ceXML - an XML vocabulary for building and civil engineering

Reinout van Rees
Delft University of Technology - Faculty of Civil Engineering and Geosciences
Section Civil Engineering Informatics

ceXML - an XML vocabulary for building and civil engineering

by Reinout van Rees

Published August 2000

Dedication

For Annie, who wants to marry me the 10th of November 2000 and for my parents, for their support all those years!

Table of Contents

Preface 15

1. General introduction 15
2. Structure of the document 15
3. Typographical conventions 16
4. Personal remarks 16

I. Outlining the problem 17

1. eConstruct project 19
    - 1.1. What will eConstruct do 19
    - 1.2. Relevance to this graduation project 19
2. Current state of communication in the building and construction industry 21
    - 2.1. Usage of communication technologies in the building and construction industry 21
        - 2.1.1. The nature of the industry 21
        - 2.1.2. Communication defined 22
        - 2.1.3. The current state of electronic communication 23
        - 2.1.4. Specific problems for European communication 24
    - 2.2. Existing solutions for electronic communication 25
        - 2.2.1. Standard for the exchange of product model data (STEP) 25
        - 2.2.2. Industry Foundation Classes (IFC) 26
        - 2.2.3. Electronic Data Interchange (EDI) 26
        - 2.2.4. Use of XML within IFC, EDI and STEP 27
    - 2.3. Conclusion 27
3. The goal of this research project 29
    - 3.1. Suggestions for improvement 29
    - 3.2. Possible advantages 30
    - 3.3. Description of the goal 32
        - 3.3.1. Discussion of the actual need 32
        - 3.3.2. Short goal description 33
        - 3.3.3. The objectives of this research 33

II. The building stones: XML and other initiatives 35

4. Introduction on communicating with a vocabulary written in XML 37
    - 4.1. Internet technology as a possible new solution 37
    - 4.2. General working of XML 38
    - 4.3. Making an XML vocabulary 40

4.4. Conclusion 40
5. Initiatives which influenced ceXML 43
5.1. Electronic Business XML (ebXML) 43
5.1.1. Introduction on ebXML 43
5.1.2. ebXML's context mechanism 44
5.1.3. Semantic level 46
5.1.4. Business processes 47
5.1.5. Influence on ceXML 47
5.2. Global Engineering Networking (GEN) 48
5.2.1. Introduction on GEN 48
5.2.2. Architecture of GEN 49
5.2.3. Usage of GEN 50
5.2.4. Influence of GEN on ceXML 51
5.3. LexiCon 52
5.3.1. Inner workings of LexiCon 52
5.3.2. Underlying model 53
5.3.3. Functional unit and technical solution 54
5.3.4. Influence on ceXML 55
5.4. Conclusions 55
III. Design and implementation of the vocabulary 57
6. Design of the vocabulary 59
6.1. Detailed requirements specification 59
6.1.1. Goals 59
6.1.2. Requirements 60
6.2. Overview of the structure 61
6.3. Business Processes model 62
6.4. The envelope model 64
6.5. The message model 65
6.5.1. Difference between a model and a view 65
6.5.2. The need for both a model and a view 66
6.5.3. The model 67
6.5.4. The view 72
6.6. Selection of elements 75
7. Prototype implementation 79
7.1. Implementation method 79
7.1.1. Unix style of programming 79
7.1.2. Programming language and tools 79

7.1.2.1. XSL/T 80
7.1.2.2. Python 80
7.1.2.3. Sed 81
7.1.2.4. CSS 81
7.1.3. Usability 81
7.2. Generic utilities 82
7.2.1. ceXML convenience functions 82
7.2.2. Selecting a language 83
7.2.3. Propagation of characteristics 83
7.2.4. Rejoining of characteristics 84
7.2.5. Conclusions 84
7.3. Dealing with the vocabulary and the views 84
7.3.1. The DTD of the message vocabulary 85
7.3.2. The filling of the message vocabulary 85
7.3.3. Generating the uni-lingual views 85
7.3.4. Creating the English and Dutch catalogs 86
7.3.5. Conclusions 86
7.4. Visualising 86
7.4.1. Visualising the filled message vocabulary 86
7.4.2. Visualising the catalogs 88
7.4.3. Conclusions 90
7.5. Using business processes and contexts 90
7.5.1. The message that is send 91
7.5.2. Sending and receiving the message 93
7.5.3. Processing the message 93
7.5.4. Sending a reply 94
7.5.5. Conclusions 95
7.6. Conclusions 96
8. Conclusions 97
8.1. Conclusions from Part I in *ceXML - an XML vocabulary for building and civil engineering*: *outlining the problem* 97
8.2. Conclusions from Part II in *ceXML - an XML vocabulary for building and civil engineering*: *the building stones: XML and other initiatives* 97
8.3. Conclusions from Part III in *ceXML - an XML vocabulary for building and civil engineering*: *Design and implementation of the vocabulary* 98
8.4. Successfulness of this research project 98
8.4.1. Investigating the state of the art 99

8.4.2. The design and implementation of a vocabulary 99
8.4.3. Prototype implementation..100
8.5. Perspective for the building and construction industry..........................100
8.6. My personal appraisal of this research project 101
IV. Appendices 103
A. Deeper introduction on XML and related technologies...........................105
A.1. eXtensible Markup Language (XML) itself.................................105
A.1.1. Introduction to XML...105
A.1.2. Structure of XML...106
A.1.3. Usage of XML 107
A.2. eXtensible Stylesheet Language/Transformation part (XSL/T) 108
A.2.1. Background 108
A.2.2. Working of XSL/T 109
A.2.3. Example usage: transforming database output...................110
A.2.4. Compiling stylesheets 113
A.2.5. Possible use for ceXML...113
A.3. Stylesheet languages for visualisation 114
A.3.1. Difference Cascading Style Sheets (CSS) and eXtensible Stylesheet Language/Formatting Objects (XSL/FO)............114
A.3.2. Usage of stylesheets...114
A.3.3. Use for ceXML 115
A.4. XML Namespaces...116
A.4.1. Working..116
A.4.2. Design decisions involving namespaces 117
A.4.3. Conclusions..117
A.5. XML Linking 117
A.5.1. Introduction..118
A.5.2. Working..118
A.5.3. Possible uses 119
A.5.4. Possible use for ceXML...120
A.6. Cocoon - Apache’s xml effort..120
A.6.1. Background 121
A.6.2. Working of cocoon...121
A.6.3. Possible use for ceXML...121
B. STEP explained in more detail..123
C. Reading Unified Modelling Language (UML) diagrams..........................125
C.1. Use Case diagram...125

C.2. Class diagram 125
D. Python listings 127
D.1. cexmlhelper.py 127
D.2. dtdbuilder.py 128
D.3. treeview.cgi 129
D.4. print.py 130
E. XSL/T listings 131
E.1. language-.xslt 131
E.2. propagate.xslt 132
E.3. rejoin.xslt 134
E.4. search.xslt 138
E.5. print_answer.xslt 140
F. Document Type Definitions (DTD’s) 143
F.1. cexml_message.dtd 143
F.2. cexml_en.dtd 145
F.3. cexml_nl.dtd 146
F.4. pce_en_procurement.dtd 147
G. XML files 151
G.1. cexml.xml 151
G.2. catalog.xml 155
Bibliography 157
Glossary 159
Colophon 165

List of Figures

3-1. PalmPilot with project management software ... 31
5-1. Logical architecture of GEN according to [Radeke, 1998] ... 49
5-2. Handling of data in GENial ... 50
5-3. The GENial interface ... 51
5-4. The structure of LexiCon ... 53
5-5. UML diagram of the LexiCon meta-model ... 54
5-6. Explanation functional unit and technical solution ("the hamburger model") [W. Gielingh, 1988] ... 55
6-1. Overview of the relations between the Business Process, the message and the envelope vocabulary ... 61
6-2. Business Process Use Case 1 ... 62
6-3. Business Process Use Case 2 ... 63
6-4. Business process model ... 64
6-5. Model of the envelope vocabulary ... 65
6-6. The message model - 1 ... 68
6-7. Example of a tree structure ... 68
6-8. The message model - 2 ... 69
6-9. The message model - 3 ... 70
6-10. The message model - 4 ... 71
6-11. The message model - 5 (final version) ... 71
6-12. Both Use Cases together ... 72
6-13. Initial version of the example view ... 73
6-14. Initial version of the example view - XML version ... 74
6-15. Definitive version of the example view ... 75
6-16. Definitive version of the example view - XML version ... 75
7-1. Dutch view on the filled message vocabulary ... 87
7-2. English view on the filled message vocabulary ... 87
7-3. The Dutch catalog in English ... 89
7-4. The content of both catalogs in English ... 89
7-5. The working of the context-manager ... 91
7-6. Screen-shot of the printed answer in Acrobat Reader ... 94
C-1. Use Case diagram example ... 125
C-2. Class diagram example ... 126

List of Examples

4-1. XML example - The plain information without any indication about what it really is ... 38
4-2. XML example - The same information, now tagged to make an xml file out of it. ... 39
7-1. request1.xml ... 92
7-2. sender.py ... 93
A-1. XML syntax example ... 106
A-2. XSL/T - basic example ... 109
A-3. XSL/T example - database table structure ... 110
A-4. XSL/T example - xml file before transformation ... 110
A-5. XSL/T example - xml file after processing with cocoon's SQL processor ... 111
A-6. XSL/T example - the stylesheet ... 112
A-7. XSL/FO example ... 114
A-8. CSS example ... 115
A-9. Example use of namespaces to distinguish between two title tags, one indicating a chapter's title, the other indicating a person's title. ... 116

Preface

Engineers never lie, we just approximate the truth.

1. General introduction

This document is my Master's Thesis (*afstudeerverslag*) for Civil Engineering Informatics. The work for this thesis for the largest part also constituted my work for the eConstruct project. This is a European project for the building and construction industry. ceXML serves as a first prototype of eConstruct's own XML vocabulary, bcXML. It shows off several implementation possibilities. Working in this international project brought me in contact with a whole new research world, which I liked, therefore prompting me to continue as a PhD student on the same subject.

My supervisors are:

- Prof. ir. Frits Tolman
- Dr. ir. Reza Beheshti
- Dr. ir. Michel Böhms

2. Structure of the document

Chapter 1 describes the eConstruct project. Started in January 2000, the eConstruct (Electronic business in the building and CONSTRUCTion industry: preparing for the new internet) project aims to make electronic communication a reality in the - still mainly paper based - building and construction industry. During my graduation, I was enlisted in this project and I will also continue on eConstruct as a PhD student.

Chapter 2 is a survey on the current state of communication in the building and construction industry. A comparison with other industries clearly outlines the specific problems present in the building and construction industry. This chapter

zooms in mainly on the *electronic* communication, this being the subject of this project.

The purpose of Chapter 3 is to serve as a guideline for the rest of the document, showing the direction the research has to take.

XML is a major building stone for this research project and therefore deserves a chapter of it's own describing it. (Chapter 4)

The purpose of Chapter 5 is to provide a basis for the design and implementation described in Chapter 6 and Chapter 7. My design and implementation are influenced mainly by three initiatives, who will serve as the underlying basis: ebXML (electronic business XML), GEN (Global Engineering Networking) and the LexiCon (a set of programs made by STABU).

Chapter 6 describes the underlying design of the vocabulary: the models. Each model is depicted using an Unified Modelling Language (UML) diagram.

The goal of Chapter 7 is to describe the implementation of the vocabulary of the previous chapter. It has to be a proof-of-concept.

The last chapter first lists the most important conclusions from the entire document. This is followed by a comparison of the results and the original objectives. It closes with a personal appraisal of the research project.

3. Typographical conventions

There are a few typographical conventions I used throughout this document. Text in *italics* is either important or it signifies a specific concept, like a *floor to stand on*. Text in `mono-type font` signifies computer code or tag names. There is a difference between built object and `builtobject`. In the latter case, I indicate the element from an XML file.

4. Personal remarks

Thanks go to Jan Peter, Jako, Reza, Michel, Frits, Martijn, Maurits, Herman and Annie for proofreading this document and providing useful input and for finding many errors.

I. Outlining the problem

Table of Contents

1. eConstruct project 19
2. Current state of communication in the building and construction industry 21
3. The goal of this research project...29

Chapter 1. eConstruct project

"If hate and war could solve anything, don't you think they'd have solved it a long time ago?"
- Geoff Mann

Started in January 2000, the eConstruct (Electronic business in the building and CONSTRUCTion industry: preparing for the new internet) project aims to make electronic communication a reality in the - still mainly paper based - building and construction industry. Wide-spread electronic communication will greatly improve the effectiveness and efficiency of both the industry as a whole and individual companies.

1.1. What will eConstruct do

eConstruct will design and build a vocabulary for electronic communication. For this, eConstruct will use the Internet and especially the new XML[1] (eXtensible Markup Language) technology. Just like a vocabulary for the language of a human being, this one defines what can be communicated about buildings, bridges, construction contracts, progress reports, etcetera, but now by computers. Software will be made and adapted to show the feasibility of the concept. This will not be "just a good concept", it will be supported by applications and integrated in a number of existing platforms. eConstruct will actively push adoption of the technology.

1. XML is explained in Section 4.2.

1.2. Relevance to this graduation project

For my graduation, I have been enlisted into the eConstruct project. After graduation, I will continue working on eConstruct as a PhD student. For eConstruct I have mainly investigated XML and related technologies, many of which findings have been used in eConstruct's first report. This report contains basic information about XML and the surrounding standards, including how best to apply them. Related initiatives are also highlighted.

ceXML serves as a prototype for eConstruct's own XML vocabulary, bcXML (building and construction XML). The intended purpose of my graduation project was to have me investigate a lot of possible technologies, to make me grab a few useful ones and to let me build a prototype with it. This should produce a good insight in the usability of the chosen technologies and provide useful input for the eConstruct project.

This document contains most of the text I produced for eConstruct. The graduation project itself basically is a stripped-down and somewhat premature version of eConstruct: to investigate XML, to make a simple vocabulary and to make an prototype implementation, showing the feasibility and usability of the vocabulary.

Chapter 2. Current state of communication in the building and construction industry

Lies, damned lies and project estimates

This chapter is a survey on the current state of communication in the building and construction industry. A comparison with other industries clearly outlines the specific problems present in the building and construction industry. This chapter zooms in mainly on *electronic* communication, the subject of this project.

First, the characteristics of the industry are outlined. This is followed by three current solutions to the electronic communication problem. They haven't gained widespread acceptance in the building and construction industry because of the specific industry characteristics.

2.1. Usage of communication technologies in the building and construction industry

This section outlines the nature of the industry, because this nature brings about specific problems which have a large impact on the kind of communication. A definition of the communication at hand and the current state is also provided.

2.1.1. The nature of the industry

The nature of the building and construction industry is best described by noting the differences with other major industries:

- There are no players large enough to drive and prescribe certain technological developments. Even the big players are small compared to the combined size of all small 1-to-10-men companies and especially when compared with the Volkswagens, Unilevers and Nokias of other industries.
- The profits generated by the industry generally aren't high, so no large investment to solve big generic problems is available[1].
- The supplier and subcontractor relationships are manifold, ever changing and diverse, resulting in more complex communication and supply chains than in most other industries.
- Though many who work in the building and construction industry are highly skilled, most of them are not highly educated[2]. Most of them are blue collar workers with a few white collars working at the company's office. Especially the large amount of very small companies consists almost solely of blue collar workers. For the blue collars, using modern computer equipment and it's possibilities will, probably correctly, not be the first natural thing on their mind.

2.1.2. Communication defined

To prevent confusion, we first need to define *communication technology*. This is basically the use of technologies to communicate with other people, thus, everything from the postal service to wireless phones and Internet. This report restricts itself to the electronic kind of communication and specifically the indirect

1. An exception is when a specialised piece of equipment is needed for a large project. Or that the project itself requires a huge and expensive planning effort. See the storm surge barrier in the Scheld estuary in the Netherlands, where a few vessels were purpose-build for that single project and were not used ever since. These huge investments are normally covered by the project's price tag. The incentive for the one paying the constructor for a specific project *also* to pay for the costly development of a solution to a generic problem that persists across the entire industry will be low. The conclusion is that there are no large investments available to solve big generic problems.

2. By highly educated I mean those who have studied civil engineering, management etc. as opposed to those who learned to be masons, road builders, etc. The distinction is important, for (in most cases) the first group will have had much more contact with computers etc. than the second group.

way of communication. Telephones and such are not considered, because they are just a different way of talking directly. So it is about electronically exchanging information that is normally communicated on paper.

Written information is very important in building and construction. Many things have to be explicitly stated on paper because of the following reasons:

- One needs to be able to verify the safety of a construction. For example the structural strength of a building, the fire resistance, the resistance of a stadium to supporters jumping up and down in unison. This verification cannot be performed without specific, written data.
- The legal system, government regulations and the current building practice need verifiable information, specifications, written deadlines, etcetera. Legal reasons in this case mean both commercial legislation (enforcement of contracts, agreed-upon standards of work) and building legislation (regulations for building safety, strength, etcetera).
- The complexity of construction projects and the multitude of partners co-operating makes it impossible to rely on just a few people's minds to store all the required information.

In the building and construction industry, there is a huge amount of information that needs to be communicated. Technical data is needed by the subcontractors. Also the safety checkers need to be able to verify exactly the correctness of all technical data. The project planning must be communicated to every worker, taking the form of work orders. Also suppliers need to know when to deliver which goods. To stress it again, all this has to be done in writing.

Current practice in the building and construction industry is mainly paper-based. If an electronic way of communication could reach the same level of trust and - also important - the same legal status, it could replace much of the paper-based communication, with it's advantages of speed and accuracy.

2.1.3. The current state of electronic communication

For electronic communication (also: using computers) between partners, the only two presently viable options are the following:

Standardise on a specific platform and one specific set of applications:

> This option leads to enormous costs for firms, because they practically need to buy every major program in existence since most building firms take part in multiple projects. Every project may standardise on a specific solution, but for every project this will be a different choice, hence the problem.

Use the lowest common denominator between various platforms and applications:

> The problem with this option is that the common subset supported by all available programs is too small. If you strip off every program's functionality that isn't supported by all other programs, the functionality drops enough to render the program unusable.

As both options are entirely unattractive, the effect is that the communication is *not done electronically*. It is not feasible either because of the costs involved, or because practically no sensible communication is possible. The majority of communication is done on paper. Because the distribution of paper based documents is slow (using a fax is the fastest method) and takes effort, getting the right information to the right persons on time is difficult. This lowers the effectiveness and efficiency of the building and construction industry when compared to other industries. Electronic communication can counter this by allowing for an automated distribution of information with unprecedented speed, provided that a suitable way of communicating electronically is found.

2.1.4. Specific problems for European communication

Communicating in a European setting poses two additional problems, the language problem and the classification problem.

The language problem is quite straightforward. A *door* in English is a *deur* in Dutch and a *Tür* in German. The problems are much bigger when it is no longer just a case of translating one word into another. Different languages (and countries) often attach different meanings to concepts. A *concept* can be something that is easily understood, like a *door*[3] . But when you talk about the *first floor*, some languages mean the floor situated at street level, while other languages use the concept *first*

3. Even with an easy concept like *door*, you have the problem of whether or not you just mean the actual door or also the door frame, the lock and the paint work.

floor to indicate the first floor above street level, basically counting from zero. Also, when talking about *foundation* and *superstructure*, one country indicates everything mounted upon the foundation poles with *superstructure*, while another country includes the bottom floor when talking about *foundation*.

Communication across borders requires a mechanism to provide a mapping from one concept onto another, which is a sizable job in Europe.

2.2. Existing solutions for electronic communication

Already some initiatives have tried to facilitate electronic communication: IFC (Industry Foundation Classes, Section 2.2.2) which is specific to the building and construction industry, STEP (STandard for the Exchange of Product model data, Section 2.2.1) and EDI (Electronic Data Interchange, Section 2.2.3), with the last two being more generic approaches. These three initiatives are introduced in the following three sections.

2.2.1. Standard for the exchange of product model data (STEP)

The STandard for the Exchange of Product model data (STEP) constitutes a standard way of dealing with product data. The STEP standard is very strict. The fact that testing for conformance to standard is build into the standard testifies of it's solidity. A detailed outline of the inner workings of STEP can be found in Appendix B.

Practically, STEP is used by the major players in the automobile, shipbuilding, oil drilling and airplane manufacturing business. They prescribe STEP to their selected suppliers, which they are able to do by means of their sheer size. Given the limited use they get out of it, the software is too expensive for small companies (like many

building and construction companies!), since it doesn't add much in the sense of additional profits. The variety of contacts, suppliers, goods, one-off deals etc. is simply too big to achieve economy of scale on a sufficient large number of them to warrant using a standardised STEP product exchange with an expensive software package.

2.2.2. Industry Foundation Classes (IFC)

IFC (Industry Foundation Classes) is a counterpart of STEP, especially for the building and construction industry. It is meant for representing electronically *things* that occur in a constructed facility, both physical things like doors and floors and abstract things like organisation, space, cost. Everything is divided into classes. A class describes common characteristics of a range of similar things. For instance, a class can describe load-bearing things like floors, bridges, elevators. It concentrates on the load-bearing characteristics those things have.

Classes can describe both physical objects (like *a bridge*) and abstract attributes etcetera (like *the costs*). So, multiple classes together can provide the information associated with one physical object. Attributes added to a class help to describe the object further, like *length* and *height* of a bridge.

Relationships also are defined in IFC (in data). For instance, a *light bulb* is *operated* by a *switch*. Relationships are important in defining object behaviour in ways that mimic the behaviour of the real world artifacts.

To provide more useful and controlled access to the objects, interfaces are defined. For each useful viewpoint, an interface is provided. For instance, from the architectural point of view , the shape and location should be accessible. And from the cost viewpoint, information about the costs, maintenance frequency, etc. is useful.

IFC is quite usable, but - as is the case with STEP - the effort and costs of using IFC software seem too large for the small companies, because of the same reasons.

2.2.3. Electronic Data Interchange (EDI)

EDI (Electronic Data Interchange) is the structured exchange of data between

applications in different companies [Raman, 1999]. The data is transmitted using pre-defined codes in one big unreadable string (that is, not readable by humans). It is used by many of the largest companies to link to their suppliers, replacing paper work orders, documents, confirmations, etc. By allowing computers to talk to each other without human intervention, the communication rolls along with fewer errors and human mistakes. *Garbage in, garbage out* can be prevented in this way.

Communication between computers needs to be done in codes, at least that's the EDI viewpoint. In EDI the standard set of codes (provided by EDIFACT, Electronic Data Interchange For Administration, Commerce and Trade) is used for the common, generic usage. A lot of companies have made their own code tables, extending EDIFACT, to suit their specific needs. Since both communicating sides need to understand what is communicated, openness about the product data is needed. Many times so-called EAN (European Article Number) numbers are used when indicating products. These EAN numbers are the numbers used on barcodes.

The current practice in EDI shows that it is mostly used by big companies who prescribe their EDI messaging to their suppliers. Because EDI has to be integrated pretty well with the background inventory system and similar systems, this excludes (again) the many small businesses. Only if a lot of them get together and come up with a common system, it is feasible.

2.2.4. Use of XML within IFC, EDI and STEP

All these initiatives are now trying to map their systems into xml vocabularies, but using the same system with a new XML overcoat will not change the *system itself*, though it may make it more accessible. The system itself, however, is not available to facilitate cheap and easy access to information.

2.3. Conclusion

Three major existing solutions are available. Due to it's specific nature (fragmented, low budget, many relationships), the building and construction industry isn't using any of these solutions. So there is no industry-wide standard for the building and

construction industry and none of the three existing solutions comes close to being that industry-wide standard.

The first conclusion is that the building and construction industry lags behind when compared with other industries. The second conclusion is that a simple, cheap electronic solution is needed. The third is that a mapping of languages and concepts onto each other is needed.

Chapter 3. The goal of this research project

"If it moves, salute it. If it doesn't move, pick it up. If you can't pick it up, paint it" - U.S. military slogan

The purpose of this chapter is to serve as a guideline for the rest of the document, showing the direction the research has taken.

This chapter starts of with showing possible improvements, followed by two examples. It ends with a formal goal description for this research project.

3.1. Suggestions for improvement

In the previous chapter, it has been shown that the building and construction industry lags behind when compared with many other industries regarding electronic communication. This can and must be remedied, as it is a known fact in this industry that the lack of good communication is estimated to raise the costs of building up to 100% when compared to an ideal case.

A second field in which improvement has to be made is internationalisation. Especially in Europe, the industry is becoming more and more internationally oriented. Major projects have to be offered by tendering[1] to all interested parties in Europe. Suppliers are increasingly starting to operate internationally. At the same

1. Tendering means to *submit* a project proposal to a number of contractors. All major construction works have to be "tendered" Europe-wide to all interested constructors. They then are allowed to *accept* the tender and to present their bid (the amount of money) for which they are prepared to take on the project.

time, national languages, classifications and different legislations still hamper this trend.

The cost of connectivity has decreased the last few years through the rapid growth of the Internet and intranets. This opens up opportunities for an increase in both the quantity and the quality of communication, but the uptake till now has been somewhat disappointing. As an illustration, in the Netherlands, STABU (a Dutch organisation which builds an information system covering the entire building process) is trying hard to get all the building companies connected to the Internet before the end of the year 2000, as half of the companies doesn't yet have Internet access.

3.2. Possible advantages

An increasingly effective communication paves the way for severe reductions of lost time, less re-doing of work because of miscommunication, a much more effective and up-to-date planning, etc. A few examples might illustrate this more clearly:

- A cement truck is caught in a traffic jam and will be unable to deliver the cement before it has hardened. This causes the building crane to have nothing to do for three hours, which it has to catch up with later. Since the crane is on the critical path of the project, the entire project suffers a three hour delay. If the information about the delay had been available immediately *and* if the project planning had been easily accessible and adaptable, the crane might have gotten other work to do in the three hour time frame. Yes, a mobile phone could have been used to contact the lead contractor, but you still need a good, readily available and flexible project planning to be able to re-schedule on such a short notice *and* get the information to the right persons in time.
- Walking around on the building site, a supervisor notices an upcoming lack of roof tiles and so he grabs his PalmPilot, looks at the information about the roof, selects the list of tile elements, selects the right one and wirelessly orders another pallet of those tiles through the company ordering system. The company truck delivers them at the end of the morning, just before the current stock runs out.
- Many big projects are international projects, with the central office and planning/designing staff in one country and the actual working site in another

country. Once a supervisor finds a problem in the design and needs to have it redesigned, he normally has to make a phone-call to the central office, manage to contact the right person in charge, explain to him the exact problem *and* the exact part of the site that has the problem. Once all that has been made clear, the design has to be changed. The changed design is then printed out and *mailed* by post to the site, taking at least a day. Had the project information been available over the internet, the supervisor could have clicked on the specific part in a three dimensional view of the project, typed in his comments and send it off. This mechanism at least ties the correct information to the right part. After the design has been changed, the changes are automatically available at the site, because the central information has been made available over the internet.

- A certain piece of equipment has broken and you need a replacement meeting the original design criteria. It should not be larger, it should not weigh more and it should not use more energy. You contact a search engine on the internet which you feed with the design criteria that have to be met and it searches the various supplier's catalogs that are known to that search engine for parts meeting the criteria. It then visualises a list of possible replacements with their price-tag, possible delivery dates and additional characteristics and you can choose the item best fitting your schedule and your wallet.

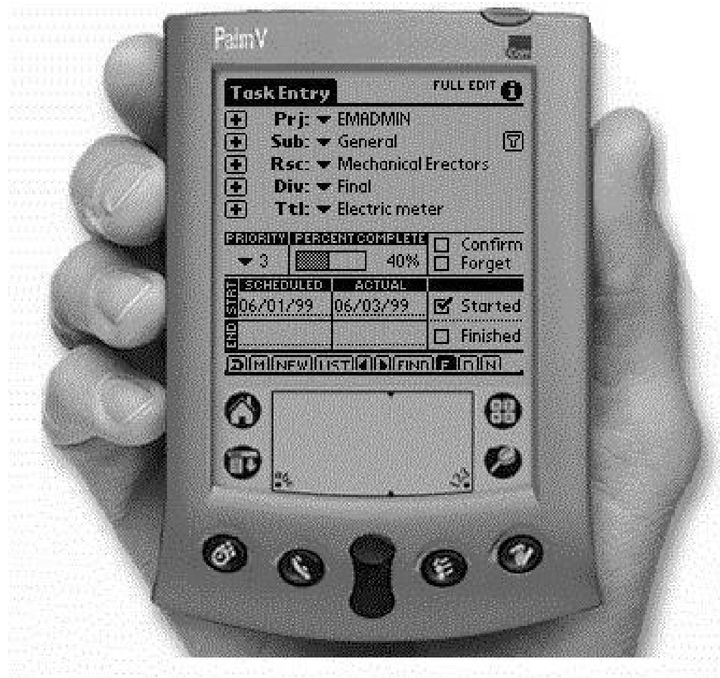

Figure 3-1. PalmPilot with project management software

3.3. Description of the goal

3.3.1. Discussion of the actual need

As said before, the building and construction industry needs to increase it's capacity to use communication and to provide information. This will lead to lower costs and better constructions. It has to be very affordable and easy to use, or the majority of small companies will not even bother to consider using new communication technology. So the Internet (with it's cheap and easy access) seems to be the best candidate for the position of communication medium. But the medium alone is not enough, a means to communicate the message is also needed, which in turn also has

to be affordable and easy. Simple HTML and webpages are not enough, HTML is only markup (how it looks like) and there is not a drop of meaning ("this is a door"). We can argue that XML will fulfil the need for meaningful communication.

3.3.2. Short goal description

The subject of this research is whether XML can be used to provide a common set of notions (a vocabulary) for the building and construction industry so that *meaningful* exchange of information by means of the Internet can become possible.

3.3.3. The objectives of this research

To restrict the scope of research, I compiled the following three objectives. They are guided partly by my role in the eConstruct project, (investigating XML and related technologies) and by the need to provide a prototype as a proof-of-concept.

- Investigating the state of the art of XML and related technologies, as well as related vocabularies and related developments like EDI (Electronic Data Interchange) and PDT (Product Data Technology). The Delft University of Technology had to investigate this for the eConstruct project. This investigation had to result in a public document outlining a 'baseline' for eConstruct. The content of Chapter 4, Chapter 5 and Appendix A has partly been used in eConstruct's first public document (eConstruct's deliverable D101, see www.econstruct.org).
- The design and implementation of a vocabulary, separate from eConstruct's work at making eConstruct's vocabulary (named bcXML, Building and Construction XML). My task is to make a simple prototype vocabulary, allowing the mapping of one language into another. Required input was the LexiCon meta-model (Section 5.3). The model has to deal with prefab concrete elements, a field eConstruct decided to concentrate upon for the time being, mostly because the Greek partner in the project is a supplier of prefab concrete elements.
- Testing of the vocabulary by means of a simple application. A vocabulary by itself is not enough to prove that the concept can work. A prototype has to be designed in order to test it. eConstruct concentrates it's effort at first on the

buying/selling phase, because that is a regular, known form of e-commerce and it will be relatively easy to gain widespread acceptance of that part of eConstruct's functionality (thereby paving the way for usage of eConstruct's bcXML for other purposes). To preserve the link with eConstruct, the prototype will concentrate on the buying and selling phase also.

II. The building stones: XML and other initiatives

Table of Contents

4. Introduction on communicating with a vocabulary written in XML 37
5. Initiatives which influenced ceXML 43

Chapter 4. Introduction on communicating with a vocabulary written in XML

"Apathy: If We Don't Take Care of the Customer, Maybe They'll Stop Bugging Us" - de-motivational poster

XML is a major building stone for this research project and therefore deserves a chapter of it's own.

The first section sums up the advantages of the Internet for communication. A deficit of the current Internet is the lack of information about *what* is communicated. In the second section, XML (eXtendible Markup Language) is introduced as a means to communicate something and at the same time including information about what is communicated. This means that computers will be able to "talk" to each other meaningfully. The third section describes how to create a vocabulary enabling communication. Appendix A contains a much more detailed explanation of XML and related technologies, skipped in this chapter to keep it compact.

4.1. Internet technology as a possible new solution

The Internet is a technology that fulfils much of the need for a usable communication medium:

- Dirt cheap. Simple modem dialup service for small companies. The big ones (should) already have a decent connection for current Internet usages.

- Usability. Many people are familiar with surfing the Internet. If much of the building and construction information can be made available through Internet pages, the information will be usable by virtually everybody.
- Platform and implementation independence. When care is taken to adhere to the World Wide Web Consortium (W3C) standards, the client's computer, operating system and browser won't make a difference. Be it a small browser on PalmOS or Mozilla on one of the company's Sun workstations.

There is, however, one problem: the way the Internetpages currently are made. The only information now contained within the pages is the actual text and images and information on how to display it. This means that a human can *read* a page and is able to determine that the third paragraph is about a certain type of bricks that can be bought. But no computer will be able to determine which words distinguish product names, which addresses, etc. Currently, it is all about presentation of information for reading by the human eye.

What is needed is electronic communication along the lines of EDI (Electronic Data Interchange, Section 2.2.3), exchanging information (almost) without human intervention, automating away the parts where human attention is not needed.

4.2. General working of XML

"Consider the modern database: sleek, efficient and able to retrieve records in the blink of an eye. Data representation, management and storage have risen to heights we dared not dream of only 10 years ago. But ironically, despite these achievements, the hippest, most cutting-edge data management technology today is (drumroll please ...) delimited text." [infoworld.com]

XML, eXtendible Markup Language, is the most cutting edge data management technology. It also is just a way of dealing with text files. You start with the information you want to send to somebody else, like this:

```
List of items

roof tiles 450 red rooster
wooden planks 20 5x20 cm
```

Example 4-1. XML example - The plain information without any indication about what it really is

The next stage is to "tag" everything in above message, that is, you put a nametag on it. In XML this is done with <nametag>information you want to tag with that nametag</nametag>. So the nametag is enclosed in "<" and ">" signs to form the opening tag, followed by the information and it is closed with a closing tag. The closing tag is the same as the opening tag, only with a "/" before the nametag. Example 4-2 shows how this works out if we change Example 4-1 into an XML format:

```
<orderform>
<description>List of items</description>
<item>
<name>roof tiles</name>
<quantity>450</quantity>
<type>red rooster</type>
</item>
<item>
<name>wooden planks</name>
<quantity>20</quantity>
<type>5x20 cm</type>
</item>
</orderform>
```

Example 4-2. XML example - The same information, now tagged to make an xml file out of it.

This information can be fed into the company's internal ordering system. For example, by using a PalmPilot to enter the data into an application while walking on the building site. Once a (wireless) connection is made via the Internet to the company's office, the information is send over in XML format and at the office, the ordering system reads the XML file and acts upon it. The interesting thing is that

the receiving application now knows whether it is an orderform, a damage report or an timesheet. Also it can distinguish between a type*number* and a quantity.

Sending over information is nothing special. Sending over data in some pre-agreed format is nothing special. The major advantage of XML is that it is a standard. It is a standard way of tagging information with *meaning*. Almost every computer programming language in existence has got one or more modules to read an XML file, to write one, to change one, etc. On the consumer side, the only thing you notice is an increase in possibilities because it is much easier to (quickly) program additional functionality.

4.3. Making an XML vocabulary

In Section 4.2, information was tagged with nametags. In order for computersystems to understand those tags, it is necessary to define beforehand which tags are allowed. This enables computersystems to decide which action to take when confronted with an <orderform> tag (namely, feeding it to the order processing part). In an analogy to human speech, the specification of the language is called a vocabulary. In XML terms, this vocabulary is called an *XML schema* or an *Document Type Definition* (DTD).

The language you speak electronically is both enabled and restrained by the vocabulary. A big vocabulary with very loose grammar results in rich expressive possibilities, but on the downside, it results in hard-to-understand messages. The opposite is true also. A small vocabulary with strict rules allows you to communicate only a limited scope of messages, but the messages are very clear to understand. How to proceed? Designing a vocabulary is a difficult task. That is why the upcoming Chapter 5 looks at three existing initiatives for good current practice in designing vocabularies.

4.4. Conclusion

The Internet is a good medium for communication. Access is cheap and available to all, generic tools (like browsers) are freely available, almost everybody can use it. XML is an Internet technology that provides the possibility to use a vocabulary to

tag information with nametags, giving meaning to the text that is communicated. A vocabulary has to be designed, which is a hard task. The next chapter therefore looks at three examples of how to accomplish that.

Chapter 5. Initiatives which influenced ceXML

People are stupid and will believe anything because either they want to believe it or are afraid it might be true - wizard's first rule

The purpose of this chapter is to provide a basis for the design and implementation described in Chapter 6 and Chapter 7. My design and implementation are influenced mainly by three initiatives, who will serve as the underlying basis.

This chapter introduces ebXML (electronic business XML), GEN (Global Engineering Networking) and the LexiCon (a set of programs made by STABU). These are initiatives which provide useful ideas on how to design and use a vocabulary. Each initiative is introduced. Some interesting aspects will be discussed at a deeper level, and at the end of each part the influence on ceXML is summed up. The chapter concludes with a summary of all the influences distilled from the initiatives, to serve as an input to Chapter 6.

5.1. Electronic Business XML (ebXML)

ebXML (Electronic Business XML) was started by the United Nations, who earlier developed EDI. It is an initiative in which many important and acknowledged global players co-operate. This results in work of high quality, fuelled by extensive experience. In this section, the interesting parts of ebXML are introduced, resulting in a list of ideas that will be adopted in ceXML.

5.1.1. Introduction on ebXML

The mission of ebXML is *to provide an open XML-based infrastructure enabling the global use of electronic business information in an interoperable, secure and consistent manner by all parties* [www.ebxml.org]. The project was jointly initiated by the UN (United Nations) and OASIS (Organisation for the Advancement of Structured Information Standards) and aims to produce a framework for sending and receiving electronic business information within an 18 month time frame (the planned end date is mid 2001). This framework includes XML standards, protocols and software.

5.1.2. ebXML's context mechanism

The collective experience available in ebXML is huge. Most of the large companies that either work with or provide frameworks for electronic business are present. In current systems, two ways of dealing with information form the mainstay of solutions.

- You make a model that allows you to describe in a precise way all information that is needed for one specific industry. This has the disadvantage that this system will not work well the moment you have to exchange information with another industry. No industry stands on it's own: a steel mill needs to order pencils and needs to deal with an accounting system for payments. Both of these fields will *not* be covered by the industry specific information model (which deals with steel).
- You make a generic model that is applicable to a large number of industries in order to provide a solve-all solution. But this model will not be able to capture the level of detail needed to exchange information regarding e.g. the steel fabrication process.

ebXML acknowledges the fact that both the industry-specific initiatives and the solve-all generic approaches do not work. Initial research pointed out the need for a different approach. As a solution to above problems, it was decided to work with the concept of *contexts*, explained below.

A context might be a geographical one (USA, Europe, Germany), it might indicate an industry (insurance, building, automobile), etc. Using the contexts, a basic set of

core components provided by ebXML can be extended to facilitate speaking about e.g. a [building project] in [Europe] using the [Dutch] classification system. This mechanism allows for a notion of inheritance. A core component can be viewed as a tag (like `<address>`), but it is more a concept of it's own. It can be used (when properly extended through the context mechanism) to hold street-city-state-type addresses in the US in one context internet addresses in another context, etcetera. Phrased in technical terms: the context mechanism modifies and extends an initial small, very generic model[1] in such a way that it is suited for holding information in that specific context.

For example, ebXML has given one of it's core components the name `party`. Party as in "*this contractor is one of the parties involved in building this bridge*"). The tag `party` always contains information on the identity of the party and how to contact the party. This is achieved through a `name` and an `address` tag. For some situations, this might be enough, but `party` can be extended to include tags like `street` and `state` in the USA, while in the Canadian context `province` will be used instead of `state`. As a side note, the context is not supposed to be hierarchical, which in this case means that first selecting the context *USA* and after that *insurance* should be the same as first selecting *insurance* and then *USA*, both should mean "I'm talking about insurance in the USA setting".

Work is underway to come up with an XML format specifying these context rules. A rule consists of actions to be taken to expand a given vocabulary to contain the tags needed to talk in the context the rule is valid for. Normally, the starting vocabulary will be ebXML's core component vocabulary. This vocabulary contains basic tags like `address` and `contract`. An early try-out (showed on a recent ebXML conference) demonstrated this extension of the core component vocabulary with additional terms.

Because XML Schema, the language that will be used in the near future to create a vocabulary in XML, is itself specified in an XML format, this process could well be done using XSL/T (see Section A.2), an XML technology to change XML files. The advantage is that it can be done completely with XML technology, making it usable in a lot of circumstances.

This mechanism is powerful, but it means that at the same time the resulting

1. ... or XML schema, or DTD, depending on the way you look at the problem.

schemas might not be stable enough. Something that is generated is *not* something that is set in stone. But if the mechanism is well-defined and predictable, this disadvantage will not be a problem. For ceXML (the vocabulary which is the subject of my research), the context mechanism seems to be the right choice. Having to accommodate different languages, classification systems and uses, as described in Chapter 3, flexibility might be the right approach.

Because no ambiguity is allowed in any core component tag, ebXML provides strict definitions for every element they create (at the moment about 30).

5.1.3. Semantic level

Semantics is basically the "meaning"-part that is associated with language. Grammar is the syntax, the structure of a language. Semantics is what ties the word *tree* to the big brown piece of wood with green leaves (or needles).

ebXML leaves the semantics mostly to industry groups. With the semantics, I mean the tags that are needed to specify, to *give words to*, items defined in the industry. To rephrase that, semantics are the words - the words that are needed to communicate *meaningfully* about the things that are relevant to an industry.

In the ebXML system, the semantics that are needed are provided from two sides:

- ebXML provides the few tags that are needed by everybody: the core components. They are the ones used for very generic tasks, like identifying the two trading parties, their addresses, contract, signature etc.
- Specific industries must take care of the task of creating a set of tags needed to communicate about things specific to their industry. For the building and construction industry, eConstruct should provide these tags.

You can draw an analogy to the development of a normal (human) language. Here also you have a basic set of words, sayings and concepts. Every industry or group adds his own semantics to the pool of words. From the King James Version bible-speak in some churches to the unintelligible ramblings of computer enthusiasts and the technical phrases of a civil engineer, every group has group-specific semantics.

To make it practical for eConstruct: eConstruct should create the message while ebXML takes care of the envelope and the transport. ebXML creates the semantics needed to talk about where to send the message and what to do with it, eConstruct has to create the semantics needed to talk about the needed strength for a hollow-core slab with a span of 5 meters.

eConstruct has to watch ebXML's development carefully in order to be able to integrate when the specifications are stable. Arguably, ceXML cannot integrate at the moment with something that does not exist. It is good, however, to remain focused on the fact that the trading and communication stuff will be provided for by initiatives like ebXML. ceXML (and it's big brother bcXML) can concentrate on the specific building and construction *semantics*.

5.1.4. Business processes

ebXML also has a technology called *business processes* that defines what *is* to be done with the data and what *can* be done with the data. By strictly specifying business processes, also those processes are subject to automation. As an example, you can specify what has to be done to get permission to build a house. If that is known, several steps can be automated, making it easier to deal with them. There are specific issues with the building and construction industry regarding the way a conversation between two possible partners rolls along, which eConstruct should integrate into the business processes framework. For instead of relying on purpose-build one-time solutions, it is good to follow ebXML's example: devise a mechanism for the dealing with business processes.

Again, this is not directly feasible for ceXML because ebXML is not finished yet, but the idea is sound. Therefore also in ceXML, transactions will be driven by data, not by programming logic.

5.1.5. Influence on ceXML

As indicated earlier, using ebXML (in the form of a finished product) is not possible because it is not finished yet. Much work that is done by ebXML, however, seems extremely useful. Therefore the following parts are chosen for inclusion in the ceXML prototype.

- The context mechanism. Using contexts to adapt the vocabulary to that context.
- The distinction between general addressing-like items (the core components) and the building/construction specific items.
- The use of specified business processes to ease the processing of the XML requests and answers send back and forth.

5.2. Global Engineering Networking (GEN)

The GEN project (Global Engineering Networking) is an effort at creating a network where engineers can get the information they need and where suppliers can provide the information needed by engineers. This section will describe GEN, show the architecture of GEN and the usage. At the end, the specific influence GEN has on ceXML is mentioned.

5.2.1. Introduction on GEN

In engineering enterprises, close co-operation with other firms is needed. One firm cannot hold all knowledge, technologies, resources, products, etc. GEN's purpose is to create a *virtual enterprise*, tying multiple suppliers, vendors, resellers, etcetera together into one big *virtual* enterprise. Currently, easy exchange of the needed information is hard to achieve. Different sources (catalogues, CD's, Internet) have to be searched to get to the needed information.

GEN wants to remedy this situation and has compiled the following list of requirements [Radeke, 1998] for the architecture:

- Various sources of information should be accessible in an uniform way.
- Standardised classifications should be used to access the information.
- Extension of classifications should be possible to allow for customisation to a company's needs.

- Information must be accessible, not only in the 'correct' classified place, but also when it has a strong relationship with information elsewhere in a classification tree.
- Reuse of existing data, whatever the source.
- Companies should be able to maintain their data on their own, due to political, security and consistency reasons.

5.2.2. Architecture of GEN

Normal systems for electronic commerce are working with a purpose-build local system. GEN uses existing classifications to store information and allows the distribution of the information amongst a number of sites. Figure 5-1 depicts the usage of GEN as a method of co-operation on which solutions can be build and to which existing systems can connect.The channel of communication is the Internet. Each company's internal system is connected to that channel of communication by means of a so-called *GEN co-operation layer*. This co-operation layer uses it's own meta data (basically: it uses it's own vocabulary) to communicate with the other GEN sites.

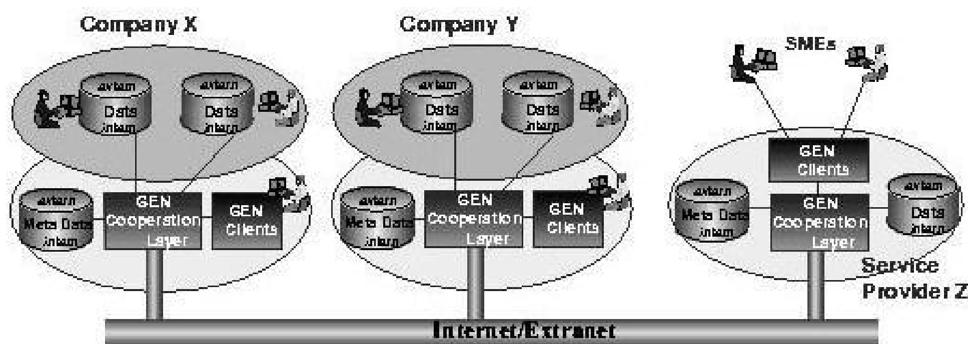

Figure 5-1. Logical architecture of GEN according to [Radeke, 1998]

The GENial project (Global Engineering Networking, Intelligent Access Libraries) is a project that had to provide an implementation of GEN. So, a reference implementation of a GEN infrastructure has been made. GENial can be used in the following three ways:

- Company-specific, as an added service for their website.
- Across multiple suppliers (e.g. of an supplier association)
- For an entire domain, all along the entire supply chain.

The architecture of GENial uses an two-step method to get useful data from the suppliers to the users. GEN uses XML as an intermediate exchange format. Data is first extracted from the suppliers data source (whatever that may be) and converted to the GENial XML format. Next, the data is ready for integration into the GENial infrastructure (whatever the implementation). The same route can be followed on the way back to the end user (Figure 5-2).

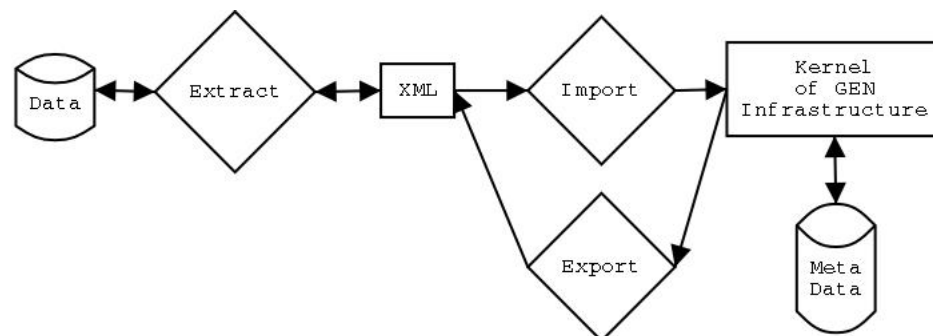

Figure 5-2. Handling of data in GENial

5.2.3. Usage of GEN

The interface presented to the user in GENial (Figure 5-3) depicts the way GENial is to be used. As said, GENial is a reference implementation of a GEN

infrastructure. In the upper left part, the user selects a classification system of his choice. A tree-like representation of that classification is available in the lower left part. Once the user has made his choice (in this case, the element *door* in the *lexicon* classification), the choice appears in the right part of the screen with a list of all attributes and a means to specify restrictions for those attributes. So, `width` can be set to be *1.2 m* and `door type` to *sliding*. Next, the search button can be pressed and all doors meeting those criteria that are available in the GENial system are listed.

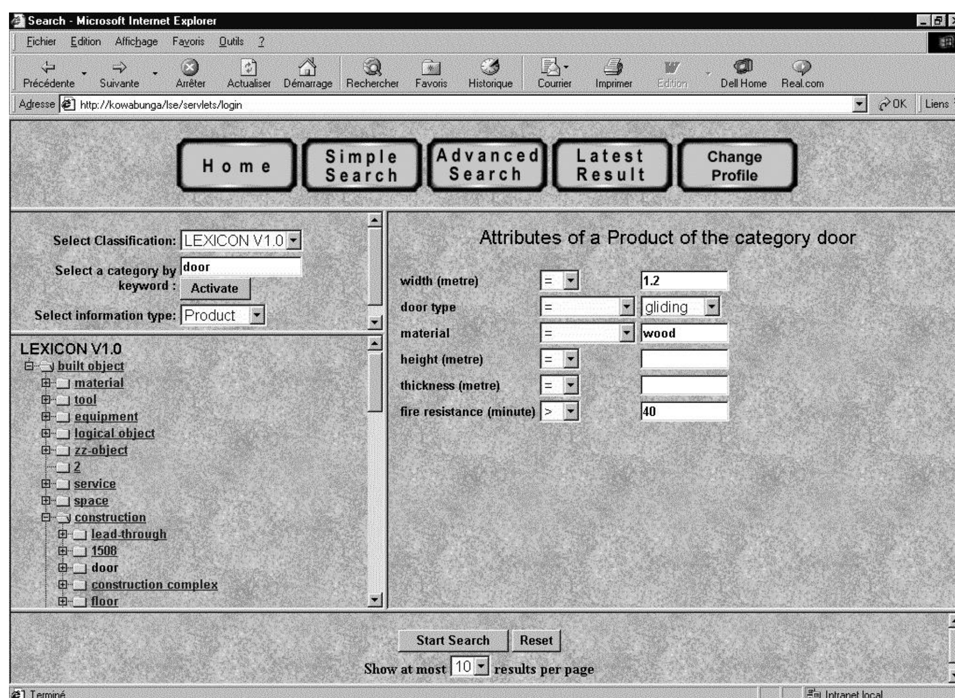

Figure 5-3. The GENial interface

5.2.4. Influence of GEN on ceXML

The use of XML is arguably effective as an intermediate format between the

original data and the infrastructure of the actual content management and the content providing layer. The need to make this intermediate format easily usable must also be met by ceXML. The question remains whether one xml format will be sufficient for *both* the exchange with the original data *and* for the actual sending of information. Are there different needs for both cases and, thus, should the format be different?

Another influence is the way user interaction is done. Using a classification and known attributes to select the needed information seems like the most straightforward and simple way of handling the interaction with the user.

5.3. LexiCon

The LexiCon set of programs [2] and way of working is one of the main building blocks of the eConstruct project. The LexiCon will provide the means to deal with multiple classifications and languages, making it possible to *identify* objects regardless of the language or classification system that is used. It is called LexiCon (with a capital C in the middle) to make clear that it is a specific product and thereby to distinguish it from the generic term *lexicon*.

5.3.1. Inner workings of LexiCon

The LexiCon consists basically of three parts:

The LexiCon Explorer

A computer program used to view, create and edit vocabularies (lexicons) according to the LexiCon meta-model.

2. The LexiCon is made by STABU, a Dutch organisation which (amongst other tasks) builds a information system covering the entire building process.

The LexiCon SpecExplorer

A program to view, create and edit information about a specific project or object in the 'language' defined by a specific vocabulary (a lexicon created by above LexiCon tool).

The LexiCon meta-model

This meta-model specifies the rules to which an implementation or a message must comply to be compatible with the other parts of LexiCon. The LexiCon meta-model defines what can be said and expressed using the LexiCon system[3].

To sum it all up, the *meta-model* specifies the rules by which one can create a vocabulary (a lexicon) in the *Lexicon Tool*. That vocabulary is used by the *LexiCon SpecExplorer* to describe real-world objects or projects (Figure 5-4).

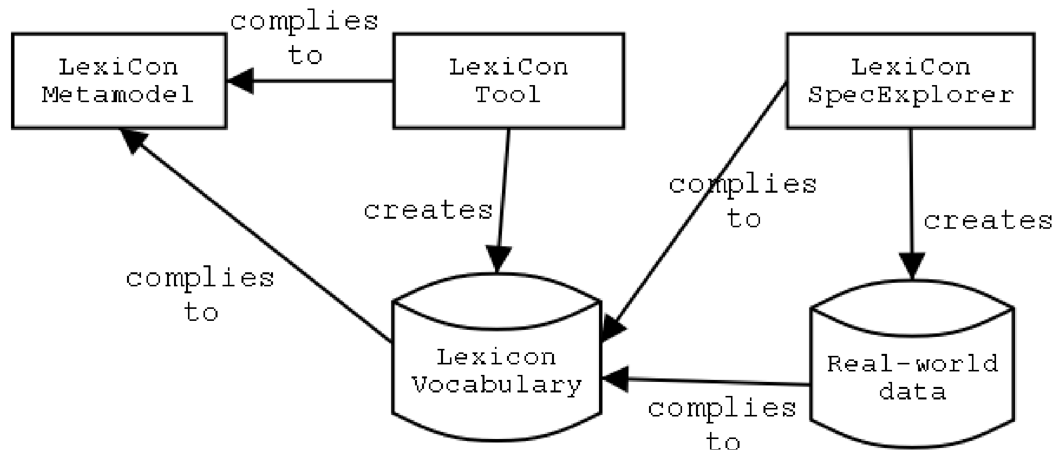

Figure 5-4. The structure of LexiCon

3. When seen this way, the information that is put into the LexiCon system is basically a model. I present it in this way because I use the information in the LexiCon system as a vocabulary. This vocabulary serves as a model to which the communication in my system complies. Thus, the level above that vocabulary is - from this viewpoint - a meta-model.

5.3.2. Underlying model

Figure 5-5 shows the meta-model of the LexiCon, simplified for clarity. This meta-model will be explained more fully in Section 6.5.3 and is the basis of the ceXML model.

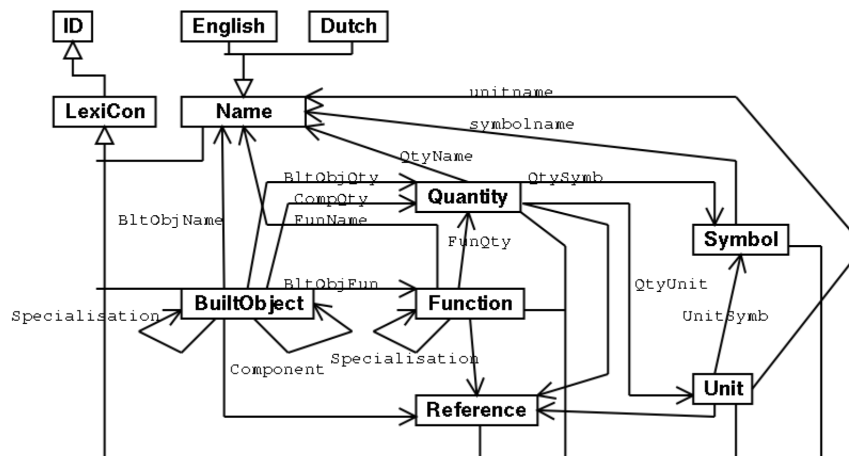

Figure 5-5. UML diagram of the LexiCon meta-model

5.3.3. Functional unit and technical solution

An important concept for the LexiCon is the FU/TS concept (meaning Functional Unit/Technical Solution). The idea comes from the General AEC[4] Reference Model (GARM) by Wim Gielingh. That boils down to a *divide and conquer*-like way of dealing with an object description. On one side one has a functional unit *"something to stand on"* and on the other side you have the technical solution *"a floor supported by beams"*. The attributes and requirements of the functional concept are to be met

4. Architecture, Engineering and Construction

by the technical solution, so that the requirements for the total solution are appropriately propagated to the technical subparts.

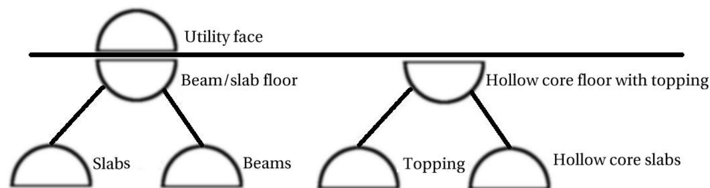

Figure 5-6. Explanation functional unit and technical solution ("the hamburger model") [W. Gielingh, 1988]

5.3.4. Influence on ceXML

One of the main points to be covered in this graduation project is testing the exchange of building/construction data with the LexiCon. The kernel of the LexiCon is the meta-model it uses. Within this meta-model, the LexiCon can describe items in multiple languages and it can link items to multiple classifications. These possibilities make it a needed ingredient of eConstruct and, likewise, this LexiCon meta-model will serve as a design guideline in the creation of a ceXML vocabulary. I call it a *design guideline* (I will not use the exact meta-model). Another reason for this is that I will implement the meta-model in XML[5], which will bring about some changes to the meta-model.

---

5. "Implementing the meta-model in XML" in this case means that I write a DTD according to the information in the meta-model. Any XML file complying to that DTD can be said to comply to the meta-model.

5.4. Conclusions

The conclusions from this chapter are best summed up by listing the most important influences on ceXML compiled from the previous three sections:

- Clearly ebXML has got the appropriate context mechanism, the distinction between generic and specific information and the idea to use specified business processes as a tool to handle all the data.
- GEN provides ideas for a good user interface and provokes thoughts on how to exchange information with existing systems.
- LexiCon's influence is the way the vocabulary should look like.

III. Design and implementation of the vocabulary

Table of Contents

6. Design of the vocabulary 59
7. Prototype implementation.. 79

Chapter 6. Design of the vocabulary

"A simple, clear purpose and simple, clear principles give rise to complex and intelligent behaviour. Complex rules and regulations give rise to simple and stupid behaviour." - Dee Hock"

This chapter describes the underlying design of the vocabulary: the models. Each model is depicted using an Unified Modelling Language (UML) diagram. For those unknown to this graphical notation, please read Appendix C for a quick introduction.

The chapter starts with listing the requirements for the vocabulary and the prototype. Next, the structure of the entire vocabulary is outlined, followed by the three parts it consists of.

6.1. Detailed requirements specification

6.1.1. Goals

The vocabulary (Chapter 6) and the prototype (Chapter 7) have two goals:

Proof-of-concept:

The concept consists of the most interesting parts of ebXML, GEN and the LexiCon. These interesting parts have been mentioned in last chapter's conclusions (see Section 5.4). The prototype and the vocabulary have to *prove* that this concept is usable.

Ease of understanding

The prototype and vocabulary serve to illustrate the working of the concept. Therefore it should be easy to understand, also for people without much experience in civil engineering informatics.

6.1.2. Requirements

The following requirements serve as a guideline in designing the vocabulary and the prototype.

It should be simple:

In order to make the concept clear, the vocabulary and prototype have to be simple. Stated bluntly: a proof-of-concept is no proof if the proof is not understood.

There should be separate generic tags and (industry) specific tags:

ebXML provides a basic set of tags (the *Core Components*) containing generic data like names and addresses. Industry specific tags are added to these core components to get the full range of tags needed to communicate. Divide and conquer. (Section 5.1.3).

It should use the LexiCon meta-model:

The LexiCon meta-model provides the functionality needed to communicate in multiple languages, which is very important for eConstruct. As one of the main points of this graduation project is testing this multi-lingual exchange of building and construction data, the LexiCon meta-model is mandatory.

It should apply a context mechanism:

The vocabulary must use the context mechanism as used in ebXML. With this mechanism, a core set of tags is modified according to a certain context (Section 5.1.2). After the modification, the core set has been enhanced with tags that are needed to communicate in that context.

It should use Business Processes:

> ebXML uses specified Business Processes to aid in the processing of requests and answers. A message should not have to include more information besides just the name of the business process in effect and the current step in that business process. The system should be able to process the message based solely upon that information.

It should use an interface with both a tree view and a list of characteristics:

> The interface of GEN (Figure 5-3) will be used by ceXML. Using a tree-like representation of a classification to select the needed item is the most straightforward and simple way of handling the interaction with the user. A list of characteristics of the selected item must be displayed next to the tree-view.

6.2. Overview of the structure

Figure 6-1 shows the structure and the parts of which the vocabulary exists. First you need a *message* (Section 6.5) containing the actual civil engineering information you want to communicate. To facilitate the various operations that have to be carried out on the message (like actually sending it), an *envelope* (Section 6.4) is placed around it containing generic information *about* the message contained within. To aid in the processing, an external list of *Business Processes* (Section 6.3) is maintained, containing information on what to do upon arrival of a message, information that tells the computer system how to process the message. The Business Process info is placed within the envelope.

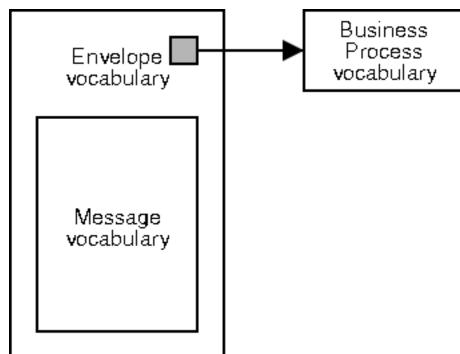

Figure 6-1. Overview of the relations between the Business Process, the message and the envelope vocabulary

6.3. Business Processes model

By specifying Business Processes in a strict way, computers can be told how to roll along a transaction from begin to end. See Figure 6-2 for an example: a contractor wants to know if a supplier can supply him with certain parts, he receives back the items the supplier has to offer, orders a certain part and receives back a confirmation of the order. The same business process is used in case a supervisor notices a shortage of certain parts on the building sites and orders an additional shipment from the company warehouse (the second example mentioned in Section 3.2). Figure 6-3 shows a second possible business process, this time a supervisor needing information about a specific part of a project, possibly to change the completion percentage recorded for that part in the company records from 60% to 100%. Or to find out which subcontractor installed the faulty window in order to send him a repair order.

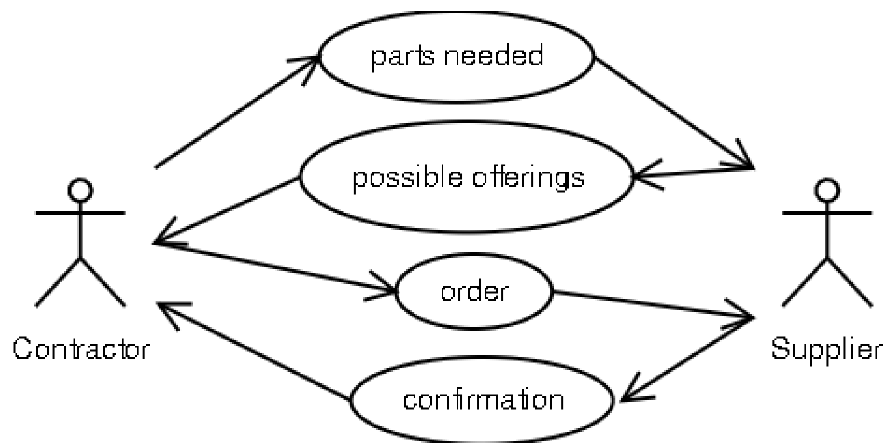

Figure 6-2. Business Process Use Case 1

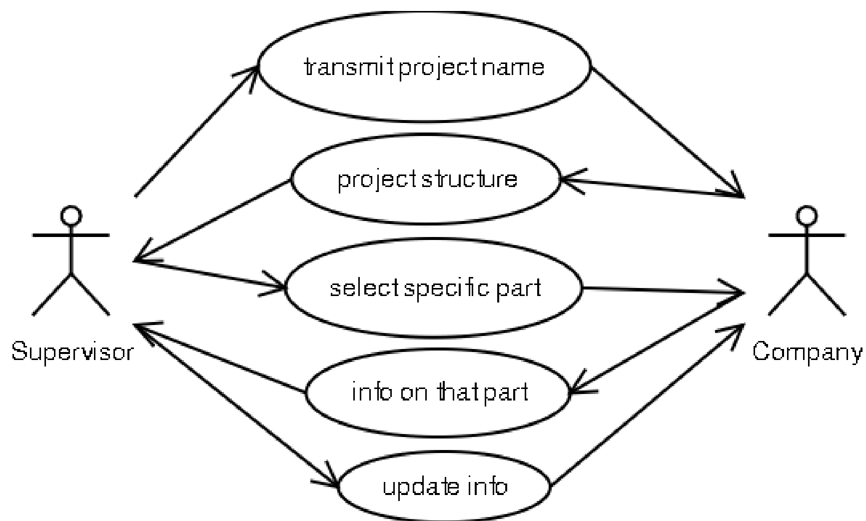

Figure 6-3. Business Process Use Case 2

To get basic functionality, the following two items are needed:

- A single step of a business process, like *update info* or *select specific part*.
- An entire business process, consisting of a number of steps. Both Use Cases (Figure 6-2 and Figure 6-3) are considered one business process.

The complete model is shown in Figure 6-4.

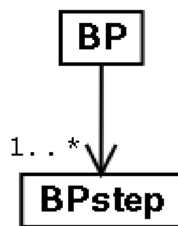

Figure 6-4. Business process model

6.4. The envelope model

The envelope model should contain the information needed to complete whatever the user wants to accomplish with the message. This is the generic information, as opposed to the actual message itself, which is the subject of the next section (Section 6.5). The term *envelope* was chosen because of it's clarity when compared with *generic vocabulary*.

First and foremost, you need an identification of the parties[1] on both sides. Secondly, you need a place to store the info on business processes and a place to

1. Party in this case means an individual or a company taking part in the communication. It is the term used within the ebXML project for the communicating partners on both the receiving and the sending side.

store the message. For a proof-of-concept, Figure 6-5 provides a model that contains enough information to be usable in simple applications.

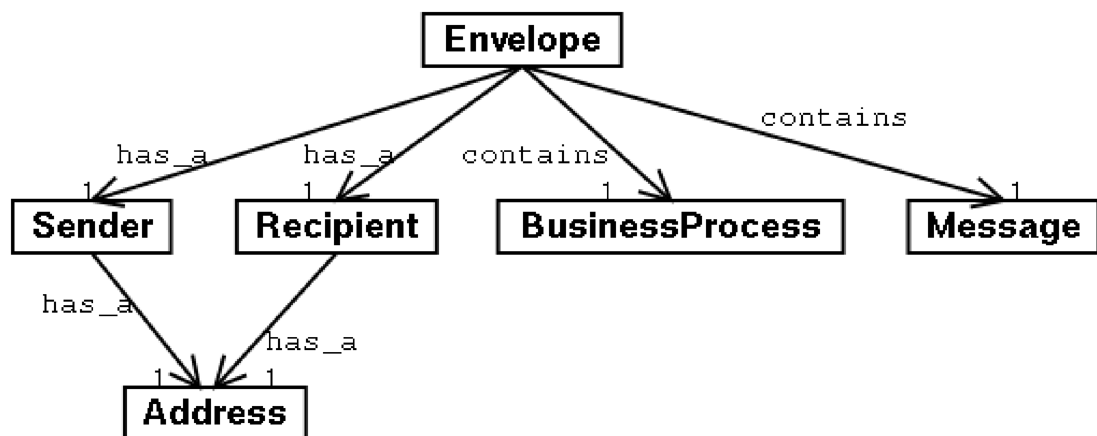

Figure 6-5. Model of the envelope vocabulary

6.5. The message model

The message model is called that way to distinguish it from the envelope model. The message model contains the actual building and construction data. For actually sending over data, a more simplified version of the model (a "view") is used

This section starts with a discussion outlining the difference between a model and a view. This is done first in order to place the message model in the right perspective. The section also describes why *both* a model and a view are needed to describe the actual message. Next, both the model and the view are presented.

6.5.1. Difference between a model and a view

A model, as the two previous sections showed, *is a way to describe something in a standardised manner*. A Business Process consists of one or more Business Process `steps`, as you can see in Figure 6-4.

A view is best described by saying that it is a *view on a model*. Let's explain it by drawing an analogy with databases. Databases are an application area where views are used a lot. An entire database is quite complicated and consists of many items[2] . Many items only serve a database-internal function and aren't meant to be seen by the users. A view in database technology terms means a selection of the items, needed for a specific purpose. A user of that view can do exactly what he needs to do, without being bothered by the underlying complexity. A view (now in the context of this document!) is a simplification of the actual model which can be exchanged losslessly with the model[3].

6.5.2. The need for both a model and a view

Reality is complex. Civil engineering works are complex. In order to describe them accurately, the model should be able to accommodate the complexity. Stated differently, the model should be able to describe the reality accurately. When a model only allows you to talk about *walls* and *floors*, you can describe a building. But you're already missing the foundation, the windows, etc. Reality in this case is definitely more complex than the model can accommodate.

---

2. *Items* is not the technical correct term, it should be fields and tables. I use it to make the document more understandable. Databases normally consist of multiple normalised tables, which means that they are stock full of fields that are only there in order to be able to re-assemble the information stored in those tables. That is why views are so handy, that way the user is not bothered with `UsrGrpID` and `InternalArticleNumber` fields.

3. Lossless exchange between the model and it's view of course only is possible when the view is used correctly and therefore only so when there is no possibility for the user to circumvent the specific constraints put into the software. Or, another possibility, the view contains all information needed by the model, only arranged in a different and more user friendly way.

To handle complexity, one of the most important steps is to *categorise* reality. This means to split up reality in multiple categories, e.g. floors, walls, foundation, services, etcetera. A wall can then be categorised further in concrete wall, wooden wall, etcetera. There are two ways to accommodate complexity through categorisation:

- Make a *complex* model. Model as much specific characteristics of the reality as feasible, categorising within the model itself.
- Make a *generic* model. Capture the characteristics of the reality, phrasing them in a limited set of generic terms, leaving the actual categorisation work to one who uses the model.

For example, the first model would have a `loadbearingfunction` element, the second model would have *a* `function` with a `name` of `load_bearing`.

The requirement (Section 6.1.2) that the model has to be simple excludes the first (the complex) model. The problem with the second model, however, is that the generic terms used (which are perfectly well suited to describe reality) are less usable from a user point of view. As an analogy, a database's internal structure, which probably will be the best and most efficient way to represent the information stored in it, most probably will not correspond to the way the user looks at the information. In the same way, a generic model does not represent the user's viewpoint most of the time. So, a view will be used to accommodate certain specific uses. The model itself will be used for generic storage and to facilitate exchange.

An additional bonus of using a view is that it constrains the user. This may sound negative, but by allowing only a limited view on the total model, the amount of possible errors also is reduced. To illustrate this, look again at the database example. In his view the user only sees the fields he needs. If he would have access to all database fields, it would be easy for him to type in a wrong customer_number in the order database that does not correspond to the intended customer. In a view, he would have selected a specific customer and the system would have taken care of filling in the correct customer_number.

As a conclusion, a generic model will be used to describe reality, but for specific uses, a view will be used to simplify the use of the model.

6.5.3. The model

The model I use is based on the LexiCon meta-model (Figure 5-5). Below, I will explain the resulting model. This also serves as an explanation of the original LexiCon meta-model.

First off, we start with a `builtobject`. That is an item which stands for one part of a civil engineering object. Like, for instance, a wall, a door, a tunnel. Nothing keeps you from using it to indicate a service, like paint-work (as applied on/for an object). But since it will most often be used to describe a "touchable" object, I stick with the naming of the LexiCon. A `builtobject` is the base building block, to which all other information is connected, like names, characteristics and sub-parts.

builtobject

Figure 6-6. The message model - 1

Secondly, I add two things. The `builtobject` on the one side has `characteristics`[4] of it's own and on the other side it serves as a category for other `builtobjects`. The `characteristics` will be explained later on. The fact that the `builtobject` serves as a category is because all the `builtobjects`, one way or another, have to be categorised in categories of builtobjects. Everything below this category still is a, say, door, but then a more specialised one. A sliding door, a revolving door, a armoured door. This is central to the LexiCon meta-model this model is based upon. All `builtobjects` which serve as a category, together form a tree-like structure. They are categorised in one big hierarchy according to specialisation (see Figure 6-7.

4. In the modelling world, *characteristics* is used to indicate the characteristics of a class, of a kind of object. As soon as the object is a specific object ("*that* door"), they are normally called *properties*.

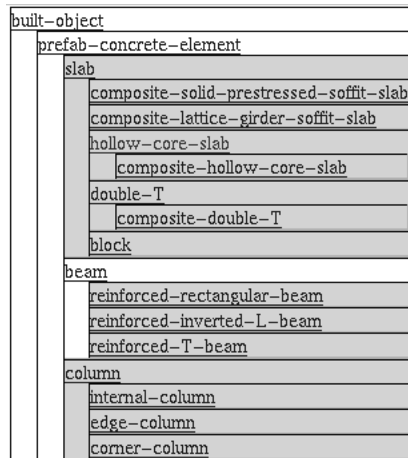

Figure 6-7. Example of a tree structure

So, the specialisation can be used to build a hierarchy of `builtobjects`, while the `characteristics` are listed separately, they are `characteristics` of that specific type of `builtobject`.

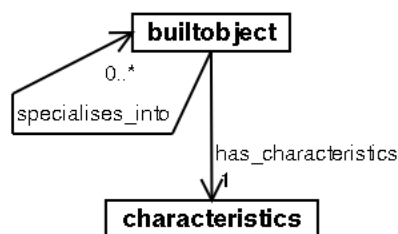

Figure 6-8. The message model - 2

Next, the `characteristics` are fleshed out. The `builtobject` can have an `amount`, `quantifications` and `functions` associated with it through it's characteristics. An `amount` is used to indicate the number of `builtobjects`. This

is used when describing that something consists of `amount` items of *this* `builtobject`.

`Quantification` is used to describe measurable `characteristics` of the `builtobject`, like weight, length, height.

`Functions` are used to describe features one desires from or who are associated with the `builtobject`. For example *load_bearing* or *fire_resistance*.

Through the subparts-relation, also `builtobjects` are connected to the `characteristics`, this is a means to identify parts which are directly related (though not in the sense of specialisation!), such as a doorknob on a door or glass in a window.

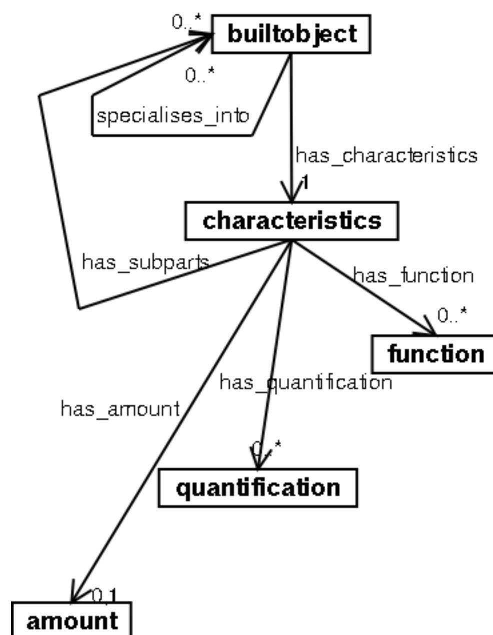

Figure 6-9. The message model - 3

Added to this is one new object, `unit`. `Unit` is a unit as in *millimetre* or *pound per square inch*.

Two relations are added. a `function` has an optional `quantification`, so the function *fire-resistance* can be quantified with *time to failure*. A `quantification` can have an `amount` (50) and a `unit` (minutes), for instance. This can be used for a fire-resistance function which has a TTF (time to failure) of 50 minutes.

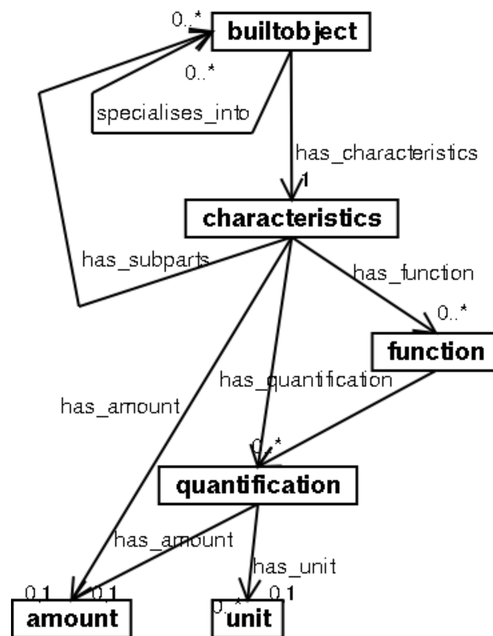

Figure 6-10. The message model - 4

Lastly, all objects (except `characteristics` and `amount`) have one or more `names` for identification and presentation purposes. Each `name` has an attribute `language` to indicate the language. This way, multi-linguality is achieved. A door can both have an English name *door* and a Dutch name *deur*. It is the same built object, but it can be identified both in Dutch and in English.

The only elements without a `name` are `amount` (which is just a number) and `characteristics` (which only serves as a container to the elements below it).

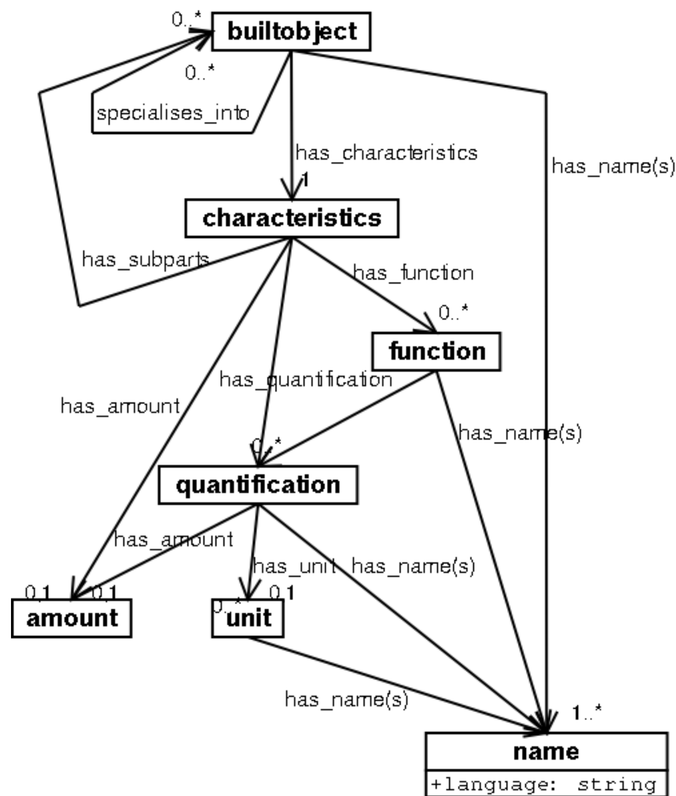

Figure 6-11. The message model - 5 (final version)

6.5.4. The view

As indicated before, the model is good for storing a hierarchy and ordering the different built objects, but for simple uses it is too awkward to use. In order to prove the concept of views (explained in Section 6.5.2), one simple view is enough. The two Use Cases can help determine the view. Both Use Cases are repeated in Figure 6-12 for convenience.

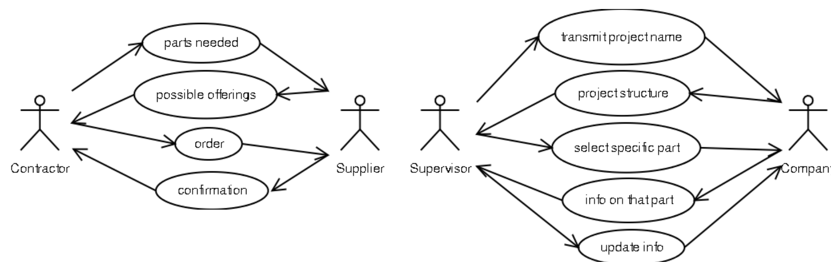

Figure 6-12. Both Use Cases together

The left-hand Use Case is mostly about simple parts. The contractor wants 500 rooftiles with a certain performance and 4 concrete beams with a span of 4 meter. He gets as a response 7 possible types of rooftiles and beams. He orders 500 items of one type of roof-tile.

The right-hand Use Case first needs an entire project hierarchy to be transmitted, but after that only single pieces of information are needed.

A useful view in both cases is to have a simple list of items with no hierarchy and substructures: just a one-dimensional list of single items. This means a very simple view with no need for much structure. The information that is contained within the model for one specific `builtobject` (like `function`) should be available too within the view. But here lies a possibility for more ease of use. In the model, the information is represented by items that have `names`. In the view, when you select only a single language, the function-with-a-name duo can be changed into a single item named after the `name` of the `function`. And instead of communicating a `builtobject` with a `name` of *roof-tile*, you can communicate a `roof-tile`. In this way, no information is lost, but the usability has risen considerably. The result of an example view can be seen in Figure 6-13 and Figure 6-14.

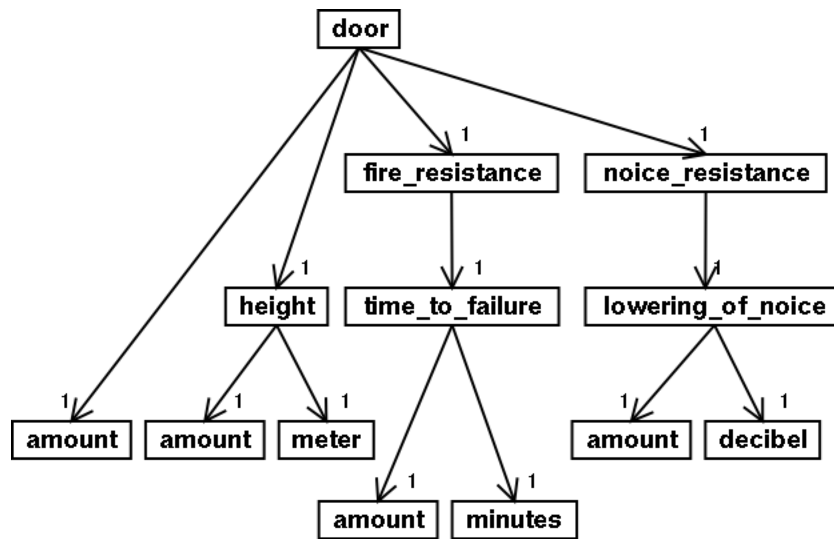

Figure 6-13. Initial version of the example view

```
<door>
  <amount>..</amount>
  <height>
    <amount>..</amount>
    <meter/>
  </height>
  <fire-resistance>
    <time-to-failure>
      <amount>..</amount>
      <minutes/>
    </time-to-failure>
  </fire-resistance>
  etcetera
</door>
```

Figure 6-14. Initial version of the example view - XML version

This can be done even simpler by using attributes, which does away with the `unit` and `amount` items. This results in Figure 6-15 and Figure 6-16.

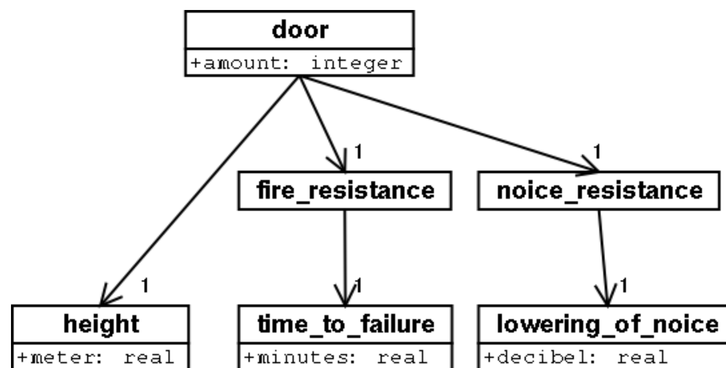

Figure 6-15. Definitive version of the example view

```
<door amount="..">
  <height meter=".."/>
  <fire-resistance>
    <time-to-failure minutes=".."/>
  </fire-resistance>
  etcetera
</door>
```

Figure 6-16. Definitive version of the example view - XML version

6.6. Selection of elements

A problem that has to be solved is how to select elements. There is a tree of `builtobjects`, in which there are names, functions, quantifications, etcetera. They themselves are stored separately to save time and effort. A certain function will probably be shared by a number of `builtobjects`. Also the `builtobjects` contain other `builtobjects`. So there has to be some mechanism of pointing towards other items.

For this, there are two feasible ways:

- Use ID's to identify items. An ID is an unique number or text that is used to identify exactly one item. `door id="abc123"`, for example.
- Use the names associated with the items to identify them. This means that you identify a specific item by searching for an item of that type with the name "door", for instance.

The ID mechanism has the advantage of not being bound to a specific language. Using names to identify items has the disadvantage that there will very probably be multiple items with the same name. An ID can be used to select exactly one specific item without any inherent language problems. A language can be trusted, however, to be logical enough to ensure that two items with the same name are more or less similar. If a real problem persists, it is necessary to use an alternative word to describe the item. A large number of conflicts can be found, however, in the naming of quantifications. Depth can be, for instance, the depth of a lake (a vertical unit of measurement) or the depth of a window (a horizontal unit of measurement). This can be solved by using a more descriptive unit, like *construction-height* (instead of just *height*) to indicate the vertical dimension of a bridge, and *clearance* for the vertical dimension that indicates the room available for ships to pass underneath the bridge.

The ID mechanism has the disadvantage that you have to be very careful with the ID's you give to items. Whatever happens, a door item will still be found by looking for the item with the name "door". Nothing keeps an ID from changing if there are no fixed rules for handing out ID's. Especially when ID's are automatically generated by computer systems of various suppliers, there are severe problems.

Programmatically, there is no difference between searching for a item with a certain name and searching for an item with a certain ID. To avoid adding ID elements to

the language, I choose to select items by looking at their names[5].

5. Not using ID's is a choice for my prototype, not a choice I would necessarily make for the final eConstruct system. A well devised ID system, probably using reasonably descriptive text as ID, gives the system much more freedom regarding the names given to items. In a real-world system there will not be much enthusiasm for a programmatic restriction on names. But for my prototype, the name system doesn't pose any problems and is much easier than having to hand out ID's to all items *and, more important, keeping them in sync*.

Chapter 7. Prototype implementation

"The world is like a book that can be read in two different ways. We read in it the power of creation, to create something out of nothing. And we read in it the power to destroy, reducing something to nothing" - Rabbi Pinchas of Koritz

This chapter's goal is to implement the vocabulary of the previous chapter. It has to be a proof-of-concept.

First, the way of implementing is explained. After that, the resulting code and files are presented in groups, first the generic utilities. Then the vocabulary and views are made and filled. The filled vocabulary and views are visualised and, finally, used for a search.

7.1. Implementation method

The goal of this section is to present the programming and the tools used.

7.1.1. Unix style of programming

In order to keep everything simple, I have chosen to follow the so-called UNIX style of programming. This means that I build a lot of small programs (in this case: many XSL/T stylesheets) that do only one thing (and do that well). These small programs are then linked together in various ways to provide the needed

functionality. This as opposed to the much-used style of programming which builds one big does-it-all program.

7.1.2. Programming language and tools

I used four programming languages/tools for the prototype. They were chosen for their ease of use and their technological appeal.

7.1.2.1. XSL/T

Being the most used and useful technology surrounding XML, this stylesheet language is used to transform one XML format into another[1]. By transforming an XML file multiple times by consecutive stylesheets, quick results can be achieved. The extensive use that is made of this technology in the XML world makes it indispensable for this prototype. An explanation of XSL/T is found in Section A.2.

There are many XSL/T stylesheet processors, I chose the Xalan processor from Apache's XML project because it is one of the most actively developed processors, keeping up with even the latest standards and achieving nearly 100% standards compliance. This turned out to be a good choice, because I did not find a single sign of strange, non-standard behaviour. Xalan is implemented in Java and therefore has to be called from a Java program or it has to be executed on the command-line (I chose the latter).

7.1.2.2. Python

Python is part of the so-called family of *scripting languages*. It has an easy and clear syntax, it allows for quick programming (typical python programs are 10% the size of comparable C++ or Java programs) and it has a large amount of modules that provide extra functionality. It is perfectly suited to the task of gluing together bits

1. In technical terms, you map an incoming XML tree into an outgoing XML tree according to rules specified in the XSL/T stylesheet. The outgoing format is allowed to be something different from XML, though, in case that should prove useful.

and pieces of functionality into one program. Because it is an interpreted language, it allows one to try out many things, which proved very useful.

There are four areas in which I used python mostly:

- Processing files with XSL/T stylesheets by calling Java on the command-line and passing all the right arguments.
- Guiding files through a series of XSL/T stylesheets using temporary files.
- Reading and interpreting command-line parameters.
- Serving as a CGI program, called from a web browser.

7.1.2.3. Sed

Sed is a small utility that processes text files according to command-line parameters. In this it serves a similar purpose as XSL/T, only on a much smaller scale. I used it for some tasks that were hard to do in XSL/T, like generating a DTD out of an XML file. A DTD is not XML compliant and contains a number of characters which XSL/T does not like. So they had to be left out and were later added to the file by means of a simple Sed command. I tried to use the right tool for the right job, not trying to see every problem as a nail, having XSL/T's hammer in my hand.

7.1.2.4. CSS

For displaying some XML files, I used CSS. This allowed me to view an XML file in a browser without having to generate lots of HTML to create the desired look&feel, since CSS can be used to specify how a particular XML tag should be visualised. For an XML and CSS compliant browser, I used the newest release of Mozilla (the former Netscape browser), because it supported all the new standards well.

7.1.3. Usability

The goal of the prototype was to serve as a proof-of-concept. It was not intended to

be the most good-looking, blindingly-fast program possible. Therefore I made only a few browser interfaces. For most of the functionality it doesn't even make much sense to provide an interface, for most functionality is more the server-side kind. This results in a number of command-line tools.

The speed of the programs also needs to be mentioned. The processing of the XSL/T stylesheets takes up most of the time and is *extremely slow*. Every stylesheet takes about a second to process, which adds up to 5-10 seconds per program (which consists of multiple stylesheet processings). This is due to the fact that the python program executes every stylesheet by starting up a shell with a Java command-line. So, the results are not passed directly from XSL/T process to XSL/T process (which is possible), but are stored in intermediate files. So every time, Java has to start, the file has to be read, the XSL/T processor has to be run and the result written back to a file. This is about the slowest way to implement it, but that does not mind much for a prototype. Implementing everything in Java would have made it faster, but that also means much more programming effort and it means you have to deal with a big Java disadvantage: there is no direct way to pass parameters to a Java program when called as a CGI program from a web browser. For my prototype, this was absolutely necessary. But the much larger amount of programming work when choosing Java was enough reason by itself not to do it.

7.2. Generic utilities

This section describes various generic utilities used. These utilities are used by most of the other programs to complete much-used tasks.

7.2.1. ceXML convenience functions

`cexmlhelper.py` (see Section D.1) is a small python program that provides basic functions that can be called from the other python scripts. It serves the purpose of keeping the amount of code in the other files down.

`set_linux_classpath()`

Sets the classpath needed to execute the Java programs. Uses the specific classpath info on my linux machine, hence the name.

`process_with_xslt(in_file, xslt_file, parameters=[])`

This function processes `in_file` with XSL/T stylesheet `xslt_file` (using the optional `parameters` to set variables in the stylesheet).

Some other small functions

These print out for instance the header needed when communicating an XML or HTML file back to the user by means of a web server. Also a function to remove temporary files is available.

7.2.2. Selecting a language

A step that is often necessary is to remove all languages except one from a file. The language part is all situated within the `name` tags. These tags have an attribute `language`. So, all `name` tags with an attribute `language` different from the language to be retained should be removed.

An additional pitfall to be avoided is to strip away the `name` tags with a `language` of `"si"`, because they are used within units using the S.I. system for measurements (like m and kg).

This task is performed by an XSLT stylesheet, named `language-.xslt` (see Section E.1). It reads through the entire file, removing all `name` tags except those specified in the language which is to be preserved *and* the units which are specified in S.I.

7.2.3. Propagation of characteristics

`propagate.xslt` (see Section E.2) is used to propagate the characteristics of parents to all children. As said before, `builtobjects` are stored in a tree-like structure using specialisation. For reasons of efficiency, the file containing the central tree-like structure does not repeat the characteristics of a `builtobject`

endlessly in all it's children. For presentation and other purposes, it is however necessary to have those characteristics copied from the parent to it's children, also: propagated. This stylesheet does the job.

7.2.4. Rejoining of characteristics

Again for reasons of efficiency (read: file-size) and because the same is done by the LexiCon (upon which model the ceXML message model is based), not the full information on the `quantifications` and `functions` is included every time they are part of a `builtobject`'s `characteristics`. Instead, only a reference is made in the form of `<quantification><name language="en">span</name></quantification>`. This should eventually be changed into the full `quantification`, like:

```
<quantification>
<name language="en">span</name>
<name language="nl">overspanning</name>
<amount/>
<unit>
<name language="si">m</name>
</unit>
</quantification>
```

In this `quantification`, of course the `unit` tag also should be expanded. This rejoining of place-holders with their original intended full information is done by the `rejoin.xslt` stylesheet (see Section E.3).

7.2.5. Conclusions

The LexiCon system can largely be implemented in XML. Simple XSL/T stylesheets can achieve a lot of the needed functionality.

7.3. Dealing with the vocabulary and the views

The purpose of the items covered in this section is to get some "building material" for the visualisation and procurement section (following later on) by filling the central multi-lingual vocabulary and two uni-lingual views.

7.3.1. The DTD of the message vocabulary

From the message model (Figure 6-11), I created a DTD (Section F.1). This DTD contains a few extra attributes for `builtobject`, which are used by some of the stylesheets and python programs. They have nothing to do with the actual model.

7.3.2. The filling of the message vocabulary

From an English book with data on concrete elements [Goodchild, 1997], I extracted all prefab concrete elements (PCE), because prefab concrete elements were to be the terrain upon which eConstruct would initially concentrate. All data was entered into an XML file (Section G.1) both in Dutch and in English. *The reader is encouraged to take a detailed look at the actual XML file in Section G.1, because it will really show the possibilities of this technology*. From now on, I will use the phrase *filled ceXML vocabulary* to indicate this file, to keep in sync with the phrase *filled LexiCon vocabulary*, much used in the eConstruct project.

7.3.3. Generating the uni-lingual views

As explained in Section 6.5.2, the message vocabulary needs to be scaled down to a view in order to be usable for exchanging information by an "ordinary" supplier or customer. Following the ideas presented in Section 6.5.4, `dtdbuilder.py` (Section D.2) generates either a Dutch or an English view DTD. For this it uses `propagate.xslt, rejoin.xslt` and `language-.xslt`. To prove their

existence, both DTD's are included in this document; the English one in Section F.2 and the Dutch one in Section F.3.

7.3.4. Creating the English and Dutch catalogs

The English catalog was filled with a few `composite-solid-prestressed-soffit-slabs` and `reinforced-rectangular-beams` and the Dutch catalog with `kanaalplaat` (hollow core slabs), `voorgespannen-rechthoekige-ligger` (reinforced rectangular beams) and `voorgespannen-T-ligger` (prestressed T-beams). Both catalogs are only filled with elements in their own native language! Only the Dutch is included, in Section G.2.

7.3.5. Conclusions

Both the model and the view are capable of holding real-world building and construction data. Also, the model can automatically generate the views.

7.4. Visualising

This section takes care of the user interface and proves in a visible way the multi-lingual capabilities.

7.4.1. Visualising the filled message vocabulary

A CGI script (again, python) named `treeview.cgi` was used to generate a the Internet page showing a tree structure on the left-hand side and the characteristics of the selected object on the right-hand side. The source of this script is included in Section D.3, the only CGI script I'll include. The output nicely resembles the look of GEN. Multi-linguality is not a problem, as Figure 7-1 and Figure 7-2 show.

The view is constructed by taking the filled message vocabulary (`cexml.xml`) and by processing it with `propagate.xslt, rejoin.xslt, language-.xslt` and `builtobjecttree.xslt`. This last one (not included in the listings) formats the resulting propagated, rejoined and pruned-of-all-but-one-language tree into below screen-shots.

Figure 7-1. Dutch view on the filled message vocabulary

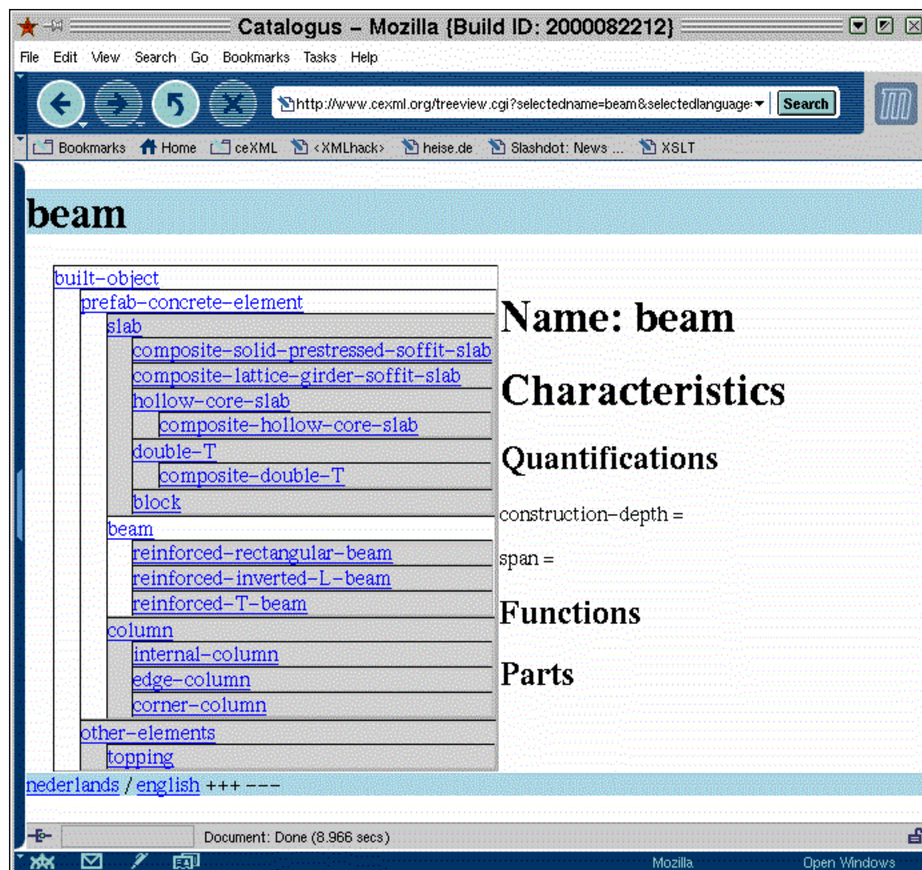

Figure 7-2. English view on the filled message vocabulary

7.4.2. Visualising the catalogs

The catalogs are transformed into viewable pages by a more elaborate approach. First a catalog is transformed using `view_to_cexml.xslt`, which transforms the catalog (which is in view format) back into a format which complies to `cexml_message.dtd`. This file is, as it has been generated from a view, uni-lingual. The resulting `builtobjects` are included in the filled message vocabulary `cexml.xml` by means of a Sed command. Those `builtobjects` have

been marked by an attribute `mode="view"` After that, propagate and rejoin are, again, used to fill those `builtobjects`, after which `strip.xslt` is called to strip out all `builtobjects`, except those marked with `mode="view"`. The propagation and rejoining have added the missing language, so that the result is, again, multi-lingual.

A CGI script is used to view the file generated above, calling `language-.xslt` to end with only the language of choice. Figure 7-3 shows the Dutch catalog in English and Figure 7-4 shows *both* catalogs combined in English.

Figure 7-3. The Dutch catalog in English

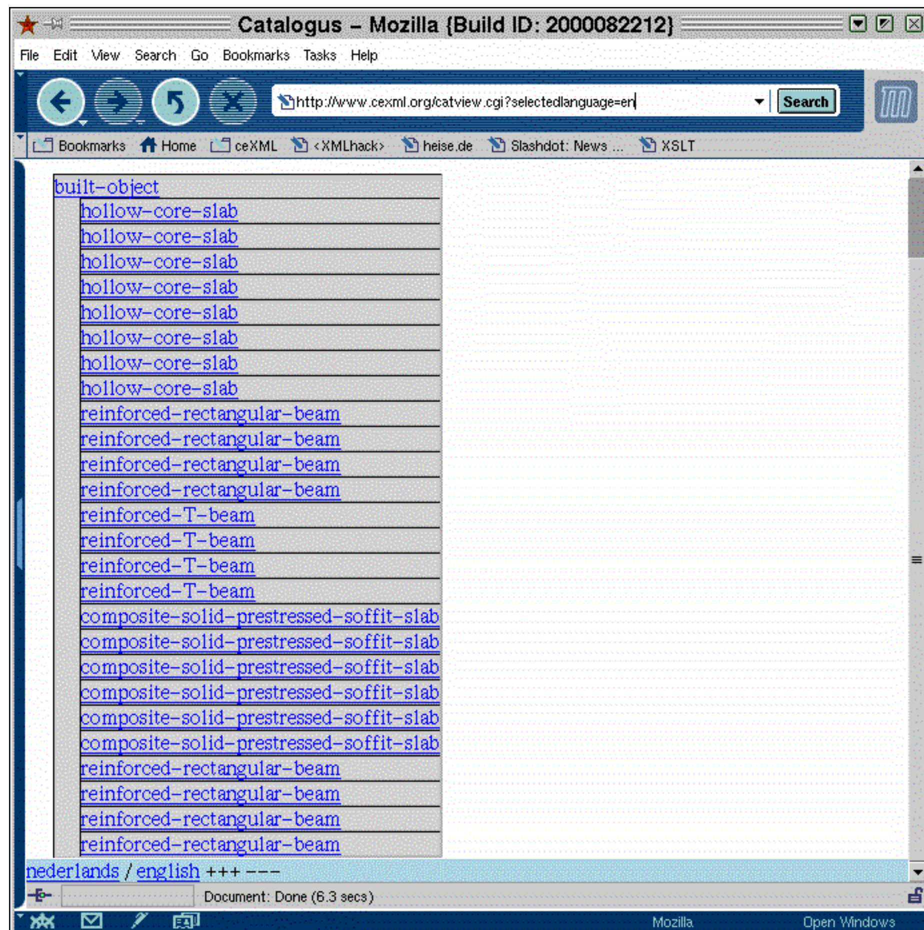

Figure 7-4. The content of both catalogs in English

7.4.3. Conclusions

Dealing with multiple languages is not a problem, even in visual user interfaces using common browser technology. The chosen model enables an easy generation of a GEN-like interface.

7.5. Using business processes and contexts

This section follows a message from it's inception through the first two steps of the business process of Figure 6-2. The message is first created, then it is send and consecutively received by the receiver. He then processes the message and sends a reply.

7.5.1. The message that is send

To send a message, first the envelope, message and business process model have to be integrated into one DTD. With this DTD it should be possible to send a message containing the following functionality:

- Addressing (from/to)
- Name of business process and the step therein
- List of built objects in *view* format

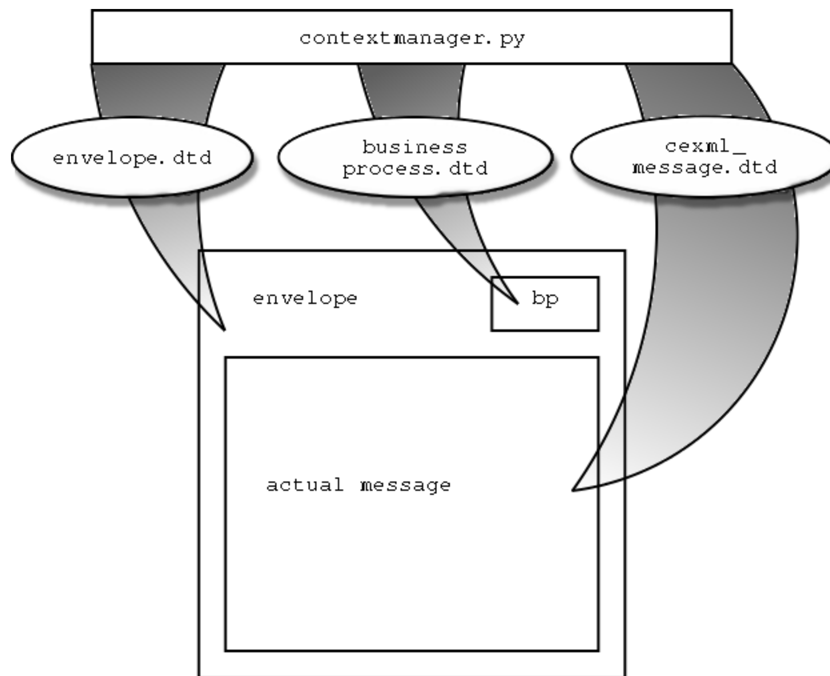

Figure 7-5. The working of the context-manager

The `contextmanager.py` script takes as it's input the desired language, the business process and the field (like "prefab concrete elements"). It's output is a DTD, suitable for sending messages in the selected language, with the business process and in the selected field. In Section F.4 you find a sample DTD, generated for communicating in the English language about prefab concrete elements. This DTD was used to fill a message, which is included below (Example 7-1). This message is a request from a contractor to a central repository (www.cexml.org, a fake address) which has knowledge of the two catalogs owned by the English and the Dutch supplier.

```
<!DOCTYPE envelope SYSTEM "pce_en_procurement.dtd">
<envelope>
  <sender>
    <!-- The addresses are in a format useful for a
         quickly hacked prototype :-) -->
    <address>../contractor/</address>
  </sender>
```

```
  <recipient>
    <address>../cexml/</address>
  </recipient>
  <businessprocess>
    <bpname>procurement</bpname>
    <bpstep>parts_needed</bpstep>
  </businessprocess>
  <message>
    <cexml>
      <hollow-core-slab amount="5">
        <span m="5"/>
      </hollow-core-slab>
      <reinforced-rectangular-beam amount="4">
        <span m="5"/>
      </reinforced-rectangular-beam>
      <double-T amount="8">
        <span m="5"/>
      </double-T>
    </cexml>
  </message>
</envelope>
```

Example 7-1. request1.xml

7.5.2. Sending and receiving the message

The message is send by feeding above request (Example 7-1) to the small `sender.py` script (Example 7-2).

```
#!/usr/bin/python
import cexmlhelper
import os
import sys
cexmlhelper.set_linux_classpath()
temp=cexmlhelper.process_with_xslt("%s"%sys.argv[1],"../cexml/dispatcher.xslt")
os.system("exec `cat %s` %s"% (temp, "%s"%sys.argv[1]))
```

Example 7-2. sender.py

This script runs the request trough a stylesheet that extracts a command-line out of it. It takes for instance the name of the business process, prepends the address of the recipient (which was specified in the form of a Unix pathname, every actor has it's own directory), thereby forming the name of an executable command. The location

of the request is passed along as a command-line parameter so that the executed program can read it.

7.5.3. Processing the message

The called program has the name of the business process (in this case `procurement`). In much the same way as the process described in Section 7.4.2, the actual message is stripped from it's containing envelope and (in reaction to the specific business process step), it is joined with both the English and the Dutch catalog. The information again is "propagate"d and "rejoin"ed. A new process is the `search.xslt` XSL/T stylesheet, which can be found in Section E.4. This stylesheet looks for all `builtobjects` with a `mode="search"` attribute. It then searches for a `builtobject` exact matching the searched-for name and the - in this case - correct desired `quantification` *span*. Then all not-searched-for `builtobjects` are stripped out and the reply basically is ready.

7.5.4. Sending a reply

The reply can be send in much the same way, but it is more illustrative if I format the answer for printing. This is done by converting it (by using `print_answer.xslt`, Section E.5) to a file with XSL/FO instructions. This is then processed with xml.apache's FOP (xsl/FO Processor) and the resulting PDF file is printed. This is done by the small script `print.py` (Section D.4). The result can be seen in Figure 7-6.

Figure 7-6. Screen-shot of the printed answer in Acrobat Reader

7.5.5. Conclusions

It is possible to have a context-manager assemble a purpose-built DTD which can be used to write a message that can be send from one party to another. For this, only the data contained within the message is needed, no extra information needs to be specified e.g. on the command-line. Searching for a specific item in both catalogs is

possible, whatever the language of the catalog. Also it is possible to generate nicely formatted output on the fly.

7.6. Conclusions

The LexiCon system can largely be implemented in XML. Simple XSL/T stylesheets can achieve a lot of the needed functionality.

Both the model and the view are capable of holding real-world building and construction data. Also, the model can automatically generate the views.

Dealing with multiple languages is not a problem, even in visual user interfaces using common browser technology. The chosen model enables an easy generation of a GEN-like interface.

It is possible to have a context-manager assemble a purpose-built DTD which can be used to write a message that can be send from one party to another. For this, only the data contained within the message is needed, no extra information needs to be specified e.g. on the command-line. Searching for a specific item in both catalogs is possible, whatever the language of the catalog. Also it is possible to generate nicely formatted output on the fly.

Chapter 8. Conclusions

This chapter first lists the most important conclusions from the entire document. This is followed by a comparison of the results and the original objectives. It closes with a personal appraisal of the research project.

8.1. Conclusions from Part I in *ceXML - an XML vocabulary for building and civil engineering*: *outlining the problem*

Three major existing solutions are available. Due to it's specific nature (fragmented, many relationships), the building and construction industry isn't using any of these solutions. So there is no industry-wide standard for the building and construction industry and none of the three existing solutions comes close to being that industry-wide standard.

The building and construction industry lags behind when compared with other industries and to solve that problem, a simple, cheap electronic solution is needed. In the European setting, a mapping of languages and concepts onto each other is needed.

8.2. Conclusions from Part II in *ceXML - an XML vocabulary for building and civil engineering*: *the building stones: XML and other initiatives*

The Internet is a good medium for communication. Access is cheap, generic tools (like browsers) are freely available, almost everybody can use it. XML is an Internet

technology that provides the possibility to use a vocabulary to tag information with nametags, giving meaning to the text that is communicated.

The three most important influences on ceXML are:

- Clearly ebXML has got the appropriate context mechanism, the distinction between generic and specific information and the idea to use specified business processes as a tool to handle all the data.
- GEN provides ideas for a good user interface and provokes thoughts on how to exchange information with existing systems.
- LexiCon's influence is the way the vocabulary should look like.

8.3. Conclusions from Part III in *ceXML - an XML vocabulary for building and civil engineering*: *Design and implementation of the vocabulary*

The LexiCon system can largely be implemented in XML. Simple XSL/T stylesheets can achieve a lot of the needed functionality.

Both the model and the view are capable of holding real-world building and construction data. Also, the model can automatically generate the views.

Dealing with multiple languages is not a problem, even in visual user interfaces using common browser technology. The chosen model enables an easy generation of a GEN-like interface.

It is possible to have a context-manager assemble a purpose-built DTD which can be used to write a message that can be send from one party to another. For this, only the data contained within the message is needed, no extra information needs to be specified e.g. on the command-line. Searching for a specific item in both catalogs is possible, whatever the language of the catalog. Also it is possible to generate nicely formatted output on the fly.

8.4. Successfulness of this research project

The successfulness of this research project is determined by comparing the outcome with the original objectives

8.4.1. Investigating the state of the art

Investigating the state of the art of XML and related technologies, as well as related vocabularies and related developments like EDI (Electronic Data Interchange) and PDT (Product Data Technology).

My research into XML and related technologies has been successful. Much accurate information has been gathered. Most of this information has been used for eConstruct's first publication. The related vocabularies have not received as much attention, only so much as needed to gain insight in their working. The LexiCon received more attention, it being the basis for my model. The related developments have been merely glanced at, except ebXML, which provided me with lots of input for the prototype. I also attended part of an ebXML meeting in Brussels.

8.4.2. The design and implementation of a vocabulary

The design and implementation of a vocabulary, separate from eConstruct's work at making eConstruct's vocabulary (named bcXML, Building and Construction XML). My task is to make a simple prototype vocabulary, allowing the mapping of one language into another. Required input was the LexiCon meta-model (Section 5.3). The model has to deal with prefab concrete elements, a field eConstruct decided to concentrate upon for the time being, mostly because the Greek partner in the project is a supplier of prefab concrete elements.

The vocabulary and surrounding techniques have been mostly based upon the LexiCon meta-model and the ebXML way of working. The model turned out to work real fine, while the idea of using a simplified view for e.g. catalogs also proved usable. The amount of elements I used to fill the model is a bit low, though. Especially the functions have not been filled in and only a marginal amount of

quantifications. But for a prototype it was enough to prove that the concept works. Mapping one language into another worked fine. It should be noted that only simple translation was implemented, without any difficult mapping of not-completely-similar concepts.

8.4.3. Prototype implementation

Testing of the vocabulary by means of a simple application. A vocabulary by itself is not enough to prove that the concept can work. A prototype has to be designed in order to test it. eConstruct concentrates it's effort at first on the buying/selling phase, because that is a regular, known form of e-commerce and it will be relatively easy to gain widespread acceptance of that part of eConstruct's functionality (thereby paving the way for wide-spread usage of eConstruct as a whole). To preserve the link with eConstruct, the prototype will concentrate on the buying and selling phase also.

The prototype implementation is a collection of well-crafted XSL/T stylesheets providing useful basic functionality and a handful of hacked-together python scripts to tie it all together. Part of those scripts are used as CGI programs to visualise either the filled vocabulary or a catalog. Those web representations provided the desired GEN-like interface, but without built-in search functionality. Almost all python scripts were happily translating vocabularies into views and vice versa *and* were translating back and forth between the Dutch and the English language. The multi-lingual capabilities and the conversion from a vocabulary to a view are by far the brightest gems of this prototype.

The final test to use the context mechanism and business processes to get a question-answer session rolling worked out, but not (to my taste) in a very well-crafted way. It *did* prove, however, that the context mechanism works, that business processes can be used to roll along a transaction solely based upon the data in the original request. The action undertaken upon receiving of the request is the best indicator for the possibilities of the entire system: *an English request for two specific built objects with a span of 5 meters returns the four matching elements from both an English and a Dutch catalog*. To top it off, the resulting answer is formatted for printing using a simple XSL/FO stylesheet.

8.5. Perspective for the building and construction industry

What is the perspective for the building and construction industry as seen from this research? It is the fact that the Internet and XML are going to change the industry deeply. For the first time, two essential pieces are in place which are needed to dramatically increase the efficiency of the communication in the industry. First, the internet provides a cheap and easy communication medium, suitable for even the smallest constructor firms. Second, XML provides the means to exchange virtually all data and information in a way that is usable and (financially) affordable by any firm, big or small.

So, this research showed that the Internet and XML are *the* way ahead for the building and construction industry. But to make it all happen, an framework is needed. XML *does* provide the means to exchange data and information, but it needs a vocabulary. This vocabulary will be provided by the eConstruct project (in it's vocabulary bcXML). In this research I made a quick prototype, having limited time and using only a limited set of possible techniques. This prototype allready provided prove of the fact that meaningful communication about construction elements indeed *is* possible in multiple languages.

This shows that eConstruct will be able to present both the Internet and XML in a usable way to the building and construction industry, providing the industry with an undreamed-off level of communication possibilities. For this, eConstruct needs to

- create a vocabulary with enough expressive power to communicate meaningfully
- re-use and connect to existing solutions in order to quickly gain critical mass
- provide programs or web interfaces that allow cheap and easy usage of the system, as well as connections to existing programs

8.6. My personal appraisal of this research project

- The learning experience was extensive. I gained a lot of knowledge, also from the various eConstruct meetings. It really sparked my interest for this field of research
- I managed to attain almost all objectives of this research project. The prototype proves that it is possible to use XML to communicate meaningfully in the building and construction industry without being constrained by different languages.
- The prototype could to my taste have been made prettier. The functionality is in place and is working, but it is a bit too much of a "dirty hack". But, when the day is over, it is a prototype after all.

I was sent out into the wide, wide world to scout the terrain and to bring back interesting technologies, useful for the building and construction industry. This I have done, choosing my *own* roads and picking up those items *I* found interesting or useful. The reader should regard this document as a scout's report, showing the possibilities of the terrain and depicting the rich lands ahead. It is my advice, as a scout, to order the entire column to march down the road and to cultivate the territory ahead, for the Internet and XML *are* going to be very, very useful for the building and construction industry.

IV. Appendices

Table of Contents

A. Deeper introduction on XML and related technologies 105
B. STEP explained in more detail 123
C. Reading Unified Modelling Language (UML) diagrams 125
D. Python listings 127
E. XSL/T listings 131
F. Document Type Definitions (DTD's) 143
G. XML files 151

Appendix A. Deeper introduction on XML and related technologies

Abstract

This section elaborates on the introduction given in Section 4.2.

In this section XML (eXtensible Markup Language) and a selection of related technologies is discussed. XML is in effect a standardised way of dealing with data. This makes it possible for a number of additional technologies to surface and blossom together with XML, thereby making XML more powerful and useful. The first section introduces XML as it stands by itself, the rest of the sections deals with all the standards that supplement and enhance XML.

A.1. eXtensible Markup Language (XML) itself

XML itself is only the way data is tagged with information about the data. This section describes this central technology, central to the rest of the XML framework.

A.1.1. Introduction to XML

XML stands for eXtensible Markup Language. It is a subset of SGML, the Structured General Markup Language, which was a promising technology, but had a reputation of intense complexity stemming from the enormous levels of customisability and flexibility [St. Laurent, 1999], thus making it too difficult and unattractive to receive a large following. XML is much simpler and smaller, though it allows for flexibility undreamed of for many developers and users. The syntax is quite akin to that of another SGML-descendant, HTML. XML uses an easy to

understand HTML-like syntax. For most uses, XML will probably completely replace SGML.

```
      <chapter>
      <title>Coffee</title>
      <para>
The delicious aroma surrounding the coffee machine puts
a smile on <emphasis>many</emphasis> faces.
      </para>
      </chapter>
```

Example A-1. XML syntax example

A.1.2. Structure of XML

XML - as a technology - consists of two parts.

Content

The XML file itself, the file containing the markup and the data, as in above example.

Semantics

The XML Schema (newer technology) or Document Type Definition (DTD) (older technology). At the beginning of the XML file, a reference is made to a specific DTD. The file used to make this very document points to the DocBook 3.1 DTD, which specifies tags like `<chapter>` and `<emphasis>`. The DTD specifies the allowed tags *and their hierarchy*. A section2 is only allowed within a section1, for example. Also specified are the allowed attributes like `colour="blue" strength="B35"`.

Additionally, for XML files that aren't meant just for data storage or data exchanges between computers, information on how to visualise the xml data is needed. For these needs, a third part is necessary:

Visualisation

One or more associated visualisation stylesheets. For every XML schema, there should be one or more stylesheets, specifying how to display/print/export/save an XML file associated with the schema. For example, it is possible to use a html-stylesheet with the XML-file which contains this document, a print-stylesheet, etc. This is only needed for XML files that need viewing/printing/etc.

A.1.3. Usage of XML

By use of this threefold model (XML-file, schema, stylesheet), this technology is a good example of the maxim "divide and conquer". The data can be stored neatly, readable and object-wise in a simple text format. The format (specified in the schema) itself is adaptable to whatever need there may be *in a well-defined way*. Whatever format one chooses, any XML-enabled program can read the information, provided the schema is accessible to that program. To read it in this case means that the program can build a tree-like representation of the data because of the hierarchical nature of XML. To some applications, this is enough to be able to use the data. This is the case when XML is used as a simple way to store information only meant to be read by a specific application which knows what to do with it. For world wide web-like applications, an associated stylesheet is needed. A program which converts XML-documents like this article, which uses the DocBook 3.1 schema, to a printable format will need information which specifies that a `<para>...</para>` pair indicates a block of text with one cm above and below, margins of 3 cm and with a TimesRoman font. Likewise, with the same data, the same schema and a different stylesheet, another program can easily generate a set of webpages from the same source.

Likewise, a set of XML-files with various schema's and stylesheets can be used to represent a building. Some parts, like an elevator, get their own XML file because it is being supplied by a subcontractor. The file `elevator.xml` is referenced in the main XML document. It is now possible to read all files and use a stylesheet (which

must be supplied with all the schema's) to generate a nice-looking viewable model of the entire building, including a moving elevator. The stylesheets - in this case - must be able to specify the information contained in the XML files in a way suited for VRML (Virtual Reality Modelling Language, a file format for generating three dimensional images and animations). Current W3C (world wide web consortium) research includes XSL/T (eXtendible Stylesheet Language/Transformation part), which allows transformations from one format into another (see Section A.2).

In the same way, a complete list of needed parts can be generated, provided the schema ensures the XML files to contain that information and again provided a usable stylesheet is available.

XML is, in essence, a standardised way to deal with metadata. The schema is the place to specify the metadata, the actual XML file is where the data is placed, using XML's standard way to tag the various parts of information in it with the metadata specified in the schema.

A.2. eXtensible Stylesheet Language/Transformation part (XSL/T)

XSL/T (eXtensible Stylesheet Language/Transformation part) is the single most useful standard alongside XML itself. It allows for transforming one format into another. Beware of confusion, for a lot of technologies have got the name *stylesheet* in them. This section deals with the transforming stylesheet XSL/T, while the next session deal with the two formatting stylesheets (named XSL/FO and CSS).

A.2.1. Background

XSL/T was designed as a part of XSL, the eXtensible Stylesheet Language. XSL/T adds the Transformation of one XML document into another. XSL itself provides (besides XSL/T's transformation) formatting of XML documents for on-line reading, printing, speech programs, etc. (see Section A.3 At the moment, XSL/T

has gained much support from XML users in contrast to the not yet widely used formatting part of XSL (named XSL/FO).

The difference between transformation and formatting might not be so clear at first. With *formatting* is meant the way it should look. A paragraph should be indented by 2cm, font size 11pt and a black colour. With *transformation* is meant the transformation of e.g. an entire document into a table of contents and just a list of all available section abstracts.

A.2.2. Working of XSL/T

A program capable of performing XSL/T processing takes as it's input both an XML file and the stylesheet document mentioned in the XML file by use of an `<?xml-stylesheet href="sample.xslt" rel="stylesheet" type="text/xsl"?>`-tag. The output can be any format, but normally a valid XML file or HTML is returned. Both the ingoing and the outgoing XML file are internally represented by a tree-like structure. The stylesheet transforms and re-arranges the incoming tree into the outgoing tree.

The stylesheet itself is a valid XML file, all instructions for the XSL/T processor are given by means of XML tags. The stylesheet contains multiple templates. Each template has a rule where it is matched against. The contents of the template are instructions on how to transform the part of the tree which was matched by the template rule. To transform a `<emphasis>`-tag in the incoming file to HTML's `<i>`-tag, the following template can be used:

```
<xsl:template match="emphasis">
<i>
<xsl:apply-templates />
</i>
</xsl:template>
```

Example A-2. XSL/T - basic example

This piece of code means that if the processor encounters an `<emphasis>`-tag in the incoming tree, it should follow the instructions within the template. That means putting the `<i>` opening and closing tag in the output tree and to continue processing within the `<i>`-tag by applying any additional templates that might be matched further down the tree.

Of course, tags like `<xsl:if>` and `<xsl:while>` are also available to make conditional processing possible. Also some support for variables and calculation is available, all geared towards being able to perform all needed document transformations, including constructing a *table of contents* from a well-structured document and so on.

A.2.3. Example usage: transforming database output

Much information is - and will be - contained in relational databases. In this example I will extract information out of a testdatabase containing information about a few Internet pages I maintain at a local testsite. The database structure is as follows:

```
CREATE TABLE pages (
url char(100) DEFAULT ″ NOT NULL,
title char(200),
section char(10),
subject char(10),
date char(8),
printable char(100),
PRIMARY KEY (url)
);
```

Example A-3. XSL/T example - database table structure

This database is accessed using a JDBC connection. Every database vendor has some tool of sorts to output XML information from it's databases, but in this case I use an XML page processed by Cocoon (part of Apache.org's XML server project, see Section A.6).

```
<connectiondefs>
<connection name="page_connection">
<driver>org.gjt.mm.mysql.Driver</driver>
<dburl>jdbc:mysql://localhost/rr_doc</dburl>
<username>rr</username>
<password>Secret_password</password>
</connection>
</connectiondefs>

<filelist>  <!--
the filelist tag is irrelevant to the example -->
<query connection="page_connection">
  select * from pages order by section,subject,date
</query>
</filelist>
```

Example A-4. XSL/T example - xml file before transformation

When the XML file containing this code fragment is processed by Cocoon's SQL processor it removes above tags and replaces the query tag with the following fragment:

```
<filelist>  <!--
the filelist tag is irrelevant to the example -->
<ROWSET>
<ROW ID="0">
<url>writings/if5950/if5950_article/t1.html</url>
<title>Use of XML with project databases and Java</title>
<section>article</section>
<subject>xml</subject>
<date>20000202</date>
<printable></printable>
</ROW>
<ROW ID="1">
<url>writings/werkplan/werkplan/t1.html</url>
<title>Werkplan</title>
<section>info</section>
```

```
<subject>graduating</subject>
<date>20000315</date>
<printable>writings/werkplan/werkplan.pdf</printable>
</ROW>
</ROWSET>
</filelist>
```

Example A-5. XSL/T example - xml file after processing with cocoon's SQL processor

Every row is contained within `<ROW>`-tags and the actual data is contained within tags named after the column name in the database. I wanted to convert this information into a normal HTML table. For that purpose I use the following stylesheet (also processed by Cocoon):

```
<xsl:template match="filelist">
<table>
<tr>
<td>Title</td>
<td>Section</td>
<td>Subject</td>
<td>Date</td>
<td></td>
</tr>
<xsl:for-each select="ROWSET">
<xsl:for-each select="ROW">
<tr>
<td>
<a href="{url}"><xsl:value-of select="title"/></a>
</td>
<td><xsl:value-of select="section"/></td>
<td><xsl:value-of select="subject"/></td>
<td><xsl:value-of select="date"/></td>
<td>
<xsl:if test="not(printable = ″) ">
<a href="{printable}">Printable version</a>
</xsl:if>
```

```
</td>
</tr>
</xsl:for-each>
</xsl:for-each>
</table>
</xsl:template>
```

Example A-6. XSL/T example - the stylesheet

A.2.4. Compiling stylesheets

A very recent development allows one to compile an XSL/T stylesheet into native code (using c++) or Java bytecode. The resulting program can receive an XML file as input. The output is the same as if you had processed the XML file with an XSL/T processor and the original stylesheet. Compiling a XSL/T stylesheet into a program has massive speed advantages, because a generic stylesheet processor needs to be able to do everything and the compiled stylesheet only has to be able to use it's own stylesheet.

A.2.5. Possible use for ceXML

There are a few possible roles I see for the usage of XSL/T:

- As above, converting database output to more suitable XML formats
- Generating easily accessible HTML pages for use in ordinary browsers from the more specialised XML data used by ceXML.
- Binding together multiple datasources in a platform- and vendor-neutral way.
- Selecting information from a ceXML file to suit the needs of the user (e.g. only the list of materials)

Note should be taken that these are only possible roles, there are many more technologies that can be used. The advantage I see at the moment is that using

XSL/T means using a native XML technology instead of using a purpose-written program. Using as much native XML technologies as feasible makes for a nice overall solution which can be adapted and re-used to one's heart's content.

A.3. Stylesheet languages for visualisation

As said on the W3C-pages: *By attaching style sheets to structured documents on the Web (e.g. HTML), authors and readers can influence the presentation of documents without sacrificing device-independence or adding new HTML tags*. Separation of content and presentation, one of the holy grails of the Internet. In effect, you want one file containing information expressed with meaningful tags and another one containing instruction on how to present that tagged information to the user.

A.3.1. Difference Cascading Style Sheets (CSS) and eXtensible Stylesheet Language/Formatting Objects (XSL/FO)

CSS means Cascading Style Sheets (you can lay some styles on top of each other, so to say). XSL/FO means eXtensible Style Sheets / Formatting Objects.

Both CSS and XSL/FO are formatting stylesheets in their own right. Both use the same underlying formatting model. Both have knowledge about formatting like `font-size` and `margin-left`, but the syntax they use to access this formatting is different.

The biggest difference is that XSL/FO is the formatting part of XSL. The other half of XSL is XSL/T, the part that is capable of changing the information, rearranging it, extracting parts, adding titles, etc. Both can be used independently (and especially XSL/T mostly *is*), but using the same language and way of expressing for both the transformation and the formatting is an advantage.

A.3.2. Usage of stylesheets

The stylesheet controls the way the information is displayed to the user, either in print or on screen. Tags are matched against rules and the relevant formatting information is applied. In XSL/FO, making `<scream>` into bold text is done with the following rule:

```
<xsl:template match="scream">
<fo:inline-sequence font-weight="bold">
<xsl:apply-templates/>
</fo:inline-sequence>
</xsl:template>
```

Example A-7. XSL/FO example

The same is done as follows in CSS:

```
scream { display: inline; font-weight: bold; }
```

Example A-8. CSS example

A.3.3. Use for ceXML

Both technologies do not offer anything spectacular to ceXML, but stylesheets are simply needed to present information to the user in a well-formatted way. With plain HTML a lot can be solved, but when one uses XML, information *has* to be added indicating how it should be displayed, since no browser can possibly know how to display a `<contractfooter>`-tag.

A.4. XML Namespaces

This section discusses XML Namespaces, a method of resolving conflicts between similar tagnames having separate meanings. Also a choice between the use of namespaces and the ebXML context mechanism is discussed.

A.4.1. Working

When designing an XML schema, many designers re-use other schemas. It is quite common to use a subset of HTML when you need a piece of formatted text somewhere in your XML files. But, when using elements from multiple sources (read: XML schemas), you need to distinguish between the elements. `<title>` can be the title of a chapter or the title of a person. When one tag exists in multiple schemas, you need a way of telling them apart.

XML's way of telling tags apart is to add a namespace reference in front of the tags. A namespace reference is a string added in front of the tagname, separated by a colon. The definition of that namespace reference has to be done beforehand, linking the reference (in itself just a string) with the intended XML schema. See Example A-9

```
<doc:chapter xmlns:doc='http://docdoc.com'
  xmlns:titles="http://title_references.net">
<doc:title>Title of chapter</doc:title>
<doc:para>
  This is a sample paragraph about
<titles:title>Prof. Dr.</titles:title> Broadsword
  demonstrating namespaces.
</doc:para>
</doc:chapter>
```

Example A-9. Example use of namespaces to distinguish between two title tags, one indicating a chapter's title, the other indicating a person's title.

There are more possibilities, for example defining a default namespace for all unspecified tags within a tag (handy when including HTML for documentation). But above example covers the basics.

A.4.2. Design decisions involving namespaces

The mentioned example of including some HTML markup within an XML document is one of the normal uses of namespaces. It can, however, also be used to glue together different classifications or different models. ceXML could define a set of general tags (including addressing etc.), which can be complemented by allowing tags from, say, a prefab concrete namespace. This can be used as an alternative to ebXML's context mechanism (see Section 5.1.2).

Using namespaces results in an XML-only solution, which can be readily used by everyone. The context mechanism needs a purpose-build software solution to accompany it. But the context mechanism is more flexible and it can generate schemas without any need for action on the client side.

The namespace mechanism has been devised to tell apart tags with the same name but a different meaning. The context mechanism has been devised to construct a set of tags applicable to a certain intersection of contexts. As the intended usage of ceXML is to communicate about building and construction data, it is likely that one has to deal with a number of different classification systems and models. This means that it will be difficult to assemble the right set of tags, excluding the namespace mechanism for that task.

A.4.3. Conclusions

Namespaces can be useful, but their use is limited. For the detailed assembling of the right set of tags for communicating a about a specific building or construction, the context mechanism is best suited. For separating included documentation (probably in an HTML-like format), the actual tags containing the construction data and the generic addressing-like tags, the namespace mechanism is the best method to prohibit possible tagname conflicts.

A.5. XML Linking

This section deals with XML Linking, the XML technology needed if you intend to link together various pieces of information (like a technical drawing and the XML files containing the information about the various parts).

A.5.1. Introduction

Formerly also known as XLink and XLL (eXtensible Linking Language), XML Linking provides the mechanism needed by XML to interconnect. It's role is comparable to HTML's `<a href="something.html">...</a>` tag, but with many more possibilities. It should be noted, though, that there is practically no browser support for the additional functionality. Of course, XML Linking only works in XML enabled programs.

A.5.2. Working

Any XML element can be used as a link by adding an `xlink:type` attribute to the element, provided the `xmlns:xlink="http://www.w3.org/1999/xlink"` namespace is enabled for that element. As an example, `<author xlink:type="simple" xlink:href="mailto:R.vanRees@ct.tudelft.nl" >Reinout</author>` is an author tag which points towards an email address. This `type="simple"` usage is in fact the old HTML kind of linking.

The `type="extended"` usage is the other way XML Linking can be used. An extended link is best described as a *directed labelled graph*, connecting multiple nodes with arcs. Do not mistake the extended link to mean just one link, it can be, but normally it is a set of links (better: arcs). It is composed of arbitrary tags having the following attributes:

`type=extended`

Indicates an element containing elements which form a directed labelled graph.

`type=locator`

Indicates an element which points toward a remote location. That is, this element serves as a node in the graph, pointing towards a remote resource with a mandatory `href="somewhere.xml"` attribute.

`type=resource`

Indicates an element which serves as a *local* node. That is, the element doesn't point towards a remote location, the information inside the element itself is pointed to.

`type=arc`

Indicates an element describing one or more arcs (or: links) between nodes using `xlink:from="..."` and `xlink:to="..."` tags. The text on the dots are the allowed origin(s) and destination(s) of the arc. `xlink:to="Dutch_mirror"` means that this arc points towards all nodes which have the `xlink:role="Dutch_mirror"` attribute.

A.5.3. Possible uses

There are three major ways in which to use XML Linking:

The HTML way

Any XML element can be made to behave just like the `<a href="...">...</a>` tag of HTML, with the same purpose and usage pattern. Just simple hyperlinking.

One-to-many relations

One local *resource* can point to multiple *locator*s (remote information). This way, clicking on a link may bring up a small menu of possible choices listing a few mirror sites, for example. Or, in an eventual online version of the lexicon, it might link one Dutch term to the three possible alternative Norwegian terms.

Out-of-line links

Used this way, the document containing the XML Links is called a *linkbase*. It

does not contain any of the information which the links point to. It contains links that point to information on other pages, possibly commenting on it, providing extra information, etc. When proper tools or browsers become available, a user might load an external linkbase and use it for browsing a vendor's online documentation, all the while comparing this vendor's products to the equivalent products of two other firms. Or an image of a building crane might with one linkbase redirect a click on part of the image to the online documentation and with another linkbase redirect it to an enlarged picture of that specific part.

A.5.4. Possible use for ceXML

When ceXML is going to utilise XML instead of HTML for client side presentation, at least the `xlink:type="simple"` usage is mandatory. Due to the (normally) many contractors in a building and construction environment, the out-of-line links (linkbase) looks like a good way to glue together various bits of information without the need to centralise all those bits of information. Also the one-to-many possibilities built into XML Linking make it possible to present different views on a specific part (either the supplier's information on the part, an enlarged photo, it's maintenance record, a error reporting form, etc.)

XML Linking is a good tool which can solve some problems with designing an XML vocabulary for the building/construction industry and civil engineering, but it desperately lacks common browser support. Therefore, when using current browsers, much has to be emulated using javascript or like technologies. But the XML Linking W3C working draft's last call period ended 20 March 2000, so the the technology is almost solid, which will make applications rapidly available. Especially since the Mozilla browser effort has completed all but the last bits of basic XML functionality, XML Linking can be expected fairly soon to be available in a mainstream browser. Also Explorer can be expected to follow soon.

A.6. Cocoon - Apache's xml effort

This section describes one of the efforts at creating an XML based document server. While documents aren't ceXML's main focus, it is a handy way to extract information from XML files containing (for example) technical data and to format it automatically for presentation on the world wide web.

A.6.1. Background

Apache is the dominant web server (60% of all sites run it), available for almost all platforms. Cocoon is part of the xml.apache.org effort at creating a set of standards-based XML solutions. Cocoon is the part which provides the central XML server. It is implemented in Java with a very modular architecture, allowing for example a choice between various XML parsers. Most of the building blocks of xml.apache.org have been donated by IBM, Sun and Oracle in order to integrate them into one powerful whole.

A.6.2. Working of cocoon

Cocoon, being programmed in Java, uses the Java Server Pages (JSP) mechanism. Every web server with JSP can redirect files with a specific extension (like `.xml`) to Cocoon, while serving normal HTML pages, images, etc. itself. According to processing instructions, Cocoon then routes the requested document through a series of transformation steps before presenting the information to the user. To name a possibility, it can start with an XML file containing SQL statements, it then queries the database, replacing SQL with the database results in XML tags, then transforms it using an XSL/T stylesheet into a readable XHTML file. Fairly quickly a way to integrate many existing scripting solutions into the Cocoon framework will be completed.

A.6.3. Possible use for ceXML

The possibility to use standard XML tools like XSL/T for transformations and XSL:FO for formatting into, say, pdf make Cocoon an attractive framework for a low-cost server side solution to use with ceXML. Though there is a choice of

various parsers and processors, home-brewed solutions can be made and integrated fairly easily. When web server functionality is needed by ceXML, Cocoon can provide a quick initial solution, if not more.

Appendix B. STEP explained in more detail

The inner workings of STEP are interesting enough to include an outline in this document. Because it does not fit well in the flow of the main text, this information is included in this appendix.

The STEP technology consists of five parts:

Application protocols

These protocols describe specific application fields. Not only do they describe which data is to be used to describe a product, but also *how* the data is to be used.

Integrated information resources

These are the common resources that are used by the application protocols. Everything is divided into separate resources, so that re-use is possible. A number of resources is earmarked especially for it's generic character in order to encourage their use, so that the costs of developing a new standard (using STEP) can be kept down.

Implementation & conformance

This part describes how to map the models which have been formally specified in STEP into a format used in the specific STEP implementation. Also this part contains information on how to perform conformance[1] testing.

Abstract test suites

In co-operation with above conformance specification (from the implementation & conformance part), this part provides the test data and criteria used to assess the conformance of a software package to the certain application protocol.

---

1. Conformance means whether or not the data in question is conforming to the standard. This standard has to be known and formally specified in order to enable a check for standard conformance.

Description methods

This is the bottom layer of STEP, explaining how to describe all STEP related models and software. An important part is the EXPRESS language reference, defining the data modelling language used in STEP.

Appendix C. Reading Unified Modelling Language (UML) diagrams

In this document, only a limited set of symbols from UML (Unified Modelling Language) is used. Two kinds of diagrams, the Use Case diagram and the class diagram are presented. Again, this is not a complete introduction on UML, only what is needed to read the diagrams in this document.

C.1. Use Case diagram

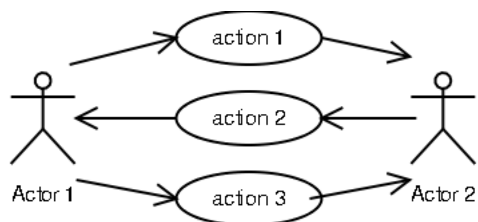

Figure C-1. Use Case diagram example

Two actors want to interact with each other. Actor 1 performs action 1 towards Actor 2, who performs Action 2 towards Actor 1 and so on. An actor can be someone like a *contractor* and an action can be something like *request catalogue*.

C.2. Class diagram

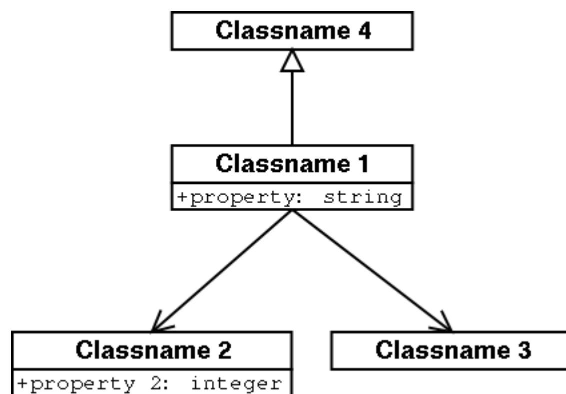

Figure C-2. Class diagram example

A class is a single item. Like a `builtobject` or a `function`. They are depicted using a rectangle. If there are any attributes (that is, some internal information for that class), an extra box is drawn in the rectangle and the attribute information is placed in the second box. A class `door` can have the attribute `language="english"`.

A line from one class to another indicates a relationship, For example, a `quantification` has both an `amount` and a `unit`. A line with a triangle at it's end indicates an inheritance relation, the attributes and other characteristics of the item having the triangle next to it are inherited by the item on the other end of the line.

Appendix D. Python listings

D.1. cexmlhelper.py

```
#!/usr/bin/python

# This module contains some helper functions to keep the length of the
# files that use this module within reasonable bounds. The functions
# all provide standard functionality needed to work with ceXML and
# xml.apache.org's Xalan parser and Xerces XSLT processor.

import tempfile
import os

# initialise tempfile's first part of the temporary filenames
tempfile.template = "cexmlhelper"
# Start a list containing the generated temporary files, allowing for
# a clean-up when they're not needed anymore
list_of_tempfiles = []

def set_linux_classpath():
    # Sets the classpath. This is a specific implementation for my
    # personal linux system, someday this will be more
    # generic. Perhaps I'll even write a nice wrapper for more
    # systems :-)
    os.putenv("CLASSPATH","/usr/lib/jdk1.1/lib/classes.zip:"+
    "/usr/share/java/xerces.jar:/usr/share/java/xalan.jar")

def process_with_xslt(in_file, xslt_file, parameters=[]):
    # feeds "in_file" to "xslt_file" using the optional
    # "parameters". The "parameters" must be a python list of
    # key-value tuples. The return value is the name of the temporary
    # file containing the output.
    commandline = "java org.apache.xalan.xslt.Process"
    # Add the in_file
    commandline = commandline + " -in %s" % in_file
    # Add the xslt_file
    commandline = commandline + " -xsl %s" % xslt_file
    # Generate a temporary filename and add it as the process' output
    out_file = tempfile.mktemp()
    list_of_tempfiles.append(out_file)
    commandline = commandline + " -out %s" % out_file
    if len(parameters) > 0:
        i = 0
        while i < len(parameters):
            commandline=commandline+" -param %s %s" % parameters[i]
            i=i+1
    # add some stuff to suppress annoying output
    commandline = commandline + " -Q -QC"
    # run the commandline
    os.system(commandline)
    # return the output filename (a temporary one)
    return out_file

def print_xml_header():
```

```
    # Prints the header needed to send the file over HTTP. Normally
    # not needed, because print_xml_file() calls it automatically, but
    # you never know...
    os.system("echo Content-type: text/xml")
    os.system("echo")

def print_html_header():
    # Prints the header needed to send the file over HTTP. Normally
    # not needed, because print_html_file() calls it automatically, but
    # you never know...
    os.system("echo Content-type: text/html")
    os.system("echo")

def print_xml_file(xml_file):
    # Prints the xml header and the xml_file
    print_xml_header()
    os.system("cat %s" % xml_file)

def print_html_file(html_file):
    # Prints the html header and the html_file
    print_html_header()
    os.system("cat %s" % html_file)

def remove_temporary_files():
    i = 0
    while i < len(list_of_tempfiles):
        os.system("rm %s" % list_of_tempfiles[i])
        i=i+1
```

D.2. dtdbuilder.py

```
#!/usr/bin/python

# This cgi script is used to display the central ceXML xml file
# containing all the builtobjects, quantifications, languages,
# etcetera. Also, some small editing possibilities are available.

import sys
import os
import cexmlhelper
import tempfile

try:
    selected_language = sys.argv[1]
except:
    selected_language = "en"

# Set the classpath
cexmlhelper.set_linux_classpath()
# First, propagate all characteristics of the parents over all the
# children
temporary_file = cexmlhelper.process_with_xslt("cexml.xml",
                                               "propagate.xslt")
# Then, re-join the complete function, unit and quantification info
# with the builtobject tree
temporary_file = cexmlhelper.process_with_xslt(temporary_file,
```

```
                                                "rejoin.xslt")
# Next, strip out the un-needed languages
temporary_file = cexmlhelper.process_with_xslt(temporary_file,
                                                "language-.xslt",
                                                [("selectedlanguage",
                                                  "\"'%s'\"" %
                                                  selected_language)] )
# change it to the DTD of a view
temporary_file = cexmlhelper.process_with_xslt(temporary_file,
                                                "view.xslt")
# then, correct some errors which aren't dealt with easily in XSLT
temporary_file_2 = tempfile.mktemp()
cexmlhelper.list_of_tempfiles.append(temporary_file_2)
os.system("sed 's/()\*>/EMPTY>/g' %s > %s" % (temporary_file,
      temporary_file_2)  )
os.system("sed 's/|)/)/g' %s > %s" % (temporary_file_2, temporary_file) )

try:
    resulting_dtd = sys.argv[2]
    os.system("cp %s %s" % (temporary_file, resulting_dtd))
except:
    os.system("cat %s" % temporary_file)

cexmlhelper.remove_temporary_files()
```

D.3. treeview.cgi

```
#!/usr/bin/python

# This cgi script is used to display the central ceXML xml file
# containing all the builtobjects, quantifications, languages,
# etcetera. Also, some small editing possibilities are available.

import cgi
import os
import cexmlhelper

# Read in the parameters passed to this script using the CGI POST
# method
form = cgi.FieldStorage()
try:
    selected_language = form["selectedlanguage"].value
except:
    selected_language = "en"
try:
    selected_name = form["selectedname"].value
except:
    selected_name = "beam"
try:
    mode = form["mode"].value
except:
    mode = "normal"
try:
    new_name = form["newname"].value
except:
    new_name = "test"
```

```
try:
    edit_language = form["editlanguage"].value
except:
    edit_language = "en"

# Set the classpath
cexmlhelper.set_linux_classpath()
# First, propagate all characteristics of the parents over all the
# children
temporary_file = cexmlhelper.process_with_xslt("cexml.xml",
                                               "propagate.xslt")
# Then, re-join the complete function, unit and quantification info
# with the builtobject tree
temporary_file = cexmlhelper.process_with_xslt(temporary_file,
                                               "rejoin.xslt")
# Next, strip out the un-needed languages
temporary_file = cexmlhelper.process_with_xslt(temporary_file,
                                               "language-.xslt",
                                               [("selectedlanguage",
                                                 "\"'%s'\"" %
                                                 selected_language)] )
# Finally, prepare the info for visualisation
temporary_file = cexmlhelper.process_with_xslt(temporary_file,
                                               "builtobject_tree.xslt",
                                               [("selectedname",
                                                 "\"'%s'\"" %
                                                 selected_name),
                                                ("selectedlanguage",
                                                 "\"'%s'\"" %
                                                 selected_language)] )
cexmlhelper.print_xml_file(temporary_file)
cexmlhelper.remove_temporary_files()
```

D.4. print.py

```
#!/usr/bin/python
import cexmlhelper
import os
import sys
cexmlhelper.set_linux_classpath()
# process the first commandline argument (which should be a file)
# with the print_answer.xslt stylesheet
temp=cexmlhelper.process_with_xslt("%s"%sys.argv[1],"print_answer.xslt")
# Call the xml.apache FOP processor on the commandline
os.system("java org.apache.fop.apps.CommandLine %s answer.pdf"% temp)
# print the resulting PDF file
os.system("lpr answer.pdf")
```

Appendix E. XSL/T listings

E.1. language-.xslt

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:html="http://www.w3.org/1999/xhtml"
xmlns:xlink="http://www.w3.org/1999/xlink"
version="1.0"
>
  <xsl:output doctype-system="cexml_message.dtd"
      indent="yes"
      method="xml"
      />
  <!-- This default declaration is needed immediately at the top,
  otherwise the xslt sheet won’t even know this variable exists. The
  true value is passed to this xslt sheet on the command line -->
  <xsl:param name="selectedlanguage" select="’default value’"/>

  <xsl:template match="/">
    <xsl:apply-templates select="cexml"/>
  </xsl:template>

  <xsl:template match="cexml">
    <xsl:element name="cexml">
      <xsl:apply-templates select="builtobject" />
      <xsl:apply-templates select="quantification" />
      <xsl:apply-templates select="function" />
      <xsl:apply-templates select="unit" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="builtobject">
    <xsl:element name="builtobject">
      <xsl:call-template name="copy-attributes"/>
      <xsl:apply-templates select="name" />
      <xsl:apply-templates select="characteristics" />
      <xsl:apply-templates select="builtobject" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="characteristics">
    <xsl:element name="characteristics">
      <xsl:call-template name="copy-attributes"/>
      <xsl:apply-templates select="builtobject" />
      <xsl:apply-templates select="amount" />
      <xsl:apply-templates select="quantification" />
      <xsl:apply-templates select="function" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="name">
    <!-- If a "name" tag is found, remove it, unless it is in the
    desired language OR in the language "si", indicating an S.I. unit
    -->
```

```
    <xsl:if test="@language=$selectedlanguage or @language='si'">
      <xsl:element name="name">
<xsl:call-template name="copy-attributes"/>
<xsl:value-of select="."/>
      </xsl:element>
    </xsl:if>
  </xsl:template>

  <xsl:template match="quantification">
    <xsl:element name="quantification">
      <xsl:call-template name="copy-attributes"/>
      <xsl:apply-templates select="name" />
      <xsl:apply-templates select="amount" />
      <xsl:apply-templates select="unit" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="function">
    <xsl:element name="function">
      <xsl:call-template name="copy-attributes"/>
      <xsl:apply-templates select="name" />
      <xsl:apply-templates select="quantification" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="unit">
    <xsl:element name="unit">
      <xsl:call-template name="copy-attributes"/>
      <xsl:apply-templates select="name" />
    </xsl:element>
  </xsl:template>


  <xsl:template name="copy-attributes">
    <xsl:for-each select="@*">
      <xsl:copy />
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

E.2. propagate.xslt

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:html="http://www.w3.org/1999/xhtml"
xmlns:xlink="http://www.w3.org/1999/xlink"
version="1.0"
>
  <xsl:output doctype-system="cexml_message.dtd"
      indent="yes"
      />

  <!-- The purpose of this XSLT stylesheet is to propagate the of a
  builtobject to all his children. This is by having every element
  copy the characteristics of his parents. The copied characteristics,
```

```
  quantification and function) are marked by setting attribute
  copied="yes". -->

  <xsl:template match="/">
    <xsl:apply-templates select="cexml"/>
  </xsl:template>

  <xsl:template match="cexml">
    <xsl:element name="cexml">
      <xsl:apply-templates select="builtobject" />
      <xsl:apply-templates select="quantification" />
      <xsl:apply-templates select="function" />
      <xsl:apply-templates select="unit" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="builtobject">
    <xsl:element name="builtobject">
      <xsl:call-template name="copy-attributes"/>
      <xsl:apply-templates select="name" />
      <xsl:apply-templates select="characteristics" />
      <xsl:apply-templates select="builtobject" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="characteristics">
    <xsl:element name="characteristics">
      <xsl:call-template name="copy-attributes"/>
      <xsl:apply-templates select="builtobject" />
      <xsl:apply-templates select="amount" />
      <xsl:apply-templates select="quantification" />
      <xsl:if test='../@mode!="search"'>
<!-- add all ancestor's quantifications here -->
<xsl:apply-templates
        select="ancestor::builtobject/ancestor::builtobject/child::characteristics/child::quantification"
        mode="propagate"/>
      </xsl:if>
      <!-- continue normally with the functions -->
      <xsl:apply-templates select="function" />
      <xsl:if test='../@mode!="search"'>
<!-- add all ancestor's functions here -->
<xsl:apply-templates
        select="ancestor::builtobject/ancestor::builtobject/child::characteristics/child::function"
        mode="propagate" />
      </xsl:if>
    </xsl:element>
  </xsl:template>

  <xsl:template match="name">
    <xsl:element name="name">
      <xsl:call-template name="copy-attributes"/>
      <xsl:value-of select="."/>
    </xsl:element>
  </xsl:template>

  <xsl:template match="amount">
    <xsl:element name="amount">
      <xsl:value-of select="."/>
    </xsl:element>
  </xsl:template>
```

```
  <xsl:template match="quantification">
    <xsl:element name="quantification">
      <xsl:call-template name="copy-attributes"/>
      <xsl:apply-templates select="name" />
      <xsl:apply-templates select="amount" />
      <xsl:apply-templates select="unit" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="function">
    <xsl:element name="function">
      <xsl:call-template name="copy-attributes"/>
      <xsl:apply-templates select="name" />
      <xsl:apply-templates select="quantification" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="unit">
    <xsl:element name="unit">
      <xsl:call-template name="copy-attributes"/>
      <xsl:apply-templates select="name" />
    </xsl:element>
  </xsl:template>


  <xsl:template name="copy-attributes">
    <xsl:for-each select="@*">
      <xsl:copy />
    </xsl:for-each>
  </xsl:template>

<!-- TEMPLATES SPECIFIC TO THIS STYLESHEET BELOW THIS LINE -->

  <xsl:template match="quantification" mode="propagate">
    <xsl:element name="quantification">
      <xsl:call-template name="copy-attributes"/>
      <xsl:attribute name="copied">yes</xsl:attribute>
      <xsl:apply-templates select="name" />
      <xsl:apply-templates select="amount" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="function" mode="propagate">
    <xsl:element name="function">
      <xsl:call-template name="copy-attributes"/>
      <xsl:attribute name="copied">yes</xsl:attribute>
      <xsl:apply-templates select="name" />
    </xsl:element>
  </xsl:template>

</xsl:stylesheet>
```

E.3. rejoin.xslt

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```
xmlns:html="http://www.w3.org/1999/xhtml"
xmlns:xlink="http://www.w3.org/1999/xlink"
version="1.0"
>
  <xsl:output doctype-system="cexml_message.dtd"
      indent="yes"
      />

  <xsl:template match="/">
    <xsl:apply-templates select="cexml"/>
  </xsl:template>

  <xsl:template match="cexml">
    <xsl:element name="cexml">
      <xsl:apply-templates select="builtobject" />
      <xsl:apply-templates select="quantification" />
      <xsl:apply-templates select="function" />
      <xsl:apply-templates select="unit" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="builtobject">
    <xsl:element name="builtobject">
      <xsl:call-template name="copy-attributes"/>
      <xsl:choose>
<xsl:when test='self::node()[@mode="view"]'>
  <xsl:param name="target">
    <xsl:value-of select="string(./name)"/>
  </xsl:param>
  <!-- search in the tree for a builtobject with a builtobject
  parent in order *not* to find some builtobject with a
  characteristics parent... Then copy that builobject's names
  -->
  <xsl:apply-templates
  select="//builtobject/builtobject[name=string($target)]"
  mode="snatch" />
</xsl:when>
<xsl:otherwise>
  <xsl:apply-templates select="name" />
</xsl:otherwise>
      </xsl:choose>
      <xsl:apply-templates select="characteristics" />
      <xsl:apply-templates select="builtobject" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="characteristics">
    <xsl:element name="characteristics">
      <xsl:call-template name="copy-attributes"/>
      <!-- all other elements below characteristics need rejoining,
      except "amount" -->
      <xsl:if test='not(../@mode="search")'>
<xsl:apply-templates select="builtobject" mode="rejoin"/>
<xsl:apply-templates select="amount" />
<xsl:apply-templates select="quantification" mode="rejoin" />
<xsl:apply-templates select="function" mode="rejoin" />
      </xsl:if>
      <xsl:if test='../@mode="search"'>
<xsl:apply-templates select="builtobject" />
<xsl:apply-templates select="amount" />
<xsl:apply-templates select="quantification" />
```

```
<xsl:apply-templates select="function" />
      </xsl:if>
    </xsl:element>
  </xsl:template>

  <xsl:template match="name">
    <xsl:element name="name">
      <xsl:call-template name="copy-attributes"/>
      <xsl:value-of select="."/>
    </xsl:element>
  </xsl:template>

  <xsl:template match="amount">
    <xsl:element name="amount">
      <xsl:value-of select="."/>
    </xsl:element>
  </xsl:template>

  <xsl:template match="quantification">
    <xsl:element name="quantification">
      <xsl:call-template name="copy-attributes"/>
      <xsl:apply-templates select="name" />
      <xsl:apply-templates select="amount" />
      <xsl:apply-templates select="unit" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="function">
    <xsl:element name="function">
      <xsl:call-template name="copy-attributes"/>
      <xsl:apply-templates select="name" />
      <xsl:apply-templates select="quantification" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="unit">
    <xsl:element name="unit">
      <xsl:call-template name="copy-attributes"/>
      <xsl:apply-templates select="name" />
    </xsl:element>
  </xsl:template>

  <xsl:template name="copy-attributes">
    <xsl:for-each select="@*">
      <xsl:copy />
    </xsl:for-each>
  </xsl:template>

<!-- BELOW HERE ARE THE SPECIFIC TEMPLATES FOR REJOIN -->

  <xsl:template match="builtobject" mode="rejoin">
    <xsl:element name="builtobject">
      <xsl:call-template name="copy-attributes"/>
      <xsl:param name="target">
<xsl:value-of select="string(./name)"/>
      </xsl:param>
      <!-- search in the tree for a builtobject with a builtobject
      parent in order *not* to find some builtobject with a
      characteristics parent... Then copy that builobject's names -->
      <xsl:apply-templates
      select="//builtobject/builtobject[name=string($target)]"
```

```
      mode="snatch" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="quantification" mode="rejoin">
    <xsl:element name="quantification">
      <xsl:call-template name="copy-attributes"/>
      <xsl:param name="target">
<xsl:value-of select="string(./name)"/>
      </xsl:param>
      <!-- search for a quantification immediately below the cexml
      element with the correct name -->
      <xsl:apply-templates
      select="/cexml/quantification[name=string($target)]"
      mode="snatch" />
      <xsl:apply-templates select="amount" />
      <xsl:apply-templates select="unit" mode="rejoin"/>
    </xsl:element>
  </xsl:template>

  <xsl:template match="function" mode="rejoin">
    <xsl:element name="function">
      <xsl:call-template name="copy-attributes"/>
      <xsl:param name="target">
<xsl:value-of select="string(./name)"/>
      </xsl:param>
      <!-- search for a function immediately below the cexml
      element with the correct name -->
      <xsl:apply-templates
      select="/cexml/function[name=string($target)]"
      mode="snatch" />
      <xsl:apply-templates select="quantification" mode="rejoin" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="unit" mode="rejoin">
    <xsl:element name="unit">
      <xsl:call-template name="copy-attributes"/>
      <xsl:param name="target">
<xsl:value-of select="string(./name[1])"/>
      </xsl:param>
      <!-- search for a unit immediately below the cexml
      element with the correct name -->
      <xsl:apply-templates
      select="/cexml/unit[name=string($target)]"
      mode="snatch" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="quantification" mode="snatch">
    <xsl:apply-templates select="name" />
  </xsl:template>

  <xsl:template match="function" mode="snatch">
    <xsl:apply-templates select="name" />
    <xsl:apply-templates select="quantification" mode="rejoin"/>
  </xsl:template>

  <xsl:template match="*" mode="snatch">
    <xsl:if test='not(self::node()[@mode="view"])'>
      <xsl:apply-templates select="name" />
```

```
    </xsl:if>
  </xsl:template>

</xsl:stylesheet>
```

E.4. search.xslt

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:html="http://www.w3.org/1999/xhtml"
xmlns:xlink="http://www.w3.org/1999/xlink"
version="1.0"
>
  <xsl:output doctype-system="cexml_message.dtd"
      indent="yes"
      />

  <xsl:template match="/">
    <xsl:apply-templates select="cexml"/>
  </xsl:template>

  <xsl:template match="cexml">
    <xsl:element name="cexml">
      <!-- search for every builtobject with a builtobject parent
      (excluding the builtobjects which are characteristics), with the
      extra requirement that they have the attribute mode="search" -->
      <xsl:apply-templates
      select='//builtobject/builtobject[@mode="search"]'  mode="search" />
      <xsl:apply-templates select="quantification" />
      <xsl:apply-templates select="function" />
      <xsl:apply-templates select="unit" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="builtobject">
    <xsl:element name="builtobject">
      <xsl:call-template name="copy-attributes"/>
      <xsl:apply-templates select="name" />
      <xsl:apply-templates select="characteristics" />
      <xsl:apply-templates select="builtobject" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="characteristics">
    <xsl:element name="characteristics">
      <xsl:call-template name="copy-attributes"/>
      <xsl:apply-templates select="builtobject" />
      <xsl:apply-templates select="amount" />
      <xsl:apply-templates select="quantification" />
      <xsl:apply-templates select="function" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="name">
    <xsl:element name="name">
```

```
      <xsl:call-template name="copy-attributes"/>
      <xsl:value-of select="."/>
    </xsl:element>
  </xsl:template>

  <xsl:template match="amount">
    <xsl:element name="amount">
      <xsl:value-of select="."/>
    </xsl:element>
  </xsl:template>

  <xsl:template match="quantification">
    <xsl:element name="quantification">
      <xsl:call-template name="copy-attributes"/>
      <xsl:apply-templates select="name" />
      <xsl:apply-templates select="amount" />
      <xsl:apply-templates select="unit" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="function">
    <xsl:element name="function">
      <xsl:call-template name="copy-attributes"/>
      <xsl:apply-templates select="name" />
      <xsl:apply-templates select="quantification" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="unit">
    <xsl:element name="unit">
      <xsl:call-template name="copy-attributes"/>
      <xsl:apply-templates select="name" />
    </xsl:element>
  </xsl:template>


  <xsl:template name="copy-attributes">
    <xsl:for-each select="@*">
      <xsl:copy />
    </xsl:for-each>
  </xsl:template>

<!-- TEMPLATES SPECIFIC TO THIS STYLESHEET BELOW THIS LINE -->

  <xsl:template match="builtobject" mode="search">
    <!-- I'm a barbarian, but I'm only implementing the search for
    "span" (or "overspanning" in Dutch) at the moment.
    So: search for
    * a builtobject with the correct name
    * with the correct value for quantity "span"
    -->
    <xsl:param name="current_name">
      <xsl:value-of select="./name"/>
    </xsl:param>
    <xsl:param name="current_amount">
      <xsl:value-of
      select='./characteristics/quantification[name="span"]/amount'/>
    </xsl:param>
    <xsl:for-each
    select='//builtobject/builtobject[name=$current_name][@mode!="search"]'>
      <xsl:if
```

```
      test='characteristics/quantification[name="span"]/amount=$current_amount'>
<xsl:element name="builtobject">
  <xsl:call-template name="copy-attributes"/>
  <xsl:attribute name="mode">found</xsl:attribute>
  <xsl:apply-templates select="name" />
  <xsl:apply-templates select="characteristics" />
</xsl:element>
      </xsl:if>
    </xsl:for-each>
  </xsl:template>

</xsl:stylesheet>
```

E.5. print_answer.xslt

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:fo="http://www.w3.org/1999/XSL/Format"
version="1.0"
>

  <xsl:template match="/">
    <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
      <!-- defines page layout -->
      <fo:layout-master-set>
<fo:simple-page-master master-name="normal"
       height="29.7cm"
       width="21cm"
       margin-top="1.0cm"
       margin-bottom="1.0cm"
       margin-left="2.5cm"
       margin-right="2.5cm">
  <fo:region-before extent="1.5cm"/>
  <fo:region-body margin-top="3cm"/>
  <fo:region-after extent="1.5cm"/>
</fo:simple-page-master>
      </fo:layout-master-set>
    <xsl:apply-templates select="cexml"/>
    </fo:root>
  </xsl:template>

  <xsl:template match="cexml">
    <fo:page-sequence master-name="normal">
      <fo:flow>
<fo:block font-size="50pt"
  font-family="sans-serif"
  line-height="50pt"
  space-after.optimum="50pt"
  background-color="gray"
  color="white"
  text-align="center">
  Articles found
</fo:block>
<xsl:apply-templates select="builtobject" />
      </fo:flow>
```

```
    </fo:page-sequence>
  </xsl:template>

  <xsl:template match="builtobject">
    <!-- print the name of the builtobject pretty big -->
    <fo:block font-size="30pt"
      font-family="sans-serif"
      line-height="30pt"
      space-after.optimum="15pt"
      color="gray"
      text-align="left">
      <xsl:value-of select='name[@language="en"]' />
    </fo:block>
    <!-- print the supplier's name -->
    <fo:block font-size="18pt"
      font-family="sans-serif"
      line-height="18pt"
      space-after.optimum="15pt"
      color="black"
      margin-left="1cm"
      text-align="left">
      Supplier: <xsl:value-of select='@supplier' />
    </fo:block>
    <xsl:apply-templates select="characteristics/quantification" />
  </xsl:template>

  <xsl:template match="quantification">
    <!-- print the quantification -->
    <fo:block font-size="18pt"
      font-family="sans-serif"
      line-height="18pt"
      space-after.optimum="15pt"
      color="black"
      margin-left="1cm"
      text-align="left">
      <xsl:value-of select='name[@language="en"]' />=
      <xsl:value-of select='amount' />
      <xsl:value-of select='unit/name[@language="en"]' />
    </fo:block>
  </xsl:template>

</xsl:stylesheet>
```

Appendix F. Document Type Definitions (DTD's)

F.1. cexml_message.dtd

```
<!--  ceXML message model DTD
      Started 2000-08-14 by Reinout van Rees
R.vanRees@ct.tudelft.nl

        This DTD is used to make instances of the message model
        according to the specification found in the report written for
        my graduation project

-->


<!--    ***********************************************************
        The element "name" serves to hold a string indicating a
        certain built object, a unit, a symbol or a
        quantification. Needed for this is a attribute ,
        indicating the language this specific "name" uses. For this,
        you should use xml:lang, especially designed for this
        purpose.
        ***********************************************************
-->

<!--            element_name    rule -->
<!ELEMENT       name            (#PCDATA)>
<!--            target_element  attr_name       attr_type default-->
<!ATTLIST       name            language        NMTOKEN   #IMPLIED >

<!--    ***********************************************************
        A "symbol" is used to represent either a unit or a
        quantification. It is used as a shorthand version.
        e.g. centimeters => cm
        It has possibly one name
        ***********************************************************
-->

<!--            element_name    rule -->
<!ELEMENT       symbol          (name?)>

<!--    ***********************************************************
        A "unit" is a unit of measurement, like centimeters. It has
        at least one name.
        ***********************************************************
-->

<!--            element_name    rule -->
<!ELEMENT       unit            (name*)>


<!--    ***********************************************************
```

```
        An "amount" is a placeholder for a number.
        ************************************************************
-->

<!--            element_name    rule -->
<!ELEMENT       amount          (#PCDATA)>

<!--    ************************************************************
        A "quantification" either quantifies a function or it
        quantifies a built object. So it has a unit and an optional
        amount. Also it can have one or more names.
        ************************************************************
-->

<!--            element_name    rule -->
<!ELEMENT       quantification  (name+,amount*,unit*)>
<!--            target_element  attr_name       attr_type default-->
<!ATTLIST       quantification  copied          CDATA   #IMPLIED >


<!--    ************************************************************
        A "function" identifies a function of a built object. It can
        have one or more names and possibly a few quantifications.
        ************************************************************
-->

<!--            element_name    rule -->
<!ELEMENT       function        (name+,quantification*)>
<!--            target_element  attr_name       attr_type default-->
<!ATTLIST       function        copied          CDATA   #IMPLIED >


<!--    ************************************************************
        "characteristics" are used to group the characteristics of
        built objects (for clarity)
        The characteristics are the subparts (built objects), an
        optional amount, optional quantifications and optional
        functions.
        ************************************************************
-->

<!--            element_name    rule -->
<!ELEMENT       characteristics (builtobject*,
                                amount?,
                                quantification*,
                                function*)>


<!--    ************************************************************
        A "builtobject" is the central building block of the
        vocabulary, indicating the actual built objects. They can
        have other builtobjects in a hierarchy below them and they
        one "characteristics". They also can have names.
        ************************************************************
-->

<!--            element_name    rule -->
<!ELEMENT       builtobject     (name+,
                                characteristics?,
                                builtobject*)>
<!-- the following are attributes that I needed to quickly hack
```

```
together a working prototype. They'd better be solved in a different
way, imo -->
<!--            target_element  attr_name       attr_type default-->
<!ATTLIST       builtobject     copied          CDATA   #IMPLIED >
<!ATTLIST       builtobject     mode            CDATA   #IMPLIED >
<!ATTLIST       builtobject     selectID        CDATA   #IMPLIED >
<!ATTLIST       builtobject     supplier        CDATA   #IMPLIED >


<!--    ************************************************************
        A "cexml" element serves as a container for the major
        elements, so that in *one* xml file, there can be:
        * a tree of builtobjects
        * a list of functions
        * a list of quantifications
        * a list of units
        * a list of symbols
        * a list of names
        ************************************************************
-->

<!--            element_name    rule -->
<!ELEMENT       cexml           (builtobject,
                                (function|
                                quantification|
                                unit|
                                symbol|
                                name)*)>
```

F.2. cexml_en.dtd

```
<!ELEMENT cexml (prefab-concrete-element| slab|
composite-solid-prestressed-soffit-slab| topping|
composite-lattice-girder-soffit-slab| hollow-core-slab|
composite-hollow-core-slab| topping| double-T| composite-double-T| topping|
block| beam| reinforced-rectangular-beam| reinforced-inverted-L-beam|
reinforced-T-beam| column| internal-column| edge-column| corner-column|
other-elements| topping)*>
<!ELEMENT prefab-concrete-element EMPTY>
<!ATTLIST prefab-concrete-element amount CDATA  #REQUIRED >
<!ELEMENT slab (span| width| thickness| loadbearing)*>
<!ATTLIST slab amount CDATA  #REQUIRED >
<!ELEMENT composite-solid-prestressed-soffit-slab (span| width| thickness|
loadbearing)*>
<!ATTLIST composite-solid-prestressed-soffit-slab amount CDATA  #REQUIRED >
<!ELEMENT composite-lattice-girder-soffit-slab (span| width| thickness|
loadbearing)*>
<!ATTLIST composite-lattice-girder-soffit-slab amount CDATA  #REQUIRED >
<!ELEMENT hollow-core-slab (span| width| thickness| loadbearing)*>
<!ATTLIST hollow-core-slab amount CDATA  #REQUIRED >
<!ELEMENT composite-hollow-core-slab (span| width| thickness| loadbearing)*>
<!ATTLIST composite-hollow-core-slab amount CDATA  #REQUIRED >
<!ELEMENT double-T (span| width| thickness| loadbearing)*>
<!ATTLIST double-T amount CDATA  #REQUIRED >
```

```
<!ELEMENT composite-double-T (span| width| thickness| loadbearing)*>
<!ATTLIST composite-double-T amount CDATA  #REQUIRED >
<!ELEMENT block (span| width| thickness| loadbearing)*>
<!ATTLIST block amount CDATA  #REQUIRED >
<!ELEMENT beam (construction-depth| span)*>
<!ATTLIST beam amount CDATA  #REQUIRED >
<!ELEMENT reinforced-rectangular-beam (construction-depth| span)*>
<!ATTLIST reinforced-rectangular-beam amount CDATA  #REQUIRED >
<!ELEMENT reinforced-inverted-L-beam (construction-depth| span)*>
<!ATTLIST reinforced-inverted-L-beam amount CDATA  #REQUIRED >
<!ELEMENT reinforced-T-beam (construction-depth| span)*>
<!ATTLIST reinforced-T-beam amount CDATA  #REQUIRED >
<!ELEMENT column (length| area)*>
<!ATTLIST column amount CDATA  #REQUIRED >
<!ELEMENT internal-column (length| area)*>
<!ATTLIST internal-column amount CDATA  #REQUIRED >
<!ELEMENT edge-column (reinforcement| length| area)*>
<!ATTLIST edge-column amount CDATA  #REQUIRED >
<!ELEMENT corner-column (reinforcement| length| area)*>
<!ATTLIST corner-column amount CDATA  #REQUIRED >
<!ELEMENT other-elements EMPTY>
<!ATTLIST other-elements amount CDATA  #REQUIRED >
<!ELEMENT topping (thickness)*>
<!ATTLIST topping amount CDATA  #REQUIRED >
<!ELEMENT span EMPTY>
<!ATTLIST span m CDATA  #REQUIRED >
<!ELEMENT width EMPTY>
<!ATTLIST width m CDATA  #REQUIRED >
<!ELEMENT thickness EMPTY>
<!ATTLIST thickness m CDATA  #REQUIRED >
<!ELEMENT construction-depth EMPTY>
<!ATTLIST construction-depth m CDATA  #REQUIRED >
<!ELEMENT length EMPTY>
<!ATTLIST length m CDATA  #REQUIRED >
<!ELEMENT area EMPTY>
<!ATTLIST area m_2 CDATA  #REQUIRED >
<!ELEMENT reinforcement EMPTY>
<!ATTLIST reinforcement prcnt CDATA  #REQUIRED >
<!ELEMENT loadbearing EMPTY>
```

F.3. cexml_nl.dtd

```
<!ELEMENT cexml
(geprefabriceerd-betonnen-element|plaat|voorgespannen-vloerplaat-met-
toplaag|toplaag|voorgewapende-vloerplaat|kanaalplaat|kanaalplaat-met-toplaag|toplaag|dubbele-
T|dubbele-T-met-toplaag|toplaag|blok|ligger|voorgespannen-rechthoekige-ligger|voorgespannen-
omgekeerde-L-ligger|voorgespannen-T-ligger|kolom|interne-kolom|wandkolom|hoekkolom|overige-
elementen|toplaag)*>
<!ELEMENT geprefabriceerd-betonnen-element EMPTY>
<!ATTLIST geprefabriceerd-betonnen-element amount CDATA  #REQUIRED >
<!ELEMENT plaat (overspanning|breedte|dikte|dragen-van-belasting)*>
<!ATTLIST plaat amount CDATA  #REQUIRED >
<!ELEMENT voorgespannen-vloerplaat-met-toplaag
(overspanning|breedte|dikte|dragen-van-belasting)*>
<!ATTLIST voorgespannen-vloerplaat-met-toplaag amount CDATA  #REQUIRED >
<!ELEMENT voorgewapende-vloerplaat
```

```
(overspanning|breedte|dikte|dragen-van-belasting)*>
<!ATTLIST voorgewapende-vloerplaat amount CDATA  #REQUIRED >
<!ELEMENT kanaalplaat (overspanning|breedte|dikte|dragen-van-belasting)*>
<!ATTLIST kanaalplaat amount CDATA  #REQUIRED >
<!ELEMENT kanaalplaat-met-toplaag
(overspanning|breedte|dikte|dragen-van-belasting)*>
<!ATTLIST kanaalplaat-met-toplaag amount CDATA  #REQUIRED >
<!ELEMENT dubbele-T (overspanning|breedte|dikte|dragen-van-belasting)*>
<!ATTLIST dubbele-T amount CDATA  #REQUIRED >
<!ELEMENT dubbele-T-met-toplaag
(overspanning|breedte|dikte|dragen-van-belasting)*>
<!ATTLIST dubbele-T-met-toplaag amount CDATA  #REQUIRED >
<!ELEMENT blok (overspanning|breedte|dikte|dragen-van-belasting)*>
<!ATTLIST blok amount CDATA  #REQUIRED >
<!ELEMENT ligger (constructiehoogte|overspanning)*>
<!ATTLIST ligger amount CDATA  #REQUIRED >
<!ELEMENT voorgespannen-rechthoekige-ligger (constructiehoogte|overspanning)*>
<!ATTLIST voorgespannen-rechthoekige-ligger amount CDATA  #REQUIRED >
<!ELEMENT voorgespannen-omgekeerde-L-ligger (constructiehoogte|overspanning)*>
<!ATTLIST voorgespannen-omgekeerde-L-ligger amount CDATA  #REQUIRED >
<!ELEMENT voorgespannen-T-ligger (constructiehoogte|overspanning)*>
<!ATTLIST voorgespannen-T-ligger amount CDATA  #REQUIRED >
<!ELEMENT kolom (lengte|oppervlakte)*>
<!ATTLIST kolom amount CDATA  #REQUIRED >
<!ELEMENT interne-kolom (lengte|oppervlakte)*>
<!ATTLIST interne-kolom amount CDATA  #REQUIRED >
<!ELEMENT wandkolom (wapening|lengte|oppervlakte)*>
<!ATTLIST wandkolom amount CDATA  #REQUIRED >
<!ELEMENT hoekkolom (wapening|lengte|oppervlakte)*>
<!ATTLIST hoekkolom amount CDATA  #REQUIRED >
<!ELEMENT overige-elementen EMPTY>
<!ATTLIST overige-elementen amount CDATA  #REQUIRED >
<!ELEMENT toplaag (dikte)*>
<!ATTLIST toplaag amount CDATA  #REQUIRED >
<!ELEMENT overspanning EMPTY>
<!ATTLIST overspanning m CDATA  #REQUIRED >
<!ELEMENT breedte EMPTY>
<!ATTLIST breedte m CDATA  #REQUIRED >
<!ELEMENT dikte EMPTY>
<!ATTLIST dikte m CDATA  #REQUIRED >
<!ELEMENT constructiehoogte EMPTY>
<!ATTLIST constructiehoogte m CDATA  #REQUIRED >
<!ELEMENT lengte EMPTY>
<!ATTLIST lengte m CDATA  #REQUIRED >
<!ELEMENT oppervlakte EMPTY>
<!ATTLIST oppervlakte m_2 CDATA  #REQUIRED >
<!ELEMENT wapening EMPTY>
<!ATTLIST wapening prcnt CDATA  #REQUIRED >
<!ELEMENT dragen-van-belasting EMPTY>
```

F.4. pce_en_procurement.dtd

```
<!ELEMENT envelope (sender,recipient,businessprocess,message)>
<!ELEMENT sender (address)>
<!ELEMENT recipient (address)>
<!ELEMENT address (#PCDATA)>
```

```
<!ELEMENT message (cexml)>
<!ELEMENT cexml (prefab-concrete-element|slab|
composite-solid-prestressed-soffit-slab|topping|
composite-lattice-girder-soffit-slab|hollow-core-slab|
composite-hollow-core-slab|topping|double-T|composite-double-T|topping|
block|beam|reinforced-rectangular-beam|reinforced-inverted-L-beam|
reinforced-T-beam|column|internal-column|edge-column|corner-column|
other-elements|topping)*>
<!ELEMENT prefab-concrete-element EMPTY>
<!ATTLIST prefab-concrete-element amount CDATA  #REQUIRED >
<!ELEMENT slab (span|width|thickness|loadbearing)*>
<!ATTLIST slab amount CDATA  #REQUIRED >
<!ELEMENT composite-solid-prestressed-soffit-slab (span|width|thickness|loadbearing)*>
<!ATTLIST composite-solid-prestressed-soffit-slab amount CDATA  #REQUIRED >
<!ELEMENT composite-lattice-girder-soffit-slab (span|width|thickness|loadbearing)*>
<!ATTLIST composite-lattice-girder-soffit-slab amount CDATA  #REQUIRED >
<!ELEMENT hollow-core-slab (span|width|thickness|loadbearing)*>
<!ATTLIST hollow-core-slab amount CDATA  #REQUIRED >
<!ELEMENT composite-hollow-core-slab (span|width|thickness|loadbearing)*>
<!ATTLIST composite-hollow-core-slab amount CDATA  #REQUIRED >
<!ELEMENT double-T (span|width|thickness|loadbearing)*>
<!ATTLIST double-T amount CDATA  #REQUIRED >
<!ELEMENT composite-double-T (span|width|thickness|loadbearing)*>
<!ATTLIST composite-double-T amount CDATA  #REQUIRED >
<!ELEMENT block (span|width|thickness|loadbearing)*>
<!ATTLIST block amount CDATA  #REQUIRED >
<!ELEMENT beam (construction-depth|span)*>
<!ATTLIST beam amount CDATA  #REQUIRED >
<!ELEMENT reinforced-rectangular-beam (construction-depth|span)*>
<!ATTLIST reinforced-rectangular-beam amount CDATA  #REQUIRED >
<!ELEMENT reinforced-inverted-L-beam (construction-depth|span)*>
<!ATTLIST reinforced-inverted-L-beam amount CDATA  #REQUIRED >
<!ELEMENT reinforced-T-beam (construction-depth|span)*>
<!ATTLIST reinforced-T-beam amount CDATA  #REQUIRED >
<!ELEMENT column (length|area)*>
<!ATTLIST column amount CDATA  #REQUIRED >
<!ELEMENT internal-column (length|area)*>
<!ATTLIST internal-column amount CDATA  #REQUIRED >
<!ELEMENT edge-column (reinforcement|length|area)*>
<!ATTLIST edge-column amount CDATA  #REQUIRED >
<!ELEMENT corner-column (reinforcement|length|area)*>
<!ATTLIST corner-column amount CDATA  #REQUIRED >
<!ELEMENT other-elements EMPTY>
<!ATTLIST other-elements amount CDATA  #REQUIRED >
<!ELEMENT topping (thickness)*>
<!ATTLIST topping amount CDATA  #REQUIRED >
<!ELEMENT span EMPTY>
<!ATTLIST span m CDATA  #REQUIRED >
<!ELEMENT width EMPTY>
<!ATTLIST width m CDATA  #REQUIRED >
<!ELEMENT thickness EMPTY>
<!ATTLIST thickness m CDATA  #REQUIRED >
<!ELEMENT construction-depth EMPTY>
<!ATTLIST construction-depth m CDATA  #REQUIRED >
<!ELEMENT length EMPTY>
<!ATTLIST length m CDATA  #REQUIRED >
<!ELEMENT area EMPTY>
<!ATTLIST area m_2 CDATA  #REQUIRED >
<!ELEMENT reinforcement EMPTY>
<!ATTLIST reinforcement prcnt CDATA  #REQUIRED >
<!ELEMENT loadbearing EMPTY>
```

```
<!ELEMENT businessprocess (bpname, bpstep+)>
<!ELEMENT bpname (#PCDATA)>
<!ELEMENT bpstep (#PCDATA)>
```

Appendix G. XML files

G.1. cexml.xml

This is the central vocabulary, containing all the information about prefab concrete elements, their characteristics, units, etcetera in both English and Dutch.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE cexml SYSTEM "cexml_message.dtd">
<cexml xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:html="http://www.w3.org/1999/xhtml">
  <builtobject>
    <name language="en">built-object</name>
    <name language="nl">gebouwd-object</name>
    <characteristics/>
    <builtobject>
      <name language="en">prefab-concrete-element</name>
      <name language="nl">geprefabriceerd-betonnen-element</name>
      <characteristics/>
      <builtobject>
<name language="en">slab</name>
<name language="nl">plaat</name>
<characteristics>
  <quantification>
    <name language="en">span</name>
  </quantification>
  <quantification>
    <name language="en">width</name>
  </quantification>
  <quantification>
    <name language="en">thickness</name>
  </quantification>
  <function>
    <name language="en">loadbearing</name>
    <name language="nl">dragen-van-belasting</name>
  </function>
</characteristics>
<builtobject>
  <name language="en">composite-solid-prestressed-soffit-slab</name>
  <name language="nl">voorgespannen-vloerplaat-met-toplaag</name>
  <characteristics>
    <builtobject>
      <name language="nl">toplaag</name>
      <name language="en">topping</name>
    </builtobject>
  </characteristics>
</builtobject>
<builtobject>
  <name language="en">composite-lattice-girder-soffit-slab</name>
  <name language="nl">voorgewapende-vloerplaat</name>
  <characteristics/>
</builtobject>
<builtobject>
  <name language="en">hollow-core-slab</name>
```

```
    <name language="nl">kanaalplaat</name>
    <characteristics/>
    <builtobject>
      <name language="en">composite-hollow-core-slab</name>
      <name language="nl">kanaalplaat-met-toplaag</name>
      <characteristics>
        <builtobject>
  <name language="nl">toplaag</name>
  <name language="en">topping</name>
        </builtobject>
      </characteristics>
    </builtobject>
  </builtobject>
  <builtobject>
    <name language="en">double-T</name>
    <name language="nl">dubbele-T</name>
    <characteristics/>
    <builtobject>
      <name language="en">composite-double-T</name>
      <name language="nl">dubbele-T-met-toplaag</name>
      <characteristics>
        <builtobject>
  <name language="nl">toplaag</name>
  <name language="en">topping</name>
        </builtobject>
      </characteristics>
    </builtobject>
  </builtobject>
  <builtobject>
    <name language="en">block</name>
    <name language="nl">blok</name>
    <characteristics/>
  </builtobject>
        </builtobject>
        <builtobject>
  <name language="en">beam</name>
  <name language="nl">ligger</name>
  <characteristics>
    <quantification>
      <name language="en">construction-depth</name>
    </quantification>
    <quantification>
      <name language="en">span</name>
    </quantification>
  </characteristics>
  <builtobject>
    <name language="en">reinforced-rectangular-beam</name>
    <name language="nl">voorgespannen-rechthoekige-ligger</name>
    <characteristics/>
  </builtobject>
  <builtobject>
    <name language="en">reinforced-inverted-L-beam</name>
    <name language="nl">voorgespannen-omgekeerde-L-ligger</name>
    <characteristics/>
  </builtobject>
  <builtobject>
    <name language="en">reinforced-T-beam</name>
    <name language="nl">voorgespannen-T-ligger</name>
    <characteristics/>
  </builtobject>
        </builtobject>
```

```
      <builtobject>
<name language="en">column</name>
<name language="nl">kolom</name>
<characteristics>
  <quantification>
    <name language="en">length</name>
  </quantification>
  <quantification>
    <name language="en">area</name>
  </quantification>
</characteristics>
<builtobject>
  <name language="en">internal-column</name>
  <name language="nl">interne-kolom</name>
  <characteristics/>
</builtobject>
<builtobject>
  <name language="en">edge-column</name>
  <name language="nl">wandkolom</name>
  <characteristics>
    <quantification>
      <name language="en">reinforcement</name>
    </quantification>
  </characteristics>
</builtobject>
<builtobject>
  <name language="en">corner-column</name>
  <name language="nl">hoekkolom</name>
  <characteristics>
    <quantification>
      <name language="en">reinforcement</name>
    </quantification>
  </characteristics>
</builtobject>
      </builtobject>
    </builtobject>
    <builtobject>
      <name language="en">other-elements</name>
      <name language="nl">overige-elementen</name>
      <characteristics/>
      <builtobject>
<name language="nl">toplaag</name>
<name language="en">topping</name>
<characteristics>
  <quantification>
    <name language="en">thickness</name>
  </quantification>
</characteristics>
      </builtobject>
    </builtobject>
  </builtobject>
  <quantification>
    <name language="en">span</name>
    <name language="nl">overspanning</name>
    <amount/>
    <unit>
      <name language="si">m</name>
    </unit>
  </quantification>
  <quantification>
    <name language="en">width</name>
```

```
  <name language="nl">breedte</name>
  <amount/>
  <unit>
    <name language="si">m</name>
  </unit>
</quantification>
<quantification>
  <name language="en">thickness</name>
  <name language="nl">dikte</name>
  <amount/>
  <unit>
    <name language="si">m</name>
  </unit>
</quantification>
<quantification>
  <name language="en">construction-depth</name>
  <name language="nl">constructiehoogte</name>
  <amount/>
  <unit>
    <name language="si">m</name>
  </unit>
</quantification>
<quantification>
  <name language="en">length</name>
  <name language="nl">lengte</name>
  <amount/>
  <unit>
    <name language="si">m</name>
  </unit>
</quantification>
<quantification>
  <name language="en">area</name>
  <name language="nl">oppervlakte</name>
  <amount/>
  <unit>
    <name language="si">m_2</name>
  </unit>
</quantification>
<quantification>
  <name language="en">reinforcement</name>
  <name language="nl">wapening</name>
  <amount/>
  <unit>
    <name language="si">prcnt</name>
  </unit>
</quantification>
<function>
  <name language="en">loadbearing</name>
  <name language="nl">dragen-van-belasting</name>
</function>
<unit>
  <name language="si">m</name>
  <name language="en">meter</name>
  <name language="nl">meter</name>
</unit>
<unit>
  <name language="si">m_2</name>
  <name language="en">square-meter</name>
  <name language="nl">vierkante-meter</name>
</unit>
<unit>
```

```
    <name language="si">prcnt</name>
    <name language="en">percent</name>
    <name language="nl">procent</name>
  </unit>
</cexml>
```

G.2. catalog.xml

This XML file is the Dutch catalog, which complies to a DTD generated out of the central vocabulary using the contexts "Dutch language" and the field "prefab concrete elements".

```
<!DOCTYPE cexml SYSTEM "cexml_nl.dtd">
<cexml>
  <kanaalplaat amount="">
    <overspanning m="5"/>
    <dikte m="0.110"/>
  </kanaalplaat>
  <kanaalplaat amount="">
    <overspanning m="5"/>
    <dikte m="0.150"/>
  </kanaalplaat>
  <kanaalplaat amount="">
    <overspanning m="6"/>
    <dikte m="0.150"/>
  </kanaalplaat>
  <kanaalplaat amount="">
    <overspanning m="6"/>
    <dikte m="0.200"/>
  </kanaalplaat>
  <kanaalplaat amount="">
    <overspanning m="7"/>
    <dikte m="0.150"/>
  </kanaalplaat>
  <kanaalplaat amount="">
    <overspanning m="7"/>
    <dikte m="0.200"/>
  </kanaalplaat>
  <kanaalplaat amount="">
    <overspanning m="7"/>
    <dikte m="0.220"/>
  </kanaalplaat>
  <kanaalplaat amount="">
    <overspanning m="7"/>
    <dikte m="0.250"/>
  </kanaalplaat>
  <voorgespannen-rechthoekige-ligger amount="">
    <overspanning m="4"/>
    <constructiehoogte m="0.252"/>
  </voorgespannen-rechthoekige-ligger>
  <voorgespannen-rechthoekige-ligger amount="">
    <overspanning m="5"/>
    <constructiehoogte m="0.288"/>
  </voorgespannen-rechthoekige-ligger>
```

```
  <voorgespannen-rechthoekige-ligger amount="">
    <overspanning m="6"/>
    <constructiehoogte m="0.332"/>
  </voorgespannen-rechthoekige-ligger>
  <voorgespannen-rechthoekige-ligger amount="">
    <overspanning m="7"/>
    <constructiehoogte m="0.386"/>
  </voorgespannen-rechthoekige-ligger>
  <voorgespannen-T-ligger amount="">
    <overspanning m="4"/>
    <constructiehoogte m="0.246"/>
  </voorgespannen-T-ligger>
  <voorgespannen-T-ligger amount="">
    <overspanning m="5"/>
    <constructiehoogte m="0.286"/>
  </voorgespannen-T-ligger>
  <voorgespannen-T-ligger amount="">
    <overspanning m="6"/>
    <constructiehoogte m="0.334"/>
  </voorgespannen-T-ligger>
  <voorgespannen-T-ligger amount="">
    <overspanning m="7"/>
    <constructiehoogte m="0.384"/>
  </voorgespannen-T-ligger>
</cexml>
```

Bibliography

Raman, Dick. *XML/EDI Cyber assisted business in practice*TIE Holding 1999. ISBN 90-805233-2-1

Radeke, dr. Elke. *GEN global engineering networking final report* Siemens C-LAB 1999

McLaughlin, Brett. *Java and XML* O'Reilly 2000. ISBN 0-596-00016-2

Eckstein, Robert. *XML pocket reference* O'Reilly 1999. ISBN 1-56592-709-5

Lutz, Mark. *Python pocket reference* O'Reilly 1998. ISBN 1-56592-500-9

Walsh, Norman & Muellner, Leonard. *DocBook, the definitive guide* O'Reilly 1999. ISBN 1-56592-580-7

Goodchild, C.H. *Economic concrete frame elements* British Cement Association 1997. ISBN 0-7210-1488-7

www.ebxml.org

www.w3.org

www.eConstruct.org

www.xml.com

www.xml.org

www.xmlhack.com

Glossary

Apache

The most widely used web server. A web server is the remote program that gives you an Internet page when you click a link in your browser.

ceXML

The name chosen for my vocabulary. It stands for *Civil Engineering XML*.

CSS

Cascading Style Sheets. A way to specify how things should look when an Internet page is displayed in your browser.

DTD

Document Type Definition. A separate file that specifies the rules to which an XML file must adhere.

eConstruct

Electronic business in the building and CONSTRUCTion industry: preparing for the new internet. The European project I'm enlisted in. It's aim is to make electronic communication a reality in the still mainly paper based construction industry.

ebXML

Electronic Business XML. A joint initiative of all big players in the ecommerce business (only Microsoft is missing) to provide an open platform for electronic business.

EDI

Electronic Data Interchange. The exchange of information using article numbers and codes by big, monolytic industries.

GEN/GENial

Global Engineering Networking. An effort at creating a network where engineers can get the information they need and where suppliers can provide the information needed by engineers. GENial is an implementation of GEN.

IFC

Industry Foundation Classes. A model for the building and construction industry, meant for the electronic representation of things that occur in a constructed facility, both physical and abstract (like services).

LexiCon

The LexiCon is a set of programs coupled with it's own meta-model which is built by the Dutch company STABU in order to create a information system covering the entire building process.

Mozilla

An Internet browser. Descendant from the well-known Netscape. Is still in beta, but supports most of the newest standards and tries actively to be as standards-compliant as possible.

PDT

Product Data Technology. A generic term for models that deal with product data (like doors or pencils).

Python

A programming language. Member of the family of so-called scripting languages. Especially good at rapid prototyping and for gluing together various bits and pieces of other programs.

Sed

A small command line program that can modify text files according to command-line parameters.

STEP

STandard for the Exchange of Product model data. This constitutes a standard way of dealing with product data. Mainly used in big companies in the automobile, shipbuilding, oil drilling and airplane manufacturing business.

tag

A tag is - in the XML world - a nametag that is connected to a piece of information. If you want to indicate that the word "this" should be in italics, you would - in XML - place a starting-nametag at the beginning of the word "this" and a closing-nametag at the end. In XML the nametag is enclosed in less-than and greater-than signs. It works out like this: `<italic>this</italic>`.

UML

Unified Modelling Language. A standard way of drawing diagrams, Use Cases, etcetera.

Use Case

In the Unified Modelling Language (UML), a Use Case is a concept, a way of thinking about a problem. The problem is described (or better:depicted) by means of *actions* between *actors*.

XML

eXtensible Markup Language, a text format which is able to tag parts of the data contained within (as text) with meta-information. An example: <italic>this ought to be in italics</italic>.

XSL

eXtensible Stylesheet Language. It consists of two parts, XSL/FO for formatting and XSL/T for transformation. They are designed to work together, but XSL/T has proven very useful as a stand-alone technology, thereby taking on a life of it's own.

XSL/FO

eXtensible Stylesheet Language/Formatting Objects part. Like CSS, this is a language used to specify the visual representation of XML. The difference is that XSL/FO is also used to format an XML directly for printing.

XSL/T

eXtensible Stylesheet Language/Transformation part. A language to specify rules with which to transform an incoming XML file into another format (probably also an XML format).

Colophon

This document was made using DocBook. DocBook is an XML DTD for making structured documents. Practice what you preach, so to say. Editing was done with the emacs editor, which has support for working with XML files. Diagrams were made with Dia, still very much in beta, but freeware (as opposed to the horribly expensive Rational Rose and the likes of it). Images were made with Gimp. All editing, programming and diagram drawing was done on Linux, unbeatable for this kind of work, especially for the server-side Internet programming.