the base class for all data in the registry. However, starting from this object would unnecessarily complicate the discussion.

### 3.5.8.4.1 RegistryEntry

The information model is organized around the *RegistryEntry*, which is already defined above as an object that provides metadata about any repository item. Most other classes in the model are actually specializations of this class, since it has several essential attributes. We will discuss three of them in more detail.

One important function of the registry is *life cycle management* of the objects it contains: an object is first *Submitted* to the Registry, but before it is available for use it must be *Approved* by the Submitting Organisation (SO). When this SO plans on removing the object from the registry (or whatever the reason might be), it can change the status of its entry to *Deprecated*, which means no new references to the object can be made, but existing references remain valid. The last step in the life cycle is *Withdrawn*, when the object is removed: we should repeat that the *RegistryEntry* and its related repository item are separate objects, so the *RegistryEntry* and any references to it may remain to exist, even if the object has been removed. This information is kept in a *status* attribute, and we will see that the *Registry Services Specification* offers methods to handle this life cycle management.<sup>89</sup>

A second, even more important attribute of a *RegistryEntry* is *objectType*. This attribute indicates the type of repository item a *RegistryEntry* contains information about, so here we meet all the information that is stored in the registry for shared use by a business community. For completeness, and as another reminder of the importance of the registry, we list the possible values below.<sup>90</sup>

Unknown, Process, Role, UMLModel, XMLSchema, CPP, CPA, Transport, SoftwareComponent, Package, ExternalLink, ExternalIdentifier, Association, Classification, AuditableEvent, User, Organization.

---

<sup>89</sup> As a matter of fact, there seems to be a dissimilarity between the two specifications on this point: in the *Registry Services Specification*, the last status of an object is said to be “Removed”.

<sup>90</sup> The first nine values are “ExtrinsicObjects”, normally pertaining to some model in one of the other specifications (you should recognize them). The rest are “IntrinsicObjects”, defined in this specification.

Third, there is the *accessControlPolicy* attribute, which brings up the *security* aspects of the registry. An *AccessControlPolicy* is an object (not on the figure) that defines a set of *Permissions* for authorized access to an object. These *Permissions* are expressed as *Privileges* that are granted to users (either software programs or humans) or groups of users, which may be identified by a digital certificate.

Furthermore, each registry object has a *name*, a *description*, an *AccessControlPolicy* (see below), and an identifier, which will normally be a UUID (*Universally Unique Identifier*) and is generated on submission, similar to what we have seen with UDDI. Objects can have a version, which is useful if it may be updated. There are several more attributes, all with associated methods, but discussing all of them would lead us too far. Many attributes are defined as *Classification* schemes, which we will meet next.

#### **3.5.8.4.2 Association, Classification and ClassificationNode**

We already know from the *Business Process* and *Core Component* specifications that classifications are very important in specifying context (remember the idea of maximum reuse through separation of contextual and generic information).<sup>91</sup> In fact, classifications apply to almost all objects in the registry. This information is recorded by associating a *RegistryEntry* with a *ClassificationNode* by means of a *Classification*. This *Classification* is actually a special form of an *Association*, and all these objects are specializations of *RegistryEntry* itself.

*Associations* play an important role in the registry. They can be used to define many-to-many relationships between *any* two registry objects and are therefore extensively used throughout the whole model. Pre-defined *associationTypes* are available, such as *HasMember*, *Contains*, *UsedBy* and many others.

*Classifications* are used to classify a repository item, by associating its *RegistryEntry* with a *ClassificationNode*. Many nodes together can be structured into a tree, which is called a *classification scheme* (also called *ontology* here, or *taxonomy* as we have seen

---

<sup>91</sup> This association and classification information will also prove to be very useful to execute queries on the registry, as we will see in the Registry Services Specification.

before). Apart from the usage in attributes of the RegistryEntry, examples of classifications are *Industry*, *Product* and *Process*. These terms themselves are also registered as parentless nodes (called *root classification nodes*), and define the classification scheme by forming the root of its tree structure.

A RegistryEntry may be classified by many classification schemes. Moreover, a classification – since it is a RegistryEntry itself – can be associated with more than one ClassificationNode, which is necessary for an unambiguous definition of the context(s): for instance, a company may be classified with the node “Belgium”, but to express that it is actually *located* in Belgium, its Classification might additionally be associated with a ClassificationNode called *isLocatedIn*. Unlike UDDI, the ebXML RR does not define support for specific standard taxonomies: any taxonomy may be registered as a tree of classification nodes.

#### **3.5.8.4.3 AuditableEvent (and Use, Organization and PostalAddress)**

The registry also provides audit trail functionality, for which *AuditableEvents* are recorded: they record *Events* that have an effect on the state of a RegistryEntry. This may be a change in the object’s life cycle, for which the same pre-defined values as mentioned above are recorded, or the result of a client request. With each event, a *timestamp* and the *User* who has caused the event is recorded. A User has several attributes on his own, and is affiliated with an *Organization*, which has a *PostalAddress*.

#### **3.5.8.4.4 Other Objects**

The other objects require less explanation. In summary, they are:

- *ExternalIdentifier*, to provide additional identification information (such as a DUNS number, which is also used by UDDI)
- *ExternalLink*, which may contain a URI, pointing to a website with additional information for instance.
- *Package*, which defines a group of related objects, enabling operations to be performed on the entire group, for instance.

- *Slot*, to add arbitrary attributes to an object, which makes the model easily extensible.<sup>92</sup>

This concludes the overview of the Information Model.

### 3.5.8.5 Registry Services Specification

The *Registry Services Specification* defines the interface a user can use to interact with the registry. The communication between the user and the registry takes place using the *Message Service*, and is normally defined by a CPA. An appendix to the specification provides DTDs for the message payloads. Since much of this functionality implicitly follows from the above discussion of the *Registry Information Model*, we will keep this discussion short.

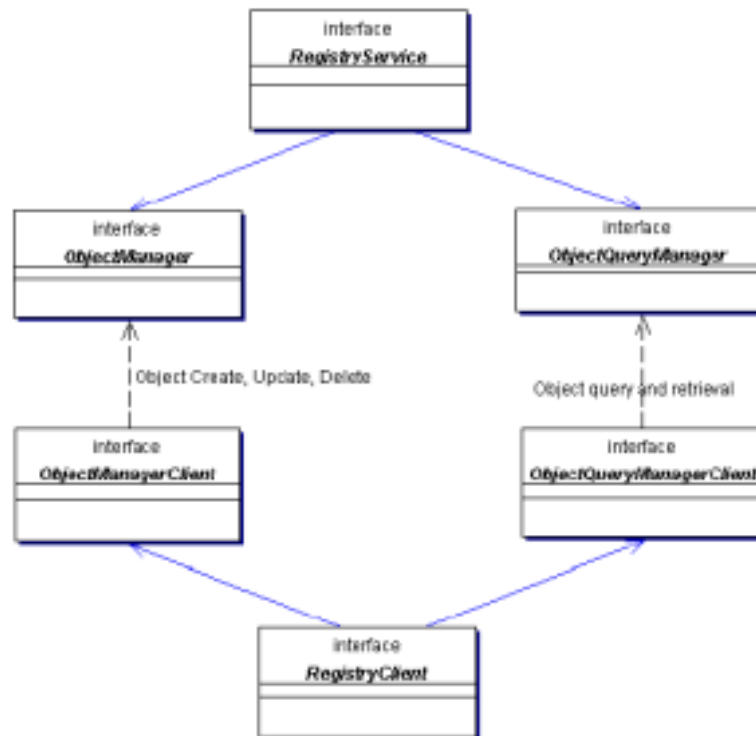
As we will see below, interfaces are defined both on both sides, the *RegistryService* and the *RegistryClient*. Depending on the place where the *RegistryClientInterface* is implemented, three possible “architectures” are proposed, quite similar to what we have seen with UDDI:

- a *thin client* scenario (using a web browser), where this interface is placed on the server side
- a *fat client* scenario, with a “Registry Browser application” implementing the *RegistryClient Interface* on the client side
- in case there is no human user involved, automatic interaction with the registry may use a *RegistryClient Service* that is exposed by a *server side business component*.

Again, we will use a simple UML diagram as a starting point for discussing the interfaces:

---

<sup>92</sup> Because of its general applicability, it seems that a *Slot* could be compared to UDDI’s *tObject* - although its exact functionality is left unspecified, and many of a *tObject*’s functions are fulfilled by *Classifications* and other objects here.



**Figure 24: ebXML Registry Interfaces**  
 (Source: ebXML Registry Information Model v0.90)

On the server side, we see the main *RegistryService* interface, which actually serves only to give access to the other two interfaces:

- *ObjectManager*, which deals with everything regarding *life cycle management*,
- *ObjectQueryManager*, to handle any *queries* on the registry information.

Corresponding services are present on the client side, with the *RegistryClient interface*. It gives the *RegistryService* access to two *call-back* client interfaces, which are used to notify the client of the results of their requests.

The *Object Management Service* can be used to register, update or delete all objects and metadata in the registry. This is similar to functions we have seen in the discussion of UDDI – although of course more extensive, due to the abundance of information types that are managed. We have already mentioned the *life cycle* of an object, and will not discuss the interfaces and associated methods of the Object Management service.

We believe it is most useful to concentrate on the *Object Query Management Service*. This service allows a client to query RegistryEntities in several ways: a *Browse and Drill Down Query*, *Filtered Query*, or *SQL Query* the latter being an optional feature that may be supported by the registry. Additionally, there is a *Content Request Query*.<sup>93</sup> These four possibilities, together with a few examples, will be covered in the following subsections.

Note: It is worth mentioning that the *W3C XQuery* syntax is considered for a future version of the specification.

### 3.5.8.5.1 Browse and Drill Down Query

Using this query method, a “drills dow” of a Classification hierarchy in the registry is done. It is supported by three complementary types of requests<sup>94</sup>:

1. A list of *root classification nodes* (representing different classification schemes) can be obtained by sending a *GetRootClassificationNodes* request.
2. For a specific parent (possibly a root node), a client can get one or more of levels of its sub tree using a *GetClassificationTree* request.
3. Ultimately, the entries classified by the combination of one or more ClassificationNodes may be obtained with a *GetClassifiedObjects* request. All descendants of the specified nodes are returned.

### 3.5.8.5.2 Filtered Query<sup>95</sup>

A *Filtered Query* can be used in many different ways. The query is always directed against a single class, and specifies one or more *class filters*. A filter contains one of a set of predefined *predicate clauses*, specifying for instance that a string must contain a

---

<sup>93</sup> The specification is unclear about the *GetContentRequest* query, which can in fact considered as a special type of query (see the description). It is not clear whether support for this query type is required, but this seems logical.

<sup>94</sup> The numbering of the requests does not imply that they *have* be performed successively, but rather indicates a logical order.

<sup>95</sup> It is not perfectly clear, but the specification seems to differentiate between a *Filtered Query* and a *Filter Query*. In fact, *Filtered* seems to refer to a query that uses a Filter, which will be contained in an *AdHocQueryRequest*, and may be a *Filter Query*.

certain sub string, or a value must be larger than a certain value. We will meet some examples below.

A Filtered Query is always sent inside an *AdHocQueryRequest*. The response message with the query result for a Filtered Query may either contain the *content itself* of one or more repository items, together with some metadata, or *only metadata* (a set of *RegistryEntries*). The first case regards a *ReturnRepositoryItem* query, which specifies a *ReturnRegistryEntry* query (see below) to specify the items *and* RegistryItems to be retrieved. This is an alternative to the *GetContentRequest* query, which only returns the item's content.

In the latter case, the *AdHocQueryRequest* contains a *FilterQuery* element, which itself contains one of several types of queries, which each return instances corresponding to different types of registry objects we have seen above. In summary, these are:

- *RegistryEntryQuery*,
- *AuditableEventQuery*,
- *ClassificationNodeQuery*,
- *RegistryPackageQuery*,
- *OrganizationQuery*,

Additionally, there is a *ReturnRegistryEntry* query, which can be used to compose an XML document containing metadata, i.e. the information is described in the *RegistryEntry itself*. This is done by including a *RegistryEntryQuery* together with the desired type(s) of metadata.

Note: The specification itself makes it quite difficult to grasp, but we would like to illuminate the following: A *ReturnRepositoryItem* query is *contained* in an *AdhocQueryRequest*, and *contains* a *ReturnRegistryEntry* query, which *contains* a *RegistryEntryQuery*, that finally specifies the registry entry instances and their associated repository items.

To illustrate the querying capabilities, we will take two examples from the specification. The first example selects all classification nodes that are root nodes (parent = ""), and

with the name containing the words “product code” or “product type”). In fact, this selects the nodes contained in the *product code* and *product type* classification schemes.

```
<FilterQuery>
  <ClassificationNodeQuery>
    <ClassificationNodeFilter>
      (name CONTAINS "product code" OR
       name CONTAINS "product type")
      AND
      parent EQUAL ""
    </ClassificationNodeFilter>
  </ClassificationNodeQuery>
</FilterQuery>
```

**Example 15: ClassificationNodeQuery Example**

The second example uses a *ReturnRegistryEntry* request which will return an XML document containing a RegistryEntry for a certain Purchase Order DTD identified by “urn:com:xyz:po:325”, together with all its classification information and the associations to other DTDs by which it may have been superseded or replaced.

```
<ReturnRegistryEntry>
  <RegistryEntryQuery>
    <RegistryEntryFilter>
      id EQUAL "urn:com:xyz:po:325"
    </RegistryEntryFilter>
  </RegistryEntryQuery>
  <WithClassifications/>
  <WithSourceAssociations>
    <AssociationFilter>
      associationType EQUAL "SupercededBy" OR
      associationType EQUAL "ReplacedBy"
    </AssociationFilter>
  </WithSourceAssociations>
</ReturnRegistryEntry>
```

**Example 16: ReturnRegistryEntry Query Example**

### 3.5.8.5.3 Content Request Query

This is in fact a “special” kind of query, since it simply specifies a list object references for repository items that have to be retrieved. If no errors occur, the response message only contains the payloads in several MIME parts – referenced within a Manifest in the header, of course.

### 3.5.8.5.4 SQL Query

As an alternative syntax, a registry may also accept queries in SQL syntax. As long as the *XQuery* syntax is not available, this provides an interesting way to specify more complex queries. An *SQLQuery* should be specified using a subset of the language, based on SQL '92, which is however extended to support *stored procedures* and certain special routines. It is also sent inside an *AdhocQueryRequest*. The exact syntax and the routines are specified in an appendix to the specification.

Note: Of course, the use of SQL queries does *not* imply that a relational database must be used to implement the repository.

The response message to an SQL query will always be a list of references to RegistryEntries. Since support for this type of queries is optional, we will only give an example. It returns a list of references (*id* values) for all RegistryEntries with a name containing “Acme”, and a version number greater than 1.3.

```
SELECT id FROM RegistryEntry WHERE name LIKE '%Acme%' AND
  objectType = 'ExtrinsicObject' AND
  majorVersion >= 1 AND
  (majorVersion >= 2 OR minorVersion > 3);
```

**Example 17: Registry SQL query**

### 3.5.8.6 Further Issues

The Registry Services Specification also includes a section about security. The current version says to take a “minimalist approach”, in that “Any known entity can publish content and anyone can view published content.”. There are four sections, dealing with the *Integrity of Registry Content* (which supposes at least that objects are digitally signed when submitted), *Authentication* on a per-request basis, *Confidentiality* and *Authorization*.

Since the specification continuously refers to the *ebXML Security Specification*, which is however not yet complete, and some issues are not even addressed yet, we will not discuss these matters here. Future versions of the ebXML Registry/Repository should include more security, and also things like a *Publish/Subscribe* service and *Logging* capabilities.

### 3.5.9 Marketing/Awareness and Education

It is obvious that the contribution of this project team to the specifications is minimal. Then, why do we even mention it ? Because, in our *personal* opinion, it seems to be the *ugly duckling* of the project teams, and should receive more attention. We think it is worth mentioning that, for instance, the navigation of the ebxml.org website could be made easier. There are many interesting slide shows and explanatory documents available on the ebXML.org website, but they are not uniform enough and are spread over different locations, so the problem is sometimes how to *find* them.

It would serve little use to explain the reasons for this opinion in detail, and of course, the situation may improve once the 18-month timeframe is over and the specifications are approved. However, as we have mentioned earlier, marketing is *not* to be neglected in developing standards. In fact, we hope that the information given in this thesis can provide a structured overview, together with a deeper examination of some aspects, in order to make life easier for people who want to get to know ebXML.

### 3.5.10 ebXML: Conclusion

After this voluminous discussion of the ebXML specifications, the reader's impression may be quite opposite to what it was after Part 2: while XML gave the impression of making everything so simple, this Part makes you realize that Rome still can't be built in one day, no matter how good a framework ebXML may be. However, it is good to recall the intended *modularity* of ebXML: not only this means that a company does not have to implement all specifications at the same time, but also that you do not have to understand them all in detail.

Instead, as with the XML standards, it is important to keep the "big picture" in mind of how ebXML is organized. What you should remember about the Business Process and Core Component teams, is that they respectively provide a methodology for process modelling and for defining core blocks of business data. This is the basis for defining the message content. The messages will be transported in a reliable, secure manner, according to the Message Service specification. In addition, a party's capabilities for

supporting business processes and the associated message exchange are specified by Collaboration Protocol Profiles. Two such profiles may be combined into a Collaboration Protocol Agreement to establish the “rules of engagement” for the actual interaction between two parties. Finally, the mechanism that connects all these parts, by providing storage for any data that must be shared, is the Registry.

If you recall the discussion about the value of EDI in developing new frameworks, you will certainly realize that ebXML has a good starting-point : the concepts in its semantic layer are borrowed from EDI (the *Open-edi Reference Framework*), while the XML framework itself is clearly related to the eCo Architecture. [Alshuler, 2000] Since we already mentioned ebXML several times when we developed our *generic framework*, it is not surprising that ebXML may indeed be considered as a “complete” framework, covering all relevant aspects of the B2B collaboration problem domain. The combined play of the *Business Operational View* and the *Functional Service View*, to constitute an architecture centered around *business transactions* and *collaborations*, provides a well-founded, overall approach to B2B eCommerce. Building on this framework, chances are real that a “*single, global electronic market*” will finally be born.

## **General Conclusion**

At the very start of this thesis, we have questioned the point of describing Internet-related technologies and standards, as any paper document is destined to be “outdated” by the time it appears under the eyes of the reader. Keeping this caveat in the back of our mind, we have tried both to leave out aspects that are obsolete, and to focus on topics which have a good chance “to make it” in the upcoming months or years.


Leaving the introductory *Part 1: General Overview of XML* out of the discussion now, the first main area of attention has been the *The Family of XML Standards* in Part 2. The importance of these standards is unquestionable: they provide the basis for almost every recent initiative that is concerned with electronic business. The foundations were formed by the XML specification itself (with its DTDs), and seem to be stable now. Equally important are the mechanisms for accessing and manipulating XML documents, offered by DOM, SAX, XPath and XSLT, which also have reached an appropriate level of maturity.

On the modelling side, the expressiveness and power of DTDs left a lot to be desired. Therefore, a very recent event that is crucial to the further development of XML-based initiatives, and eCommerce in particular, is the advancement of the *XML Schema* specification to *Recommendation* status on May 2, 2001. We have seen that these XML Schemas add essential features to XML modelling, which allow for a better definition of constraints, data typing, inheritance, and more powerful validation, among others. It’s a sign on the wall that the ebXML project teams recently added an XML Schema (where applicable) to several specification documents.

This ebXML framework is the central topic of *Part 3: Electronic Business and ebXML*. The reader might wonder why we have almost exclusively discussed this framework (apart from SOAP and UDDI), while there are many other initiatives that may deserve our attention. The main reason for this should already be clear: its broadness, the fact that it comprises the whole B2B collaboration area. However, there is an underlying, second reason: in our intention to provide the reader with information that is relevant *at the time he is reading it*, we have taken up the daunting albeit very interesting challenge to describe a framework that is not finished yet, or more precisely, that is *in full*

*development*. On one hand, this has greatly complicated the assignment (the reader should take a look at the dates and versions of the specifications, as noted in the text) and in principle there is the risk that one or more of the specifications will not be supported by the ebXML community on the final voting. On the other hand, chances are small that any *substantial* changes will be made over the specification versions used in this thesis. Moreover, the perspectives are optimistic, one example being that an important vertical framework, *RosettaNet*, has recently announced support for the ebXML Message Service specification.

We believe it would serve little use to repeat the ebXML components and mechanisms once more – as the ebXML conclusion immediately precedes this general conclusion. Rather it should be mentioned that, in our optimism, we should not forget that the ebXML framework is *extremely* young: at the time of writing, the 18-month development timeframe has just reached its end with the final ebXML meeting in Vienna, where the approval of all specifications will be voted. While it is assumed that this approval will be no problem, we should not forget that the *only* implementation experience of ebXML is formed by a couple of Proof-of-Concept demos. These demos, in which many software industry giants have participated, demonstrate the viability of ebXML, not in the least because they show that specifications from initiatives like *RosettaNet*, the *Open Applications Group* and others can indeed be successfully “plugged in” into ebXML horizontal framework. And yet, the mere fact that these are *only* demos brings us right back down to earth: it will be up to vendors, end-users and others to correctly implement and use the ebXML specifications. The upcoming months will be ever more exciting, as they will show *if* and *how* electronic commerce between companies all over the world may be facilitated, and thereby enabled, using the ebXML framework described in this thesis.



## **Last-Minute Appendix**

As the last pages of this thesis were rolling out of the printer, an ebXML Press Release about the approval of the ebXML specifications on the ebXML meeting in Vienna was issued. It is unfortunately too late to incorporate this information into the thesis, but at the same time it would be wrong not to mention it. Therefore, we have included the full text of the press release here. For further information we refer to the website, <http://www.ebxml.org>.

### **ebXML Approved**

#### **UN/CEFACT and OASIS Deliver on 18-Month Initiative for Electronic Business Framework**

Geneva, Switzerland and Boston, MA, USA; 14 May 2001 -- UN/CEFACT and OASIS today announced that participants from around the world approved ebXML specifications at a meeting in Vienna, Austria on 11 May 2001. ebXML, which began as an 18-month initiative sponsored by UN/CEFACT and OASIS, is a modular suite of specifications that enables enterprises of any size and in any geographical location to conduct business over the Internet. Using ebXML, companies now have a standard method to exchange business messages, conduct trading relationships, communicate data in common terms and define and register business processes.

"The level of involvement and cooperation between industry groups, vendors and users that has been demonstrated in the ebXML initiative is unprecedented," said Klaus-Dieter Naujok of IONA, chair of ebXML and member of the UN/CEFACT Steering Committee. "We applaud the work of all the organizations around the world who have collaborated on the development of ebXML."

ebXML specifications and technical reports are available at no charge on <http://www.ebxml.org>. Approved specifications include ebXML Technical Architecture, Business Process Specification Schema, Registry Information Model, Registry Services, ebXML Requirements, Message Service and Collaboration-Protocol Profile and Agreement. Accepted ebXML Technical Reports include: Business Process and Business Information Analysis

Overview, Business Process Analysis Worksheets & Guidelines, E-Commerce Patterns, Catalog of Common Business Processes, Core Component Overview, Core Component Discovery and Analysis, Context and Re-Usability of Core Components, Guide to the Core Components Dictionary, Naming Convention for Core Components, Document Assembly and Context Rules, Catalogue of Context Drivers, Core Component Dictionary, Core Component Structure and Technical Architecture Risk Assessment.

" ebXML implementations are already being announced, and we expect the rate of deployment to accelerate quickly," said Patrick Gannon, chair of the OASIS Board of Directors. Mr. Gannon pointed to recent announcements of ebXML integration and support from industry groups that represent a significant variety of user communities. RosettaNet, a consortium of more than 400 companies in information technology, electronic components and semiconductor manufacturing, plans to integrate support for the ebXML Messaging Services Specification in future releases of RosettaNet's Implementation Framework (RNIF). The Global Commerce Initiative, which represents manufacturers and retailers of consumer goods, chose to base their new Internet protocol standard for trading exchanges and B2B communications on ebXML. Other industry organizations, such as the Automotive Industry Action Group, Health Level Seven, Open Applications Group, Open Travel Alliance, SWIFT and formal international and North American EDI standards bodies, have also been active participants in the ebXML initiative.

Adoption, implementation and maintenance of the ebXML specifications will be conducted by UN/CEFACT and OASIS under the auspices of a Memorandum of Understanding, which was signed by the two organizations in Vienna. Coordination of this work will be achieved through a joint UN/CEFACT and OASIS management committee.

[[http://www.ebxml.org/news/pr\\_20010514.htm](http://www.ebxml.org/news/pr_20010514.htm)]

## **List of Figures**

<i>Figure 1: Tree structure of the preliminary FamousQuotes XML document. ....</i>	<i>12</i>
<i>Figure 2: The XML example after applying the stylesheet .....</i>	<i>15</i>
<i>Figure 3: Snapshot overview of the XML Family .....</i>	<i>65</i>
<i>Figure 4: The eCo Architecture .....</i>	<i>85</i>
<i>Figure 5: Circular view of Electronic Business Collaboration .....</i>	<i>90</i>
<i>Figure 6: B2B Collaboration Process, with ebXML Specifications fitting in.....</i>	<i>97</i>
<i>Figure 7: Business Transactions as seen by the Open-edi Reference Model.....</i>	<i>100</i>
<i>Figure 8: Use Case for UN/ CEFACT Modelling Methodology.....</i>	<i>102</i>
<i>Figure 9: ebXML Business Operational View .....</i>	<i>104</i>
<i>Figure 10: ebXML Functional Service View.....</i>	<i>107</i>
<i>Figure 11: "Pyramid" layer view of the ebXML specifications .....</i>	<i>112</i>
<i>Figure 12: Relationship of Business Process Specification with UMM and CPP/CPA .....</i>	<i>113</i>
<i>Figure 13: Reuse in the Business Process Model .....</i>	<i>115</i>
<i>Figure 14: Choreography of Business Signals.....</i>	<i>119</i>
<i>Figure 15: Components of the Message Service Handler.....</i>	<i>129</i>
<i>Figure 16: Basic SOAP Message Structure with Example .....</i>	<i>135</i>
<i>Figure 17: An example SOAP Architecture .....</i>	<i>138</i>
<i>Figure 18: ebXML Message Structure .....</i>	<i>143</i>
<i>Figure 19: Forming a CPA from T wo CPPs.....</i>	<i>156</i>
<i>Figure 20: The use of SOAP in UDDI .....</i>	<i>163</i>
<i>Figure 21: A Business Registration, contained in a businessEntity element .....</i>	<i>164</i>
<i>Figure 22: The roadmap for UDDI.....</i>	<i>169</i>
<i>Figure 23: Registry Information Model - High level public view .....</i>	<i>173</i>
<i>Figure 24: ebXML Registry Interfaces.....</i>	<i>178</i>

## **List of Examples**

<i>Example 1 : Preliminary version of a FamousQuotes XML Document.....</i>	<i>11</i>
<i>Example 2 : A sample XSL Stylesheet .....</i>	<i>15</i>
<i>Example 3 : The DTD for the FamousQuotes Collection .....</i>	<i>25</i>
<i>Example 4: The XML Schema for the FamousQuotes Collection.....</i>	<i>34</i>
<i>Example 5: The (simplified) "output" of a SAX parser. ....</i>	<i>41</i>
<i>Example 6: XPath Expression Examples .....</i>	<i>44</i>
<i>Example 7: The XSLT example revisited.....</i>	<i>48</i>
<i>Example 8: An XSLT Stylesheet transforming XML data into SQL entries. ....</i>	<i>54</i>
<i>Example 9: A simple XQuery example – taken from example "Q9" in the XQuery Working Draft.....</i>	<i>58</i>
<i>Example 10: XSLT version of example "Q9" in the XQuery Working Draft. ....</i>	<i>60</i>
<i>Example 11: Illustration of XLink.....</i>	<i>64</i>
<i>Example 12: Structure of a sample Business Process Specification.....</i>	<i>121</i>
<i>Example 13: Context Rules Example .....</i>	<i>127</i>
<i>Example 14: Structure of the Collaboration Protocol Profile (CPP) (Source: Adapted from the Collaboration-Protocol Profile and Agreement Specification Version 0.95).</i>	<i>152</i>
<i>Example 15: ClassificationNodeQuery Example .....</i>	<i>181</i>
<i>Example 16: ReturnRegistryEntry Query Example.....</i>	<i>181</i>
<i>Example 17: Registry SQL query.....</i>	<i>182</i>

## **Sources**

Note: Since almost all topics covered in this thesis are the result of very recent (or even incomplete) work, the Internet has served as the main source of information. Therefore, a list of **Websites** is first given, which serve as a general source of information, containing the specifications, background information and links to reference material. Since the exact location of web documents could change, these websites should be used as a starting point. Following, the list of sources is subdivided in three sections: *Specifications and Whitepapers*, *Articles*, *Books*. The **Specifications** form the actual basis for most of the writings, and as such their correctness is usually unquestioned – as far as they are finished, of course. The term **Whitepapers** should be understood as “documents from a reliable source (usually a standards organisation) which are not considered as a specification”. Although the **Articles** come from websites which cannot always be considered as “authorative”, comparison between several documents should have filtered out any imprecisions. As far as the **Books** are concerned, only one book is mentioned. I would like to stress that this book has *not* served as the major source of information, rather the opposite – in fact, it was already partly outdated at the time of writing.

### **a) Websites**

ebXML: <http://www.ebXML.org>

OASIS: <http://www.oasis-open.org>

UDDI: <http://www.uddi.org>

UN/CEFACT: <http://www.uncefact.org>

W3C : <http://www.w3.org>

#### *Additional:*

OASIS XML Cover Pages: <http://xml.coverpages.org> - This website is generally regarded as *the* starting point for information about XML- and SGML-related matters.

RosettaNet: <http://www.rosettanet.org>

Open Applications Group : <http://www.openapplications.org>

## **b) Specifications and Whitepapers**

### 1. W3C (in order of occurrence in the thesis)

- W3C. (2000) Extensible Markup Language (XML) 1.0 (Second Edition).  
<http://www.w3.org/TR> (Recommendation)
- W3C. (2000) *XHTML™ 1.0: The Extensible HyperText Markup Language - A Reformulation of HTML 4 in XML 1.0*. <http://www.w3.org/TR> (Recommendation)
- W3C. (1999) *HTML 4.01 Specification* <http://www.w3.org/TR> (Recommendation)
- W3C Recommendation. (1999) *Namespaces in XML*. <http://www.w3.org/TR> (Recommendation)
- W3C. (1999) *Associating Style Sheets with XML documents*. <http://www.w3.org/TR> (Recommendation)
- W3C. (1999). *XML Schema Requirements*. <http://www.w3.org/TR> (Note)
- W3C. (2001). *XML Schema Part 0: Primer*. <http://www.w3.org/TR> (Recommendation)
- W3C. (2001). *XML Schema Part 1: Structures*. <http://www.w3.org/TR> (Recommendation)
- W3C. (2001). *XML Schema Part 2: Datatypes*. <http://www.w3.org/TR> (Recommendation)
- W3C. (1998) *Document Object Model (DOM) Level 1*. <http://www.w3.org/TR> (Recommendation)
- W3C. (2001) *Document Object Model (DOM) Level 2 Specifications*.  
<http://www.w3.org/TR> (Recommendation)
- W3C. (1999) *XML Path Language (XPath) Version 1.0*. <http://www.w3.org/TR> (Recommendation)
- W3C. (2001) *XML Pointer Language (XPointer) Version 1.0*. <http://www.w3.org/TR> (Working Draft)
- W3C. (2000) *Extensible Stylesheet Language (XSL) Version 1.0* <http://www.w3.org/TR> (Candidate Recommendation)
- W3C. (1999) *XSL Transformations (XSLT) Version 1.0*. <http://www.w3.org/TR> (Recommendation)
- W3C. (2000) *XSL Transformations (XSLT) Version 1.1* <http://www.w3.org/TR> (Working Draft)
- W3C. (2001). *XML Query Requirements*. <http://www.w3.org/TR> (Working Draft)

W3C. (2001). *XML Query Use Cases*. <http://www.w3.org/TR> (Working Draft)

W3C. (2001) *XML Query Data Model*. <http://www.w3.org/TR> (Working Draft)

W3C. (2001) *XML Query Algebra*. <http://www.w3.org/TR> (Working Draft)

W3C. (2001) *XQuery: A Query Language for XML*. <http://www.w3.org/TR> (Working Draft)

W3C. (XML Linking Language (XLink) Version 1.0 <http://www.w3.org/TR> (Proposed Recommendation)

W3C. (2000) *Simple Object Access Protocol (SOAP) 1.1* <http://www.w3.org/TR> (Note)

W3C. (2000) *SOAP Messages with Attachments*. <http://www.w3.org/TR> (Note)

W3C. (2001) *XML Protocol Requirements*. <http://www.w3.org/TR> (Working Draft)

W3C. (2001) *XML-Signature Syntax and Processing*. <http://www.w3.org/TR> (Candidate Recommendation)

## 2. ebXML

Business Process Team. (2001). *The ebXML Business Specification Schema Version 1.0*. <http://www.ebxml.org/specdrafts/>

Core Components Team. (2001). *The ebXML CC Dictionary Entry Naming Conventions v1.0*. <http://www.ebxml.org/specdrafts/>

Core Components Team. (2001). *The ebXML Methodology for the Discovery and Analysis of Core Components v1.0*. <http://www.ebxml.org/specdrafts/>

Core Components Team. (2001). *The role of context in the re-usablility of Core Components and Business Processes v1.0*. <http://www.ebxml.org/specdrafts/>

Core Components Team. (2001). *The ebXML specification for the application of XML based assembly and context rules v1.0*. <http://www.ebxml.org/specdrafts/>

Transport Routing and Packaging Team. (2001). *The ebXML Message Service Specification v0.99*. <http://www.ebxml.org/specdrafts/>

Trading Partners Team. (2001). *The Collaboration-Protocol Profile and Agreement Specification Version 0.95*. <http://www.ebxml.org/specdrafts/>

Registry & Repository Team. (2001). *The ebXML Registry Information Model v0.90*. <http://www.ebxml.org/specdrafts/>

Registry & Repository Team. (2001). *The ebXML Registry Services v0.90*. <http://www.ebxml.org/specdrafts/>

ebXML. (2000a). *ebXML White Paper: Enabling Electronic Business with ebXML*.  
<http://www.ebxml.org/>

ebXML. (2000b). *ebXML Glossary v0.95*. <http://www.ebxml.org>

### 3. UDDI

UDDI. (2000). UDDI Technical White Paper. <http://www.uddi.org/>

UDDI. (2000). UDDI Data Structure Reference V1.0. <http://www.uddi.org/>

UDDI. (2000). UDDI Executive White Paper. <http://www.uddi.org/>

### 4. Other

eCo. (1999) *eCo Architecture for Electronic Commerce Interoperability*.  
<http://eco.commerce.net>

UN/EDIFACT. (1995) *UN/EDIFACT Glossary*. <http://www.unece.org/trade/untdid/>

## c) Articles

Alshuler L. (2000). *Schema Repositories: What's at Stake ?* <http://www.xml.com/>

Balen H. (2000). *Deconstructing Babel: XML and application integration*.  
<http://www.adtmag.com/>

Bordes W. & Dumser J. (2000) *SOAP: Simple Object Access Protocol*.  
<http://www.techmetrix.com/trendmarkers/>

Bos L. (1999). *December 1999 : Overview of XML Family of Standards*.  
<http://www.infoloom.com/gcaconfs/WEB/>

Box. D. (2000). *A Young Person's Guide to The Simple Object Access Protocol*.  
<http://msdn.microsoft.com/>

Dodds L. (2001a). *Time to Refactor XML?* <http://www.xml.com/>

Dodds L. (2001b). *Does XML Query Reinvent the Wheel?* <http://www.xml.com/>

Dongwon L. & Chu W. (2000) *Comparative Analysis of Six XML Schema Languages*.  
<http://cobase-www.cs.ucla.edu/>

Drummond R. (2001). *EbXML to Incorporate SOAP*. <http://www.advisor.com/>

DuCharme B. (2000). *Combining Stylesheets with Include and Import*.  
<http://www.xml.com/>

Dumbill, E. (2000). *Distributed XML*. <http://www.xml.com/>

- Glushko R. (2000). *How XML Enables Internet Trading Communities and Marketplaces*. <http://www.infoloom.com/gcaconfs/WEB/>
- Holman G. (2000). *What is XSLT?* <http://www.xml.com/>
- Kay M. (2001). *What kind of language is XSLT?* <http://www-106.ibm.com/>
- Kleinman R. (2001). *Vertical XML Standards and ebXML*. <http://www.sun.com/xml/>
- Kotok A. (1999). *XML and EDI Lessons Learned and Baggage to Leave Behind*. <http://www.xml.com/>
- Kotok A. (2000). *ebXML: Assembling the Rubik's Cube*. <http://www.xml.com/>
- Landgrave T. (2001). *XML: Explained by Tim Landgrave*. <http://www.techrepublic.com/>
- Lenz E. (2001) *XQuery: Reinventing the Wheel?* <http://www.xmlportfolio.com/>
- Maler E. (2001a). *XML Linking: State of the Art*. <http://www.sun.com/xml/>
- Maler E. (2001b) *XML Linking: An Executive Summary*. <http://www.sun.com/software/xml/>
- Parsia B. (2001). *Functional Programming and XML*. <http://www.xml.com/>
- Peat B. & Webber D. (1997). *Introducing XML/EDI...* <http://www.xmledi-group.org/>
- Prud'hommeaux E. (2000). *XML Protocol Comparisons*. <http://www.w3.org/>
- Skonnard A. (2000). *SOAP: The Simple Object Access Protocol*. <http://www.microsoft.com/>
- Sun Microsystems. (2001). *W3C XML Technical Recommendation Quick Reference*. <http://www.sun.com/xml/>
- W3C XSL Working Group. (1998). *The Query Language Position Paper of the XSL Working Group*. <http://www.w3.org/>
- W3C Press Release. (2000). *W3C Consortium Issues XML Schema as a Candidate Recommendation*. <http://www.w3.org/>
- Webber D. & Dutton A. (2000). *Understanding ebXML, UDDI and XML/edi*. <http://www.xml.org/>
- Wentworth E. (1991). *Introduction to Functional Programming Using Gofer*. <ftp://cs.ru.ac.za/>

#### **d) Books**

- Birbeck M., et al. (2000). *Professional XML*. Wrox Press Ltd Birmingham UK.