# Annotated text databases in the context of the Kaj Munk corpus

*One database model, one query language, and several applications*

Sandborg-Petersen, Ulrik

*Publication date:*
2008

*Document Version*
Publisher's PDF, also known as Version of record

# Annotated text databases in the context of the Kaj Munk corpus: One database model, one query language, and several applications

Ulrik Sandborg-Petersen

May 5, 2008

PhD thesis

Submitted in partial fulfilment
of the requirements for
the ph.d. degree.

Doctoral School in
Human Centered Informatics,
Aalborg University,
Denmark


To my wife and my daughter

# Contents

# Acknowledgements

"No man is an island," says John Donne. Likewise, no research is carried out in a vacuum. I therefore have many people to thank for their part in the successful completion of this dissertation. First and foremost, my PhD supervisor, Professor, dr.scient, ph.d. Peter Øhrstrøm, whose continuing support since 1996 it has been my privilege to enjoy. Peter Øhrstrøm embodies in his person the ideal concept of "the gracious Professor," whose support and sound advice has been invaluable for me through the years, and especially during my PhD studies. I could not have been privileged with a better supervisor.

Another person whose support I have enjoyed since 1996 is Associate Professor, teol.dr. Nicolai Winther-Nielsen. Dr. Winther-Nielsen changed the course of my life when, in 1996, he gave me a copy of Crist-Jan Doedens' PhD dissertation [Doedens, 1994], with a side-remark that I might want to implement the contents of that book some day. Little did I know that I would, in fact, do as he suggested, and as a result visit three countries, gain many friends from all over the world, and be able to do a PhD, all as a consequence of his gifting me with this book.

I thank Bob Pritchett, CEO of Logos Research Systems for hosting me at a fruitful visit in their offices in the USA during my PhD studies. I also thank Professor, Dr. Eep Talstra for hosting me at two research visits in his research group, "Werkgroep Informatica", during my PhD studies.

Professor Søren Dosenrode believed in me and was instrumental in obtaining funds for my PhD studies, for which I am grateful. My colleagues at the Kaj Munk Research Centre, cand.scient. Jørgen Albretsen and lic.iur. Andrea Dosenrode, have been pleasant colleagues whose humor and wit I have enjoyed. Another colleague at the Kaj Munk Research Centre, cand.mag. Jesper Valeur Mogensen, whose outstanding literary and linguistic skills have helped me in many practical aspects of my work, has also been a great inspiration in my research. Professor, Dr. Adil Kabbaj has provided inspiration for my work several times, and has written both the Prolog+CG software and the Amine Platform software, whose program code I have had the pleasure of having a helping hand in maintaining and extending over the past four years. Professor Kabbaj has furthermore taught me on a number of occasions about the nature of research in genereal, and research in knowledge representation in particular, for which I am grateful. I also wish to thank Associate Professor, Dr. Kirk E. Lowery for fruitful discussions about technical issues.

Although we have neither met, nor corresponded, I wish to thank Søren Hornstrup for making the texts of the Danish Old Testament from 1931 and the Danish New Testament from 1907 available on the Internet back in the early 1990'ies. I also wish to thank Sigve Bø of "Sigve Saker" in Norway (www.sigvesaker.no) for sending me his proofread, corrected version of the Danish Bible from 1931/1907. I thank Ole Madsen for pleasant interactions concerning even older versions of the Bible in Danish.

I thank Associate Professor, dr.phil. Eckhard Bick for help with an early part-of-

speech tagging and lemmatization of the Munk Corpus. I thank Center for Language Technology (CST) at the University of Copenhagen for letting me experiment with and use their part-of-speech tagger and lemmatizer.

Last, but most certainly not least, I wish to thank my wife, Lone Margrethe, for her love and support. Life is truly *abarês* with you.

# List of published articles

The articles referenced below are part of this PhD dissertation, and can be found appended to the main body of the dissertation. The names in [square brackets], for example, [RANLP2005], are used throughout the dissertation whenever referring to one of the articles. All of the articles are published, and have been peer-reviewed internationally.

**[COLING2004]** Petersen, Ulrik. (2004) *Emdros — a text database engine for analyzed or annotated text*, In Proceedings of COLING 2004, held August 23–27 in Geneva. International Commitee on Computational Linguistics, pp. 1190–1193.

**[RANLP2005]** Petersen,Ulrik. (2005) *Evaluating corpus query systems on functionality and speed: TIGERSearch and Emdros.* In: Angelova, G., Bontcheva, K., Mitkov, R., Nicolov, N. and Nikolov, N. (eds): International Conference Recent Advances in Natural Language Processing 2005, Proceedings, Borovets, Bulgaria, 21–23 September 2005, pp. 387–391.

**[FSMNLP2005]** Petersen, Ulrik. (2006a) *Principles, Implementation Strategies, and Evaluation of a Corpus Query System.* In: Yli-Jyrä, Anssi; Karttunen, Lauri; Karhumäki, Juhani (eds), *Finite-State Methods and Natural Language Processing 5th International Workshop, FSMNLP 2005, Helsinki, Finland, September 1–2, 2005, Revised Papers*, Lecture Notes in Computer Science, Volume 4002/2006, Springer Verlag, Heidelberg, New York, pp. 215–226. DOI: 10.1007/11780885_21.

**[LREC2006]** Petersen, Ulrik. (2006b) *Querying both Parallel and Treebank Corpora: Evaluation of a Corpus Query System.* In: Proceedings of International Language Resources and Evaluation Conference, LREC 2006.

**[CS-TIW2006]** Petersen, Ulrik. (2006c) *Prolog+CG: A Maintainer's Perspective*. In: de Moor, Aldo, Polovina, Simon and Delugach, Harry (eds.): *First Conceptual Structures Interoperability Workshop (CS-TIW 2006). Proceedings*. Aalborg University Press, pp. 58–71.

**[CS-TIW2007]** Petersen, Ulrik (2007) *Using interoperating conceptual tools to improve searches in Kaj Munk.* In: Pfeiffer, Heather D., Kabbaj, Adil and Benn, David (eds.): *Second Conceptual Structures Tool Interoperability Workshop (CS-TIW 2007). Held on July 22, 2007 in Sheffield, UK, in conjunction with International Conference on Conceptual Structures (ICCS) 2007*. Research Press International, Bristol, UK. ISBN: 1-897851-16-2, pp. 45–55.

**[ICCS-Suppl2008]** Sandborg-Petersen, Ulrik (2008) *An FCA classification of durations of time for textual databases.* In: Eklund, Peter and Haemmerlé, Olivier (eds): *Supplementary Proceedings of ICCS 2008*, CEUR-WS.

The article labelled [COLING2004] was written before my PhD studies commenced. It is part of this thesis because it is an important introduction to my work. Even though it was written during the last months of my MA studies, it did not form part of the basis for evaluation of my MA work. The ideas presented in [COLING2004] are foundational for large portions of the theoretical as well as practical aspects my work, and so the article finds a natural place among the other articles presented here.

# Resumé

Det centrale tema for denne ph.d.-afhandling er “annoterede tekstdatabaser”. En annoteret tekstdatabase er en samling tekst plus information on teksten, som er lagret i et computer system med henblik på nem opdatering og tilgang. “Informationen om teksten” er annoteringerne af teksten.

Mit ph.d.-arbejde er blevet udført under den organisatoriske paraply, som udgøres af Kaj Munk Forskingscentret ved Aalborg Universitet. Kaj Munk (1898–1944) var en både indflydelsesrig og flittig dramatiker, journalist, pastor og poet, hvis indflydelse mærkedes både i og udenfor Danmark i mellemkrigstiden. Han blev myrdet af Gestapo først i januar 1944 på grund af hans modstandsholdninger.

De to af Kaj Munk Forskningscentrets hovedaktiviteter, som jeg har været involveret i igennem mit ph.d.-arbejde er følgende: a) At digitalisere Kaj Munks samlede værker, og b) At gøre Kaj Munks samlede tekstproduktion tilgængelig i elektronisk form for den brede befolkning. Min afhandling afspejler disse aktiviteter ved at tage Kaj Munks værker, og gøre dem til det empiriske grundlag, det empiriske datamateriale, hvorpå min afhandlings teoretiske fremskridt er blevet testet.

Mit ph.d.-arbejde har ikke omhandlet Kaj Munks værker set fra et historisk eller litterært perspektiv. Nej, mit arbejde har været udført set med en datalogs briller, og med det formål at repræsentere annoterede udgaver af Kaj Munks værker i et computerdatabasesystem, samt med det formål at opnå mulighed for nem adgang til disse annoterede udgaver ved hjælp af et søgesprog. Derfor har det faktum, at det empiriske grundlag har været Kaj Munks samlede værker, været, i alt det for resultaterne væsentlige, ligegyldigt. Med andre ord kan resultaterne — de udkrystalliserede teorier, de opnåede metoder, og det implementerede system — nu anvendes på andre annoterede korpora, uafhængigt af deres ophav med Kaj Munks værker som det empiriske grundlag.

De teoretiske fremskridt, som jeg har opnået i løbet af min ph.d., bygger naturligvis på en række andre personers arbejde. Det primære udgangspunkt har været Dr. Crist-Jan Doedens’s arbejde, som han udførte i sin PhD-afhandling fra 1994: “Text Databases — One Database Model and Several Retrieval Languages”, Universitetet i Utrecht, Holland. I sin PhD-afhandling beskrev Dr. Doedens dels sin “Monads dot Features” (MdF) model for annoterede tekstdatabaser, dels sit “QL” søgesprog, defineret over MdF databaser.

I mit arbejde har jeg taget MdF tekstdatabasemodellen, og har både udvidet og reduceret dens omfang på forskellige områder. Dermed er jeg nået frem til den udvidede MdF model, eller “EMdF modellen” (på engelsk: “Extended MdF model”). Jeg har også taget Doedens’s QL søgesprog, og har dels reduceret en del af det til et lidt mindre del-sprog, dels udvidet QL kraftigt til at blive et “sprog med fuld tilgang” til EMdF databaser. Jeg kalder min udvidelse af QL for “MQL”.

EMdF modellen er i stand til at udtrykke næsten enhver form for annotering, der måtte kræves, for at repræsentere lingvistiske annoteringer af tekst. Som jeg viser i Kapitel 10,

så er det i hvert fald tilfældet, at alle de former for annoteringer, som vi i Kaj Munk Forskningscentret har ønsket at tilføre Kaj Munk Korpusset, kan utrykkes i EMdF modellen.

MQL søgesproget er som sagt et "sprog med fuld tilgang" til EMdF databaser, idet MQL understøtter de fire hovedoperationer på databaser: "opret", "hent", "opdater" og "slet" (på engelsk: "create", "retrieve", "update", and "delete"). Dette står i kontrast til QL, som "kun" var et sprog, hvori man kunne "hente" (på engelsk: "retrieve") visse dele af MdF modellens datadomæner.

Jeg har implementeret EMdF modellen og MQL søgesproget i mit "Emdros" korpussøgesystem. Dermed har jeg opfyldt de fleste af de krav, som Doedens stillede til et annoteret tekstdatabasesystem, hvilket jeg viser i Kapitlerne 4, 5 og 14.

Afhandlingens første del, Part I, omhandler "teoretiske fundamenter" og indeholder Kapitlerne 1 til og med 8.

Kapitel 1 introducerer afhandlingens emner, og definerer nogle af de hovedbegreber, som anvendes i afhandlingen. Kapitel 2 giver et overblik over den vigtigste litteratur indenfor afhandlingens emneområde. Kapitel 3 giver en kort introduktion til emnet "ontologi", til brug i senere kapitler. Kapitel 4 introducerer og diskuterer EMdF modellen, mens Kapitel 5 gør det samme for MQL søgesproget. Kapitel 6 introducerer og diskuterer "Neget" (på engelsk: "The Sheaf"), som er en af de datastrukturer, som en MQL søgning kan returnere. Kapitel 7 beskriver en generel algoritme for, og en klassificering af mulige strategier for, at "høste" et "Neg", det vil sige, at aflede meningsfulde resultater af et "Neg". Kapitel 8 diskuterer nogle mulige relationer imellem annotered tekst (som den kan udtrykkes i og med EMdF modellen) på den ene side, og tid på den anden siden.

Afhandlingens anden del, Part II, omhandler "Anvendelser" og indeholder Kapitlerne 9 til og med 13.

Kapitel 9 introducerer Part II. Kapitel 10 beskriver hvordan de teoretiske fundamenter lagt i Part I kan blive anvendt til at implementere Kaj Munk Korpusset i EMdF modellen. Kapitel 11 beskriver principperne bag, og funktionaliteten af, et web-baseret værktøj, som jeg har skrevet med brug af Emdros. Værktøjet har til formål at understøtte en kollaborativ annotering af Kaj Munk Korpusset, men kunne i princippet anvendes på et hvilket som helst andet korpus. Kapitel 12 er det centrale kapitel i Part II, i hvilket jeg viser, hvordan både EMdF modellen og MQL søgesproget og "høstningsalgoritmen", som alle blev udviklet i Part I, kan anvendes på de problemer, der indebæres i målet, at gøre Kaj Munks værker tilgængelige i elektronisk form for den brede befolkning. Kapitel 13 diskuterer måder, hvorpå EMdF modellen og MQL søgesproget kan anvendes til at understøttet processen med, automatisk at lokalisere citater fra ét korpus i et andet korpus. I dette tilfælde er problemet, at finde citater fra Bibelen i Kaj Munk Korpusset.

Afhandlingens tredie del, Part III, er meget kort, og indeholder kun to kapitler. Den omhandler "Perspektiver" på mit arbejde.

I Kapitel 14 diskuterer jeg måder, hvorpå EMdF modellen og MQL søgesproget kan blive udvidet, så de endnu bedre kan understøtte problemerne indebåret i at lagre og fremhente annoteret tekst. Kapitel 15 afrunder afhandlingen.

Appendix A giver den fulde grammatik for den delmængde af MQL, som svarer til Doedens's QL.

Syv allerede publicerede, internationalt peer-reviewede artikler er vedlagt afhandlingen i Appendix B, og udgør en del af bedømmelsesgrundlaget for afhandlingen..

# Abstract

The central theme of this PhD dissertation is "annotated text databases". An annotated text database is a collection of text plus information about that text, stored in a computer system for easy update and access. The "information about the text" constitutes the annotations of the text.

My PhD work has been carried out under the organizational umbrella of the Kaj Munk Research Centre at Aalborg University, Denmark. Kaj Munk (1898–1944) was an influential and prolific playwright, journalist, pastor, and poet, whose influence was widely felt — both inside and outside of Denmark — during the period between World War I and World War II. He was murdered by Gestapo in early January 1944 for his resistance stance.

The two main tasks of the Kaj Munk Research Centre in which I have been involved during my PhD work are: a) Digitizing the *nachlass* of Kaj Munk, and b) Making the texts of Kaj Munk available electronically to the general public. My dissertation reflects these tasks by taking the works of Kaj Munk as the empirical basis, the empirical sample data, on which to test the theoretical advancements made in my dissertation.

My work has thus not been about Kaj Munk or his works as seen from a historical or even literary perspective. My perspective on Kaj Munk's works has been that of a computer scientist seeking to represent annotated versions of Kaj Munk's works in a computer database system, and supporting easy querying of these annotated texts. As such, the fact that the empirical basis has been Kaj Munk's works is largely immaterial; the principles crystallized, the methods obtained, and the system implemented could equally well have been brought to bear on any other collection of annotated text. Future research might see such endeavors.

The theoretical advancements which I have gained during my PhD build on the work of a number of other people, the primary point of departure being the work of Dr. Crist-Jan Doedens in his PhD dissertation from 1994: "Text Databases — One Database Model and Several Retrieval Languages", University of Utrecht, the Netherlands. Dr. Doedens, in his PhD dissertation, described the "Monads dot Features" (or MdF) model of annotated text, as well as the "QL query language" defined over MdF databases.

In my work, I have taken the MdF text database model, and have both reduced it in scope in some areas, and have also extended it in other areas, thus arriving at the EMdF model. I have also taken Doedens's QL query language, and have reduced part of it to a slightly smaller sub-language, but have also extended it in numerous ways, thus arriving at the "MQL query language".

The EMdF model is able to express almost any annotation necessary for representing linguistic annotations of text. As I show in Chapter 10, it is certainly the case that all of the annotations with which we in the Kaj Munk Research Centre have desired to enrich the Kaj Munk Corpus, can be expressed in the EMdF model.

The MQL query language is a “full access language”, supporting the four operations “create”, “retrieve”, “update”, and “delete” on all of the data domains in the EMdF model. As such, it goes beyond Doedens’s QL, which was only a “retrieval” language.

I have implemented the EMdF model and the MQL query language in the “Emdros” corpus query system. In so doing, I have fulfilled most of the demands which Doedens placed on an annotated text database system, as I show in Chapters 4, 5, and 14.

The dissertation is structured as follows.

Part I on “Theoretrical Foundations” encompasses Chapters 1 to 8.

Chapter 1 introduces the topics of the thesis, and provides definitions of some of the main terms used in the dissertation. Chapter 2 provides a literature review. Chapter 3 provides a brief introduction to the topic of “ontology”, for use in later chapters. Chapter 4 introduces and discusses the EMdF model, while Chapter 5 does the same for the MQL query language. Chapter 6 introduces and discusses the Sheaf, which is one kind of output from an MQL query. Chapter 7 describes a general algorithm for, and a classification of possible strategies for, “harvesting” a Sheaf, that is, turning a Sheaf into meaningful results. Chapter 8 discusses the relationship between annotated text (as expressible in the EMdF model) on the one hand, and time on the other.

Part II on “Applications” encompasses Chapters 9 to 13.

Chapter 9 introduces Part II. Chapter 10 describes how the theoretical foundations laid in Part I can be used to implement the Kaj Munk Corpus in the EMdF model. Chapter 11 discusses the principles behind, and the functionality of, a web-based application which I have written on top of Emdros in order to support collaborative annotation of the Kaj Munk Corpus. Chapter 12 is the central application-chapter, in which I show how both the EMdF model, the MQL query language, and the harvesting procedure described in Part I can be brought to bear on the task of making Kaj Munk’s works available electronically to the general public. I do so by describing how I have implemented a “Munk Browser” desktop application. Chapter 13 discusses ways in which the EMdF model and the MQL query language can be used to support the process of finding quotations from one corpus in another corpus, in this case, finding quotations from the Bible in the Kaj Munk Corpus.

Part III on “Perspectives” is very short, encompassing only two chapters.

Chapter 14 discusses ways in which the EMdF model and the MQL query language can be extended to support the requirements of the problem of storing and retrieving annotated text even better. Finally, Chapter 15 concludes the dissertation.

Appendix A gives the grammar for the subset of the MQL query language which closely resembles Doedens’s QL.

Seven already-published, internationally peer-reviewed articles accompany the dissertation in Appendix B, and form part of the basis for evaluation of the dissertation.

# Part I

# Theoretical foundations

# Chapter 1

# Introduction

## 1.1 Introduction

"There are many ways to skin a cat," so the saying goes. Similarly, there are several ways to write a PhD dissertation. I have chosen the way recommended to me by my PhD supervisor, Peter Øhrstrøm, namely to publish a number of articles, and let the PhD dissertation be a hybrid between, on the one hand, a lengthy introduction, and on the other hand, the published articles. All of the articles appended to the main body of this dissertation have been peer-reviewed internationally.

The contents of this introduction, together with the published articles, reflect some of the research activities in which I have been engaged in my three years of PhD study (May 2005 through April 2008). The common thread weaving through all of my research activities has been that of "annotated textdatabases." An annotated text database is, as Doedens [1994] puts it,

> "an interpreted text, i.e. a combination of text and information about this text, stored in a computer, and structured for easy update and access. The text and the information about the text are *different* parts of the *same* database." (p. 19; emphasis in original)

The key notion in this definition is that of a "combination of text and information about this text". To me, a text by itself may be interesting and useful, but its utility can potentially increase manifold when the text is combined with information about the text, and stored in a text database. This is the basic interest, the basic wonder that has made my PhD studies enjoyable from start to finish.

Organisation-wise, my research has been carried out at the Kaj Munk Research Centre at Aalborg University, Denmark. Therefore, the empirical basis for my theoretical research has, for the most part, been that of the works of Kaj Munk. Kaj Munk (1898–1944) was a Danish playwright, pastor, poet, and journalist whose influence was widely felt, both inside and outside of Denmark in the period between World War I and World War II. He was murdered by Gestapo for his resistance stance in early January 1944.

My research has not been about Kaj Munk as a literary or even historical figure, nor about his works as literary works or historical artifacts. Rather, my research has focussed on the theoretical and practical problems centered around the task of making Kaj Munk's texts available as annotated text. This has been done from the perspective of computer science, with elements of computational linguistics and knowledge representation being

supportive disciplines under whose broad umbrellas I have found inspiration for solutions to some of the problems posed by the empirical data and the theoretical considerations. This PhD dissertation thus falls within the category of "cross-disciplinary research."

## 1.2 Empirical application

My PhD work is not only cross-disciplinary, it it is also a blend of theoretical and principal considerations on the one hand, and an empirical application of these considerations to practical problems on the other hand. The theoretical and principal considerations have led to their application upon the following practical domains of knowledge:

1. **Representation of annotated text**, both in a database management system and outside of such a system (see Chapter 10).
2. **Software development**, as applied to:
   (a) My own database management system for annotated text, called Emdros (see Chapters 4, 5, 6, and 7).
   (b) A "Munk Browser", i.e., a desktop software application which supports "browsing" the Kaj Munk Corpus, as well as searching it (see Chapter 12).
   (c) Website development for collaborative annotation (see Chapter 11).
3. **Artificial intelligence**, that is, an attempt at building an algorithm for locating quotations from the Bible in the Kaj Munk Corpus (see Chapter 13).

The most important of these applications, seen from my perspective, is the development of my "Emdros" database management system for annotated text. Before I introduce Emdros, I need to define what I mean by the terms "database system", "database management system", "database", "text database", and "annotated text database". I will define these terms in the next two sections.

## 1.3 What is a database system?

A "database", according to [Date, 1995], is not a piece of software. Rather, it is a collection of *data*:

> "A **database** consists of some collection of persistent data that is used by the application systems of some given enterprise." (Date [1995], p. 9; emphasis in original.)

Date explains that "persistent" database data

> "differs in kind from other, more ephemeral data, such as input data, output data, control statements, work queues, software control blocks, intermediate results, and more generally any data that is transient in nature." (ibid.)

Thus the nature of the data in a database is that it is persistent, i.e., neither ephemeral nor transient. In fact, it is "information" (p. 4) which is potentially useful to "some given enterprise", and which is therefore stored in a database.

Notice that Date also mentions database data as being *used* by *application systems*. These application systems are made up of layers of software that include both a database management system (DBMS) and applications running on top of the DBMS. These, in turn, make up the software that is only one kind of component in a full *database system*. Date explains (p. 5) that

> "a database system involves four major components, namely, **data**, **hardware**, **software**, and **users**." (Date [1995], p. 5; emphasis in original.)

The users are human beings wishing to make use of the data in the database system. The data is the information which the users wish to use. The hardware is a physical computer system which stores the data and which runs the database software. The database software consists, as already mentioned, of:

1. A database management system (DBMS), which is responsible for:

   (a) "the shielding of database users from hardware-level details" (Date [1995], p. 7), that is, abstracting away the details of hardware-level storage and retrieval, typically abstracted away into query language constructs.

   (b) Actually storing the data in physical database files (through the intermediary agency of the operating system on which the DBMS is running),

   (c) Providing high-level query language functionality for user- and application-initiated requests on the database.

   (d) And other functions which are less important in this context.

2. A number of software applications running on top of the DBMS. The user will typically be interacting with one of these software applications rather than the DBMS itself directly. A database application running on top of a DBMS may be domain-specific (a good example would be the "Munk Browser" described in Chapter 12), or it may be general (a good example would be the "MySQLAdmin" application, which is a tool for database administrators to interact graphically with the MySQL[1] database management system with the purpose of performing administrative tasks on one or more MySQL databases).

The concepts mentioned so far can be illustrated as in Figure 1.1. As can be seen, the hardware is a box surrounding both the physical databases (data) and the software (Operating System, DBMS, and Applications). The software parts run on top of each other in the layers shown. Finally, the users are human beings who make use of the database system through interaction with the applications.

[1] http://www.mysql.com

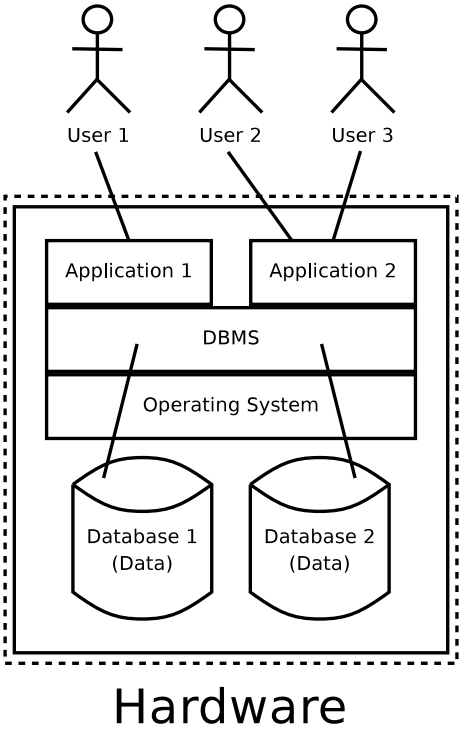



Figure 1.1: Components of a database system. The double box with an inner solid box and an outer striped box represents the hardware. Inside the hardware, we find two files (Database 1 and Database 2) containing the data. We also find the software, which I have drawn as consisting of of three layers: The application-layer, the Database Management System (DBMS), and the Operating System. The little people at the top represent users who interact with the application-layer.

## 1.4 What is an annotated text database system?

I have just defined what I mean by "database system", "database", "database management system", and "database application". These definitions are derived from those of C.J. Date, who is an authority in the field of database systems. Now I turn to another set of definitions, which are all related to the central theme of this dissertation, namely "annotated text databases".

Given that a *database* is a collection of persistent data, a *text database*, then, is a database whose primary content consists of text. As a further step, the term "*annotated text database*" can be defined as a text database which, in addition to the text itself, contains — as a significant part of the information-value in the database — *information about* that text, i.e., *annotations* of the text.

An *annotated text database management system* (ATDBMS), then, is a piece of software which performs DBMS functions on one or more annotated text databases. It is used by *annotated text database applications* in order to provide text database services to human *users*.

Finally, an annotated text database system consists of:

1. human **users**,
2. **data** in the form of annotated text databases,

3. **software** in the form of:

   (a) an operating system,

   (b) an annotated text database management system (ATDBMS),

   (c) one or more annotated text database applications,

   and

4. the **hardware** on which these three pieces of software run.

These definitions are closely tied to those of Doedens [1994], who states:

> "A *general* system, i.e., a system which is not tied to a particular application, which holds the information of an expounded text [Doedens's term for an annotated text database] in a way which makes it possible to selectively update and access the information according to its structure and content I call a **text database management system**, or **text database system** for short. In practice a text database system is a computer program. The idea of a text database management system is that it supplies services to one or more applications. Through the use of these services the applications can access the text data managed by the text database management system." (Doedens [1994], p. 21; emphasis in original.)

Note that Doedens does not, as I do, distinguish between a *text database management system* on the one hand, and a *text database system* on the other. In contrast to Doedens's conflation of these two terms, I define a *text database system* as the whole system (users + annotated text databases + hardware + software), following Date [1995]. In my terminology, a *text database management system* is then part of the software encompassed by the whole text database system.

The nature of the annotations added to the text in an annotated text database is not my concern at this point. I shall return to this question in Chapter 4. For now, it is enough to bear in mind the notion that text can be annotated, i.e., information about the text can be added outside the text, and both can be stored in an annotated text database, which in turn is managed by an annotated text database management system (ATDBMS).

The primary example of an ATDBMS with which I have been concerned in my PhD work, is a piece of software which I call "Emdros." It has served as the testbed for the majority of the ideas produced in my PhD work, and also embodies many of the principles, theories, strategies, and methods which I have developed for dealing with annotated text. In the next section, I describe Emdros from a high-level perspective, preparing the ground for some of the following chapters.

## 1.5 Emdros

Emdros is an annotated text database management system which I have been developing from around 1999 until the present. The theoretical foundation for Emdros has, in large part, been provided by Crist-Jan Doedens's PhD dissertation [Doedens, 1994] from which I have already quoted several times. Where Doedens blazed a theoretical trail in the

jungle of annotated text database theory, I have merely been following behind Doedens, implementing many of his theoretical ideas in practice, and also slightly repairing and clearing the path along the theoretical trail, thus making the theoretical trail safer for others to follow.

Doedens defined, in his PhD dissertation, a number of theoretical constructs related to the field of annotated text database theory. From a high-level perspective, these theoretical constructs include:

- A *text database model* which Doedens called the "Monads dot Features" model (or **MdF** model for short).
- The notion of ***topographicity*** (to which I return below).
- A powerful *text database retrieval language* which Doedens called "**QL**".
- A text database *retrieval language* which Doedens called "LL". Doedens also provided a formal framework for and specification of the translation of LL to QL.

In my work, I have extended the MdF model to become the "Extended Monads dot Features model" (or EMdF for short). The EMdF model is described in detail in Chapter 4, and also in [COLING2004, FSMNLP2005, RANLP2005, LREC2006]. I have also implemented a large subset of Doedens's QL, resulting in the query language which I call "MQL".

Doedens, in his PhD work, did not provide any implementation of his ideas. The majority of his ideas were described in abstract mathematical terms, and his "QL" query language, in particular, was difficult to implement due to the nature of the descriptions. As Winskel [1993] explains, there are two opposite ways of describing the semantics of formal languages:

- Denotational semantics, which describes *what* to compute, but not *how* to compute it.
- Operational semantics, which describes *how* to compute the desired result, but not *what* that result should be.

Doedens, in his PhD work, described a *denotational* semantics of the QL language. Given the level of abstraction employed in the descriptions, it was difficult for me, being a mere undergraduate in computer science, to turn those "whats" into "hows". Yet I succeeded in doing precisely this for a small subset of QL, and this result (called "MQL" for "Mini QL") was reported on in my B.Sc. thesis [Petersen, 1999]. Since then, I have greatly expanded the MQL query language, both before and during my PhD studies, as can be seen in part by following the growth of MQL through the articles appended to this dissertation, in particular [COLING2004,FSMNLP2005,RANLP2005,LREC2006]. Chapter 5 reports on the "state of the art" of the MQL query language, and also spells out what has been done during my PhD studies and what has been implemented before.

The notion of "topographicity" developed by Doedens can be briefly explained as follows. A language is "topographic" if there exists an isomorphism between the structure of the expressions in the language and the structure of the objects denoted by the language [Doedens, 1994, p. 108]. QL is a "topographic language", meaning that the structure

of a query in the language is in a direct isomorphism relationship with the structure of the database objects denoted by the query. Furthermore, there is a direct isomorphic relationship between the structure of the query and the structure of the output from the query, called a "Sheaf". Both of these tenets make QL "topographic".

A subset of MQL, too, is topographic in both of these ways. But where QL was "merely" a "retrieval-language", in which it was only possible to retrieve objects from an existing database, MQL is a "full access language", in which it is possible to create, update, delete, and retrieve all of the data domains expressible in the EMdF model. Thus MQL is, in a sense, larger than QL, since it has "create, update, delete" functionality in addition to the "retrieve" functionality provided by QL. Moreover, the topographic subset of MQL has grown since its inception in 1999, even if it has not yet reached the full potential expressed by Doedens's description of QL.

## 1.6 Recurring example

In order to be able to write meaningfully and in practical terms about theoretical matters, I shall employ a recurring example. I have chosen to use a poem by Kaj Munk, entitled "Den blaa Anemone" (in English: "the blue anemone") from 1943. It appears in Figure 1.2 on the following page both in its original Danish, and in my own English rendition. The rendition suffers from not being a word-for-word equivalent. Instead, it is hoped that the flavour of the original can be descried in the English rendition, however dimly.

This poem is perhaps the most well-known of Kaj Munk's poems, not least because it has been set to music and is sung in most elementary schools as part of any Danish pupil's learning experience. The poem provides an excellent example for the explicative purposes of this dissertation, for the following reasons: a) It is relatively short, but not too short for my purposes. Thus a "happy medium" between brevity on the one hand a complexity on the other is represented by this poem. b) The poem exists both in handwritten, original, autograph form, and in a printed form. There are slight differences between these forms, and even the handwritten autograph form has corrections and variations. This entails a certain level of complexity in the encoding and annotation of the text, since all versions should ideally be represented and annotated.

I shall return to this example poem at opportune points in the dissertation.

## 1.7 Dissertation plan

The dissertation is laid out as follows. There are three parts:

1. Theoretical foundations, in which the theoretical foundations are laid for the rest of the dissertation.

2. Applications, in which these theoretical foundations are brought to bear on a number of practical problems.

3. Perspectives, in which directions for future research are explained.

Hvad var det dog, der skete?
mit hjerte haardt og koldt som Kvarts
maa smelte ved at se det
den første Dag i Marts.
Hvad gennembrød den sorte Jord
og gav den med sit dybblaa Flor
et Stænk af Himlens Tone
den lille Anemone,
jeg planted der i Fjor.

Paa Lolland jeg den hented,
en Hilsen fra min Fødeø.
Saa gik jeg her og vented
og tænkte, den maa dø.
Den savner jo sit Skovkvarter,
sin lune Luft sit fede Ler
i denne fjendske Zone
forgaar min Anemone
jeg ser den aldrig mer.

Nu staar den der og nikker
med Smil i Jyllands skarpe Grus
ukuelig og sikker
trods Havets Storm og Gus
som om Alverdens Modstand her
har givet den et større Værd
en lille Amazone
og dog min Anemone;
jomfruelig og skær.

Hvad var det dog, der skete
Mit Hjerte koldt og haardt som Kvarts
det smelter ved at se det
den første Dag i Marts.
Jeg mindes, under Vinters Had
jeg intet mere Haab besad.
Gør med sin vaarblaa Krone
den lille Anemone
igen mit Hjerte glad?

Ja, denne rene Farve
den er mig som en Vaarens Daab
der naadig la'r mig arve
en Evighed af Haab.
Saa bøjer jeg mig ned mod Jord
og kysser ømt dit Silkeflor
en Flig af Naadens Trone
du lille Anemone,
hvor er vor Skaber stor!

What was it that had happened?
My heart, as hard and cold as quarts,
must melted be to see it,
this day, the first of March.
What piercéd had the darkened soil,
and giv'n it with its deep blue leaves
a touch of Heaven's tone
the little anemone
I planted there last year.

On Lolland did I fetch it,
a greeting from my isle of birth.
Then here I walked and waited,
and thought that: "It will die.
For miss it must its coppice place,
its warmish air, its fatty clay
in this so hostile zone,
is lost my anemone
I'll see it ne'er again."

Now there it stands, all nodding
with smiles in Jylland's flinty dirt,
invincible and certain
despite sea, storm, and fog.
As if the World's resistance here
has given it a greater worth.
A little amazone,
and yet, my anemone;
so virgin-like and clean.

What was it that had happened?
My heart, as hard and cold as quarts,
is melting now to see it,
this day, the first of March.
I call to mind, through Winter's hate
I had no longer any hope.
Will, with its spring-blue crown
the little anemone
again my heart make glad?

Yes, this unmixéd colour,
it is to me a bath of spring
which, graceful, lets me inherit
an eternity of hope.
I bend, then, down towards the earth
and kiss thy leaves so tenderly.
A hint of mercy's throne —
thou, little anemone!
How great is our creator!

Figure 1.2: "Den blaa Anemone", a poem by Kaj Munk (1943), with my own (almost literal) translation into English

I now describe the chapters in Part I. Later parts will have their own introductions.

In Part I, this introductory chapter is followed by a literature review (Chapter 2). I then describe and explain the concept of "Ontology" (Chapter 3). I then describe and explain the EMdF model implemented in Emdros (Chapter 4), followed by a chapter on the MQL query language of Emdros (Chapter 5). One of the data structures which can be returned from an MQL query is a "Sheaf", which contains the results from a topographic query (Chapter 6). The following chapter describes a number of strategies for "harvesting" a sheaf, i.e., for turning a sheaf into meaningful results (Chapter 7). Finally, in a chapter on "Annotated Text and Time", I expand upon the ideas presented in [ICCS-Suppl2008] (Chapter 8).

# Chapter 2

# Literature review

## 2.1 Introduction

In this chapter, I will review some of the most important literature related to my own work. The two most important aspects of my own work are: a) Database models for annotated text, and b) Query languages for annotated text. Hence, I have divided this literature review into two main sections, one on database models for annotated text (2.2), and one on query languages for annotated text (2.3). Since the field of Information Retrieval overlaps to some extent with my own work, I have included a section on relevant literature from the field of information retrieval (2.4). Since my main contribution is a "corpus query system," I then list some of the other corpus query systems available (2.5). Finally, I conclude this chapter (2.6).

## 2.2 Other models for annotated text

### 2.2.1 Introduction

What is a database model? As defined by Codd [1980] (and generally accepted in the field), a database model consists of three distinct items. A database model is, and I quote from Codd [1980]:

> "1) A collection of data structure types (the building blocks of any database that conforms to the model);
>
> 2) a collection of operators and inference rules, which can be applied to any valid instances of the data types listed in (1), to retrieve or derive data from any parts of those structures in any combinations desired;
>
> 3) a collection of general integrity rules, which implicitly or explicitly define the set of consistent database states or changes of state or both — these rules may sometimes be expressed as insert-update-delete rules."

This is different from what some authors call a "data model", or a "schema". A schema is an application of a database model to a particular problem domain. Where a database model defines what it is *possible* to express in a database (constrained by Codd's "data structure types", "operators and inference rules", and "integrity rules"), a schema defines

an *actual* set of constraints that conforms to some problem domain. For example, whereas a database model may stipulate that it is *possible* to have objects that have attributes, a schema may define that there must be objects of *particular* object types called "clause", "phrase", and "word", with certain *particular* attributes.

Database models are used for a variety of purposes, which are not just limited to storing text. Some database models support text as a side-effect of supporting other kinds of data. The variety of database models that exist becomes apparent in even a cursory overview of the literature [Angles and Gutierrez, 2008, Loeffen, 1994, Abiteboul, 1997, Christophides et al., 1994, Abiteboul et al., 1997, Blake et al., 1994, DeHaan et al., 2003, Kashyap and Rusinkiewicz, 1997, Carletta et al., 2003b, Cassidy, 1999, Evert et al., 2003, Gonnet and Tompa, 1987, Nicol, 2002, Tague et al., 1991, Tompa, 1989].

What concerns us in this dissertation, however, is not database models in general, but database models for annotated text. The survey paper of Arjan Loeffen on "Text Databases: A survey of text models and systems" [Loeffen, 1994] describes a view on database models similar to Codd, and summarizes the state of the art up until 1994 in text database models. Some of the salient models cited by Loeffen's paper will be discussed here.

In the following, I first discuss XML and SGML. Then I discuss grammar-based models of text, graph-based models of text, range-based models of text, and object-oriented models of text.

## 2.2.2 XML and SGML

### 2.2.2.1 Introduction to XML and SGML

SGML was developed during the 1970'ies and 1980'ies, and was published as an ISO-standard in 1986 [ISO, 1986]. XML was developed during the early 1990'ies, and is a reduced subset of SGML [Bray et al., 2004]. According to Robin Cover,

> "Both SGML and XML are "meta" languages because they are used for defining markup languages. A markup language defined using SGML or XML has a specific vocabulary (labels for elements and attributes) and a declared syntax (grammar defining the hierarchy and other features)."[1]

These descriptions very succinctly capture the essence of what XML and SGML are: "meta-languages" in which it is possible to define other languages. When I say "language" here, I mean "formal language", not a "natural language" (such as English, Greek, or Urdu). Formal languages have a formal syntax, and sometimes also a formal semantics. The sense of the word "formal" here is the sense described by [Winskel, 1993] and [Martin, 1991], namely a mathematical object (a "formal language", "formal syntax", or "formal semantics") which has been given a rigorous definition.

XML and SGML are thus formal "meta-languages" in which it is possible to define other formal languages. I have defined a formal language using XML, which is designed to capture the data present in the Kaj Munk Corpus. I return to this language in Chapter 10.

[1] Cover Pages, [Cover, 1986-2008]. Specific URL: http://xml.coverpages.org/sgml.html Access online April 12, 2008. Robin Cover's "Cover Pages" is the most respected online resource for information about XML and SGML.

Since I will be referring to XML in Chapter 10, I now give an extremely brief tutorial on a few important aspects of XML.

A *tag* in XML is a part of a document which is enclosed in <angle brackets>, for example, `<munktxt>` (which in my XML language starts a Munk Text). Tags are separate from the rest of the document, and form part of the *markup* of the document. In effect, a tag specifies either the *start* of an annotation, its *end*, or *both at the same time*.

An *element* in XML terminology consists of a tag and its *attributes*. The attributes of a tag are marked up as *attribute-names* with *attribute-values*, and are located inside of the tag itself. For example:

```
<metadata kind="title" value="La Parole"/>
```

Here the tag "metadata" has two attributes, with names "kind" and "value". The value of the attribute "kind" is "title", and the value of the attribute "value" is "La Parole".

XML specifies that an element can either *have content* or be *empty*. An empty element tag looks like this: `<br/>`, consisting of an "angle bracket begin", the tag name, a front slash, and an "angle bracket end". It has no content because it is both a *start-tag* and an *end-tag* in one and the same tag. Attributes may intervene bewteen the tag name and the front slash.

A tag with content, on the other hand, has a start-tag, e.g., `<munktxt>`, followed by its contents, followed by its end-tag, e.g., `</munktxt>`. The start-tag specifies where the element starts, and the end-tag specifies where it ends. This can obviously be used to specify annotations of text. Any attributes will always occur on the start-tag, and never on the end-tag.[2]

#### 2.2.2.2 XML and SGML as models of annotated text

Both XML and SGML can be said to provide models for annotated text. Or, more precisely, the model of annotated text assumed by XML and SGML can be applied to a large subset of the problems inherent in the domain of annotated text. Briefly put, both SGML and XML assume that annotated text is strictly hierarchical: All tags must be strictly hierarchically nested within each other in a tree. For example, the sequence of tags + text:

```
<page><paragraph>This is the start of a paragraph which
</page><page> continues on the next page.</paragraph></page>
```

is illegal in XML. The reason is that the end-tag for the page occurs at a place where the paragraph tag is still open (and not closed by its end-tag). This does not form a strictly nested hierarchy, but forms an overlapping model of text. Thus the natural way of representing annotated text in SGML and XML does not allow for overlapping hierarchies.

This obviously poses a problem for many applications. Take something as simple as a language defined in XML which is designed to capture page layout and document structure at once. For any given page, its start and end in the document must be marked up. This is due to the purpose of the language as a page layout markup language. At the same time, the dual purpose of this language is to describe document structure. Yet

[2] Pedants among my readership will be relieved to know that I shall not overlook the fact that an empty element tag is "both start-tag and end-tag in one and the same tag", and thus attributes may occur on the end-tag in this special case.

paragraphs and chapters (and even words) do not always stay nicely within the borders of a single page – a chapter rarely fits on a single page, and paragraphs are certainly able to extend over a page break, as the example above shows.

There have been two proposed ways of dealing with this problem in the SGML and XML communities. One is standoff markup [Thompson and McKelvie, 1997], and the other is milestones. The latter is described in the TEI Guidelines TEI [2007], where we read that milestones

> "... simply mark the points in a text at which some category in a reference system changes. They have no content but subdivide the text into regions, rather in the same way as milestones mark points along a road, thus implicitly dividing it into segments." [TEI, 2007, Section 3.10.3 ]

Thus, for example, with milestones, it is possible to say that a page starts and ends at such and such points in the document (using start-tags and end-tags), and to say with milestones that a paragraph begins and ends. For example:

```
<page><paragraphbegin/>This is the start of a paragraph which
</page><page> continues on the next page.<paragraphend/></page>
```

Here the <paragraph> start-tag and the </paragraph> end-tag have been replaced with empty tags which mark with separately named tags the start and end of the paragraph. This neatly solves the problem of representing overlapping hierarchies, but does push the burden of keeping track of the overlapping hierarchies from the XML parser onto the application which uses the XML parser.

The idea of Standoff Markup is to separate the annotations from the text itself structurally, instead using pointers between the annotation and the parts of the text which it annotates. This involves giving every word a unique identifier, which is then referred to in the annotations. I have not used standoff markup in the Munk XML.

Thus there are ways to overcome the apparent non-support of overlapping hierarchies in SGML and XML, using either standoff markup or milestones, or a mixture of both.

#### 2.2.2.3 Applications of XML to annotated text

A number of applications of XML that deal specifically with linguistic annotations have appeared in the literature. McKelvie et al. [2001] describes the XML-based MATE workbench, later expanded in the NITE project [Carletta et al., 2003a,b, 2004, Evert et al., 2003, Voormann et al., 2003]. Work on supporting linguistic queries on XML-based models of linguistically annotated text is described in Bird et al. [2005].

### 2.2.3 Grammar-based models of text

As Loeffen [1994] explains, a number of models of text have been proposed which are based on parsing text according to a formal grammar. One of the first to do so was Gonnet and Tompa [1987], which described "a new approach to modeling text", viewing texts as trees which had characters as leaves and structural information (such as words, paragraphs, headings, dictionary entry headwords, etc.) as inner nodes. This model was later expanded in Tague et al. [1991]. A similar database model may be found in Gyssens et al. [1989].

### 2.2.4 Graph-based models

The survey article of Angles and Gutierrez [2008] both surveys the state of the art in graph-based models, and provides an overview of 40 years of research in the area. Some of the models relevant to text, and mentioned in this survey, include:

- Gram [Amann and Scholl, 1992] (which, although a general model, can be applied to hypertext, as shown by the authors).
- Gonnet and Tompa [1987] (already mentioned; grammar-based models can, in general, be said also to be graph-based models, since grammars result in parse-trees, which are graphs).
- Consens and Mendelzon [1989] (which provides a visual language in which to query hypergraph-based databases, including hypertext).
- Navarro and Baeza-Yates [1995] (which defines the "Proximal Nodes" model for structured text, later expanded in Navarro and Baeza-Yates [1997] and Baeza-Yates and Navarro [2002]).

A number of graph-based models of text specifically suited to linguistic annotations have appeared in the literature. Examples can be found in [Cotton and Bird, 2002, Bird et al., 2000a,b, Bird and Liberman, 2001, Cassidy and Bird, 2000, Cassidy and Harrington, 2001].

### 2.2.5 Range-based models of text

Range-based models of text generally identify regions of text by a start-pointer and an end-pointer into the text. Examples include the seminal paper by Clarke et al. [1995] and the paper by Nicol [2002], which attempts to provide the same kind of algebra as Clarke et al. [1995]. I am not aware of any actual implementation of either of these models, except that Emdros implements a (non-proper) superset of the ideas presented in these papers. Jaakkola and Kilpeläinen [1996b] provide an algebra for structured, nested text regions, also expounded in Jaakkola and Kilpeläinen [1996a].

### 2.2.6 Object-oriented models of text

Object-oriented models of text have *objects* and *object types* as a fundamental part of the model. Doedens [1994] provides an example of an object-oriented database model. Of course, I have lated expanded Doedens's work [Petersen, 1999, 2004, Sandborg-Petersen, 2002-2008, Petersen, 2006a, 2007a, 2006b, 2005]. Another database model in the object-oriented vein is described in Abiteboul et al. [1997].

## 2.3 Query languages for annotated text

One part of a database management system is the database model which it supports. Another very important part is the means through which this data is accessed, usually through a query language.

SQL [Ullman and Widom, 1997, Date, 1995] is the most widely used database query language in business applications. It supports the relational database model [Date, 1995]. An application of SQL to the problem of interval-based querying can be found in Zhang et al. [2001], which was inspired by Kriegel et al. [2000]. Interval-based querying is important in Emdros, as we shall see in Chapter 5. Kashyap and Rusinkiewicz [1997] show how to model textual data using the Entity-Relationship model, and how to query it using SQL. Davies [2003] shows how to implement "unlimited" linguistic annotation of text in a relational database using SQL.

Object-oriented query languages include the one defined by the Object Database Standard [Cattell and Barry, 1997], Lorel [Abiteboul et al., 1997], and Doedens's QL [Doedens, 1994]. Doedens's QL had a predecessor, called QUEST, described in [Harmsen, 1988, 1992, Doedens and Harmsen, 1990, Talstra et al., 1992]. My own MQL is another object-oriented query language [Petersen, 1999, 2004, 2007a, 2006b].

XML-based query languages include the "NXT Search" query language supported by the NITE XML system [Voormann et al., 2003, Carletta et al., 2004]. In addition, Bird et al. [2005] extend the well-known XPath language [W3C, 2007] to support queries on linguistic structures encoded in XML. Cassidy [2002] applied XQuery [Boag et al., 2005] to the problem of querying linguistically annotated text, while DeHaan et al. [2003] provides a "comprehensive XQuery to SQL translation using dynamic interval encoding". Bird et al. [2000a] described a query language which could be applied to Annotation Graphs; the latter is an XML schema for representation of multi-layered linguistic annotation of text. The work of Baeza-Yates and Navarro [2002] supports XML-based queries on the Proximal Nodes database model mentioned above. Similar in scope is Jaakkola and Kilpeläinen [1996a], which defines the SGrep query language over nested text regions, which can be applied to XML. Blake et al. [1994] show how to implement an SGML-based database model in SQL, and how to query it using SQL.

The TIGERSearch query language was designed to support treebanks, or databases containing linguistic annotations of text in the form of trees [König and Lezius, 2003, Lezius, 2002a,b]. A related language, VIQTORIA [Steiner and Kallmeyer, 2002], used visual representation to search treebanks. Mettler [2007] provided a re-implementation of TIGERSearch, adapting it for use in parallel treebanks. Kallmeyer [2000] describes an implementation of a query language on syntactically annotated corpora. Aswani et al. [2005] show how to index and query linguistically annotated text using the GATE system [Cunningham et al., 2002, Cunningham and Bontcheva, 2003].

Others which could have been cited in this section will be cited below in Section 2.5 on "Other corpus query systems".

## 2.4 Relation to Information Retrieval

The subject of this thesis is "annotated text". I see this as distinct from "raw text", which is the view imposed on text by most Information Retrieval researchers. In Information Retrieval, the goal is to locate the answer to questions ("queries") in *documents*. To this end, there exist only two levels of text: The *token* (or word), and the *document*. One could choose to view this arrangement as an annotated text: The text (tokens) is annotated with one layer of information (document-boundaries). However, for the purposes of this dissertation, I shall choose to define most Information Retrieval techniques as being outside the scope of my research.

Some techniques from Information Retrieval are, however, very applicable to the problems which I have attempted to solve during my PhD work. In particular, the following techniques apply very well:

- **Indexing of text.** Indexes are indispensable for fast querying, also in the problems which I have attempted to solve. The seminal paper by Bayer and McCreight [1972] showed that it was feasible to obtain speed-increases even with no further investment in new hardware (by using B-Trees, introduced in this article). Since then, a lot of variants of B-Trees have appeared, including R-Trees [Guttman, 1984, Lee et al., 2003] and R*-Trees [Beckmann et al., 1990] to name but a few. Other indexing methods include inverted files [Larson, 1984, Zobel et al., 1998], PAT-Trees [Gonnet et al., 1992], Signature Files [Zobel et al., 1998], and the index compression techniques described in [Zobel and Moffat, 2006, Frakes and Baeza-Yates, 1992, Baeza-Yates and Ribeiro-Neto, 1999]. See also Fujii and Croft [1993], which applies text indexing techniques to Japanese, and Kornacker [1999], which describes a high-performance, extensible indexing technique.

- **Result ranking.** When a number of results have been retrieved, it may be important to the user of the database system that the results appear in an order which makes sense from the user's perspective. This order can be difficult to compute if there is no a-priori ordering which makes sense.[3] Some of the techniques which fall under the category of "query result ranking" are described in the survey by Zobel and Moffat [2006], and are also described in Baeza-Yates and Ribeiro-Neto [1999].

## 2.5 Other corpus query systems

This thesis would not be complete without a review of the most important pieces of software comparable to Emdros. Therefore, I give a short review here.

Several systems are aimed at doing word-level searches only. They include Corpus Work Bench [Christ, 1994, Christ et al., 1999, Christ, 1998], SARA and its successor XAIRA[4] (designed to search the British National Corpus), Professor Mark Davies's online search tool[5] , and others.

Other tools are aimed at searching structured text, including syntactic annotations and/or speech-based data. They include:

- SGrep [Jaakkola and Kilpeläinen, 1996a] (designed for searching general XML).

- XQuery [Boag et al., 2005, DeHaan et al., 2003] (again designed for searching general XML).

- TGrep2 [Rohde, 2004] (designed for searching the Penn Treebank [Marcus et al., 1994a,b]).

[3] In the case of a Biblical corpus, it might make sense to present the results in the order imposed by the traditional canonical ordering of books, and at a lower level, the chapters and verses which they contain. In the context of the Kaj Munk Corpus, however, it is not obvious what the ordering of search results should be. One proposal might be to present the results in the order in which Kaj Munk wrote them (where known). Another proposal might be to use the techniques on query result ranking in the literature mentioned.

[4] http://www.xaira.org

[5] http://davies-linguistics.byu.edu

- VIQTORYA [Steiner and Kallmeyer, 2002] (designed for searching the VERBMOBIL German corpus).
- TIGERSearch [Lezius, 2002a,b, König and Lezius, 2003, Mengel and Lezius, 2000] (designed for searching general treebanks, including the German newspaper corpus, TIGERCorpus [Brants et al., 2002, Brants and Hansen, 2002]).
- STASearch [Mettler, 2007] (a reimplementation of the TIGER query language [König and Lezius, 2003, Lezius, 2002a], extended with capabilities for querying parallel treebanks).
- LPath [Bird et al., 2005] (designed for querying XML-based treebanks).
- Emu [Cassidy, 1999, Cassidy et al., 2000, Cassidy and Bird, 2000, Cassidy and Harrington, 2001] (designed for querying annotated speech).
- MATE [McKelvie et al., 2001, Mengel, 1999] (the predecessor of NITE).
- NITE [Carletta et al., 2003a, 2004, Evert et al., 2003, Carletta et al., 2003b, Voormann et al., 2003, Carletta et al., 2002] (for querying XML-based corpora which use the NITE object model [Carletta et al., 2003b, Evert et al., 2003]).
- GATE[6] [Cunningham et al., 2002, Cunningham and Bontcheva, 2003, Aswani et al., 2005] (which was designed as a general language engineering platform).
- Ellogon[7] [Petasis et al., 2002] (which was designed as a general language engineering platform, a competitor with GATE).[8]
- Linguist's Search Engine[9] [Resnik and Elkiss, 2005] (which was designed as a general-purpose linguistic search engine).
- Manatee [Rychlý, 2000] (which was designed for very fast retrieval of linguistic markup in very large corpora).
- The Sketch Engine[10] [Kilgariff et al., 2004] (which builds on Manatee).
- CorpusSearch[11] (which was designed for searching treebanks in Penn Treebank format, in particular, the Penn-Helsinki Parsed Corpora of Historical English).
- Finite Structure Query (FSQ) [Kepser, 2003] (which was designed for querying treebanks, using a logic-based query language).
- Netgraph [Mírovský et al., 2002] (which was designed to search through the Prague Dependency Treebank).

[6] http://gate.ac.uk

[7] http://www.ellogon.org

[8] Incidentally, the Ellogon Object Model as described in [Petasis et al., 2002] looks very much like the original MdF model.

[9] http://lse.umiacs.umd.edu:8080/

[10] http://www.sketchengine.co.uk

[11] http://corpussearch.sourceforge.net/

- The Layered Query Language (LQL) system [Nakov et al., 2005] (which was designed to search through annotated MEDLINE abstracts).

There are others, but these are some of the salient ones in the literature.

## 2.6 Conclusion

In this chapter, I have reviewed some of the most important literature related to my own work. More citations will come in later chapters, at places where the literature is relevant to mention. The major sections of this chapter have followed the two major themes of my own work, namely: a) Database models for annotated text (2.2), and b) Query languages for annotated text (2.3). Since the field of Information Retrieval contains literature and themes which are relevant to my own work, I have also cited some of the most important literature from this field (2.4). Finally, I have cited most of the other "corpus query systems" available (2.5).

# Chapter 3

# Ontology

## 3.1 Introduction

Sowa [2000, p. 492] defines the two notions "ontology" and "*an* ontology" in the following way:

> "The subject of *ontology* is the study of the *categories* of things that exist or may exist in some domain. The product of such a study, called *an ontology*, is a catalog of the types of things that are assumed to exist in a domain of interest *D* from the perspective of a person who uses a language *L* for the purpose of talking about *D*."

Thus *an ontology* is a catalog of categories in some domain of interest, while *ontology* is the study of categories in some domain. For a similar view, see Nilsson [2001].

In later chapters, and in [CS-TIW2007], I shall make use of the notion of "an ontology", and so I need to discuss the notions involved.

The notion of ontology goes back at least to Aristotle, who invented the way of definition by means of genus, species, and differentiae [Sowa, 2000, p. 4]. In this way of defining what a thing is, a thing (species) is defined by saying what its genus is, along with a list of differentiae that show how the species differs from the genus. For example, an elephant (species) is a mammal (genus) with the following differentiae: An elephant has four legs, grey hide, a trunk, and tusks (in the male).[1]

The word "ontology" itself, however, does not go back to Aristotle. Although it has Greek roots, it was actually constructed by Jacob Lorhard in 1609, as Øhrstrøm et al. [2005] report.

The rest of the Chapter is laid out as follows. First, I discuss Type and instance as key notions in ontology (3.2). I then discuss various relations that may obtain between types (3.3), in particular the "is-a" relation. I then discuss the notions of "supertype" and "subtype", which flow out of the definition of the "is-a" relation as a partial order on types. In this section, also briefly discuss lattice-structures as a natural way of representing ontologies (3.4). Finally, I conclude the chapter.

[1] Biologists would most probably beg to differ with this definition, but since this is not a thesis in biology, I shall not concern myself with finding the exact definition in the literature. What is important here is the method of definition, not the definition itself.

## 3.2 Type and instance

A *type* (or ontotype, or concept type) is an abstract entity which consists of three things:

1. A *name*. We use this name whenever referring to the type.

2. An *extension*. That is, the set of all things that exist in the domain of interest, and which are *instances* of this type. An instance, in turn, is a thing (physical or abstract, real or imagined) which can be categorized as being of the type in question.

3. An *intension*. That is, the set of properties that are common to all of the instances in the extension of the type.

For example, the type which we may give the name "dog" has as *extension* all of the dogs in the world. Or at least, if not all dogs in the whole world, then all dogs in the domain of interest *D*. We may be interested, for example, only in the domain of "mammals from comics", in which case the extension of "dog" would include such instances as "Snoopy", "Odie", "Dogbert", etc. These venerable dogs might be excluded from the extension of the type "dog", however, if the domain of interest was not comics or imaginary dogs, but real dogs. Thus it is important to specify what the domain of interest *D* is.

The *intension* of the type "dog" is the set of properties which are common to all dogs. Again, the domain of interest D might have an influence on the intension. For example, for dogs in comics, it is usually not a requirement that they be able to breathe real air with nitrogen-atoms, oxygen-atoms, carbon dioxide, and other gases, whereas this is usually a member of the set of properties in the intension of real-world dogs. That is, comic-dogs are usually not required to ever have been alive in any real sense, whereas real-world dogs must have been alive at some point for them to be in the extension of the type "dog".

## 3.3 Ontological relations

Given that an ontology is "a catalog of the types of things that are assumed to exist in a domain of interest", it is usually a good idea to structure such a catalog in some way. One way of doing so is to maintain *relations* between types. A relation is here understood as a structural link which links two or more entities into a structure.

The relation which is most often used in structuring ontologies is the "is-a" relation. If type *B* is-a type *A*, then type *B* is a species of the genus *A*, with some differentiae. This usually entails that the extension of *B* is smaller than the extension of *A*, since the intension of *B* has more properties than the intension of *A*.

For example, the following relationships would obtain:

- elephant *is-a* mammal.
- sedimentary rock *is-a* rock.
- employee *is-a* person.
- artefact *is-a* object.

There are other relations than is-a which may obtain between types. WordNet [Fellbaum, 1998] is a good example of what may be considered an ontology which implements more relations than the is-a relation. For example, part-whole (meronymy) relationships may obtain (e.g., "steering wheel" is part-of "car"). For a discussion of the relations employed by WordNet, see Miller et al. [1990].

## 3.4 Supertypes, subtypes, and lattices

The notions of "supertype" and "subtype" are closely related to the notions of "genus" and "species", yet are also distinct from these notions. A type *A* is a supertype of type *B*, if and only if *B* is-a *A*. However, the is-a relation is a partial order, which, among other properties, means that the is-a relationship is *transitive*. For example, if type *C* is-a type *B*, and *B* is-a type *A*, then it also holds that *C* is-a type *A*. Thus, for example, "elephant" is-a "mammal", and "mammal" is-a "vertebrate", and therefore, it is also true that "elephant" is-a "vertebrate". Thus we need to be able to distinguish between "direct supertype" and merely "supertype", where the former is a type in our catalog of types (that is, our ontology) which is only one is-a step away from the type of which it is a supertype. That is, with respect to the is-a partial order, the direct super type of *B* is a least element of the set $\mathscr{T}$ of types *A* for which it holds that *B* is-a *A*.

Conversely, the "subtype" relation is the opposite of the "supertype" relation. Thus, if *B* is-a *A*, then *B* is a subtype of *A*. The same distinction between "direct subtype" and "subtype" can be made as for supertype, but in the opposite direction: A direct subtype *B* of some type *A* is a greatest element of the set $\mathscr{T}$ of types *B* for which it holds that *B* is-a *A*.

The reason it is called a "*super*-type" lies in the nature of the is-a relation as a partial order. Partial orders on sets may be viewed as chains of elements of the sets, with the partial order relation standing between the elements in the chain. For example, the partial order $\leq$ on the set of the integers has a chain which looks like this:

$$1 \leq 2 \leq 3 \leq 4 \leq 5 \leq \ldots$$

In ontological terms, if the "is-a" relation is denoted by "$\leq$", then:

$$\text{elephant} \leq \text{mammal} \leq \text{vertebrate}$$

One way of thinking about these chains is that they are vertical. If so, then it is natural from the Latin roots of "super" and "sub" that a "super"-type lies "above" its "sub"-type, which lies "below" its "super"-type.

I said that the is-a relation is a partial order on types. It is well-known that partial orders give rise to structures known as lattices [Insall and Weisstein]. A lattice is a directed, acyclic graph with certain properties that make them very well suited to the problem of structuring types in ontologies. We shall see examples of lattice structures in later chapters.

## 3.5 Conclusion

In this Chapter, I have briefly outlined some of the most important notions in the field of ontology. First, I have discussed the way in which types may be defined in terms of their

name, their extension, and their intension. The extension is the set of all instances of the type, whereas the intension is the set of properties common to all instances of the type. I have then discussed ontological relations which may obtain between types, in particular, the “is-a” relation. This relation is a partial order on sets of types. From the definition of “is-a” as a partial order on sets of types, the notions of “supertype” and “subtype” flow, as does the notion of a “lattice of types”. We shall return to examples of lattices of types in later chapters.

# Chapter 4

# The EMdF model

## 4.1 Introduction

In this chapter, my topic is the EMdF text database model for annotated text. The EMdF model sprang out of Doedens's [1994] work on the MdF (Monads dot Features) model, and is an extension of the MdF model.

In Section 4.2, I start by recounting the demands on a text database model which Doedens defined in his PhD thesis. This is done to form a backdrop against which to show in what ways I have extended the MdF model in order to meet the full requirements which Doedens stipulated must be met of a text database model.

In Section 4.3, I discuss the original MdF model of Doedens. I do not discuss the full MdF model, since I find some of it irrelevant to the further discussion, but the core of the MdF model is reformulated in formal terms.

In Section 4.4, I discuss my own extensions of the MdF model, resulting in the EMdF (Extended MdF) model. The extensions mainly have the goal of making the MdF model implementable.

In Section 4.5, I show how one can implement the EMdF model using a relational database engine as a backend. Thus the fruit of Section 4.4, namely to make the MdF model implementable, is borne out in practice in a recipe for implementation in Section 4.5.

In Section 4.6, I discuss another implementation, namely an in-memory implementation of an EMdF database.

Finally, I conclude the chapter.

## 4.2 Demands on a database model

### 4.2.1 Introduction

In order to be able to know whether his work was adequate, Doedens identified and defined thirteen demands on a text database model which, in his opinion, it was good for a text database model to meet. In order to be able to talk intelligently about these demands in later sections and chapters, I here offer a reformulation of Doedens's demands (4.2.2), followed by a critique and extension of Doedens's demands (4.2.3).

### 4.2.2 Doedens's demands

Doedens [1994] defines thirteen demands on a database model, of which his own EMdF model and QL query language only meet the first ten, plus part of the twelfth. The demands, which I state here in my own formulation, are as follows [Doedens, 1994, pp. 27–30]:

**D1. Objects:** We need to be able to identify separate parts of the text and its annotation. This can be done with 'objects'.

**D2. Objects are unique:** We need to be able to identify every object uniquely.

**D3. Objects are independent:** The existence of an object in the database should be possible without direct reference to other objects.

**D4. Object types:** Grouping objects into "object types" with like characteristics should be possible. (Note how this is similar to the notion of "concept types" mentioned in Chapter 3.)

**D5. Multiple hierarchies:** It should be possible to have several "views" of the same data, as specified in the annotations. For example, in a Biblical database, it should be possible to have both a "document-view" (books-chapters-verses) and a "linguistic view" (phrases, clauses, sentences). These two hierarchies should be able to coexist.

**D6. Hierarchies can share types:** It should be possible for an object type to be a part of zero *or more* hierarchies.

**D7. Object features:** Objects should be able to have "features" (or "attributes"), with values being assigned to these "features". (Note how this is similar to the notion of "properties" or "attributes" of concept types mentioned in Chapter 3.)

**D8. Accommodation for variations in the surface text:** For example, two different spellings of the same word should be attributable to the same word object.

**D9. Overlapping objects:** It should be possible for objects of the same type to "overlap", i.e., they need not cover distinct parts of the text.

**D10. Gaps:** It should be possible for objects to cover a part of the text, then have a "gap" (which does not cover that part of the text), and then resume coverage of the text — and this should be possible to do arbitrarily many times for the same object (i.e., it should be possible for an object to have arbitrarily many gaps).

All of the demands above have to do with the data structures that a database model defines. In the terms of Codd [1980] (see Section 2.2.1 on page 29), this is equivalent to Codd's first point, which is a "collection of data structure types". In order to obtain Codd's second and third points, Doedens defines three more demands:

**D11. Type language:** We need a language in which we can define both object types and the features that those object types contain.

**D12. Data language:** We need a strongly typed language in which it is possible to express creation, deletion, update, and retrieval of all of the data domains in the model.

**D13. Structural relations between types:** It should be possible to express structural relations between object types declaratively, for example, that a sentence consists of words.

### 4.2.3 Critique of Doedens's demands

Doedens went on in his PhD work to describe a database model (MdF) and a number of retrieval languages (QL and LL) which, taken together, meet demands D1-D10 and part of D12. Demands D11 and D13 were left for further research, and the "create", "update", "delete" parts of D12 were also left unspecified – only the "retrieve" part of D12 was partially specified in Doedens's work.

In my work, I have implemented full support for Doedens's demands D1-D10. I have also implemented full support for Doedens's D11 and D12 demands. That is, it is possible in MQL to declare object types and their features (D11), and it is possible to create, update, delete, and retrieve all of the data domains of my EMdF model (D12). I have not implemented support for Doedens's D13 demand. Very early versions of Emdros did support the kind of declarations needed by demand D13, but I took this functionality out, for two reasons: First, it proved to be a performance bottleneck on the insertion and update of data to have to check and maintain the integrity of these structural relations. And second, at the time I did not see a need for these kind of declarations. I have since then come up with a number of reasons for supporting such declarations, but I have not had the time to implement them (see Section 14.2 on page 175).

It is possible to offer both a critique and a extension of Doedens's demands, something which I shall attempt to do now.

With respect to D5 ("Multiple hierarchies: It should be possible to have several "views" of the same data"), I offer the following critique: When using the word "hierarchies", Doedens must have thought strictly in terms of structural (i.e., part-whole) hierarchies. There are two points of critique which I want to raise in this respect. First, there are also other types of hierarchy than structural: The Aristotelian "genus-species" kind of hierarchy should be mentioned too. Second, the "multiple views" should not be defined in terms of different object types; that is, it should be possible to express several "views" of the same data within the same object type. For example, it should be possible to express that a given sentence can be analyzed in several different ways. Doedens mentions such an analysis (pp. 80–82), and offers a solution within his own MdF model. Thus I have two points of critique, each leading to a new demand. I now make these further demands explicit:

**D5.1. (Multiple) inheritance:** It should be possible to declare that an object type is a subtype of one or more object types. This would mean that the newly declared object type (let us call it *A*) had the union of the features of its supertypes, with possible renaming to avoid conflicts (let us, without loss of generality, call the supertypes *B* and *C*), plus any features that might distinguish *A* from *B* and *C*. Then *A* would be said to *inherit* the features of *B* and *C*, and *A* would be a subtype of both *B* and *C*. Single inheritance should be possible, too, in which an object *D* inherited from a single object type *E*.

This would be useful, among other times when querying. For example, if we had an object type "Phrase" which defined, among other features, a "phrase_function",

then we might further define an object type “NP”, which was a subtype of “Phrase” and therefore had the same features as “Phrase”, as well as any other features which might pertain to NPs (e.g., whether there is an article or another determiner in the NP). In addition, we might define an object type “VP”, which inherited from “Phrase” and also had other features of its own. Then, whenever querying, we might want to specify either an “NP” or a “VP” explicitly, but we might also “not care”, and simply search for “Phrase”. This would find both NPs and VPs, whereas search for “NP”s would not also find “VP”s.

**D5.2. Multiple parallel analyses:** It should be possible to specify (when creating an object of a given object type) that this object is a part of a specific analysis or annotation of a given piece of text. It should also be possible to express, within the same object type, that two objects belong to different analyses of the same text.

This is possible both within the MdF model and within the EMdF model. A different way of specifying the parallel analyses than the way offered by Doedens (p. 82) would be to declare, on each object type which might be a part of several anlyses, a feature which was simply an integer — let us call it “ambiguity_set”. Then objects whose value for this “ambiguity_set” was the same (e.g., 1) would belong together.

With respect to D8 (“accommodation for variations in the surface text”), it should not only be possible to describe variations in spelling for the same word; it should also be possible to express two or more different texts which are very similar, yet which have differences at various points in the exact number of words (e.g., words being deleted or inserted in one text with respect to another text). There is a whole field within theology devoted to the study of such differences, namely the field of *textual criticism*. For an introduction to textual criticism of the New Testament, see Greenlee [1995]. It would be good if a text database model supported such “insertion” and “deletion” of words. I therefore define another demand on textual database models, designed to encapsulate this need:

**D8.1. Multiple parallel texts:** It should be possible to express parallel texts which have only slight differences (such as a word that has been inserted or deleted in one text with respect to another text).

This is probably possible to do within the MdF and EMdF models, but further research is needed to establish a good way to do it.

## 4.3 The original MdF model

### 4.3.1 Introduction

The original MdF model had many characteristics, and defined many concepts and data structures. In order to be able to describe the EMdF model, which is an extension of a subset of the MdF model, I now describe the parts of the MdF model which are pertinent to the later discussion. Much of this discussion has been presented several times in my published papers (e.g., [COLING2004, RANLP2005, FSMNLP2005, LREC2006]). Therefore, the following discussion will be brief.

### 4.3.2 Monads

In the MdF model, a monad is simply an integer with a specific meaning: It is a "smallest, indivisible unit" of the backbone of the database, also known as the "monad stream". The "monad stream" is the sequence of monads which make up the sets of monads of all objects in a database. The sequence of the integers (1, 2, 3, ...) defines the *logical reading order* of the database. This is conceptually different from the physical reading-order, which may be left-to-right, right-to-left, top-to-bottom, or combinations of these.

A monad is an integer. As such, it can be a member of a set of monads, in the usual Zermelo-Fraenkel sense of sets. Objects are sets of monads with concomitant attributes (known as features). We now turn to objects, followed by Object Types and Features.

### 4.3.3 Objects

An object in the MdF model is a pair, $(M,F)$ where $M$ is an arbitrary set of monads (it may even be empty), and $F$ is a set of value-to-attribute assignments $(f_i, v_i)$, where $f_i$ is the $i^{th}$ feature (or attribute), and $v_i$ is the value of $f_i$. Whenever speaking about an object $O = (M,F)$, we will usually say that $O$ is synonymous with $M$, such that, for example, a monad $m$ may be "a member of $O$" (i.e., $m \in O$) when we really mean $m \in M$. Features-values are denoted with "dot-notation", i.e., "$O.f$" means "The value of feature $f$ on the object $O$".[1]

### 4.3.4 Object types

Objects may be grouped into *object types*. An object type has many similarities with the conceptual types described in Chapter 3: Object types are abstract entities which group instances with similar attributes. The instances may have different values for these attributes. One difference between conceptual types as described in Chapter 3 and the Object Types of the MdF model, is that Object Types may not (as described by Doedens) inherit characteristics from other object types. Indeed, Doedens did not even describe a language in which to declare object types or their features; this was left for further research, and acknowledged as such [Doedens, 1994, p. 258].

Doedens defined a number of special object types, three important ones being "all_m", "pow_m", and "any_m".

There is only one object of type all_m for any given database; it is the object having no features and consisting of the monad set which is the big-union of all monad sets of all objects in the database. That is, the sole object of object type all_m is the set of monads consisting of all monads in use by all objects. This may, of course, have gaps, if there are unused monads.

Similarly, "pow_m" is the object type having objects which have no features, and which are monad sets which are drawn from the power set (hence "pow_m") of the sole object in the all_m object type.

Finally, "any_m" is the object type having objects with no features, and where each object consists of precisely one monad from the sole object of object type all_m.

[1] This formalization of $O = (M,F)$ is not present in Doedens's work, but is, in my opinion, a fair interpretation and formalization of Doedens's intent.

### 4.3.5 Features

In the MdF model, an object type may have zero or more *features*. A feature is a strongly typed attribute which may take on a range of values in specific instances (i.e., objects) of the object type. For example, a "Word" object type is likely to have a "surface" feature, whose type is "string". Or a "Phrase" object type may have a "function" feature, whose values are drawn from the strings "Subject", "Object", "Predicate", etc.

Doedens described features as functions which assigned, for each unique object, a value to that feature of that object. In so doing, he was careful about not restricting the co-domains of feature-functions: He left it up to the implementor to decide what should be acceptable co-domains of feature-functions, that is, he left it up to the implementor both to explicify and to limit the types which a feature may take on. This is what I have done in implementing the Extended MdF model (EMdF model), to which I now turn.

## 4.4 The abstract EMdF model

### 4.4.1 Introduction

In this section, I show how my extension of the MdF model (my "EMdF model") differs from the MdF model. I also show how my extensions make the MdF model less abstrac, more concrete, and therefore more implementable.

Some of the information below has already appeared in my MA thesis. However, it appeared in an appendix, on five short pages, and said extremely little beyond what Doedens had already said, except to say that I had implemented Doedens's ideas. Therefore, this Section (i.e., 4.4) presents information not presented before for the fulfilment of any other academic degree.

The rest of Section 4.4 is laid out as follows. First, I discuss monads, then Objects get their treatment, followed by Object types, followed by Features. Finally, I discuss "Named Monad Sets", which are an innovation in relation to Doedens's work.

### 4.4.2 Monads

Monads are exactly the same in the EMdF model as in the MdF model, and serve the same purpose. For purposes of implementation-efficiency and -possibility, however, the monad stream has been limited in the EMdF model to a finite range, namely from 1 to MAX_MONAD. The monad labelled "MAX_MONAD" is currently set to 2,100,000,000 (or 2.1 billion). This choice is "large enough" for even giga-word corpora, and neatly fits within a signed 32-bit signed integer.

In the EMdF model, there are two special monads, and one special set of monads, all related: The "min_m" monad is the least monad which is currently in the big-union set of all sets of monads of all objects. That is, it is the least monad used by any object. Similarly, "max_m" is the greatest such monad, i.e., the greatest monad in use by any monad. The set all_m is defined (contrary to the definition by Doedens) to be the set of monads covering all monads in the range "min_m" to "max_m", both inclusive.

### 4.4.3 Objects

Objects are almost the same in the EMdF model as in the MdF model. However, in the EMdF model, an object is always distinguished uniquely from other objects (demand D2), not by its set of monads being unique, but by a unique integer called an "id_d" (the "_d" suffix distinguishes it from the "id_m" id in the MdF model, and is meant to mean "id_database"). All object types have a feature called "self" whose type is "id_d", and whose value for any given object is a number which is unique in the whole database. This design choice was motivated by a desire to have objects of the same object type which have exactly the same set of monads, yet which are distinct, something which is not possible in the MdF model due to the requirement that all objects within a single object type be unique in their set of monads. I have found no theoretical side-effects of this design choice which are theoretically detrimental. The choice in the MdF model to base uniqueness on the set of monads seems to be motived on Doedens's part by simplicity and clarity: Doedens writes (p. 59):

> "No two objects of the same object type may consist of the same set of monads. The reason for this restriction is that it allows us a simple and clear criterion for what different objects are."

However, Doedens also mentions that this uniqueness based on sets of monads entails a number of theoretical side-effects which are beneficial, including the fact that all objects of a given object type can be said to have an "ordinal number" (id_o) which uniquely defines it within the object type. This "ordinal" is defined in terms of a lexicographic ordering on the sets of monads of which the objects of a given object type consist. The id_o then arises because the lexicographic ordering is what mathematicians call "well-ordered" [Allenby, 1991, p. 17]. Since all objects are unique in their monads, the "id_o" uniquely identifies each object.

This theoretical gain in the MdF model is lost in the EMdF model. However, the loss is not great, since:

1. The "id_o" ordinal number has been replaced with an "id_d" which is easy to compute (just take the next one available whenever creating a new object).

2. The "id_o" is expensive to compute, since it involves comparison of sets of monads.

3. Worse, the "id_o" of an object may change when a new object is inserted into the database. Hence, it is not a stable id, unless the database is stable. In contrast, the id_d of an object never changes throughout the lifetime of the object, and is never re-used for a new object.

The need for objects of the same type to consist of the same set of monads becomes apparent when one studies various linguistic theories, including Role and Reference Grammar [Van Valin and LaPolla, 1997] and X-Bar syntax [Jackendoff, 1977], in which objects are often stacked on top of each other, referencing the same string of underlying words, and thus the same set of monads. To demand that all objects be distinct in their monads would preclude having two objects of the same object type which was stacked on top of another object of the same kind. Hence this design choice on my part, of letting uniqueness be determined, not by the set of monads, but by a distinguishing id_d feature called "self".

### 4.4.4 Object types

Object types are almost the same in the EMdF model as in the MdF model. One difference is that all object types have a feature called "self", which is explained in the previous section.

Another difference is that I have, in the EMdF model, restricted the possibilities for naming an object type: In the EMdF model, the name of an object type must be a C identifier, that is, it must start with a letter (a-z) or an underscore (_), and then it may have zero or more characters in the name, which must be either letters (a-z), underscores (_), or digits (0-9). The reason for this restriction is that some database management systems (such as the early versions of MySQL which Emdros supports) require table names to be C identifiers. Since I have mapped the EMdF model to the relational database model, and since Emdros was designed to support various database management systems as backends, I needed a uniform "lowest common denominator" to share among the database backends supported. I chose "C identifiers" because of their simplicity.

I have also extended the MdF model by supporting *object range types* (which are completely different from the "range types" of Doedens's MdF). In the EMdF model, an object type may be declared (upon creation) to be one of the following object range types:

1. WITH SINGLE MONAD OBJECTS, which means that all objects of this object type must occupy exactly one monad. That is, the sets of monads of the object types are limited to containing precisely one monad (a singleton).

2. WITH SINGLE RANGE OBJECTS, which means that all objects of this object type must occupy exactly one, contiguous stretch of monads. The stretch is contiguous, meaning it may not have gaps. It may, however, consist of a single monad ("singleton stretch"), and so does not need to start and end on different monads.

3. WITH MULTIPLE RANGE OBJECTS, which means that there are no restrictions on the sets of monads of the objects of the object types: They may consists of one monad, of a single, contiguous stretch of monads, or they may consist of multiple contiguous stretches of monads (including singleton "stretches").

The reason for introducing these object range types is that it allows for more efficient storage and retrieval of sets of monads. See especially [ICCS-Suppl2008] and Chapter 8.

Furthermore, I have extended the MdF model by supporting *monad uniqueness constraints*. A "monad uniqueness constraint" is declared for the object type upon object type creation, and may be one of the following:

1. HAVING UNIQUE FIRST MONADS, which means that all objects within the object type must have unique first monads. That is, no two distinct objects in the object type may have the same first monad.

2. HAVING UNIQUE FIRST AND LAST MONADS, which means that both the first and last monads of all objects within the object type must have *both* unique first monads, *and* unique last monads.

3. WITHOUT UNIQUE MONADS, which means that there are no restrictions on the uniqueness of either the first or the last monads.

The reason for introducing these monad uniqueness constraints is again that it allows for more efficient storage and retrieval of sets of monads. Again, please see [ICCS-Suppl2008] and Chapter 8 for more information.

### 4.4.5 Features

As mentioned above, Doedens did not limit the types which a feature may take on. In implementing Emdros, I had to decide what types to support for features. I have implemented support for the following:

1. integers
2. id_ds
3. strings
4. enumerations (which are finite sets of pre-declared labels, see below)
5. ordered lists of integers
6. ordered lists of id_ds
7. ordered lists of enumeration constants

As mentioned above, all objects in the EMdF model (but not in the MdF model) have a feature called "self", whose type is "id_d", and whose value is the unique id_d given to each object.

An enumeration is a set of (label,integer) pairs. The labels are called enumeration constants, whereas the integers are called the "enumeration values" of each "enumeration constant".

This concept of enumeration is almost identical to the concept of the same name in the C and C++ programming languages. The only difference is that in the EMdF model, it is not allowed to have two enumeration constants with the same integer value within the same enumeration, whereas this is allowed in C and C++. See Stroustrup [1997, pp. 76–78].

The reason for this restriction on enumeration values is that it allows the implementation to store enumeration constants unambiguously as the integers to which they correspond. That is, because there is a one-to-one correspondence between the set of enumeration constants and the set of integers both coming from the same enumeration, the implementation can take advantage of this one-to-one correspondence and only store the integer, and only pass the integer around internally. If there were no one-to-one correspondence, the implementation would have to store the enumeration constants as strings, which would be inefficient.

One of the types which a feature may take on is "id_d". This entails that objects may point to each other. This is useful, e.g., in the case of anaphora pointing backwards to other objects (and similarly for cataphora pointing forwards); another example where this is useful would be the "secondary edges" as used in the TIGER Corpus [Brants et al., 2002, Brants and Hansen, 2002]. Tree, of course, may also be represented using id_d pointers, which point either "upwards" (to the parent), or "downwards" (to the children).

In the latter case, of course, the feature should be a "list of id_ds". Directed acyclic graphs can also be represented using lists of id_ds.

A string feature may be declared to be "FROM SET", meaning that "under the hood", strings are changed to integers for more efficient storage and retrieval. In effect, the set of actually occurring strings is given a one-to-one mapping to a set of integers of the same cardinality. The one-to-one mapping is, of course, bijective, and so may be inverted for retrieval purposes.

Any feature may be declared to be "WITH INDEX", meaning that an SQL index is created on the column containing the values of that feature. This may speed up search.

### 4.4.6 Named monad sets

I have extended the MdF model also by supporting "named sets of monads" which are not objects. That is, it is possible to declare — and give a name to — a set of monads, which may then be used as the basis of querying in a limited part of the database. For example, if one had all of Kaj Munk's works in one single database, it might be useful to declare a named set of monads for each kind of document in the database (e.g., "plays", "poetry", "sermons", "prose", etc.). Then, when searching, one could limit the search to, say, "poetry" simply by referencing this named set of monads in the query. As with objects, the named sets of monads are completely arbitrary, and may have multiple, disjoint stretches of monads. The only restriction is that they cannot be empty. Allowing them to be empty would defeat their purpose, namely of being easy-to-use restrictions on queries.

This concludes my description of the abstract EMdF model. I now turn to the relational implementation of the EMdF model.

## 4.5 The relational implementation of the EMdF model

### 4.5.1 Introduction

In this section, I describe the implementation of the EMdF model as it is currently done in the four relational database backends supported by Emdros: SQLite 3[2], SQLite 2[3], MySQL[4], and PostgreSQL[5]. I do so in two further subsections; the first is on storage of "meta-data", while the second subsection is on storage of "object data".

### 4.5.2 Meta-data

Meta-data of various kinds need to be maintained in order to handle the various data domains of the EMdF model. The implementation is trivial, in that each kind of data is implemented by a table with the requisite number of columns necessary for storing the relevant data. In Table 4.1, I list the various kinds of meta-data implemented, along with a brief comment on each.

I invite the reader to consult Petersen [2007b] for the details.

[2] http://www.sqlite.org
[3] http://www.sqlite.org
[4] http://www.mysql.com
[5] http://www.postgresql.org

| Meta-data kind | Comment |
|---|---|
| schema version | Holds the version of the SQL schema. |
| Enumeration | Holds the names of enumerations present |
| Enumeration constants | Holds the names and values of enumeration constants |
| Object types | Holds the names and other properties of object types |
| features | Holds the object types, names, types, and default values of features |
| min_m | Holds the least monad used by any object |
| max_m | Holds the greatest monad used by any object |
| monad sets | Holds "named monad sets" |
| Sequences | Hold the next "unique number" to be used for various purposes (e.g., id_ds) |

Table 4.1: Kinds of EMdF meta-data.

### 4.5.3 Object-data

In the relational implementation of the EMdF model, I have chosen to map one object type to one table, and to map each object to one row of such a table. I originally implemented object data more complexly, by having two tables for each object type: One containing the features, and one containing the monad sets, one row per maximal stretch of monads (also known as "monad set element") per object. This turned out to be inefficient, so I opted for storing the sets of monads as a marshalled string in the same row as the feature-data.

I have implemented three ways of storing monad sets, each corresponding to the three object range types defined on page 50:

1. Object types declared WITH SINGLE MONAD OBJECTS get one column for storing the monad set, namely the sole monad in the monad set (known by the column name "first_monad").

2. Object types declared WITH SINGLE RANGE OBJECTS get two columns for storing the monad set, namely: The first monad and last monad (column names: first_monad and last_monad respectively).

3. Object types declared WITH MULTIPLE RANGE OBJECTS get three columns for storing the monad set, namely: The first monad, the last monad, and a string-representation of the arbitrary set of monads. The reason for storing the first and last monad, even though this information is also contained — redundantly — in the string-representation of the arbitrary set of monads, is that this scheme allows more efficient retrieval of certain kinds of queries, including the GET OBJECTS HAVING MONADS IN as well as so-called "topographic queries". See Chapter 5 for more information.

The string-representation of the monad set is an efficiently packed delta-representation (similar to the d-compression mentioned in Zobel and Moffat [2006]), in which only the deltas (differences) between each end-point in the stretches and gaps of the set of monads are encoded. Furthermore, the integers involved are efficiently encoded using a base-64 encoding scheme.

Furthermore, each object is represented by a column storing its id_d (column name: "object_id_d"). This is obviously the "self" feature.

Finally, all other features than "self" are stored in other columns, one column per feature. The various kinds of EMdF feature type map to the following SQL type (PostgreSQL syntax has been taken as the example syntax):

1. An EMdF **integer** maps to an SQL INTEGER.
2. An EMdF **id_d** maps to an SQL INTEGER.
3. An EMdF **enumeration** maps to an SQL INTEGER, with appropriate conversion inside of Emdros.
4. An EMdF **string** maps to an SQL TEXT, if it is not declared FROM SET. This entails that strings of arbitrary length may be stored. Alternatively, if the string is declared FROM SET, it maps to an SQL INTEGER, which in turn is a foreign key in a separate table containing the strings.
5. An EMdF **list of integer** maps to an SQL TEXT, by storing the integers as strings of numbers in base-10 with spaces between, and also surrounded by spaces. This makes it easy to search by using standard SQL LIKE syntax, in that all integers will be surrounded by a space on either side.
6. An EMdF **list of id_d** maps to an SQL TEXT in exactly the same way as an EMdF list of integer.
7. An EMdF **list of enumeration constants** maps to an SQL TEXT in the same way as an EMdF list of integer, with appropriate conversion between enumeration constants and enumeration values.

The column names of features are the same as the features, except they are always prefixed by the string "mdf_". This is in order to avoid name-clashes. For example, a malicious user might declare a feature called "object_id_d", thinking that they would be able to wreak havoc by forcing Emdros to declare two different columns with the same name. Not so, since the feature-name "object_id_d" maps to the column-name "mdf_object_id_d", thus making it distinct from the column holding the id_d of the object.

## 4.6 An in-memory EMdF database

### 4.6.1 Introduction

In this section, I describe a set of data structures which I have found useful whenever having to deal with EMdF data in-memory. I start by defining an "EmdrosObject". I then describe the main data structure as it is implemented in various software libraries which I have written. I first implemented this way of representing an in-memory EMdF database in my "Linguistic Tree Constructor" software, which originates back in the year 1999.[6] I have also implemented this representation as part of my work with the Munk Corpus, this time implementing it twice: First, in the Python programming language, in order to

[6] For more information on the Linguistic Tree Constructor, please see <http://ltc.sourceforge.net>.

be able to build Emdros databases from the Munk Corpus. And second, in the Munk Browser, to which I return in Chapter 12.

The information presented in this section already appeared in kernel form in my B.Sc. thesis [Petersen, 1999], and as such has already counted towards one academic degree. The material presented here is a modernization of the ideas presented in my B.Sc. thesis, but does not constitute radically new information.

The reason I include it anyway is that I shall need to return to this data structure in Section 12.4 on page 152.

### 4.6.2 EmdrosObject

In order to be able to store Emdros objects in an in-memory EMdF database, I have found it useful to declare a class (in C++, Python, Java, or whatever language I am using) which has the following members:

1. A set of monads (implemented by the class SetOfMonads in the Emdros code base).
2. An id_d (being the object id_d of the object)
3. A map data structure mapping feature-names to strings representing the values of those data structures. I have implemented variations on this theme over the years; I have found it most useful to distinguish between "non-string values" and "string-values" inside the EmdrosObject, and to let the code calling the EmdrosObject keep track of whether something is an integer, an id_d, an enum, or a string (leaving aside lists for the purposes of this discussion). The utility of this arrangement is that it is possible to have the EmdrosObject class write itself as an MQL statement that creates the object; differentiating between string-features and non-string-features has the effect of being able to predict (in the code that writes the EmdrosObject as an MQL statement to create the object) whether to surround the feature-value with quotes or not. A further extension would be to support lists of integers, id_ds, and enums.
4. The name of the object type of the EmdrosObject (or a surrogate for the object type, so that the real object type can be looked up somewhere else).

This data neatly encapsulates what an Emdros object is.

I now describe the main data structure that holds the in-memory database.

### 4.6.3 InMemoryEMdFDatabase

In order to be able to access any EmdrosObject quickly, I have found a data structure with the following data useful and time-efficient:

1. A container data structure to hold the object types of the database. Ideally, the object types should each contain: a) The object type name, b) The features of the object type, and c) The types of the features. This makes it possible to export the whole InMemoryEMdFDatabase to MQL statements that will recreate the database on-disk.

2. A map data structure mapping object id_d to EmdrosObject object. This is the only data structure that holds the EmdrosObjects themselves.

3. A map data structuring mapping monads to the following data structure: A map mapping object type names (or surrogates) to sets of object id_ds. Whenever an EmdrosObject is inserted into the InMemoryEMdFDatabase, this map of monads mapped to object types mapped to sets of id_ds is updated in such a way that all of the monads of the EmdrosObject result in an entry in the inner set of id_ds for the object type of the EmdrosObject. For example, if an EmdrosObject has a monad set of {1,2}, and an object type of "Phrase", then the following would occur:

   (a) The outer map would be checked, whether it had an entry for the monad 1. If it did not, an instance of a map data structure mapping object type to set of id_d would be created, and this instance would be inserted into the outer map for the monad 1. On the other hand, if the outer map already had an entry for the monad 1, then this step would be skipped.

   (b) The inner map for the monad 1 would be checked to see if it had an entry for the object type "Phrase". If it did not, an instance of a set of id_ds would be created, and this instance would be inserted into the inner map for the object type "Phrase". If, on the other han, the inner map already had an entry for "Phrase", then this step would be skipped.

   (c) By now the data structure is set up for adding the id_d, so the outer map is checked for the monad 1, yielding an inner map mapping object types to sets of id_ds. This inner map is checked for the object type "Phrase", yielding a set of id_ds. Then the id_d of the EmdrosObject which we wish to insert is added to the set, and we are done for the monad 1.

   (d) The process (a)-(c) is repeated for the monad 2.

Among other questions, this data structure supports quick answering of the following questions:

1. Which EmdrosObject has such and such an id_d.

2. Which EmdrosObjects share at least one monad with such and such a set of monads (this is useful for answering the MQL statement known as GET OBJECTS HAVING MONADS IN).

3. Which EmdrosObjects are within the confines of such and such a set of monads (this is useful for answering the topographic MQL statement known as SELECT ALL OBJECTS).

Variations on this theme exist: For example, it is sometimes useful to distinguish between the starting monad of an EmdrosObject and other monads, and so information can be stored in the data structure about which objects start at a given monad, versus which objects merely have the monad, but do not start at the given monad.

Note that this data structure is highly memory-inefficient, and needs large amounts of memory for large data sets. The reason is that the id_d of an EmdrosObject is duplicated across all of the monads in the monad set of the object.

| Monad | 20001 | 20002 | 20003 | 20004 | 20005 | 20006 |
|---|---|---|---|---|---|---|
| **Word** | 1 | 2 | 3 | 4 | 5 | 6 |
| surface | Hvad | var | det | dog, | der | skete? |
| part_of_speech | PRON_INTER_REL | V_PAST | PRON_DEMO | ADV | ADV | V_PAST |
| lemma | hvad | være | det | dog | der | ske |
| **verse** | 11 | | | | | |

| Monad | 20007 | 20008 | 20009 | 20010 |
|---|---|---|---|---|
| **Word** | 7 | 8 | 9 | 10 |
| surface | Mit | vinterfrosne | Hjertes | Kvarts |
| part_of_speech | PRON_POSS | ADJ | N_INDEF_SING_GEN | N_INDEF_SING |
| lemma | mit | vinterfrossen | hjerte | kvarts |
| **verse** | 12 | | | |

Figure 4.1: Parts of “The Blue Anemone” encoded as an EMdF database. The two tables have the exact same structure. The top row represents the monads. They start at 20001 because we imagine that this document starts some distance into the collection (i.e., database). There are two object types: **word** and **verse**. The numbers in the rows headed by the object type names (in **bold**) are the “self” feature, i.e., the id_d of each object. The non-bold headings in the first column (apart from the heading “Monad”) are the feature-names. For example, the “surface” feature of the object of object type Word with id_d 1 is “Hvad” (in English, “What”). The “verse” object type has no features beyond “self”.

## 4.7 Example

In this section, I show how the EMdF model can be used to express the first two lines of Kaj Munk’s poem, “The Blue Anemone”, introduced in Section 1.6 on page 25. For the remainder of this discussion, please refer to Figure 4.1.

First, it should be obvious that the database depicted in Figure 4.1 is not the whole database: Something comes before “The Blue Anemone”, and something comes after what is shown.

Second, there are two tables in Figure 4.1. This is partly for typographical reasons (i.e., width of the page in relation to the font size), partly because the two lines of the first stanza can be divided as in the two tables.

Third, the line of integers at the top of each table is the monads. In this particular database, “The Blue Anemone” starts at monad 20001. I could have chosen any other monad for the purposes of this example, of course.

Fourth, there are two object types in this particular database: “Word” and “Verse”[7]. The object type “Word” has four features: “surface”, “part_of_speech”, “lemma”, and “self”. The latter is shown as the integers (actually id_ds) above each “Word” object. The

[7] I use the terminology “stanza” for the whole of a coherent set of verses, and “verse” for the “line”.

"verse" object type has no features apart from the "self" feature.

Fifth, it can be seen that each "verse" object occupies the set of monads that is the big-union of all of the sets of monads belonging to the words which make up the verse. This will become significant in Chapter 5, when we discuss the MQL query language.

## 4.8 Conclusion

In this chapter, I have discussed the Extended MdF model (EMdF model). The EMdF model is a reformulation of the MdF model described by Doedens [1994]. The main contribution which I have made is to make the MdF model implementable.

I have first described thirteen demands on a text database model which Doedens deemed to be requisite for a full text database model to be complete. I have shown how the MdF model only fulfilled the first ten demands, plus part of number 12, leaving number 11 and 13 completely unfulfilled. I have also given a critique of Doedens's demands, extending and refining them with new demands.

I have then given a reformulation of the pertinent parts of the original MdF model. The reformulation is a formalization of the MdF model which is not in Doedens's original work, but is my own contribution. As I have stated, I believe that my reformulation is a fair interpretation of Doedens's intent.

I have then discussed the abstract EMdF model, and how it relates to the original MdF model, showing the differences between the original and the extension. As already stated, my main contribution is to make the MdF model less abstract, more concrete, and therefor more implementable than the MdF model. The EMdF model fully meets demands D1-D10.

I have then shown in two rounds how the EMdF model can be implemented: First, in a relational database system, and second, in an in-memory database. Thus the purpose of the EMdF model has been fulfilled, namely to be implementable. I have, of course, also fulfilled this purpose in other ways, by implementing the EMdF model a number of times, most notably in the Emdros corpus query system.

# Chapter 5

# The MQL query language

## 5.1 Introduction

In Chapter 4, we saw how the EMdF model, in conjunction with the MQL query language, together constitute what Doedens called a "full access model". In the previous chapter, it was still unclear how exactly Deodens's demands D11 and D12 were going to be met. In this chapter, I show how they are met by the MQL query language. (D13 is not met by either the EMdF model, the MQL query language, or their conjunction.)

Doedens's demands D11-D13 sound as follows in my reformulation:

**D11. Type language:** We need a language in which we can define both object types and the features that those object types contain.

**D12. Data language:** We need a strongly typed language in which it is possible to express creation, deletion, update, and retrieval of all of the data domains in the model.

**D13. Structural relations between types:** It should be possible to express structural relations between object types declaratively, for example, that a Clause consists of Phrases, or that a Book consists of Chapters, Front Matter, and Back Matter.

I shall show how demands D11 and D12 are met by the MQL query language in a succession of sections, but first, I need to offer some general remarks (5.2), to discuss the design and implementation of the MQL interpreter (5.3), and to discuss the output of an MQL query briefly (5.4).

Then, getting back to the demands, I offer a section on the "Type language" (5.5) (meeting demand D11), followed by two separate sections on different parts of the Data language (meeting D12). The two sections on "Data language" are the "non-topographic" part (5.6) and the "topographic" part (5.7). Along the way, I give examples of how the various statements are used. Finally, I conclude the chapter.

## 5.2 General remarks

Although MQL is my own invention in large part, I could not have accomplished it without standing on the shoulders of a whole army of giants. MQL draws from a pool of many sources of inspiration, among which the SQL query language [Date, 1995] and the

QL language [Doedens, 1994] are chief. I have explicitly striven to model the type language and the data access language after a mixture of both SQL and QL. For example, the "SELECT FEATURES FROM OBJECT TYPE" statement is modeled after the SELECT statement of SQL, but "borrows" the convention of enclosing any object type in [square brackets] from QL:

```
// Returns a table containing the features and their types
SELECT FEATURES
FROM OBJECT TYPE
[Word]
GO
```

Notice also that *comments* may be indicated with "//". Such comments extend until the end of the line. There is another kind of comment, which is started by "/*" and ended by "*/". The former kind of comments were borrowed from the C++ programming language, while the second kind of comments were borrowed from the C programming language. The second kind of comment may extend over multiple lines if need be:

```
/* This comment spans
   multiple lines, and
   does not end until
   the 'star followed by a slash'... */
// ... while this comment ends at the end of the line
```

The keyword "GO" is used for indicating that a particular query is fully expressed, and that the MQL interpreter can start the execution of the query. The "GO" keyword is used throughout this dissertation whenever showing an MQL query, unless it is unambigous (for example, for typographical reasons) that the query is fully expressed. For example:

```
CREATE OBJECT TYPE
[verse]
GO
```

I shall use the terms "MQL query" and "MQL statement" to mean the same thing: A string of characters forming a single "thing to do" for the MQL interpreter.

## 5.3 The MQL interpreter

I have designed and implemented the MQL interpreter as a pipeline of "stages" (or "processes"), each of which has a particular function to fulfill. An overview can be seen in Figure 5.1.

The processes (or stages) involved in interpreting an MQL query start with parsing and lexing. This may be viewed as one process, since in my implementation, the parser drives the lexer.

"Parser" here means a program module which takes tokens as input and produces an Abstract Syntax Tree (AST) as output, provided that the input stream of tokens conforms to some formal grammar which the parser implements. I have specified the formal grammar of the MQL query language using the formalism employed by the "lemon"

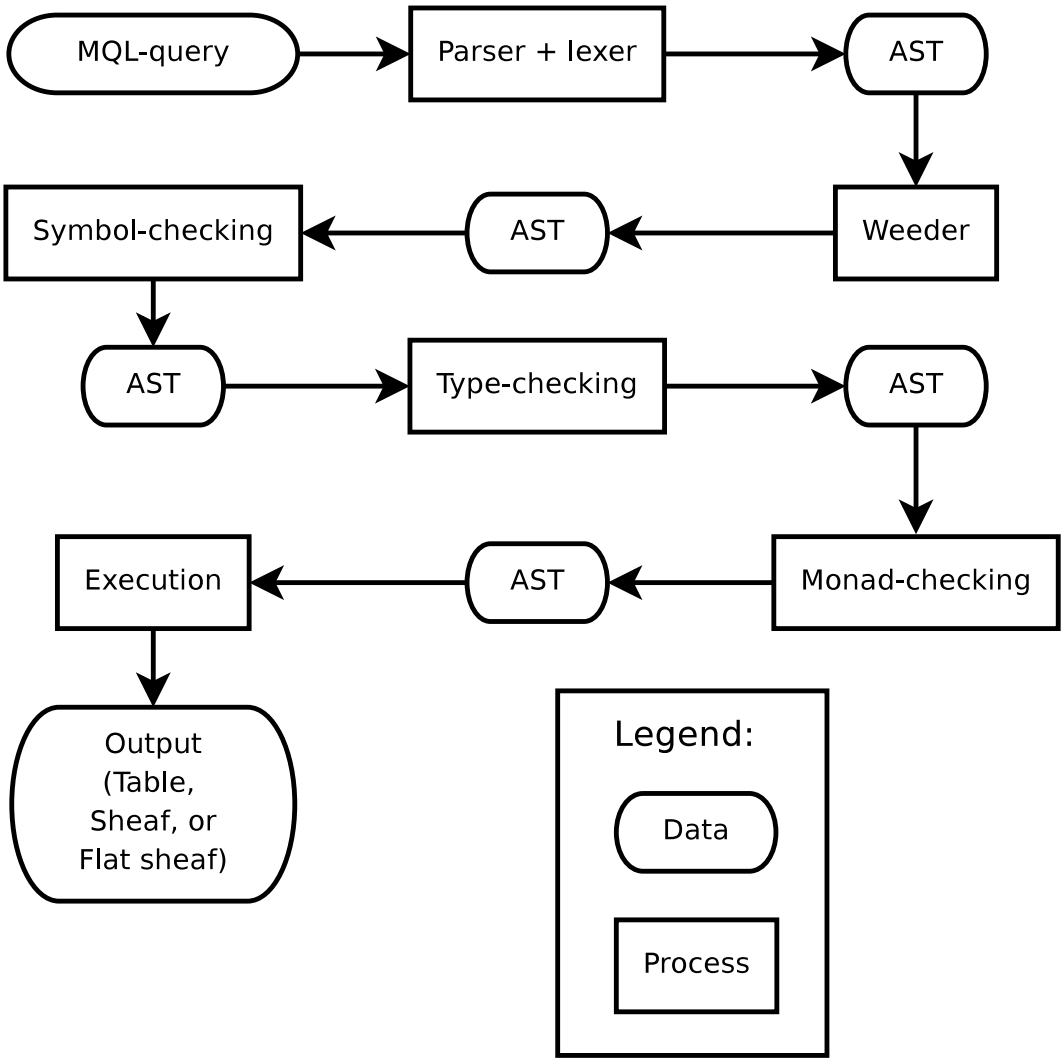



Figure 5.1: The data and compiler stages involved in the MQL interpreter. The process starts in the upper left hand corner, with an MQL query in raw text form. This is passed through the "parser" and "lexer" stage. They are counted as one stage here, for reasons discussed in the main body of the dissertation, even though they are usually counted as two stages. This produces an "AST", which stands for "Abstract Syntax Tree". The rest of the stages operate on this AST. The next stage is the Weeder, which "weeds out" certain parse-trees which are not well-formed for various reasons. The two next stages, Symbol-checking and Type-checking, also weed out parse-trees that do not meet certain criteria, specified in the MQL Programmer's Reference Guide [Petersen, 2007a]. The next stage, "monad checking", adds some information to the AST about monads, and may do some checking of monads. Finally, the query is executed in the Execution stage. The process may stop with an error at any point before the Execution stage.

parser generator, available from <http://www.sqlite.org/>. The formalism is a LALR(1) grammar, modeled after Backus-Naur Form. See Appel [1997] and Martin [1991] for introductions to the LALR(1) class of formal languages.

"Lexer" here means a program module which takes a character string (or character stream) as input, and produces tokens as output, ready to be converted by the parser into an abstract syntax tree.

That the parser drives the lexer is not the traditional order: The lexer usually drives the parser. This has been so at least since the Unix tools lex(1) and yacc(1) were invented (see also Appel [1997]).

The output of the parsing+lexing process is an "Abstract Syntax Tree", meaning a data structure which has the form of a tree (in the sense used by computer scientists), and which is an abstract representation of the pertinent parts of the parse tree deduced by the parser.

The AST now passes through a series of stages: The "weeder" checks that the parse tree is well-formed with respect to a number of constraints which are particular to certain MQL statements, the details of which need not concern us here. The "symbol checker", among other tasks, checks that all symbols (e.g., object types, features, enumerations, enumeration constants, etc.) do indeed exist — or do not exist — depending on the statement at hand. The "type checker" checks that all symbols have the right type. The "monads checker" checks for or builds certain monad sets for certain MQL statements. Finally the "Execution" stage executes the statement, based on the state of the AST after the "monads checking" stage has completed.

For any given kind of MQL statement, one or more of the stages may not actually do anything; it depends on the purpose of the statement, and what constraints exist on the statement.

This "pipeline" architecture is standard practice within the field of compiler- and interpreter-programming. See, e.g., Appel [1997] and Aho et al. [1985].

## 5.4 MQL output

The output of an MQL statement can be one of four things:

1. A table with rows and columns, where each column has a specific, specified type.

2. A "sheaf". A sheaf is the recursive data structure which is returned from a "topographic" query. The Sheaf is explored in Chapter 6.

3. A "flat sheaf". A flat sheaf is a sheaf-like data structure which is not recursive. It is also explored in Chapter 6.

4. Nothing. Some MQL statements do not return any output, but instead merely have some side-effect(s).

## 5.5 Type language

### 5.5.1 Introduction

We now finally arrive at the point where I begin to show how the MQL language meets Doedens's demands D11 and D12. In particular, this section deals with demand D11:

**D11. Type language:** We need a language in which we can define both object types and the features that those object types contain.

I do so by showing, in this section, how the MQL query language can be used to create, update, and delete the various kinds of types available in the EMdF model. The kinds of types fall neatly into four groups:

1. Databases
2. Enumerations
3. Object types and features
4. Named monad sets

There is a fifth kind of data type, which I will only discuss very briefly here. It is the data type called "index". It is used here in the sense used when speaking about a relational database system, i.e., an index on one or more columns of a table. The MQL query language supports the creation and deletion of indexes, both at a database-wide level (i.e., on all object types), and on individual features. The latter will be discussed below, but the former is an implementation detail which need not concern us here, since the details are trivial, and are only there for performance reasons.

I now discuss each of the four groups of data types listed above.

### 5.5.2 Databases

In MQL, databases can be created with the "`CREATE DATABASE`" statement, and can be deleted (or "dropped") with the "`DROP DATABASE`" statement. The user can connect to an existing EMdF database by issuing the "`USE DATABASE`" statement.

```
// Creates the database with the name munk_test,
// and populates it with empty meta-data tables
CREATE DATABASE 'munk_test'
GO

// Connects to the newly created database
USE DATABASE 'munk_test'
GO

// Drops (deletes) the database, without
// ever having filled it with any data.
DROP DATABASE 'munk_test'
GO
```

MQL follows the philosophy that one statement should do one thing, and do it well. That is, there should be one purpose of an MQL statement, and therefore, several MQL statements in combination (i.e., succession) may be needed in order to obtain a desired result. For example, the "CREATE DATABASE" statement does not immediately connect the user to the database; a separate "USE DATABASE" statement is needed for that. This is in line with the philosophy that one statement should do one thing, and do it well.[1]

### 5.5.3 Enumerations

As explained in Chapter 4, an "enumeration" is a set of pairs $(c_i, v_i)$, where each $c_i$ is an "enumeration constant" (or a label), and each $v_i$ is the integer corresponding to that label. All $v_i$'s are unique within any given enumeration.

In order to create an enumeration, the "CREATE ENUMERATION" statement is issued. This must be done before the enumeration is used to declare the type of a feature.

```
/* Implicit enumeration value assignment all the way.
   The first one is given the value 0, and the rest are
   given the value of the previous one, plus 1.
   For example, because ‘‘NOUN’’ is 0, ‘‘VERB’’ is 1,
   and because ‘‘VERB’’ is 1, ‘‘VERB_PAST’’ is 2. */
CREATE ENUMERATION
part_of_speech_e = {
   NOUN,
   VERB,
   VERB_PAST,
   VERB_PAST_PARTICIPLE,
   VERB_PAST_FINITE,
   ADJ,
   ADV,
   PREP,
   PROPER_NOUN,
   PRON_PERS
   PRON_POSS,
   PRON_INTER,
   PRON_RELA,
   CONJ
}
GO

CREATE ENUMERATION
voltage_e = {
   low = 0, // explicit value assignment
   high, // implicit value assignment; will be 0 + 1 = 1
   tristate // implicit value assignment; will be 1 + 1 = 2
}
```

[1] This philosophy is also the philosophy of the Unix operating system, and its utility has been argued many times by many over the years. See, for example, Raymond [2003].

The syntax for enumeration creation borrows both from SQL ("CREATE ENUMERATION") and from C/C++ (the 'enum_name = { /* enum-constants */ }' syntax).

It is also possible to update an enumeration:

```
UPDATE ENUMERATION
voltage_e = {
   REMOVE tristate,
   ADD highest = 2
}
GO
```

Finally, it is possible to delete (or "drop") an enumeration:

```
DROP ENUMERATION voltage_e
GO
```

The MQL interpreter will not allow this to happen if any feature on any object type uses the enumeration. This is an example of a constraint on an MQL statement which is checked in one of the compiler stages shown in Figure 5.1 on page 61, in this case, the "symbol-checking" stage. I have designed the statement this way because not having this constraint would leave the database in an inconsistent state, if some enumeration was still in use by some feature on some object type, after the enumeration had been dropped. This consideration is similar to Codd's requirement number (3) on a database model.[2]

### 5.5.4 Object types and features

In order to create an object type, the "CREATE OBJECT TYPE" statement must be issued. In its simplest form, it only takes an object type name. This will create an object type with no features (beyond the implicitly created "self" feature), no monad uniqueness constraints, and the "WITH MULTIPLE RANGE OBJECTS" object range constraint.[3] For example:

```
CREATE OBJECT TYPE
[verse]
```

It is possible to specify the object range type and the monad uniqueness constraints, as in the following examples:

```
CREATE OBJECT TYPE
WITH SINGLE MONAD OBJECTS
HAVING UNIQUE FIRST MONADS
[Token]
GO
```

[2] See Codd [1980] and Section 2.2.1 on page 29 of this dissertation.

[3] The reader is reminded that the "monad uniqueness constraint" and the "object range constraint" were introduced in Section 4.4.4 on page 50.

```
CREATE OBJECT TYPE
WITH SINGLE RANGE OBJECTS
HAVING UNIQUE FIRST AND LAST MONADS
[sentence]
GO

CREATE OBJECT TYPE
WITH MULTIPLE RANGE OBJECTS
// no monad uniqueness constraint here
[phrase]
GO

CREATE OBJECT TYPE
/* no object range constraint means
   'WITH MULTIPLE RANGE OBJECTS' */
[clause]
GO
```

In order to create a feature, the following abstract syntax is used:

*feature-name* : *feature-type* ;

This is done inside the [square brackets] of the `CREATE OBJECT TYPE` statement:

```
CREATE OBJECT TYPE
WITH SINGLE MONAD OBJECTS
HAVING UNIQUE FIRST MONADS
[Word
   surface : STRING;
   // The enumeration created before
   pos : part_of_speech_e;

   // FROM SET: uses a map from strings to integers
   lemma : STRING FROM SET;

   // Count of occurrences of this lemma in the DB.
   frequency : INTEGER;

   // Useful for syntax trees.
   parent : id_d;

   // Useful for directed acyclic syntax graphs
   parents : LIST OF id_d;

   // morph_e enum must have been declared
   morphology : LIST OF morph_e;
]
GO
```

In order to create an index on any given column in the table created for the object type in the relational database backend, the user employ the "WITH INDEX" keywords:

```
CREATE OBJECT TYPE
[Phrase
   phrase_type : phrase_type_e WITH INDEX;
   phrase_role : phrase_role_e WITH INDEX;
]
```

An already-created object type may be updated by issuing an "UPDATE OBJECT TYPE" statement. This has the effect of adding or removing columns to the table holding the objects. Default values must be given for each feature added. If no default values are given, then standard default values are used.

```
UPDATE OBJECT TYPE
[Phrase
   // Too interpretative... We shouldn't get into semantics
   // before we have mastered the syntax of this language!
   REMOVE phrase_role;

   // Ahhhh... Much more formal!
   ADD phrase_function : phrase_type_e DEFAULT Unknown;
]
```

An already-created object type may be deleted (or "dropped") with the "DROP OBJECT TYPE" statement. Only the name of the object type is needed:

```
DROP OBJECT TYPE
[verse]
```

Taken together, the creation, update, and deletion statements available for enumerations and object types meet the requirements of demand D11. However, creation, update, and deletion are not the only possible operations on these data types. It is also possible to *retrieve* the enumerations, enumeration constants, object types, and features present in any EMdF database. We shall return to this retrieval facility in Section 5.6.

## 5.6 Data language (non-topographic)

### 5.6.1 Introduction

In this section, I show how the data domains present in the EMdF model can be created, updated, deleted, and retrieved. I have appended the label "non-topographic" to the heading of this section because the topographic part of MQL is so large that it deserves its own section (5.7).

In this section, I first discuss how objects are created, updated, deleted, and retrieved (non-topographically). I then discuss the same for "named monad sets". I then discuss "retrieval of object types and features".

### 5.6.2 Objects

Objects can be created either individually or in bulk (batch-processing). The latter is much more efficient than the former if many objects are to be inserted at the same time.

In order to insert objects individually, the "CREATE OBJECT" statement must be issued.[4]

```
CREATE OBJECT
FROM MONADS = { 20001-20006 } // set of monads
[verse]
GO

// An explicit id_d can be given
CREATE OBJECT
FROM MONADS = { 20001 }
WITH ID_D = 1
[word
  surface := 'Hvad';
  part_of_speech := PRON_INTER_REL;
  lemma := 'hvad';
]
GO

/* One can also specify a list of id_ds instead
   of monads. Then the monad set will be the big-union
   of the monad sets belonging to the objects which have
   the id_ds given. */
CREATE OBJECT
FROM ID_DS = 1,2,3,4,5,6
[verse]
GO
```

In order to insert objects in bulk, the "CREATE OBJECTS WITH OBJECT TYPE" statement must be used:

```
CREATE OBJECTS WITH OBJECT TYPE [verse]
CREATE OBJECT FROM MONADS = { 20001-20006 }
WITH ID_D = 11
[] // Notice how the object type is left out
CREATE OBJECT FROM MONADS = { 20007-20010 }
WITH ID_D = 12
[]
GO
```

Objects with specific id_ds may be updated:

[4] Almost all examples in this section are drawn from the EMdF database given in Figure 4.1 on page 57.

```
UPDATE OBJECTS
BY ID_DS = 2,6
[Word
   // As opposed to V_PAST_PARTICIPLE
   part_of_speech := V_PAST_FINITE;
]
GO
```

Or one can specify a set of monads, and all objects of the given type which fall wholly within that set of monads will be updated.

```
// Only
UPDATE OBJECTS
BY MONADS = { 20004 }
[Word
   surface := 'dog';
]
GO
```

Objects may be deleted, either by monads or by id_ds:

```
// Delete all objects of the given type
// which fall wholly within the given monad set
// (in this case, the words of the first verse).
DELETE OBJECTS BY MONADS = { 20001-20006 }
[word]
GO

// This will delete all the words in the second verse.
DELETE OBJECTS BY ID_DS = 7,8,9,10
[word]
GO
```

Objects may be retrieved in a number of ways. The first and foremost is the topographic query "SELECT ALL OBJECTS", discussed in Section 5.7. The rest, more ancillary ways, are discussed here, since they are non-topographic. The first is "SELECT OBJECTS AT", which selects objects of a given type which start at a given monad.

```
// Will retrieve the word ''Hvad''.
SELECT OBJECTS AT MONAD = 20001
[word]
```

It is also possible to retrieve objects which overlap with a given set of monads, i.e., to select objects which have at least one monad in common with a given set of monads.

```
// Will retrieve the verbs ''var'' and ''skete''.
SELECT OBJECTS HAVING MONADS IN { 20002, 20006 }
[word]
GO
```

```
// Will retrieve both verses, since they both
// have at least one monad in common
// with the monad set given.
SELECT OBJECTS HAVING MONADS IN { 20005-20007 }
[verse]
```

The "SELECT OBJECTS HAVING MONADS IN" statement returns a table, with no possibility of retrieving the features or the full monad sets of objects. If one wishes to retrieve features as well as full monad sets, the "GET OBJECTS HAVING MONADS IN" statement is useful:

```
// Retrieves the words ''Hvad'', ''var'', ''det'',
// ''der'', and ''skete''. Will return a flat sheaf
// containing the objects, their id_ds, their full monad
// sets, and the two features 'surface' and 'part_of_speech'.
GET OBJECTS HAVING MONADS IN { 20001-20003, 20005-20006 }
[word GET surface, part_of_speech]
GO
```

It is also possible to retrieve the monads of objects separately:

```
GET MONADS
FROM OBJECTS WITH ID_DS = 1,2
[word]
```

It is also possible to retrieve the features of objects separately:

```
GET FEATURES surface, part_of_speech
FROM OBJECTS WITH ID_DS = 1,2
[word]
```

The "GET OBJECTS HAVING MONADS IN" statement was designed to combine the three statements "SELECT OBJECTS HAVING MONADS IN", "GET MONADS", and "GET FEATURES". The reason for introducing the "GET OBJECTS HAVING MONADS IN" statement was very simple: The three statements which it combines were very inefficient to run in succession, if one really wanted all three. Thus, although MQL was designed with the Unix-like philosophy that each statement must "do one thing, and do it well", the level of granularity at which this philosophy must be applied is open for debate, especially when performance considerations are at play.

This concludes the list of things which it is possible to do with objects in a non-topographic way. We shall return to topographic querying of objects in Section 5.7.

### 5.6.3 Monads

In this subsection, I discuss how min_m, max_m, and named monad sets may be manipulated, starting with the last.

Named monad sets were introduced in Section 4.4, and are an innovation on my part with respect to the MdF model. The purpose of named monad sets is to allow for easy

restriction of a topographic query to a specific part of the database, by supporting the use of named monad sets. The monad sets must, of course, be declared before they are used in a topographic query; otherwise, the "symbol-checking" stage of the interpreter will complain that the named monad set referenced by the topographic query does not exist.

A named monad set can be created with the "CREATE MONAD SET" statement. The examples are from a Biblical setting, namely the WIVU database [Talstra and Sikkel, 2000, Dyk, 1994, Dyk and Talstra, 1988, Hardmeier and Talstra, 1989, Talstra and Van Wieringen, 1992, Talstra et al., 1992, Talstra, 1992, 1989, 2002a, n.d., 2002b, 1998, 1997, Talstra and van der Merwe, 2002, Verheij and Talstra, 1992, Talstra and Postma, 1989, Winther-Nielsen and Talstra, 1995].

```
// From a Biblical setting,
// namely the WIVU database.
CREATE MONAD SET
Towrah
WITH MONADS = { 1-113226 }
GO

CREATE MONAD SET
Historical_books
WITH MONADS = { 113300-212900 }
GO

// Oops... also includeds non-Jeremiah books...
// this will be corrected below,
// under the discussion of UPDATE MONAD SET
CREATE MONAD SET
Jeremiah_Corpus
WITH MONADS = { 236000-368444 }
GO
```

A named monad set can also be updated using various monad set operations:

```
// Add the set of monads given
UPDATE MONAD SET
Historical_books
UNION
{ 110000-113250 }
GO

// Subtract the set of monads given
UPDATE MONAD SET
Historical_books
DIFFERENCE
{ 110000-113299 }
GO
```

```
// Replace it with the intersection
UPDATE MONAD SET
Historical_books
INTERSECT
{ 113300-212900 }
GO

// Replace 'Jeremiah_Corpus' with the monad set given
UPDATE MONAD SET
Jeremiah_Corpus
REPLACE
{ 236000-265734, 366500-368444 }
GO
```

It is, of course, also possible to delete (or drop) a monad set:

```
DROP MONAD SET
Jeremiah_Corpus
GO
```

It is also possible to retrieve the names of the monad sets present in an EMdF database:

```
SELECT MONAD SETS
GO
```

And in order to get a table listing the monad of either all monad sets, or specific monad sets, the "`GET MONAD SETS`" statement can be issued:

```
// Retrieves all monad sets
GET MONAD SETS ALL
GO

// Just retrieves one
GET MONAD SET Historical_books
GO

// Retrieves monad sets 'Towrah' and 'Historical_books'
GET MONAD SETS Towrah, Historical_books
GO
```

It is also possible to retrieve the least monad used by any object (min_m), and the greatest monad used by any object (max_m):

```
SELECT MIN_M
GO

SELECT MAX_M
GO
```

Both named monad sets, min_m, and max_m are innovations on my part with respect to the MdF model.

### 5.6.4 Retrieval of object types and features

The names of the object types available in an EMdF database can be retrieved with the "SELECT OBJECT TYPES" statement. It returns a table listing the object type names.

```
SELECT OBJECT TYPES
GO
```

The question can also be posed, which features a given object type has, using the "SELECT FEATURES" statement:

```
SELECT FEATURES
FROM OBJECT TYPE
[Word]
GO
```

Again, this will return a table containing one row for each feature, giving the feature name, its type, its default value, and a boolean saying whether it is a computed feature[5] or not.

A table containing the names of the enumerations present in an EMdF database can be retrieved with the "SELECT ENUMERATIONS" statement:

```
SELECT ENUMERATIONS
GO
```

Similarly, the enumeration constants present in a given enumeration can be retrieved as a table:

```
SELECT ENUMERATION CONSTANTS
FROM ENUMERATION part_of_speech_e
GO
```

To find out which object types have features which use a certain enumeration, the "SELECT OBJECT TYPES USING ENUMERATION" statement can be issued:

```
SELECT OBJECT TYPES
USING ENUMERATION part_of_speech_e
GO
```

[5] Only the "self" feature is said to be computed, even though it, too, is stored. Computed features is an area of further research; the intent is to provide a mechanism for specifying features as functions which operate on an object and return a value based on the stored values associated with the object. This has not been implemented yet.

| | Database | Enumeration (D11) | Object Type (D11) | Feature (D11) | Object (D12) | Monad set (D12) |
|---|---|---|---|---|---|---|
| Create | + | + | + | + | + | + |
| Retrieve | - | + | + | + | + | + |
| Update | N/A | + | + | + | + | + |
| Delete | + | + | + | + | + | + |

Table 5.1: Operations implemented on the data domains in the EMdF model.
"+" means "implemented", "-" means "not implemented", and "N/A" means "Not Applicable".

### 5.6.5 Conclusion

As should be evident from my description of the various data domains in the EMdF model, I have implemented complete support for all create, update, delete, and retrieve operations on all data domains, except for "databases". Creation of databases is supported, as is deletion. Update of databases can be said to occur at lower levels using the `CREATE` and `UPDATE` statements meant for object types, enumerations, objects, and named monad sets. However, retrieval of database names is not supported: The names of the EMdF databases present on a given system cannot be retrieved within the Emdros implementation; it must either be known by the calling application, or it must be retrieved from the system by other means.

Table 5.1 shows which operations (create, retrieve, update, delete) are available for which data domains. As can be seen, demands D11 and D12 are fully met.

So far, I have described the non-topographic data language part of MQL. I now turn to the topographic part, which has been modeled after Doedens's QL.

## 5.7 Data language (topographic)

### 5.7.1 Introduction

The topic of my B.Sc. thesis[6] was two-fold: a) A sketch of an implementation of what was the "EMdF" model as I saw it then, and b) An operational semantics for a subset of the topographic part of "MQL" as I saw it then. Since my B.Sc., I have expanded MQL in numerous ways, becoming what it is today.

In the interest of being transparent about what has been written for other degrees, and what has been written for my PhD, this section is structured as follows. First, I describe MQL as it was in my B.Sc. thesis. Then, I describe how the present-day MQL differs from the MQL of my B.Sc. thesis. Finally, conclude the section.

### 5.7.2 The MQL of my B.Sc. thesis

The MQL of my B.Sc. thesis was very limited in its expressivity, compared to the present-day MQL. Only the core of Doedens's QL was present. The operational semantics given in my B.Sc. thesis was given in a PASCAL-like language, but glossed over a lot of details, as I later found when I came to implementing it.

[6] See Petersen [1999]. The thesis is available online: http://ulrikp.org/studies.html

For the purposes of the discussion in this Chapter, let us call the MQL of my B.Sc. thesis "BMQL" (for "Bachelor MQL"), whereas the present-day MQL will be called just "MQL".

In BMQL, as in MQL, the core notion is that of "block". In BMQL, there were four kinds of blocks:

1. object_block_first,
2. object_block,
3. opt_gap_block, and
4. power_block.

I now describe these in turn.

#### 5.7.2.1 object_block and object_block_first

An object_block_first and an object_block both correspond to (and thus match) an object in the database. They both look like this in their simplest form:

```
[Clause]
```

That is, an object_block (or object_block_first) consists of an object type name in [square brackets].

An object_block (or object_block_first) can have a number of modifiers:

First, it is possible (in BMQL) to specify that an object should not be retrieved, or to specify explicitly that it should be retrieved:

```
[Phrase retrieve]
[Phrase noretrieve]
```

The first Phrase above will be retrieved, whereas the second will not.

Second, it is possible (in BMQL) to specify that an object_block may optionally be "first" or "last" in its string of blocks (a block_string). An object_block_first, on the other hand, may optionally be specified as being "first", but not "last". The precise meaning of "first" and last can only be specified with reference to the "substrate", so we return to their meaning below, when we have defined "substrate". For now, let us continue to describe object_blocks and object_block_firsts.

Third, it is possible (in BMQL) to specify an arbitrarily complex predicate in a subset of First-Order-Logic (FOL), namely FOL without quantifiers, but with AND, OR, NOT, and parentheses as connectives, and with feature-value equality-comparisons ("=") as atomic predicates (e.g., "`phrase_type = NP`"). Other comparison-operators than equality are *not* possible in BMQL. An example of a legal query could be:

```
[Phrase (phrase_type = NP AND phrase_function = Subject)
        OR (phrase_type = AdjP
            AND phrase_function = Predicate_Complement)
]
```

Fourth, it is possible (in BMQL) to specify a set of variable assignments, which makes it possible to refer back to the values of features of an object matched by an object_block(_first) further up towards the top in the query. For example:

```
var $c,$n,$g;
[Word
     $c := case;
     $n := number;
     $g := gender;
]
[Word
     case = $c AND number = $n AND gender = $g
]
```

This string of blocks (technically, a block_string) would find all pairs of Words whose case, number, and gender features agreed. The declaration "`var $c,$n,$g;`" at the top is necessary in BMQL in order to declare the variables, and must be present before the block_string starts, if any variables are used within the block_string.

Finally, the fifth modifier which may be specified on an object_block or object_block_first is an *inner blocks*. A "blocks" is a sequence of an optional variable declaration (such as was seen above), followed by a block_string. For example:

```
[Clause
    [Phrase phrase_function = Modifier]
]
```

This would find all clauses which had at least one phrase inside of the clause whose phrase_function was Modifier.

We now turn to the notion of "substrate".

#### 5.7.2.2 Substrate

The *substrate* of an MQL/BMQL query is a set of monads which defines the context within which the blocks of a block_string (i.e., a string of blocks at the same level in the query) will have to match. The idea of the substrate is that there must always exist some limiting, finite set of monads within which to execute the query at a given level of nesting.

For the outermost block_string, the substrate is given (in BQML) by the set of monads corresponding to the single object present in the all_m object type (see Section 4.3.4 on page 47), that is, the substrate is the set of monads formed by the big-union of all sets of monads from all objects in the database.

When the context of a block_string is an object_block or object_block_first, then the substrate of the inner block_string is the set of monads from the object which matched the outer object_block or object_block_first. For example:

```
[Sentence
   [Clause]
   [Clause]
]
```

In this example, the sets of monads of the two inner Clause objects must both be subsets of the set of monads belonging to the outer Sentence object. The two clause objects must also stand in sequence, at least with respect to the set of monads belonging to the outer sentence object. This means that, if there are no gaps in the set of monads belonging to the Sentence object, then, if the first Clause object ends on monad *b*, then the second Clause must start on monad $b+1$. If, however, there are gaps in the set of monads belonging to the outer Sentence object, then the rules are slightly different. Let us say, without loss of generality (wlog), that the first clause ends on monad *b*. If there then is a gap in the outer Sentence which starts at monad $b+1$ and ends at monad *c*, then the second clause *must* start on monad $c+1$.

This special rule may be circumvented by placing an exclamation mark ('!') between the two clauses:

```
[Sentence
   [Clause]
  ![Clause]
]
```

In that case, the two clauses must stand next to each other, that is, if the first clause ends on monad *b*, then the second clause *must* start on monad $b+1$.

The substrate also defines what the keywords "first" and "last" mean on an object_block or object_block_first: The keyword "first" means that first monad of the set of monads belonging to the object matched by the the object_block or object_block_first must be the same as the first monad of the substrate. Similarly, the keyword "last" means that the last monad of the set of monads belonging to the object matched by the object_block must be the same monad as the last monad of the substrate.

#### 5.7.2.3 opt_gap_block

An opt_gap_block corresponds to (and thus matches) a gap in the substrate of the context.

```
[Sentence
   [Clause]
   [gap? noretrieve] // this is the opt_gap_block
   [Phrase]
]
```

In this example, the outer sentence provides the substrate for the inner block_string. After the Clause, there may optionally be a gap in the substrate (i.e., the set of monads of the surrounding Sentence object). After the gap, a Phrase must exist. The gap is not retrieved if it is found (it would have been retrieved if the "`retrieve`" keyword had been used instead of "`noretrieve`"). If there is no gap, then the Phrase must exist right after the Clause (i.e., if the Clause ends on monad *b*, the Phrase must start at monad $b+1$).

#### 5.7.2.4 power_block

A power_block corresponds to "an arbitrary stretch of space" in the surrounding substrate. It is signified by two dots ('. .'). For example:

```
[Phrase
   [Word first]
   ..
   [Word last]
]
```

This would retrieved all Phrases inside of which we found a Word (which must be “first” in the Phrase), followed by arbitrary space (even 0 monads), followed by a Word (which must be “last” in the Phrase). The significance of the “..” power block is here that it allows finding Phrases that are more than two words long, even if they have no gaps. Had the query been:

```
[Phrase
    [Word first]
   ![Word last]
]
```

then the two words must take up the whole Phrase. The power_block, on the other hand, makes it possible to say that the Phrase may be arbitrarily long, without specifying what must be between the two Word object blocks.

Finally, BMQL allows a limiting number of monads to be specified on the power block. For example:

```
[Word part_of_speech = article]
.. < 3
[Word part_of_speech = noun]
```

In this example, there may be at most 3 monads between the article and the noun. The limitor is handy, especially if using the “..” operator on the outermost level. For example, the query:

```
[Word]
..
[Word]
```

would take an awfully long time to execute, and would consume an awful amount of RAM memory, for a decent-sized database. The reason is that the query means the following: Find all pairs of words, where the second word appears at some unspecified point after the first word. In general, the number of possibilities for this particular query is given by the following formula:

$$S = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$$

where $n$ is the number of Words in the database.

This gets quite large for decent-sized databases. For example, in a database of 400,001 words, the sum is 8,000,200,000 — or eight billion, two hundred thousand. Thus the restrictor on power blocks, though small in stature, can be put to very good use.

This concludes my summary of MQL as it was in my B.Sc. thesis.

### 5.7.3 The present-day MQL

Several innovations and/or enhancements have appeared in MQL since my B.Sc. work. In this section, I detail the most important ones.

#### 5.7.3.1 Kinds of blocks

The set of kinds of block has been expanded. The full set is now:

1. object_block
2. power_block
3. opt_gap_block
4. gap_block
5. NOTEXIST_object_block

Of these, number (4) and (5) are new. A gap_block is almost the same as an opt_gap_block, except that the gap *must* be there for for the query to match as a whole. A NOTEXIST_object_block is the same as an object_block, except that for it to match, no objects of the kind specified, and with any feature-restrictions specified, may exist within the substrate, counting from the point in the query in which the NOTEXSIT_object_block is located. Where an object_block specifies that something must exist (a $\exists$ existential quantifier), the NOTEXIST_object_block specifies that something must not exist (a $\forall\neg$ "for all"-quantifier, with inner negation).

The distinction between object_block and object_block_first has been done away with in the "new MQL".

#### 5.7.3.2 First and last

Even though the *syntactic* distinction between object_block and object_block_first has been done away with in the "new MQL", the concept has not. It has just moved from the "parsing" stage to the "weeding" stage of the interpreter. In addition, it is now possible to specify that some object must be bother first *and* last in its context:

```
SELECT ALL OBJECTS
WHERE
[Phrase
    [Word  FIRST AND LAST  pos=verb]
]
```

Here, the inner Word (which is a verb) must occupy the whole Phrase.

#### 5.7.3.3 Focus

It is possible, in the new MQL, to specify that a given object block, gap_block, or opt_gap_block must be "in focus", with the FOCUS keyword:

```
SELECT ALL OBJECTS
WHERE
[Clause
   [Word pos=verb]
   ..
   [Word FOCUS pos=noun]
   ..
   [Word pos=adjective]
]
```

This would find clauses with a verb, followed by arbitrary space (".."), followed by a noun, followed by arbitrary space, followed by an adjective. The noun would be "in focus", i.e., the "focus boolean" for that Word would be true in the resulting sheaf, whereas it would be "false" for all other objects (because they did not have the FOCUS keyword).

This is useful in many cases. For example, as argued in [FSMNLP2005], it is sometimes necessary for the user to give the corpus query system a "hint" as to what part of the query they are really interested in. This can be done with the "FOCUS" keyword.

Note that the "focus boolean" is never interpreted inside of Emdros. The interpretation of the focus boolean always lies in the application layer above the MQL layer (see [LREC2006] for an explanation of the architecture behind Emdros).

#### 5.7.3.4 Marks

Besides "FOCUS" booleans, it is possible to add a further kind of information to the query, which can be passed upwards to the application-layer. This was first discussed in Doedens [1994], and so the idea originates there. The idea is to allow the user to "decorate" a block (either an object_block, an opt_gap_block, or a gap_block) with C-identifiers which are passed back up to the application layer. The application layer is then free to either ignore the "marks", or to interpret them in a way which is fitting to the purpose at hand. For example, in a graphical query application, the following might color parts of the display:

```
SELECT ALL OBJECTS
WHERE
[Clause‘red
   [Phrase‘blue function=Pred]
]
```

This would attach the mark "red" to the Clause objects, and the mark "blue" to the Phrase object in the resulting sheaf. The graphical query application might then interpret these as instructions to color the Clause red, except for the Predicate Phrase inside the Clause, which must be colored blue.

This is also useful for specifying what is "meat" and what is "context" (cf. the extended discussion about this in [FSMNLP2005]):

```
SELECT ALL OBJECTS
WHERE
[Clause‘context‘red
   [Word‘context pos=verb]
   ..
   [Phrase‘meat function=Objc]
   ..
   [Phrase‘context function=Objc]
]
```

This would find all verbal clauses with double objects, where the first object was “meat”, and the second object was “context”. Note also how the Clause object has two marks, namely “context” and “red”. There is no limit on the number of marks a block can have, except that power blocks cannot have marks.

#### 5.7.3.5 New comparison-operators on features

In “BMQL”, it was only possible to use the equality (“=”) operator on features. In the new MQL, a range of other comparison-operators have been defined. The full set is now:

= Denotes equality

<> Denotes inequality

< Denotes “less than”

<= Denotes “less than or equal”

> Denotes “greater than”

>= Denotes “greater than or equal to”

**IN** Denotes “feature has to have a value IN the following comma-separated, parentheses-surrounded list of values”. For example: “pos IN (noun,article,adjective)”

**HAS** Denotes “this list-feature has to have the following value”. For example: “morphology HAS genitive”

~ Denotes “this feature must match the following regular expression”

**!~** Denotes “this feature must not match the following regular expression”

Note that the IN and HAS operators were added so as to accommodate comparisons with lists of values. Lists of values are new in the EMdF model since my B.Sc. thesis, and were described in Section 4.4.5 on page 51.

#### 5.7.3.6 Object references replace variables

In "BMQL", it was possible to define variables, assign feature-values to these variables, and then refer back to the variables later in the query.

In the "new MQL", variables have been replaced with the concept of "object references." The concept can best be explained by reference to a concrete query. For example:

```
SELECT ALL OBJECTS
WHERE
[Phrase phrase_type = NP
   [Word AS w1 pos=noun]
   [Word pos=adjective
         AND case = w1.case
         AND number = w1.number
         AND gender = w1.gender
   ]
]
```

This would find all NP phrases inside of which there was a noun, followed by an adjective which agreed with the noun in case, number, and gender. The "AS w1" incantation is an *object reference declaration*. It declares that the first Word must be referred to by the C-identifier "w1". Then, in the second word, the feature-comparisons can refer back to "w1" with the "dot notation": For example, "w1.case" means "The value of the w1-object's case-feature."

This is very useful, not just in ensuring agreement, but also in other cases, such as ensuring parentage. For example:

```
SELECT ALL OBJECTS
WHERE
[Clause AS c1
   [Phrase parent = c1.self]
]
```

Here, the "parent" feature of the inner Phrase is checked to see if it is the same as the outer Clause's "self" feature, i.e., whether the inner Phrase's "parent" feature "points to" the outer Clause object.

I find that this notation is less cumbersome, more convenient to type, and more intuitive than the concept of variables as developed in BMQL, even though they are largely equivalent.

#### 5.7.3.7 OR between strings of blocks

It is now possible in MQL to specify an "OR" relationship between two strings of blocks. For example:

```
SELECT ALL OBJECTS
WHERE
[Clause
   [Phrase function=Subj]
```

```
   [Phrase function=Pred]
   [Phrase function=Objc]
   OR
   [Phrase function=Pred]
   [Phrase function=Subj]
   [Phrase function=Objc]
]
```

This would find clauses in which either the order of constituents was Subject-Predicate-Object, or Predicate-Subject-Object. In other words, the "OR" construct between strings of blocks supports, among other uses, permutation of the constituents. In reality, any string of blocks may stand on either side of the OR, not just permutation of the constituents. In addition, there can be as many OR-separated strings of blocks as necessary. For example:

```
SELECT ALL OBJECTS
WHERE
[Clause
   [Phrase function=Subj]
   [Phrase function=Pred]
   [Phrase function=Objc]
   OR
   [Phrase function=Pred]
   [Phrase function=Subj]
   [Phrase function=Objc]
   OR
   [Phrase function=Objc]
   [Phrase function=Subj]
   [Phrase function=Pred]
]
OR
[Clause
   [Word pos=verb]
   [Phrase phrase_type=NP]
   [Phrase phrase_type=AdjP]
]
[Clause
   [Word pos=verb]
   [Word pos=adverb]
   [Phrase phrase_type=NP
     [Phrase phrase_type=AdjP]
     [Phrase phrase_type=NP]
   ]
]
```

#### 5.7.3.8 Grouping

It is now possible in MQL to group strings of blocks, like this:

```
[Clause
   [
     [Phrase]
     [Phrase]
   ]
]
```

The reason for including this construct in the language will become apparent in the next section.

#### 5.7.3.9 Kleene Star

It is now possible to apply a "Kleene Star" to either an object block or a group of blocks. For example:

```
SELECT ALL OBJECTS
WHERE
[Clause
   [Word pos=verb]
   [Word
         pos IN (noun,article,adjective,conjunction)
   ]* // Note star!
   [Word pos=adverb]
]
```

This would find all Clauses, inside of which we find a verb, followed by zero or more words whose part of speech is either noun, article, adjective, or conjunction, followed by an adverb.

The Kleene Star can also optionally take a set of integers, denoting the number of times that the Kleene Star must repeat:

```
SELECT ALL OBJECTS
WHERE
[Phrase phrase_type=NP
    [Word FIRST pos=article]*{0,1}
    [Word LAST pos=noun]
]
```

This would find all NPs of either 1 or 2 words in length, where the last word was a noun, and (if its length was 2), the first word would be an article. This obviously allows the specification of optional blocks.

It is also possible for the set of integers to have gaps. It is also possible for it to have "no upper limit":

```
SELECT ALL OBJECTS
WHERE
[Clause
   [Phrase function=Pred]
```

```
   [Phrase
           function IN (Subj,Objc,Cmpl,Adju)
   ]*{2-}
]
```

This would find all Clauses inside of which there was a Predicate Phrase, followed by two or more Phrases whose function was either Subject, Object, Complement, or Adjunct. Note that there is no integer after the "-" dash, denoting that there is no upper limit. Notice that, even though there is no upper limit, the Kleene Star is limited by the surrounding context, in this case, the boundaries of the Clause. Therefore, the program code calculating the results will always terminate.

It is possible for the Kleene Star to apply to groupings:

```
SELECT ALL OBJECTS
WHERE
[Phrase phrase_type=NP
   [Word FIRST pos=article]
   [Word pos=noun]
   [
      [Word pos=conjunction]
      [Word pos=article]
      [Word pos=noun]
   ]*{1-} // Note the star!
]
```

This would find all NP Phrases which consisted of an article as the first word, followed by a noun, followed by one or more iterations of the sequence "conjunction", "article", "noun". An example in English would be "The duck and the hen and the eagle".

This concludes my summary of the most important changes to MQL language since the MQL of my B.Sc. thesis.

### 5.7.4 Conclusion

In this section, I have shown what MQL is today. I have done so by first discussing what MQL looked like in my B.Sc. thesis, followed by a discussion of how the MQL of today differs from the MQL of my B.Sc. thesis. The differences are not negligible, since the expressive power of the language has increased dramatically since the MQL of my B.Sc. thesis. In particular, the "OR" construct between strings of blocks, the "grouping" construct, and the "Kleene Star" bring a lot of new expressive power, as do the new comparison operators. The "gap_block" and "NOTEXIST_object_block" are new, and the latter brings the expressivity of the language up to the level of full First Order Logic over EMdF objects, since it adds the "forall" quantifier" in addition to Fthe already-implemented "exists" quantifier. The possibility of adding "FOCUS" booleans and "marks" to the resulting sheaf opens new possibilities for allowing the user to distinguish between what is "meat" and what is merely "context" in the query (cf. [FSMNLP2005]).

Lai [2006] is a model-theoretical and logical analysis of the requirements that treebanks place on query languages designed for querying treebanks. Lai concludes that the

full expressiveness of First Order Logic (FOL) over tree-structures is required for querying treebanks in all relevant ways. In the following sense, I have fulfilled this requirement by building the present-day MQL: MQL supports the full FOL over objects, with existential *and* "forall" quantification over objects, and a full set of Boolean operators on the attributes (i.e., features) of objects. Thus both the atoms, the predicates, the Boolean connectives, the quantifiers, and the formulae of FOL are covered by MQL, defined over the objects and features of the EMdF model. While this does not constitute FOL over tree-structures, it is a step in the right direction.

Yet for all its expressive power, today's MQL still leaves a lot to be desired, compared with Doedens's QL. I discuss some of the possible future enhancements in Chapter **??**.

## 5.8 Conclusion

In this chapter, I have discussed the MQL query language. MQL, in contrast to Doedens's QL, is a "full access language", meaning it has statements for create, retrieve, update, and delete operations on the full gamut of data domains in the EMdF model. Doedens's QL was strictly a "data retrieval" language, and only operated with topographic queries. Thus I have fulfilled Doedens's demands D11 and D12.

I started the chapter by recapitulating demands D11, D12, and D13 from Doedens [1994]. I then promised to show, in this chapter, how demands D11 and D12 were fully met. I then had three sections which paved the way for fulfilling this promise, namely a section on "general remarks" (discussing lexical conventions and comments), a section on "the MQL interpreter" (discussing how the MQL interpreter is implemented), and a section on "the resulting output" (discussing the four kinds of output possible from an MQL query).

I then discussed how MQL meets the requirements of Doedens's demands D11 and D12 in three sections: First, I discussed the type language of MQL, thus showing how MQL fulfills D11. Second, I discussed the non-topographic "data language" part of MQL, which enables the creation, retrieval, update, and deletion of objects and monad sets, as well as the retrieval of the names of object types, features, and enumerations. Third, I discussed the topographic "data language" part of MQL, which resembles Doedens's QL in numerous ways, yet is also distinct from QL. Taken together, the non-topographic "data language" part, and the topographic "data language" part of MQL entail that MQL fully meets Doedens's demand D12.

Meeting demand D13 is a future research goal. I shall return to this in Chapter 14.

# Chapter 6

# The Sheaf

## 6.1 Introduction

The datastructure known as a "Sheaf" was first introduced by Doedens [1994]. I have modified the data structure to be able to implement it in practice, much as I modified the MdF model to make it implementable in practice.

In this Chapter, I discuss the Sheaf, what it is and what it is for (6.2). I then discuss the parts of the Sheaf and how they relate to the query from which it originated (6.3). Finally, I conclude the chapter.

## 6.2 What is a Sheaf?

### 6.2.1 Introduction

A sheaf is a data structure which is the result of either a topographic query or a `GET OBJECTS HAVING MONADS IN` query. As mentioned in the previous section, the Sheaf was invented by Doedens, and was described in Section 6.5, pp. 138–147 of Doedens [1994]. I have modified the sheaf in order to be able to implement it, and in order to make it suit the EMdF model and the MQL query language.

The Sheaf is a recursive data structure, by which I mean that a part of a Sheaf may contain an "inner sheaf", which then again may contain a part that contains an "inner sheaf", and so on, until there is an "inner sheaf" which is empty, and thus does not have any more inner sheaves.

There are two kinds of Sheaf:

1. Full Sheaf (which corresponds to Doedens's Sheaf), and

2. Flat Sheaf (which is a Sheaf with only empty inner sheaves).

I shall first define the Full Sheaf, and then define the Flat Sheaf in terms of the constructs found in the Full Sheaf.

### 6.2.2 Sheaf Grammar

A Full Sheaf either is failed, is empty, or has a comma-separated list of one or more Straws (we'll get to the definition of Straw in a moment). In Backus-Naur Form[1]:

```
Sheaf ::= "//"            /* failed sheaf */
      |   "//" "<" ">"    /* empty sheaf */
      |   "//" "<" Straw { "," Straw }* ">"
;
```

A Straw is a string of one or more Matched_objects:

```
Straw ::= "<" Matched_object { "," Matched_object }* ">"
;
```

A Matched_object may be of two kinds: an MO_ID_D or an MO_ID_M. An MO_ID_D has its origin in an object block, wheras an MO_ID_M has its origin in either an opt_gap_block or a gap_block.

```
Matched_object ::= /* MO_ID_D */
                   "[" Object_type Id_d Set_of_monads
                       Marks Focus_boolean Feature_values
                       Sheaf
                   "]"
               |   /* MO_ID_M */
                   "[" "pow_m" Set_of_monads
                       Marks Focus_boolean
                       Sheaf
                   "]"
;
```

The "Object_type" is an IDENTIFIER, that is, a C-identifier, just as in the EMdF model.

```
Object_type ::= IDENTIFIER   /* C-identifier */
;
```

An "Id_d", as used in the MO_ID_D, is simply an integer, representing the id_d of the object from which the MatchedObject arose:

```
Id_d ::= INTEGER             /* Any integer */
;
```

[1] Backus-Naur Form is a way of specifying the syntax of a formal language by means of a context-free grammar. In my version of Backus-Naur Form, a rule consists of a non-terminal, followed by "::=", followed by one or more "|"-separated production rules, followed by a terminating ";" (semicolon). Terminals appear either in "double quotes" or as identifiers which are ALL UPPER CASE. Non-terminals appear with the first letter capitalized, and the rest lower-case. /* Comments may be surrounded by slash-star ... star-slash, as this sentence is. */

See Martin [1991] for an introduction to the theory of context-free formal languages, and Appel [1997] for an introduction to something that looks like my Backus-Naur Form.

The “Marks” non-terminal is the list (possibly empty) of Marks as found on the block from the the Matched_object originated. See the section on “Marks” on page 80 for an explanation.

```
Marks ::= { "‘" IDENTIFIER  }*
;
```

The “Focus_boolean” non-terminal is either “false” or “true”. It is true if an only if the block from which this Matched_object originated had the FOCUS keyword (see page 80).

```
Focus_boolean ::= "false" | "true"
;
```

The “Feature_values” non-terminal is a possibly-empty list of value assignments to features on the object from which the MO_ID_D originated, and is a result of the “`GET` *feature_list*” construct on an object_block:

```
Feature_values ::= "("   ")"    /* empty */
                |   "(" Feature_value { "," Feature_value }* ")"
;
Feature_value ::= Feature_name "=" Value
;
Feature_name ::= IDENTIFIER
;
Value ::= /* Any value allowed in the EMdF model. */
;
```

An MO_ID_M has a Set_of_monads instead of an Id_d non-terminal. It also has “pow_m” as the object type. This indicates that it contains the set of monads corresponding to the gap matched by the (opt_)gap_block from which the MO_ID_M arose.

An MO_ID_D also has a Set_of_monads corresponding to the monads of the object which matched the object_block from which the MO_ID_D arose. These sets of monads are never empty, but have at least one Monad_set_element.

```
Set_of_monads ::= "{"
                  Monad_set_element { "," Monad_set_element }*
                  "}"
;
Monad_set_element ::= /* singleton */ Monad
                  |   /* range */  First_monad "-" Last_monad
;
First_monad ::= Monad
;
Last_monad ::= Monad
;
Monad ::= INTEGER
;
```

Notice that a Matched_object may have an inner Sheaf, thus making the data structure recursive. This inner Sheaf results from an inner blocks construct in an object_block, opt_gap_block, or gap_block. Notice also that a Sheaf may be empty (see above), and thus the data structure need not be infinitely recursive, but does have a "base case" in which the recursion may stop. This is the case whenever a block does not have an inner blocks construct.

## 6.3 The parts of the Sheaf

### 6.3.1 Introduction

The previous Section has focussed on the grammar of the sheaf, but said little about what it means. This Section will explore the relationship between the Full Sheaf, its parts, and the query from which it originated. This is done by discussing each of the constructs "Matched_object", "Straw", and "Sheaf" in turn. I will then also discuss Flat Sheaves.

### 6.3.2 Matched_object

A Matched_object is the product of one matching of a block which is neither a power_block or a NOTEXIST_object_block. That is, it is the product of a matching of either an object_block, an opt_gap_block, or a gap_block. An opt_gap_block and a gap_block gives rise to an MO_ID_M, which has a set of monads corresponding to the gap matched. An object_block, on ther other hand, has an object type and an id_d, both corresponding to the object which matched the object_block.

### 6.3.3 Straw

A Straw is one complete matching of one block_string. This is done right below the level of the "OR" construct between strings of blocks. For example, the following MQL query:

```
SELECT ALL OBJECTS
WHERE
[Phrase function=Subj GET function]
[Phrase function=Pred GET function]
```

would result in a Sheaf with a number of Straws, each of which would contain two Matched_objects, the first resulting from the Phrase object_block with the "function=Subj" restriction, and the second Matched_object resulting from the Phrase object_block with the "function=Pred" restriction:

```
// /* the following is a Straw */
   <
      /* The following is a Matched_object */
      < Phrase 32 { 12-13 } false (function=Subj)
          // < >    /* empty, inner sheaf */
      >,
      /* The following is a Matched_object */
```

```
      < Phrase 33 { 14 } false (function=Pred)
          // < >    /* empty, inner sheaf */
      >
   > /* ... more Straws likely to follow ... */
```

### 6.3.4 Sheaf

A Sheaf is the result of all matches of a given `blocks` construct in a query. The outermost `blocks` construct results in a Sheaf which consists of Straws of Matched_objects resulting from the `blocks` in the `block_string` at the outermost level. If, then, one of these `blocks` has an inner `blocks`, that inner `blocks` gives rise to the inner Sheaf in the outer Matched_object. For example:

```
SELECT ALL OBJECTS
WHERE
[Clause
   [Phrase first function=Time]
   [Phrase first function=Pred]
   [Phrase first function=Subj]
]
```

This would give rise to a Sheaf which contained Straws, each of which had only one Matched_object. This Matched_object would have arisen from the "Clause" `object_block`. Since this "Clause" `object_block` has an inner `blocks` (namely the `block_string` consisting of the three Phrase `object_blocks`), the MO_ID_D resulting from each Clause `object_block` would have an inner Sheaf which was not empty. Each of these inner Sheaves would contain at least one Straw, each of which would contain three Matched_objects, one for each Phrase `object_block`. These inner Matched_objects would then each contain an empty inner Sheaf, since none of the inner Phrase `object_blocks` have an inner `blocks` construct.

Thus, there is an isomorphism, not only between the structure of the query and the structure of the objects found, but also between the structure of the query and the structure of the Sheaf resulting from the query. It is this quality which makes MQL a "topographic" language.

### 6.3.5 Flat Sheaf

A "flat sheaf" may arise in two ways: First, a `GET OBJECTS HAVING MONADS IN` query may be issued, which always results in a flat sheaf. Second, the Full Sheaf of a topographic query may be post-processed in order to obtain a flat sheaf.

A flat sheaf, in essense, is a Sheaf in which all MatchedObjects of a particular object type have been grouped into one Straw. Thus, for each object type present in the corresponding Full Sheaf, a Flat Sheaf has one Straw containing all the MatchedObjects of the Full Sheaf with that particular object type. In this respect, "pow_m" is also an object type, meaning that a Flat Sheaf may contain MO_ID_Ms as well as MO_ID_Ds.

Furthermore, no MatchedObject in a Flat Sheaf has a non-empty or failed inner Sheaf. That is, all MatchedObjects in a Flat Sheaf have empty inner sheaves. Therefore, a Flat Sheaf is not recursive.

For a `GET OBJECTS HAVING MONADS IN` statement, there never is any Full Sheaf from which the Flat Sheaf has been obtained. Instead, the Flat Sheaf is constructed directly. In this case, the Flat Sheaf only has one Straw with Matched_objects of one object type, namely the one which the `GET OBJECTS HAVING MONADS IN` statement queries.

## 6.4 Conclusion

In this Chapter, I have discussed the Sheaf as one of the possible kinds of output from an MQL statement. There are two kinds of Sheaves: Full Sheaves and Flat Sheaves. A Sheaf is a list of Straws. A Straw is a list of Matched_objects. A Matched_object corresponds to one matching of one `block` in a topographic MQL query. A Matched_object may have an inner Sheaf, thus making the data structure recursive. A Straw corresponds to one whole matching of one `block_string`.

The Sheaf is a central part of the implementation of the MQL query language, and we shall return to the Sheaf data structure several times in the thesis. In the next chapter, I show how to “harvest” a sheaf into something even more useful than the Sheaf itself.

# Chapter 7

# Harvesting search results

## 7.1 Introduction

Having explained the MQL query language and its result data type (the "Sheaf") in the previous chapter, I now turn to the problem of how to display meaningful results from the Sheaf.

The Chapter is laid out as follows. First, I discuss the problem at hand from a general perspective. I call the process whereby the Sheaf is turned into meaningful, displayable results, the process of "harvesting". Second, I define a number of concepts to be used when discussing the "harvesting" algorithm. Third, I lay out a general algorithm for "harvesting" the Sheaf. Fourth, I discuss ways of determining the "hit". Fifth, I discuss various ways of extending the algorithm. And finally, I conclude the chapter.

## 7.2 The problem at hand

As explained in [FSMNLP2005], there are various strategies for implementing query languages for text databases. The one I have chosen is to separate the process of querying from the process of gathering enough data to display the results. This is because this strategy provides for the greatest number of possible uses of the query results, and provides for "genericity" of the corpus query system. As [FSMNLP2005] explains, there are many uses of the output of the topographic query-stage, including (but not limited to):

1. Statistical measurements of the results.
2. Displays of various kinds, including:
    (a) word-based concordances (e.g., Keywords In Context (KWIC) [Luhn, 1960]),
    (b) bracketed views of linguistic units [Van Valin, 2001, Bickford, 1998],
    (c) graph-structure views of linguistic directed acyclic graphs, including:
        i. tree-structure view of linguistic trees [Horrocks, 1987, Van Valin, 2001, Bickford, 1998, Jackendoff, 1977, König and Lezius, 2003, Lezius, 2002a,b, Van Valin and LaPolla, 1997],
        ii. Role and Reference Grammar trees [Van Valin and LaPolla, 1997],
        iii. dependency-based views of dependency-based analyses [Van Valin, 2001],

iv. slot-filler views of slot-filler analyses (the work of Kenneth L. Pike and others on Tagmemics, e.g., Pike and Pike [1982]),

v. systemic views of systemic analyses [Halliday, 1976/1969, 1994, Eggins, 1994, de Joia and Stenton, 1980]

(d) and many others

3. Database-maintenance tasks, including:

   (a) Creating new objects based on the monad-sets arising out of a query.

   (b) Updating existing objects based on the monad-sets arising out of a query.

   (c) Deleting existing objects based on the monad-sets arising out of a query.

All of these kinds of uses require slightly different ways of manipulating the Sheaf. Displays drawing the output on a screen or printer may require different data to be retrieved based on the nature of the display (KWIC, graph-based, bracketed, etc.), and most displays require slightly different data to be retrieved from the database.

A statistical view may need to know how many times a given word or other linguistic object occurs in the entire corpus queried, in addition to the number of times it appears in the query results. A word-based KWIC display, on the other hand, may need to retrieve a specified number of words of context for each "hit" word, while a tree-based view will require information about both words (terminals in the tree), parents (non-terminals), and the uppermost context for each "hit" (in the tree; also called the "tree root"), in addition to the "parentage" information that inheres between the nodes in the tree. A bracketing view will require much the same information.

For database-maintenance tasks, it may not be required to retrieve other data than what is already present in the sheaf. But then again, depending on the purpose, other data may, in fact, be required. The query may only return words, for example, whereas the basis of the database maintenance may require Sentences to be retrieved, based on the words retrieved at first.

The benefit of separating the process of querying from the process of display is that all of these various kinds of uses are possible from the same output from the topographic query-stage. Thus, the system becomes generic in a very real, very applied sense.

The process of querying thus has the following stages:

1. Execute the query obtained from the user. This produces a Sheaf.

2. "Harvest" the Sheaf. This is the process of "harvesting" the "kernels" (or "hits") out of the Sheaf. This process produces a list of "Solutions", where a "Solution" is a collection of data that gives enough information that the next stage can display the "hit".

3. Use the Solutions in whatever manner is required to fulfill the purpose of the system. For example, the system may display each Solution to the user in a manner which is applicable to the database at hand and the needs of the user.

The display of data is separate from the harvesting process. Again, the harvesting stage can actually be made rather generic (as we shall see below), thus providing the same kind of output for many different kinds of display.

In the following, I describe a general harvesting algorithm which I have employed many times for various tasks relating not only to my work with the Kaj Munk Corpus, but also to my general work with implementing the Emdros Corpus Query System. As explained in Chapter 1, I have implemented a generic "Emdros Query Tool", and it is precisely in this tool that the following algorithm has been implemented and refined.

But first, I need to define a number of concepts.

## 7.3 Definitions of harvesting concepts

**Hit:** The precise nature of "hit" is difficult to determine generically. Below, in Section 7.5 on page 97, I show some ways of determining the hit. Here it suffices to say that the "hit" can be characterized by a set of monads.

**Focus monad set:** A set of monads which shows which monads are in "focus", because a MatchedObject in the Sheaf both had some of the monads in the focus monad set, and had the "focus" boolean set to "true". Notice that the concept of "focus monad set" may be applied both to an entire sheaf, and to a single "hit".

**Raster monad range:** Most display-based (as opposed to statistics-based) uses of Emdros require that a certain amount of context must be shown for each "hit". A raster monad range is a stretch of monads which supplies "enough" context that at least part of a "hit" can be shown. Sometimes, a "hit" will need to cover more than one "raster monad range", especially if the "raster monad ranges" are calculated based on "raster units" (see the next definition).

**Raster unit:** A raster unit is an object type whose objects are assumed to provide "enough" context to display for any given "hit". For example, for a Biblical database, a good raster unit might be "verse". If the display is to show Kaj Munk's plays, an "actor line" may be a good raster unit. If the display is to show all of the poetry of Kaj Munk, "poetry" might be a good raster unit. In the algorithm proposed below, there can only be one raster unit object type.

**Raster monad set:** A set of monads corresponding to the big-union of all raster monad ranges of either:

1. A single hit, or
2. All hits.

**Data unit:** A data unit is an object type, some of whose objects it is necessary to retrieve in order to display a given "hit". For example, "word" or "token" is almost always a candidate to be a data unit. Similarly, if the database is a syntactic analysis, then "Clause" and "Phrase" would probably be good candidates for being data units. If the display is to show all of Kaj Munk's poetry, then, in addition to "poetry" being the raster unit, one would require that "stanza" and "line" be data units, in order to know when to break the display into lines (at "line" object boundaries) and into stanzas (at "stanza" object boundaries).

**Data unit feature:** For any given display-purpose, the monads of a data unit may not be enough; certain features might also need to be retrieved. For example, for a "Word" data unit, the features "surface", "part_of_speech", and "lemma" may need to be retrieved. For the "actor line" data unit in Kaj Munk's plays, it may be necessary to retrieve the name of the actor role which says the line. For a syntactic tree-display, the "phrase type" may need to be retrieved for Phrases, and a "parent" feature may point to the parent object of any given node in the tree, and hence should be retrieved in order to recreate the tree. And so on.

**Reference unit:** A reference unit is an object type whose features gives information which makes it possible to locate any given "hit" in the corpus. For example, in a Biblical database, the "verse" reference unit may provide information about the book, chapter, and verse of the given verse. If so, the "verse" object type is a good candidate for being a reference unit. If the display is of Kaj Munk's poetry, then the object type which bears the title of each play or poem would be a good candidate for being the reference unit. If the display is a linguistic database, there may be some object (say, Sentence or Document) which bears a feature specifying the number of the sentence or document in some catalog. And so on.

**Reference unit feature:** A feature of a reference unit which must be displayed in order to identify the reference given by the reference unit.

**Solution:** A Solution is a collection of data corresponding to a single "hit", and consisting of:

1. The hit's "hit monad set".
2. The hit's "focus monad set".
3. The hit's "raster monad ranges".
4. The data units (with data unit features) necessary for displaying the hit.
5. The reference unit object which can identify the hit.

Armed with these definitions, we can now discuss the general harvesting algorithm.

## 7.4 A general harvesting algorithm

The following algorithm was first proposed by myself in 2001 or 2002, and was first described by me in [Sandborg-Petersen, 2002-2008]. Hendrik Jan Bosman of the Werkgroep Informatica, Vrije Universiteit Amsterdam was kind enough to implement it in various prototype programs — first a C program, then a Python program, both of which had the purpose of displaying results from Emdros queries. In 2004 or 2005, I reimplemented Mr. Bosman's Python program in C++, and have since refined the algorithm.

The basic algorithm is as follows:

1. From the Sheaf, determine a list of "hits." Some of the many ways in which this can be done are described below.
2. Determine a set of "raster monad ranges".

3. Obtain the set of monads arising as the big-union of all “raster monad ranges”. Call this set of monads the “raster_monad_set”.

4. For each “data unit” to display, issue a query based on the following template:

```
GET OBJECTS HAVING MONADS IN <raster_monad_set>
[<data_unit> GET <data_unit_features>]
```

   For example:

```
GET OBJECTS HAVING MONADS IN { 1-1083, 2435-3310 }
[token GET surface, part_of_speech, lemma]
```

   Then store the results in a suitable data structure.

5. Do the same for the “reference unit”.

6. For each “hit” in the list of hits, create a “Solution” which contains enough data to display the results of the Hit. Append the Solution to a list of solutions, which in the end becomes the output from the harvesting algorithm.

Once this algorithm has run to completion, the list of Solutions can be used to display the results in a manner which is helpful to the user. Depending on how one calculates the “raster_monad_set”, the Solution object can support many different kinds of display. For example, a syntactic database likely needs a Sentence object to form the basis of the context to display for any given “hit”, while a “Keywords In Context” (KWIC) concordance may need a specified number of tokens on either side of the keyword, as explained above.

In the next section, I describe some ways that a “hit” can be determined.

## 7.5 Determining the “hit”

It is not at all obvious exactly what a “hit” is in the context of a Sheaf. As explained in [FSMNLP2005], the Sheaf is generic enough that it can support a wide range of uses. However, this genericity comes at a cost, namely that the determination of what constitues a “hit” may need to rely on domain-specific knowledge of the database at hand.

In the following, I describe a number of ways in which the “hit” may be determined generically. Other ways of determining the hit are most likely possible, given specific databases and specific user needs.

What is common to all the proposed solutions is that the “hit” is characterized solely by a set of monads. In my own experiments, this has proven to be both useful and sufficient. However, for certain kinds of database, it may be desirable also to store, with each “hit”, information about the id_d or focus state of the MatchedObject(s) from which the “hit” arose.

**Outermost:** In this strategy, the “hit” is determined by being the big-union of all MatchedObjects in each outermost Straw in the Sheaf. Thus each “hit” corresponds to one string of blocks at the outermost level in the topographic query.

**Focus:** In this strategy, the "hit" is determined by the monad set of a single MatchedObject which has the "focus" boolean set. The whole sheaf is traversed, and all MatchedObjects in the Sheaf-tree are visited. Thus, every block in the topographic query which has the "FOCUS" keyword will give rise to a "hit". Obviously, this may mean that a given Sheaf may yield an empty set of monads, if there is no "focus" boolean which is true in any MatchedObject in any Straw in the Sheaf. Such empty sets of monads should be removed from the list of hits before going to the following stages in the harvesting algorithm. This, of course, means that the list of "hits" will be empty.

**Marks:** In this strategy, the "hit" is determined by a single MatchedObject which has certain specified "marks" in its set of marks. Thus this strategy is analogous to "Focus", except that specified "marks", rather than the "focus boolean" are the basis for selecting the MatchedObjects which give rise to the "hits". The same remarks about empty monad sets and empty lists of "hits" apply here as they did for "Focus".

**Outermost_focus:** This strategy is the same as "outermost", except that only MatchedObjects in the outermost straws whose "focus" boolean is true will contribute to the monad set of a "hit". Obviously, this may mean that a given "outermost Straw" may yield an empty set of monads, if there is no "focus" boolean which is true in any MatchedObject in the Straw. Such empty sets of monads should be removed from the list of hits before going to the following stages in the harvesting algorithm. This, of course, may mean that the list of "hits" will be empty.

**Outermost_marks:** This strategy is analogous to "Outermost_focus", except that specified "marks" form the basis of the selection of the MatchedObjects which give rise to "hits". The same remarks about empty monad sets apply here as they did for Outermost_focus.

**Innermost:** In this strategy, the "hit" is calculated from a Straw in which all MatchedObjects have no inner sheaves. That is, in order to become a "hit", a Straw must contain only MatchedObjects which are terminals in the Sheaf-tree. The "hit" is then the big-union of the monad sets of all the MatchedObjects of such a Straw.

**Innermost_focus:** This strategy is the same as "Innermost", except that only the monads of MatchedObjects whose "focus" boolean is set will contribute to a hit. Again, the same remarks about empty monad sets and empty "hit" lists apply as they did for "Outermost_Focus".

**Innermost_marks:** This strategy is the same as "Innermost_focus", except that it is MatchedObjects with certain "marks" which contribute to the "hit", not MatchedObjects with the "focus" boolean set to "true". Again, the same remarks apply as they did for "Outermost_focus".

**Terminal:** This strategy is similar to "innermost", but does not have the requirement that all MatchedObjects in the straw need be terminal nodes in the sheaf-tree. It is sufficient that the MatchedObject itself does not have an inner sheaf. Obviously, this can be combined with the "focus" and "marks" strategies.

**Level:** In this strategy, the "hit" is calculated as the big-union of the monad sets of all Straws at a given level of nesting. Obviously, this can be combined with the "focus" and "marks" strategies.

**Object_type:** In this strategy, the "hit" is calculated as the big-union of certain object types present in the sheaf. Obviously, this can be combined with the "outermost", "innermost", "focus" and "marks" strategies, as well as the "level" strategy, such that only certain object types at certain levels will contribute to a "hit". This, of course, requires domain-specific knowledge of which object types may be good to use for this purpose.

**Object_type_feature:** In this strategy, the "hit" is calculated as the big-union of certain object types with certain values for certain features in the sheaf. Obviously, this can be combined with any of the "outermost", "innermost", "focus", "marks", and "level" strategies.

Having described some example strategies, I now attempt to bring some rigor to the subject at hand by abstracting some categories out of the above. There are three categories with various subtypes:

**Initial MatchedObject candidate selection:** (Also known as 'Sheaf traversal strategy'):

**All MatchedObjects:** All MatchedObjects are visited, and are possible candidates. This is the basis of "Focus" and "Marks" above.

**Outermost:** Only the MatchedObjects which are immediate children of the outermost Straws in the outermost Sheaf are candidates.

**Innermost:** Only the MatchedObjects which are immediate children of Straws containing only terminals in the sheaf-tree are candidates.

**Terminal:** Only the MatchedObjects which have no inner sheaf are candidates.

**Level:** Only the MatchedObjects at certain levels of nesting are candidates.

**Matched-Object Filtering strategy:** Once the candidate MatchedObjects are found, it may be useful to take only some of them as a basis of a hit, thereby obtaining "filtered candidates".

**Focus:** Only candidate MatchedObjects which have the "focus" boolean set to "true" are candidates.

**Marks:** Only candidate MatchedObjects which have certain specified "marks" are candidates.

**Object_type:** Only candidate MatchedObjects which arise from certain specified object types are candidates.

**Object_type_feature:** A special kind of Object_type filtering involves not only object types, but also features: Only candidate MatchedObjects which arise from certain specified object types with certain values for certain features are candidates.

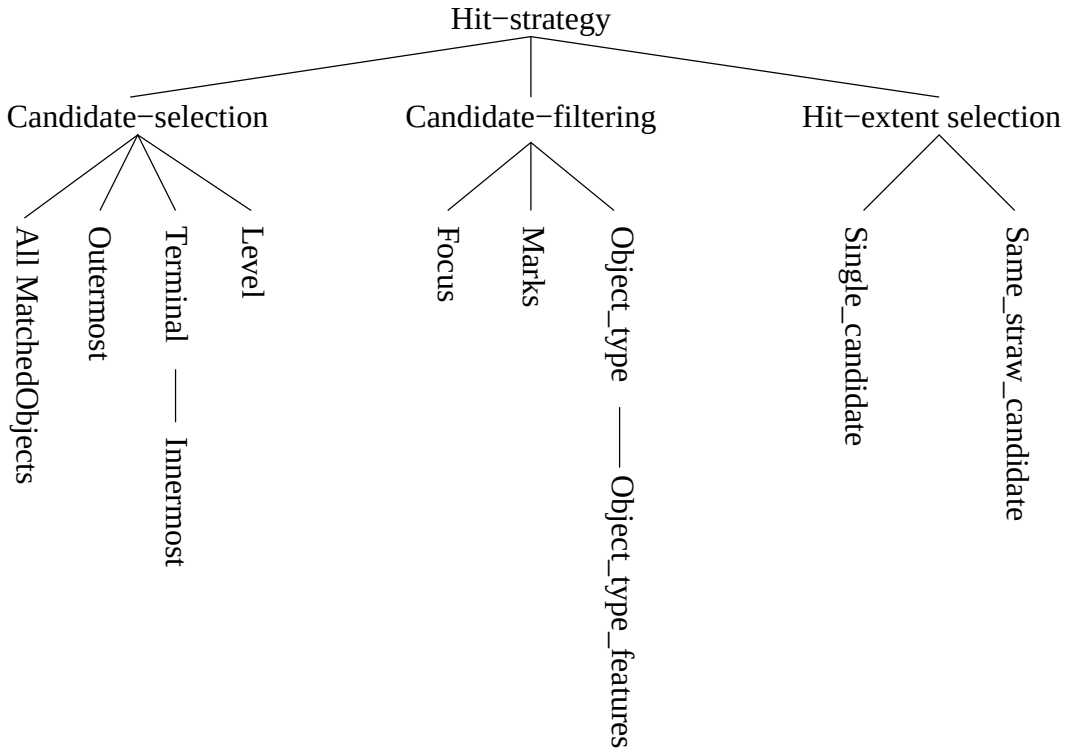



Figure 7.1: Ontology of hit strategies

**Hit-extent strategy:** Once the candidates are found and filtered, it may be useful to have each filtered candidate give rise to a hit, or it may be useful to group some of them into a single hit:

**Single_candidate:** Each filtered candidate MatchedObject gives rise to a "hit". This forms the basis of the above "Focus" and "Marks" strategies.

**Same_straw_candidate:** The monad set from each filtered candidate MatchedObject is big-union'ed with the monad sets from the other filtered candidate MatchedObjects from the same straw. This forms the basis of the above "outermost" and "innermost" strategies in the first description above.

This may be illustrated as in the ontology in Figure 7.1.

## 7.6 Extending the harvesting algorithm

There are many variations over the harvesting algorithm, some of which I list here:

**No reference unit:** It might not be necessary to have a reference unit for certain purposes.

**More than one reference unit:** For certain databases, more than one object type may need to be queried in order to obtain a full reference. For example, in a Biblical

```
SELECT ALL OBJECTS
WHERE
// Retrieve three Phrases before each hit
[Phrase]
[Phrase]
[Phrase]
// Now retrieve exactly the clauses which
// form the basis of the example hits
[Clause self IN (10321,10326,10337,...,47329)]
// Retrieve four Phrases after each hit
[Phrase]
[Phrase]
[Phrase]
[Phrase]
```

Figure 7.2: Example topographic query implementing raster_context_objects

database with no redundant data, the "verse" object type may carry information only about the verse number, while a "book" object type may carry information about the book name, and a "chapter" object type may carry information about the chapter number. This situation would require more than one object type to be queries in order to obtain all relevant reference information. The display-stage would then have to assemble this information into output which would be meaningful to the user.

**Raster context monads:** For certain types of display (e.g., KWIC concordances), it might be desirable to retrieve a certain number of tokens as context. If a token takes up precisely one monad, then this can be specified as a certain number of monads "before" the first monad of a given "hit", coupled with a certain number of monads "after" the last monad of a given "hit".

**Raster_context_objects:** For certain types of display, it may be desirable to retrieve a certain number of arbitrary raster units as context. For example, one may want to display precisely three Phrases on either side of a given "hit". In the current Emdros-implementation, this would have to be implemented with yet another topographic query, rather than a GET OBJECTS HAVING MONADS IN query. The topographic query could look something like the one in Figure 7.2. Notice that this approach would require the "hit" to carry information about the "self" feature of each "hit-Clause" (i.e., the id_d of each "hit-Clause").

## 7.7 Conclusion

In this chapter, I have described some ways to use the output of a topographic query (i.e., a Sheaf). In particular, I have described some purposes to which a Sheaf may be put, including statistical information, display of results, and maintenance of database objects (creation / update / deletion of objects).

I have described the process of querying an Emdros database as a three-step process in which: (i) the topographic query is executed, (ii) the resulting Sheaf is "harvested", and (iii) the harvested "Solutions" are put to use.

Flowing out of this, I have described a generic algorithm for "harvesting" a sheaf, i.e., an algorithm for gathering enough data on the basis of a Sheaf to be able to fulfill the purpose for which the query was executed. This description required the definition of a number of concepts, including "focus monads", "raster monad range", "raster unit", "data unit", "data unit feature", "reference unit", and "Solution", among others.

Having described the algorithm in general terms, I have then described a number of strategies which may be employed in determining what a "hit" is. This resulted in an ontology of "hit-strategies" in which three main categories of hit-strategy were identified, namely "MatchedObject Candidate Selection Strategy", "MatchedObject Candidate Filtering Strategy", and "Hit-extent strategy".

I have then described some ways in which the basic harvesting algorithm may be tweaked. Some of these ways are useful at times, though all of them require domain-specific judgment of whether they are useful or not.

Harvesting a Sheaf is a basic process in the toolbox of techniques which I have developed during my PhD, and I have used the technique many times in various sub-projects relating to Kaj Munk. We will get back to some of these sub-projects in due course. Although I conceived of the basic algorithm in 2001 or 2002 (which is before I commenced my PhD studies), the framework for describing and classifying the various "hit-strategies" (Section 7.5) and the extensions to the general algorithm (Section 7.6) have been developed purely within the time-frame of my PhD studies. In addition, the justification for the separation of the topographic query-stage from the harvesting stage has been developed in [FSMNLP2005] as part of my PhD studies, and has been expanded upon in this chapter.

# Chapter 8

# Annotated text and time

## 8.1 Introduction

Formal Concept Analysis (FCA) [Lehmann and Wille, 1995, Ganter and Wille, 1997] is a mathematically grounded method of dealing with data in tabular form, transforming it to so-called "formal contexts" which can be drawn as lattices of objects and attributes. As such, Formal Concept Analysis has a lot in common with the ontologies described in Chapter 3.[1]

FCA has many applications, not least of which is aiding a human analyst in making sense of large or otherwise incomprehensible data sets. In this paper, we present an application of FCA to the problem of classifying classes of linguistic objects that meet certain linguistically motivated criteria, with the purpose of storing them in Emdros.

The rest of the Chapter is laid out as follows. In Section 8.2, I argue that the structures which we perceive in text can be seen as sets of durations. In Section 8.3, I recapitulate the important parts of the EMdF model, and show how this view of text is modelled in Emdros. In Section 8.4, I introduce certain linguistically motivated criteria which sets of sets of durations may or may not exhibit. In Section 8.5, I analyze these criteria in order to obtain a complete catalog of the combinations of the criteria which are logically consistent. In Section 8.6, I describe and analyze the lattice which FCA produces from this catalog. In Section 8.7, I apply the results from the previous section to Emdros. In Section 8.8, I show how I have implemented these results. Finally, in Section 8.9, I conclude the Chapter and give pointers to further research.

## 8.2 Language as durations of time

Language is always heard or read in time. That is, it is a basic human condition that whenever we wish to communicate in verbal language, it takes time for us to decode the message. A word, for example, may be seen as a duration of time during which a linguistic event occurs, viz., a word is heard or read. This takes time to occur, and thus a message or text occurs in time.

In this section, I describe four properties of language which have consequences for how we may model linguistic objects such as words or sentences. The first is sequence,

[1] This Chapter is an expanded and edited version of the material found in my published article, [ICCS-Suppl2008].

the second is embedding, the third is resumption, and the fourth is "non-hierarchic overlapping relationships".

### 8.2.1 Sequence

As I mentioned above, a message (or text) is always heard or read in time. We experience the passage of time as being ordered, i.e., there is a definite direction of time: We never experience time going "backwards"; the very language which we employ about time (viz., "forwards", "backwards") betrays our notions of time as a linear sequence.[2]

Because a message is heard or read in time, and because time is perceived as being linear, there is always a certain sequence to a given text. We may call this the "hearing-order" or "reading-order". This ordering can be based on the partial order $\leq$ that exists between non-overlapping durations.[3]

Thus, if we wish to build a text database system which adequately captures the linguistic information in a text, the system must be able to maintain the sequence of the hearing- or reading-order of the text.

### 8.2.2 Embedding

Language always carries some level of structure; at the very least, the total duration of time which a message fills may be broken down into shorter durations which map to words. The durations which are filled by the words (either heard or read) are thus *embedded* inside the duration filled by the whole message.

Words, as we saw above, may be interpreted as durations during which a linguistic event occurs. However, we usually speak or write not only in words, but also in sentences. This adds another, intermediate level of structure to the message, such that words are embedded in sentences, and sentences are embedded in the whole message.

Moreover, most languages can be analyzed at levels that are intermediate between words and sentences.[4] Linguists have called these levels *phrases* and *clauses*, among other terms. For example, the sentence "The door opens towards the East" may be analyzed thus:

[The door] [opens] [towards [the East]]

Here I have added brackets, indicating the boundaries of phrases in the sentence. "The door" is usually called a *noun phrase* (NP) by linguists, as is "the East". The latter noun phrase is, in turn, embedded inside the larger *prepositional phrase* (PP) "towards [the East]".

Thus a text can be analyzed or broken down into smaller units, which in turn may be able to be further analyzed into smaller units. Even words may be analyzed into morphemes and these, in turn, may be analyzed into graphemes/phonemes [Van Valin, 2001, Van Valin and LaPolla, 1997, Eggins, 1994, Horrocks, 1987].

[2] As [Øhrstrøm and Hasle, 1995, p. 31] explains, time may also be perceived as being branching; yet what we actually experience can always be mapped to a single line, regardless of which branches are actualized.

[3] See Section 8.3 for the details.

[4] For a lucid discussion of the linguistic terms involved in the following paragraphs, see Van Valin [2001], Van Valin and LaPolla [1997].

What I have just described is the *recursive* nature of language: Linguistic units are capable of being embedded inside other, larger linguistic units, which in turn may be embedded within yet larger linguistic units. This can be described in terms of sets of durations which are ordered in a hierarchy based on the $\subset$ subset-relation between the sets.[5]

Thus, if we wish to store linguistic objects adequately within a text database system, the system must support embedding of textual objects.

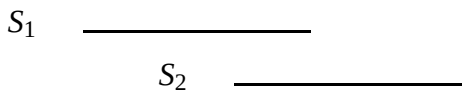
### 8.2.3 Resumption

There is, however, another important property of linguistic units which must be mentioned, namely that language is, by nature, *resumptive*. By this I mean that linguistic units are not always *contiguous*, i.e., they may occupy multiple, disjoint durations of time. For example, some linguists would claim that the sentence "This door, which opened towards the East, was blue." consists of two clauses: "which opened towards the East", and "This door ... was blue."[6] This latter clause has a "hole" in it, in terms of time, and as such must be described in terms of two durations, not one. Because language is generative, and because sentences can in principle go on indefinitely [Horrocks, 1987, pp. 14–16] this kind of resumption can in principle occur arbitrarily many times.

Hence, in order to be able to describe linguistic units adequately, a text database system must be capable of storing not just single durations, but *arbitrary sets of non-overlapping durations*.

### 8.2.4 Non-hierarchic overlap

A fourth important property of linguistic units is that they may "violate each other's borders." By this I mean that, while unit $A$ may start at time $a$ and end at time $c$, unit $B$ may start at time $b$ and end at time $d$, where $a < b < c < d$. Thus, while $A$ overlaps with $B$, they cannot be placed into a strict hierarchy based on the $\subset$ relation. This occurs especially in spoken dialog, where speaker $S_1$ may speak for a while, and then speaker $S_2$ may start his or her speaker-turn, before speaker $S_1$ has finished speaking (see also [Cassidy, 1999, Bird et al., 2000b, Cassidy and Harrington, 2001]).




Thus, if we wish to capture all kinds of linguistic information adequately in a text database system, the system must support linguistic units which not only occur in a hierarchy of embedding, but which may overlap in such a way that the units are not embedded inside each other in a strict hierarchy.

[5] I here abstract away from how the sets of time are actually realized. Later in the Chapter, I will show how the statement is true if the sets are sets of monads.

[6] For one such opinion, see McCawley [1982].

## 8.3 The EMdF model

As already indicated in Chapter 4, these four properties are indeed present in the EMdF model. I here recapitulate the EMdF model, from the perspective of "monads seen as durations of time."

The reader will recall that central to the EMdF model is the notion that textual units (such as books, paragraphs, sentences, and even words) can be viewed as *sets of monads*. A monad is simply an integer, but may be viewed as *an indivisible duration of time*.

When viewed from outside the system, a monad does not map to an instant, but rather to a duration. This is because even words are not heard (or read) in an instant, but always take a certain duration of time to hear or read, however small.

When viewed from inside the system, however, the length of the duration does not matter. This is because we view a monad as an *indivisible* duration of time. The duration is indivisible, not by nature, but because we do not wish to analyze or break it down further into smaller durations in our description of the text.

Usually, a monad maps to the duration of any given *word* in the text, but this need not be so. If we wish to analyze a text below word-level, we can stipulate that the smallest indivisible duration which we wish to treat in the EMdF model is some other linguistic unit, such as the grapheme or phoneme. Then words will consist of more than one monad.

Notice that the actual length of the duration of a monad is immaterial to the database; it has been abstracted away into the indivisibility of the monad. One monad may even map to a different actual length of time than any other monad, since the actual length depends on the actual linguistic unit to which the monad corresponds.

Since monads are simply integers, they form a well-ordered sequence (i.e., 1, 2, 3, ..., etc.). The sequence of monads can be ordered by the $\leq$ partial order on integers. Moreover, this partial order can be extended to durations of time, given that a monad represents a duration of time.

Given that textual objects such as words, sentences, and paragraphs may be viewed as sets of durations of time, it follows that objects may be embedded inside each other. This is because durations may embed, or be subsets of, each other.

These durations are represented as stretches of *monads*, which in turn map to indivisible durations of time during which linguistic events occur which we do not wish to analyze further into smaller durations of time. A single stretch of monads thus maps to a larger duration than each of the monads making up the stretch. Finally, a set of monads is made up of one or more stretches of monads, thus making the set completely arbitrary.

Since textual objects can often be classified into similar kinds of objects with the same attributes (such as words, paragraphs, sections, etc.), the reader will recall that EMdF model provides *object types* for grouping objects. An object type, though abstract in itself, may have a set of concrete *instances* of objects with actual values.

The reader will recall that an object type may have a set of *attributes*, also called *features*. Each object belonging to an object type $T$ has a value for all of the features belonging to the object type $T$. For a given database, the set of objects belonging to an object type $T$ is denoted $\text{Inst}(T)$. For a given object $O$, $\mu(O)$ denotes the $M$ set of monads from $O$.

Finally, an object $O$ is a two-tuple $(M,F)$, where $M$ is a set of monads, and $F$ is a set of value-assignments to the features of the object type $T$ to which $O$ belongs. It is important to note that $M$ may not be empty. That is: $\forall T : \forall O \in \text{Inst}(T) : \mu(O) \neq \emptyset$.

| | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 |
|---|---|---|---|---|---|---|---|---|---|
| **Word** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| surface | This | door, | which | opened | towards | the | East, | was | blue |
| **Phrase** | 10 | | 11 | 12 | | 13 | | 14 | 15 |
| phrase_type | NP | | NP | VP | | NP | | VP | AP |
| **Phrase** | | | | | 16 | | | | |
| phrase_type | | | | | PP | | | | |
| **Clause** | 17 | | 18 | | | | | 17 | |
| **Sentence** | 19 | | | | | | | | |

Figure 8.1: EMdF database example. Note how there are two rows of Phrase objects, because Phrase-13 and Phrase-16 overlap. Phrase-13 is, in effect, embedded inside Phrase-16. Notice also that Clause-17 is discontiguous, consisting of the monad set {101-102, 108-109}.

A small EMdF database can be seen in Figure 8.1. The line of integers at the top are the monads. All other integers are the "self" IDs.

## 8.4 Criteria

In this section, we introduce some linguistically motivated criteria that may or may not hold for the objects of a given object type $T$. This will be done with reference to the properties inherent in language as described in Section 8.2.

In the following, let Inst$(T)$ denote the set of objects of a given object type $T$. Let $a$ and $b$ denote objects of a given object type. Let $\mu$ denote a function which, given an object, produces the set of monads $M$ being the first part of the pair $(M,F)$ for that object. Let $m$ denote a monad. Let $f(a)$ denote $\mu(a)$'s first (i.e., least) monad, and let $l(a)$ denote $\mu(a)$'s last (i.e., greatest) monad. Let $[m_1 : m_2]$ denote the set of monads consisting of all the monads from $m_1$ to $m_2$, both inclusive.

**Range types:**

**single monad($T$):** means that all objects are precisely 1 monad long.
$\forall a \in \text{Inst}(T) : f(a) = l(a)$

**single range($T$):** means that all objects have no gaps (i.e., the set of monads constituting each object is a contiguous stretch of monads).
$\forall a \in \text{Inst}(T) : \forall m \in [f(a) : l(a)] : m \in \mu(a)$

**multiple range($T$):** is the negation of "single range($T$)", meaning that there exists at least one object in Inst$(T)$ whose set of monads is discontiguous. Notice

that the requirement is not that all objects be discontiguous; only that there exists at least one which is discontiguous.

$$\begin{aligned} & \exists a \in \text{Inst}(T) : \exists m \in [f(a) : l(a)] : m \notin \mu(a) \\ \equiv \quad & \neg(\forall a \in \text{Inst}(T) : \forall m \in [f(a) : l(a)] : m \in \mu(a)) \\ \equiv \quad & \neg(\text{single range}(T)) \end{aligned}$$

**Uniqueness constraints:**

**unique first monad($T$):** means that no two objects share the same starting monad.

$$\begin{aligned} & \forall a, b \in \text{Inst}(T) : a \neq b \leftrightarrow f(a) \neq f(b) \\ \equiv \quad & \forall a, b \in \text{Inst}(T) : f(a) = f(b) \leftrightarrow a = b \end{aligned}$$

**unique last monad($T$):** means that no two objects share the same ending monad.

$$\begin{aligned} & \forall a, b \in \text{Inst}(T) : a \neq b \leftrightarrow l(a) \neq l(b) \\ \equiv \quad & \forall a, b \in \text{Inst}(T) : l(a) = l(b) \leftrightarrow a = b \end{aligned}$$

Notice that the two need not hold at the same time.

**Linguistic properties:**

**distinct($T$):** means that all pairs of objects have no monads in common.

$$\begin{aligned} & \forall a, b \in \text{Inst}(T) : a \neq b \rightarrow \mu(a) \cap \mu(b) = \emptyset \\ \equiv \quad & \forall a, b \in \text{Inst}(T) : \mu(a) \cap \mu(b) \neq \emptyset \rightarrow a = b \end{aligned}$$

**overlapping($T$):** is the negation of distinct($T$).

$$\begin{aligned} & \neg(\text{distinct}(T)) \\ \equiv \quad & \exists a, b \in \text{Inst}(T) : a \neq b \wedge \mu(a) \cap \mu(b) \neq \emptyset \end{aligned}$$

**violates borders($T$):** $\exists a, b \in \text{Inst}(T) : a \neq b \wedge \mu(a) \cap \mu(b) \neq \emptyset \wedge ((f(a) < f(b)) \wedge (l(a) \geq f(b)) \wedge (l(a) < l(b)))$

Notice that violates borders($T$) $\rightarrow$ overlapping($T$), since violates borders($T$) *is* overlapping($T$), with an extra, conjoined term.

It is possible to derive the precise set of possible classes of objects, based on logical analysis of the criteria presented above. I now turn to this derivation.

### 8.4.1 Range types

Notice that *single monad*$(T) \Rightarrow$ *single range*$(T)$. This is because, given that $\forall a \in \text{Inst}(T) : f(a) = l(a)$, then, since $\forall a \in \text{Inst}(T) : a \neq \emptyset$, it follows that $\forall a \in \text{Inst}(T) : \forall m \in [f(a) : l(a)] : m \in \mu(a)$, due to the fact that $f(a) = l(a)$.

The converse obviously does not hold, however: An object type $T$ may be *single range* without it being *single monad*.

In the following, we treat the logical relationships that exist between the various criteria within each of the range types.

### 8.4.2 Single monad

Notice that the following relationships hold:

$$\text{single monad}(T) \wedge \text{unique first monads}(T) \rightarrow \text{unique last monads}(T) \tag{8.1}$$

$$\text{single monad}(T) \wedge \text{unique last monads}(T) \rightarrow \text{unique first monads}(T) \tag{8.2}$$

This is because, since single monad$(T)$ holds, then for all $a$, $f(a) = l(a)$. That is, if the first monad is unique, then the last has to be unique as well, and vice versa.

Notice also that:

$$\text{single monad}(T) \wedge \text{unique first monads}(T) \rightarrow \text{distinct}(T) \wedge \text{single monad}(T) \tag{8.3}$$

This is because, if all objects consist of a single monad, and all objects are unique in their single monad, meaning no two objects share the same first monad, them all distinct pairs of objects $a$ and $b$ will fulfill $\mu(a) \cap \mu(b) = \emptyset$.

Notice the the converse also holdes:

$$\text{distinct}(T) \wedge \text{single monad}(T) \rightarrow \text{unique first monads}(T) \tag{8.4}$$

This is because, if all objects consist of a single monad, and all pairs of sets of monads $\mu(a)$ and $\mu(b)$ (where $a \neq b$) have an empty intersection, then, since all sets of monads are non-empty, it follows that $f(a) \neq f(b)$, and hence unique first monads$(T)$ holds.

Notice that "violates borders" requires either "single range" or "multiple range", but cannot be realized if the range type is "single monad". This is because of the double requirement that $l(a) \geq f(b)$ and $l(a) < l(b)$, which together imply that $f(b) < l(b)$, which again implies that $f(b) \neq l(b)$. Thus "single monad" is ruled out. Thus:

$$\text{violates borders}(T) \rightarrow \neg \text{single monad}(T) \tag{8.5}$$

Notice that the inverse also holds:

$$\text{single monad}(T) \rightarrow \neg \text{violtates borders}(T) \tag{8.6}$$

### 8.4.3 Single range

Notice that if a "single range" object type is non-overlapping (i.e., all objects are distinct), it implies that all objects are unique in their first *and* last monads. This is because all sets are distinct (i.e., no object overlaps with any other object); hence, for all distinct objects $a$ and $b$, $f(a) \neq f(b) \wedge l(a) \neq l(b)$. This is because the negation, namely $f(a) = f(b) \vee l(a) = l(b)$, would imply that the objects in question shared at least one monad, namely either the first or the last (or both).

$$\text{single range}(T) \wedge \text{distinct}(T) \rightarrow \text{unique first monads}(T) \wedge \text{unique last monads}(T) \quad (8.7)$$

Notice that the converse does not hold: Given an object type holding only the two objects {1,2} and {2,3}, we have both unique first monads and unique last monads, but we also have overlapping.

Notice that if an object type is both *single range* and *non-overlapping*, it also holds that it does not violate any borders. This is because violation of borders requires overlapping to exist.

$$\text{single range}(T) \wedge \neg\text{overlapping}(T) \rightarrow \neg\text{violates borders}(T) \quad (8.8)$$

Inversely, if an object type does violate borders, it must be overlapping:

$$\text{single range}(T) \wedge \text{violates borders}(T) \rightarrow \text{overlapping}(T) \quad (8.9)$$

This is because $\text{violates borders}(T) \rightarrow \text{overlapping}(T)$, as we saw above.

### 8.4.4 Multiple range

Notice that "multiple range" conjoined with "non-overlapping" again implies "unique first and unique last monads", just as it did for "single range", and for the same reason.

$$\text{multiple range}(T) \wedge \text{distinct}(T) \rightarrow \text{unique first monads}(T) \wedge \text{unique last monads}(T) \quad (8.10)$$

The converse does not hold, however, for the same reason it did not hold for single range objects.

Notice finally that, just as for single range, non-overlapping implies non-violation of borders, since violation of borders requires overlapping.

$$\text{multiple range}(T) \wedge \neg\text{overlapping}(T) \rightarrow \neg\text{violates borders}(T) \quad (8.11)$$

The inverse also holds:

$$\text{multiple range}(T) \wedge \text{violates borders}(T) \rightarrow \text{overlapping}(T) \quad (8.12)$$

The converse of this does not hold, however: It is possible to be overlapping and not have violation of borders.

| Class name | sm | sr | mr | ufm | ulm | ds | ol | vb |
|---|---|---|---|---|---|---|---|---|
| 1.000 | X | X | | | | | X | |
| 1.300 | X | X | | X | X | X | | |
| 2.000 | | X | | | | | X | |
| 2.001 | | X | | | | | X | X |
| 2.100 | | X | | | X | | X | |
| 2.101 | | X | | | X | | X | X |
| 2.200 | | X | | X | | | X | |
| 2.201 | | X | | X | | | X | X |
| 2.300 | | X | | X | X | | X | |
| 2.301 | | X | | X | X | | X | X |
| 2.310 | | X | | X | X | X | | |
| Class name | sm | sr | mr | ufm | ulm | ds | ol | vb |
| 3.000 | | | X | | | | X | |
| 3.001 | | | X | | | | X | X |
| 3.100 | | | X | | X | | X | |
| 3.101 | | | X | | X | | X | X |
| 3.200 | | | X | X | | | X | |
| 3.201 | | | X | X | | | X | X |
| 3.300 | | | X | X | X | | X | |
| 3.301 | | | X | X | X | | X | X |
| 3.310 | | | X | X | X | X | | |

Table 8.1: All the possible classes of object types. Legend: sm = single monad, sr = single range, mr = multiple range, ufm = unique first monad, ulm = unique last monad, ds = distinct, ol = overlapping, vb = violates borders.

## 8.5 Logical analysis of the criteria

Using the relationships in the previous section, we can derive all the possible classes of object types based on the criteria given in that section.

The possible classes are listed in Table 8.1

## 8.6 FCA results

The context resulting from these tables is then processed by the Concept Explorer software (ConExp)[7]. This produces a lattice which can be seen in Figure 8.2.

## 8.7 Applications

It is immediately noticeable from looking at Figure 8.2 that “ds” is quite far down the lattice, with several parents in the lattice. It is also noticeable that “ol” is quite far up in the lattice, with only the top node as its parent. Therefore, “ds” may not be as good a

[7] See http://conexp.sourceforge.net. Also see Serhiy A. Yevtushenko. *System of data analysis "Concept Explorer"*. (In Russian). Proceedings of the 7th national conference on Artificial Intelligence KII-2000, p. 127-134, Russia, 2000.

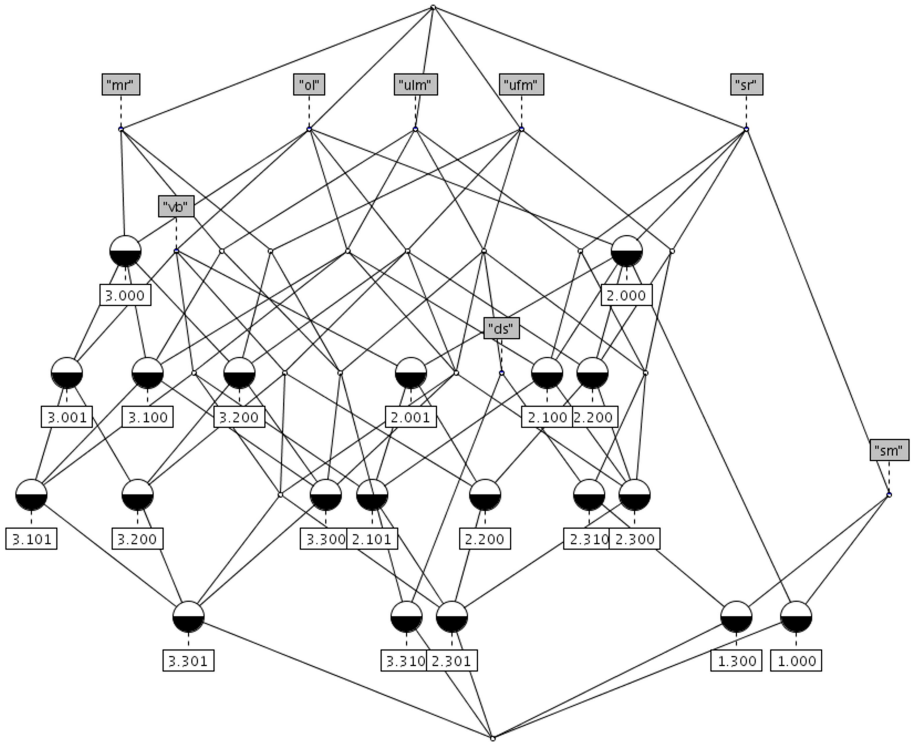



Figure 8.2: The lattice drawn by ConExp for the whole context.

candidate for a criterion on which to index as “ol”. Hence, we decided to experiment with the lattice by removing the “ds” attribute. The resulting lattice can be seen in Figure 8.3

In this new lattice, it is noticeable that the only dependent attributes are “sm” and “vb”: All other attributes are at the very top of the lattice, with only the top node as their parent. This means we are getting closer to a set of criteria based on which to index sets of monads.

The three range types should definitely be accommodated in any indexing scheme. The reasons are: First, “single monad” can be stored very efficiently, namely just by storing the single monad in the monad set. Second, “single range” is also very easy to store: It is sufficient to store the first and the last monad. Third, “multiple range”, as we have argued in Section 8.2.3, is necessary to support in order to be able to store resumptive (discontiguous) linguistic units. It can be stored by storing the monad set itself in marshalled form, perhaps along with the first and last monads.

This leaves us with the following criteria: “unique first monad”, “unique last monad”, “overlapping”, and “violates borders” to decide upon.

In real-life linguistic databases, “unique first monads” and “unique last monads” are equally likely to be true of any given object type, in the sense that if one is true, then the other is likely also to be true, while if one is false, then the other is likely also to be false. This is because of the embedding nature of language explained in Section 8.2.2: If embedding occurs at all within a single object type, then it is likely that both first and last monads are not going to be unique. Conversely, if embedding does not occur, then it is likely that “overlapping” also does not occur, in which case both “unique first monads” and “unique last monads” are going to be true (for all three range types).

Therefore, we decided to see what happens to the lattice if we remove one of the two uniqueness criteria from the list of attributes. The criterion chosen for removal was

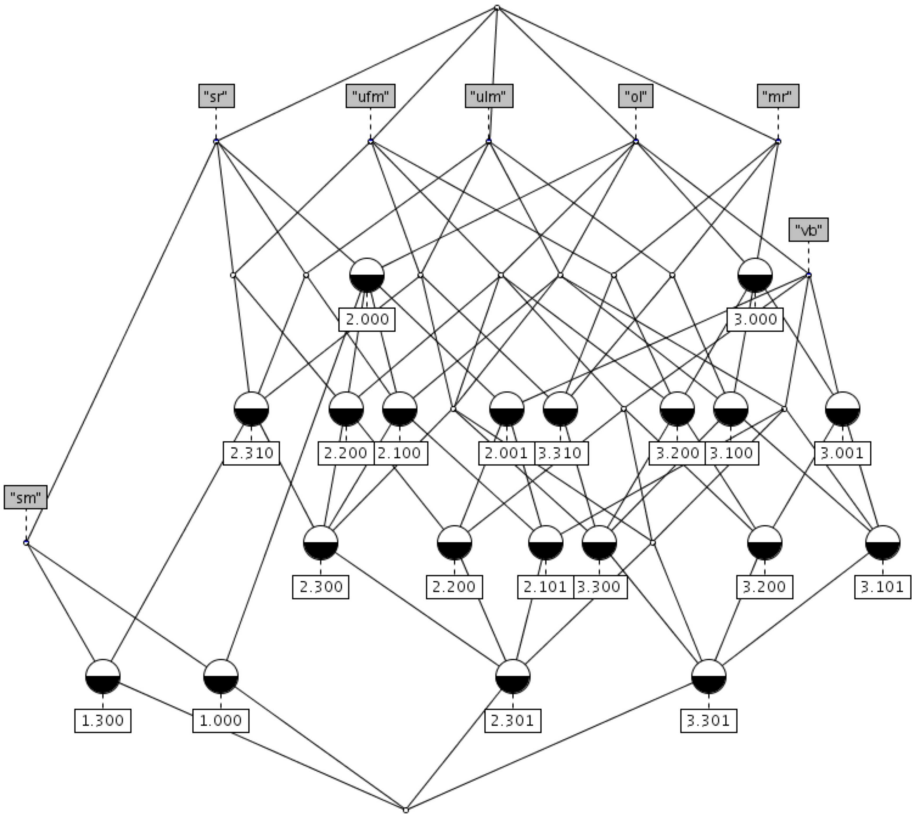



Figure 8.3: The lattice drawn without the “ds” attribute.

“unique last monads”. In Figure 8.4, the result can be seen, with “unique first monads” selected. ConExp reports that “unique first monads” subsumes 11 objects, or 55%.

Similarly, still removing “ds” and “ulm”, and selecting “overlapping”, we get the lattice drawn in Figure 8.5. ConExp reports that “overlapping” subsumes 17 objects, or 85%, leaving only 3 objects out of 20 not subsumed by “overlapping”. This indicates that “overlapping” is probably too general to be a good candidate for treating specially.

It is also noticeable that “violates borders” only subsumes 4 objects. Hence it may not be such a good candidate for a criterion to handle specially, since it is too specific in its scope.

Thus, we arrive at the following list of criteria to handle specially in the database: a) single monad; b) single range; c) multiple range; and d) unique first monads.

## 8.8 Implementation

We have already shown in Chapter 4 how these are implemented, so here we just briefly recapitulate.

The three range types (“`WITH SINGLE MONAD OBJECTS`”, “`WITH SINGLE RANGE OBJECTS`”, and “`WITH MULTIPLE RANGE OBJECTS`”) can be easily implemented in a relational database system along the lines outlined in the previous section.

The “unique first monads” criterion can be implemented in a relational database system by a “unique” constraint on the “first monad” column of a table holding the objects of a given object type. Notice that for multiple range, if we store the first monad of the

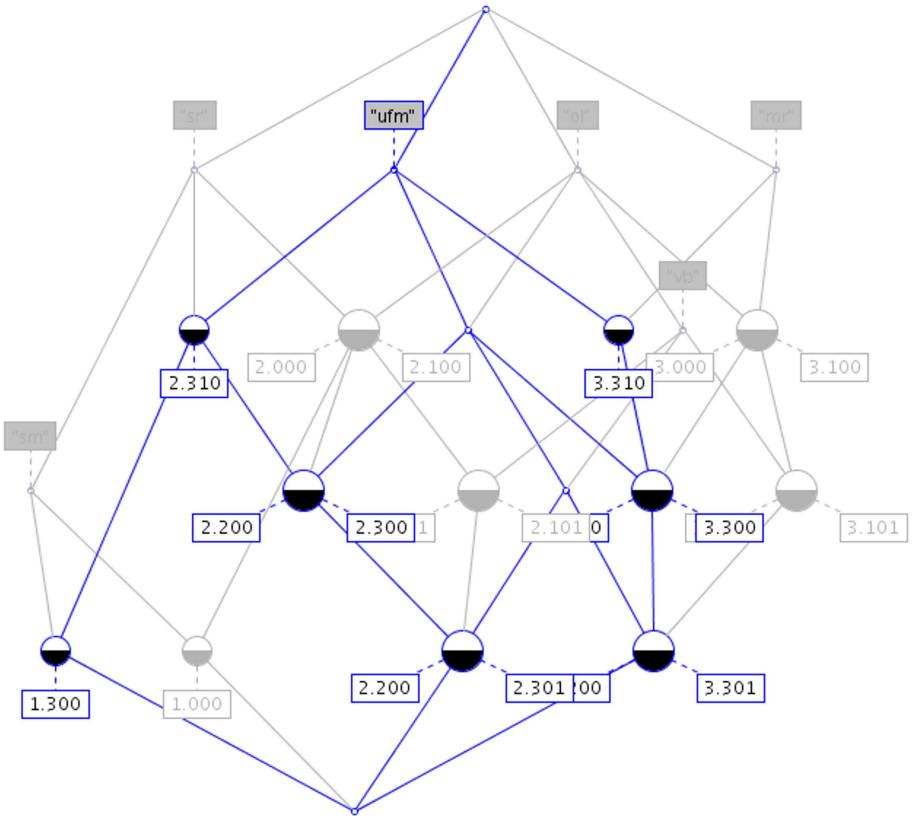



Figure 8.4: The lattice drawn without the “ds” and “ulm” attributes, and with “ufm” selected.

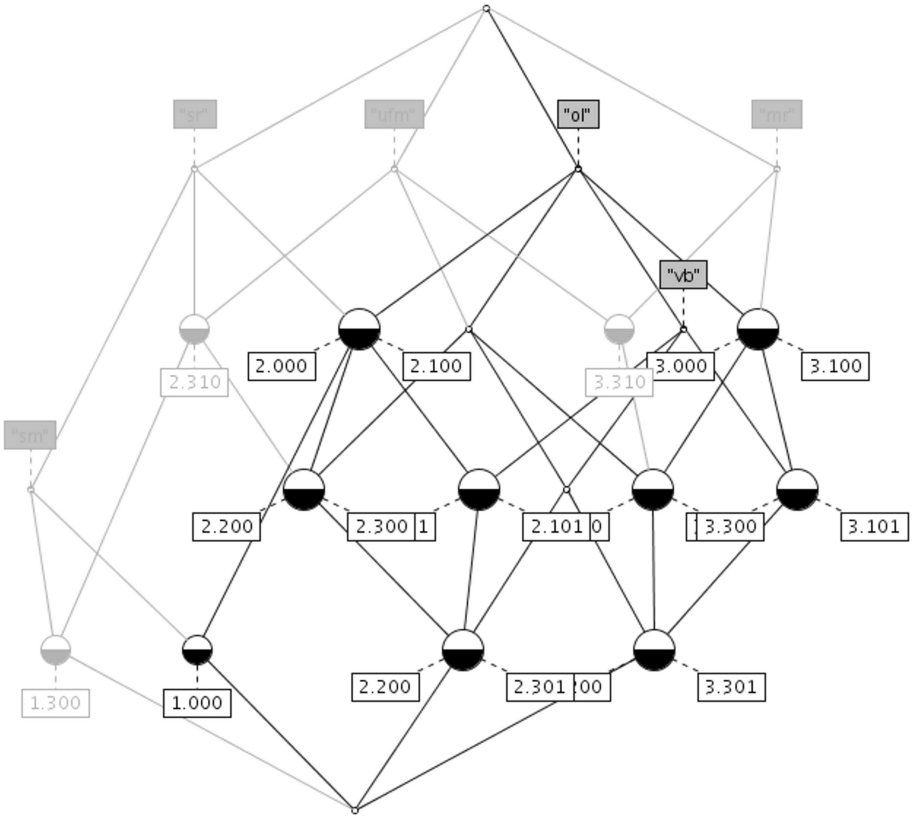



Figure 8.5: The lattice drawn without the “ds” and “ulm” attributes, and with “ol” selected.

| Backend | SQLite 3 | SQLite 2 | PostgreSQL | MySQL |
|---|---|---|---|---|
| Avg. time for DB **without** optimizations | 153.92 | 130.99 | 281.56 | 139.41 |
| Avg. time for DB **with** optimizations | 132.40 | 120.00 | 274.20 | 136.65 |
| Performace gain | 13.98% | 8.39% | 2.61% | 1.98% |

Table 8.2: Evaluation results on an Emdros database, in seconds.

monad set in a separate column from the monad set itself, this is possible for all three range types. Notice also that, if we use one row to store each object, the "first monad" column can be used as a primary key if "unique first monads" holds for the object type.

We have run some evaluation tests of 124 diverse Emdros queries against two versions of the same linguistic database[8], each loaded into four backends (SQLite 3, SQLite 2, PostgreSQL, and MySQL). One version of the database did not have the indexing optimizations arrived at in the previous section, whereas the other version of the database did. The version of Emdros used was 3.0.1. The hardware was a PC with an Intel Dual Core 2, 2.4GHz CPU, 7200RPM SATA-II disks, and 3GB of RAM, running Fedora Core Linux 8. The 124 queries were run twice on each database, and an average obtained by dividing by 2 the sum of the "wall time" (i.e., real time) used for all $2 \times 124$ queries. The results can be seen in Table 8.2.

As can be seen, the gain obtained for MySQL and PostgreSQL is almost negligible, while it is significant for the two versions of SQLite.

## 8.9 Conclusion

We have presented four properties that natural language possesses, namely sequence, embedding, resumption, and non-hierarchic overlap, and we have seen how these properties can be modeled as sets of durations of time.

We have presented the EMdF model of text, in which indivisible units of time (heard or read) are represented by integers, called "monads". Textual units are then seen as objects, represented by pairs $(M,F)$, where $M$ is a set of monads, and $F$ is a set of attribute-value assignments. An object type then gathers all objects with like attributes.

We have then presented some criteria which are derived from some of the four properties of language outlined above. We have formally defined these in terms of objects and their monads. We have then derived an FCA context from these criteria, which we have then converted to a lattice using the Concept Explorer Software (ConExp).

We have then analyzed the lattice, and have arrived at four criteria which should be treated specially in an implementation.

We have then suggested how these four criteria can be implemented in a relational database system. They are, in fact, implemented in ways similar to these suggestions in the Emdros corpus query system. We have also evaluated the performance gains obtained by implementing the four criteria.

Thus FCA has been used as a tool for reasoned selection of a number of criteria which should be treated specially in an implementation of a database system for annotated text.

Future work could also include:

1. Derivation of more, pertinent criteria from the four properties of language;

[8] Namely the WIVU database [Talstra and Sikkel, 2000].

2. Exploration of these criteria using FCA;
3. Implementation of such criteria; and
4. Evaluation of any performance gains.

# Part II

# Applications

# Chapter 9

# Introduction

Part II of my dissertation, entitled “Applications”, summarizes some of the work which I have done to apply empirically the theoretical and methodological concerns discussed in Part I.

As explained in Chapter 1, the empirical basis for my work has, to a large extent, been formed by the Kaj Munk corpus. Other empirical data sets have been employed as well, such as the BLLIP corpus [Charniak et al., 2000] and the TIGER Corpus [Brants and Hansen, 2002], both of which were used in Emdros in [LREC2006]. In addition, the Danish Bible (Old Testament from 1931 and New Testament from 1907) has formed the empirical basis for some of the work done during my PhD, as explained in Chapter 13.

The rest of Part II is laid out as follows. First, in Chapter 10, I discuss the Kaj Munk corpus and its implementation. In Chapter 11, I discuss a web-based tool which I have developed, the purpose being for a group of people to collaborate on annotating a text corpus with comments. In this case, of course, the text corpus is the Kaj Munk Corpus. In Chapter 12, I discuss the use of the implementation of the Kaj Munk corpus in a “Munk Browser”, a piece of software containing some of the works of Kaj Munk. The “Munk Browser” is going to be published by the Kaj Munk Research Centre, and be made commercially available. Finally, in Chapter 13, I discuss an algorithm which Peter Øhrstrøm and I have developed in order to locate quotations from the Bible in Kaj Munk’s works — or, in principle, in any other corpus of text.

# Chapter 10

# Implementation of the Kaj Munk corpus

## 10.1 Introduction

The Kaj Munk Corpus consists of some of the most important works by Kaj Munk. It is the product of an ongoing effort to digitize the *nachlass* of Kaj Munk. This task is performed by a team consisting of student workers as well as Academic staff. Thus I am but one cog in a larger wheel which is part of an even larger machinery, in the process of digitizing Kaj Munk's works, making Kaj Munk's works available to the general public, and performing research on Kaj Munk's texts.

The rest of the chapter is laid out as follows. First, I discuss the nature of the texts, as well as giving a very high level perspective on the digitization process (10.2). In then discuss the "why" of my choice to base the encoding of the texts on XML (10.3). I then briefly discuss the reasons for not choosing the Text Encoding Initiative (TEI) guidelines as a basis for the encoding (10.4). I then detail the most important aspects of the digitization process (10.5). I then discuss how the XML-encoded form of a Munk document becomes an Emdros database (10.6). This is important, since the rest of the research and production which can be carried out depends either directly or indirectly on the Emdros databases formed from the XML-encoded form of the Munk texts. Finally, I conclude the chapter.

## 10.2 The nature of the texts

The nature of the texts varies. Kaj Munk's works can be said broadly to fall into five categories.

1. Plays
2. Journalism
3. Poetry
4. Sermons
5. Prose

The choice of method of encoding has been influenced by this diversity. It seemed obvious that what was needed was a mechanism which could be extended to cover all of these kinds of manuscripts. XML, being extensible, seemed a good choice for dealing with this diversity.

Some of Kaj Munk's texts that we have in the Kaj Munk Archive have been published, others have not. Some have been typed on a typewriter, whereas many only exist in manuscript form, written by Kaj Munk in his own — often hard-to-read — handwriting. The published parts of the corpus could be scanned and converted to text using Optical Character Recognition (OCR) software,[1] but always with manual correction and proof-reading after the OCR-step. The other parts of the corpus (typed, hand-written) needed to be keyboarded manually, again with proof-reading and manual correction as necessary follow-up steps to take. These tasks (OCR-scanning, keyboarding, proofreading, correction) have been undertaken by the excellent student helpers of the Kaj Munk Research Centre. I am grateful for the meticulousness with which they have carried out this work.

I wish to say that my own task is not that of establishment of an "authoritative text". That is, my task is not that of the edition-philologist, namely to establish a precise, authoritative edition of Kaj Munk's text. Rather, I leave this task to others, who come after me in the process.

From a high-level perspective, the process of digitization of the Munk Corpus can be viewed as in Figure 10.1 on the facing page. Please see the caption of the figure for an explanation.

## 10.3 XML as the basis for encoding

A text is not just a string of words. For example, divisions into sentences are marked by punctuation marks in modern texts, and divisions into paragraphs, sections, and chapters are often marked. For plays, the text is structured into acts, scenes, stage directions, actor names, actor lines, and other kinds of text. This kind of structural markup is easily identified on a page by human sight, by means of conventions for typesetting. For example, a paragraph often has its first line indented, a section is marked by a heading in larger typeface than the rest of the text, and a chapter often begins on a new page with an even larger typeface for the chapter name. For plays, an actor name is often printed in boldface or italics flush-left, with a colon or a period to separate the actor name from the words which the actor says.

This kind of markup needs to be formalized (i.e., made explicit) in order for the computer to be able to process the text as a structurally marked-up text. The development and maintenance of a tagging scheme to suit the purposes of the Kaj Munk Research Centre has been one of my main tasks. I have based this development on the XML standard [Bray et al., 2004, Harold and Means, 2004], which is a standard sponsored by the World Wide Web Consortium. "XML" stands for "eXtensible Markup Language" [Bray et al., 2004], and is a specification of a syntax for a set of formal languages, in which it is possible to define one's own Markup Language. One of the benefits of the XML standard is that, when followed, the resulting data are easy to share across platforms and implementations. The reason for this ease of use is that the XML specification specifies a *syntax* which is

[1] For an introduction to some of the techniques used in Optical Character Recognition, see Matteson [1995].

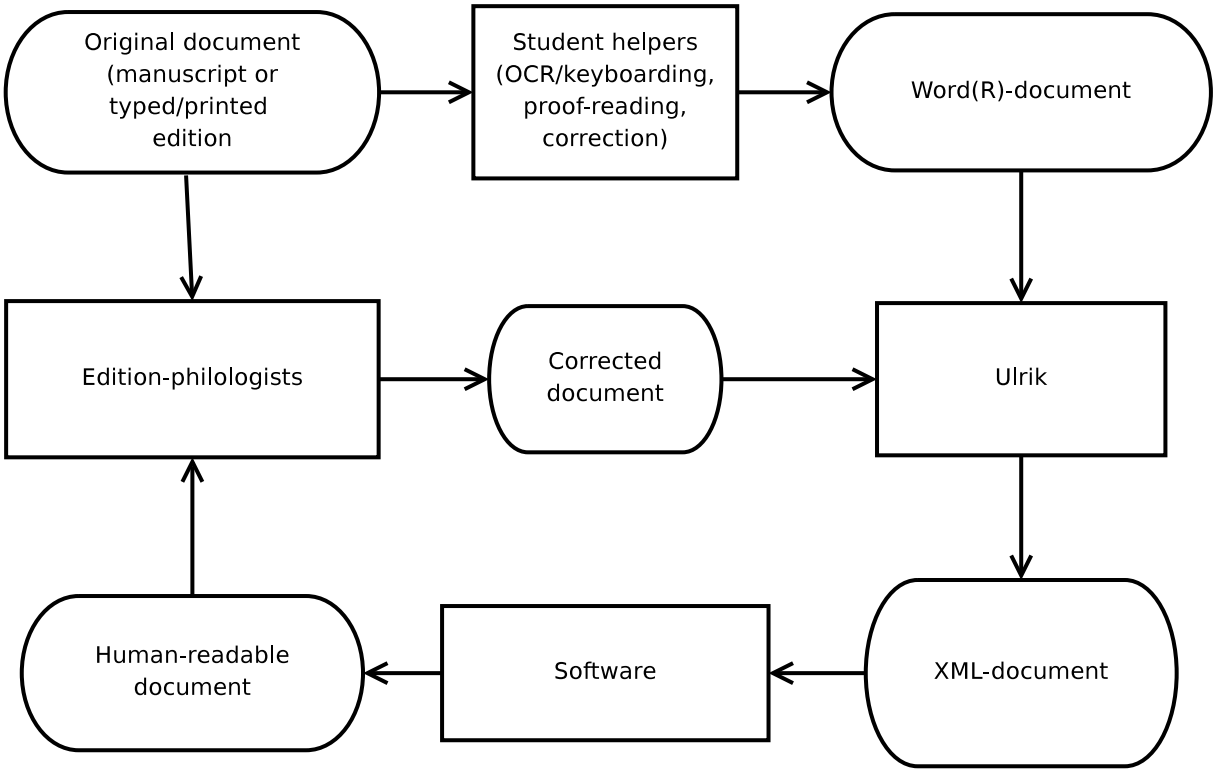



Figure 10.1: The process of digitization of the Munk Corpus, seen from a high-level perspective. The process starts in the upper left corner, with an original manuscript, either in hand-written form, or in typed or printed form. This then goes through a Student Helper, who converts the original document to a Word(R) document. This document then goes through me, who converts the document to XML. This XML document is then processed by some software that I have written, to a form which is again readable by humans. This is as far as we have got in the process to date. After this, the following steps can take place: The human-readable document produced from the XML goes through a team of edition-philologists, who compare the document with the original document(s), producing a corrected document. This is then fed through me again, who makes the necessary corrections to the XML document. At this point, the process could either stop, or go one more round, just to be sure that I had entered every correction found by the edition-philologists perfectly.

easy for software to parse, i.e., it is relatively easy to write a program (a "parser") which reads an XML document and decomposes it into its parts, which can then be interpreted by another program making use of the parser.

Ease of use is one reason for choosing XML as the standard on which to base the Kaj Munk corpus. Another reason is the ease with which XML makes it possible to separate *presentation* from *content*. For example, the XML "schema" which I have developed in order to store the Kaj Munk corpus states that an actor line must have an actor name associated with it. The precise details of the syntax for specifying this actor name, and the extent of the actor line, is immaterial for our discussion. The important point is that, using XML markup, the computer is told unambigously what part of the text is an actor name, what part is stage directions, and what part is the words which the actor speaks. In other words, the *kind* of content is made explicit (or formalized) to the computer. This is separated from the *presentation* of that content. For example, after the conversion to XML, it does not matter whether the actor name was printed boldface or italics, centered or flush left. The string of characters that made up the name of the actor has been unambiguously tagged as being an actor name. This "abstract *representation*" entails that it is possible to present it in any way desirable. For example, it is now possible to specify that, when converting the play-turned-XML back to a presentation form, all actor names should be flush-left, bold, and italic, with a colon after it. Similarly, the XML-ification entails that the data can be used for other purposes, e.g., storing in an Emdros database. The possibility of this separation of content and presentation made XML a good choice.

## 10.4 Text Encoding Initiative

I could have chosen to use the standards developed by the Text Encoding Initiative (TEI)[2] in the preparation of the Kaj Munk Corpus. The reasons for not choosing TEI as the basis for our XML schema include:

1. The complexity of the TEI. The TEI guidelines were designed to help store many kinds of scholarly work. We in the Kaj Munk Research Centre did not need all of that complexity. Granted, the TEI makes it easy to choose only those parts of the TEI "schema" (technically, a "DTD" – Document Type Definition) which are relevant to one's own research. Yet even with this choice of modules, the TEI was deemed too complex to use for our purposes.

2. Control. We wanted to retain full control over how the Munk Corpus was encoded.

3. Ignorance, or: A growing understanding of the structure of the texts. At first, we did not know exactly what a Munk text would encompass. As time went by, a clearer picture was formed, but only through actual experimentation with actual encoding of Munk texts.

4. Special needs. We found out that especially Kaj Munk's plays made some distinctions which would have been difficult to implement, or at least we did not know how to implement, within the TEI guidelines alone.

[2] Barnard and Ide [1997]. Website: http://www.tei-c.org/

## 10.5 Overview of the digitization process

The process used for digitizing the Kaj Munk Corpus is outlined in Figure 10.3 on page 129. The figure should be from top to bottom. The rectangles represent data, whereas the ellipses represent processes or programs which the data passes through in order to be transformed from one kind of data to another.

At the top, we have a work by Kaj Munk in its paper form (printed, typed, or handwritten). This is passed through a human being, who either initiates the OCR-scanning process, or who keyboards the manuscript in. This results in a Word document in which content is not separated from presentation: The actor names are laid out flush left or centered (as the case may be), but there is no formalized indication that a given string of characters is, in fact, an actor name — or any other kind of text. It is simply a Word document which must be interpreted by human sight if it is to be disambiguated as to the kinds of text involved.

This Word-document is then XML-ified. The process is that the Word document is first converted to an XML form (via Abiword[3]) which still does not separate presentation from content: It is still "just" a representation of the typeface and paragraph styles used in the Word document. This XML-representation is then simplified, by a Python script which I have written, into something that resembles very simple HTML. This document is then manually (and semi-automatically) XML-ified to XML that does separate content from presentation. I do this using a text editor with advanced search-and-replace functionality based on patterns[4]. For any given text, it is most often the case that the text has originally been typed or printed in a fairly uniform way, with given typography indicating a given kind of text. For example, within any given text, open-parentheses, followed by something in italics, followed by close-parentheses may mean "stage directions", and can then be found fairly easily using ad-hoc patterns, and converted to the XML form that strips away the parentheses and the italics, instead saying unambigously that this is a stage direction. The reason this process cannot be more automated than it has been is that the typographical conventions for marking out the various kinds of text varies slightly in almost all texts. A sample XML document can be seen in Figure 10.2 on the following page.

Now that the document has been XML-ified, and thus has had its content separated from its presentation, various kinds of use can be made of the XML-ified document. In Figure 10.3, I have shown two such uses: Namely conversion to an Emdros database, and conversion to HTML for showing in a Web browser. I shall explain the process of conversion to an Emdros database more fully in Section 10.6. For now, it suffices to say that the XML is passed through an XML parser, and is then handed to a Python script which transforms the XML either to Emdros MQL, or to HTML.

## 10.6 Conversion to an Emdros database

In order to make the most of Kaj Munk's texts, I have devised a process whereby the texts are enriched with annotations. In order to perform this enrichment, the processes

[3] Abiword is an Open Source word-processor. See http://www.abisource.com

[4] The editor is Emacs, and the patterns are regular expressions. For an introduction to regular expressions, see Martin [1991].

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE munktxt SYSTEM "munkschema.dtd">
<munktxt>
<poetry>
<metadata>
        <metadataitem kind="title" value="Den blaa Anemone"/>
        <metadataitem kind="yearwritten" value="1943"/>
        <metadataitem kind="yearpublished" value="0"/>
</metadata>
<lyrik>
<strofe>
<vers>Hvad var det dog, der skete?</vers>
<vers>mit hjerte haardt og koldt som Kvarts</vers>
<vers>maa smelte ved at se det</vers>
<vers>den første Dag i Marts.</vers>
<vers>Hvad gennembrød den sorte Jord</vers>
<vers>og gav den med sit dybblaa Flor</vers>
<vers>et Stænk af Himlens Tone</vers>
<vers>den lille Anemone,</vers>
<vers>jeg planted der i Fjor.</vers>
</strofe>
</lyrik>
</poetry>
</munktxt>
```

Figure 10.2: XML version of the first stanza of “The Blue Anemone”.

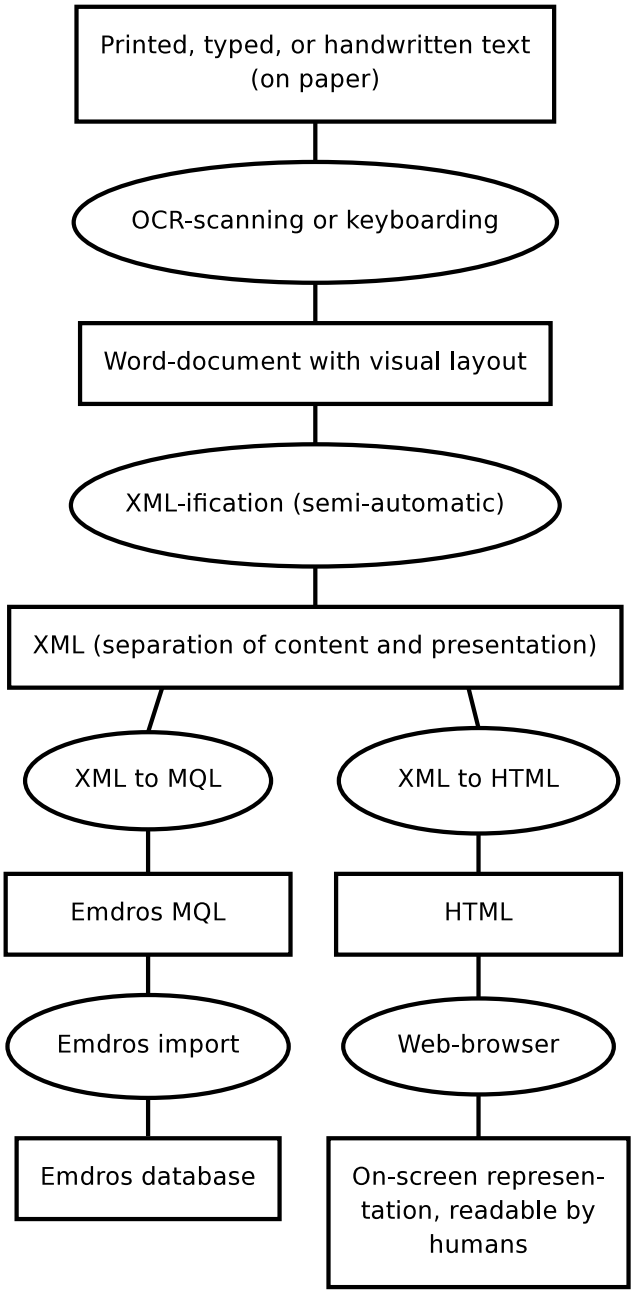



Figure 10.3: Overview of the digitization process. Should be read from top to bottom. The rectangles represent data, whereas the ellipses represent processes that the data passes through in order to be transformed.

described below are carried out. The running example given throughout will parts of the first stanza of “The blue Anemone”.

**Formatting removal:** The XML version of a Kaj Munk text may contain markup that specifies the kind of text involved, as well as some markup that specifies formatting features, such as “italics” or “paragraph start / end”. In order to be able to abstract away from these formatting-details, the XML is transformed to its raw text form. This is done using a very simple Python program which I have written, which merely parses the XML and strips off all of the markup, leaving only the words of the text. Whitespace separates every word, and two newlines separate parts of the content that do not belong together (because, e.g., they are separated by paragraph boundaries, or actor line boundaries).

The output of this process looks like this:

```
Hvad var det dog, der skete?

Mit vinterfrosne Hjertes Kvarts
```

**Orthography modernization:** In 1948, the Danish language underwent an orthography reform. Since all of Kaj Munk’s texts were written before this reform, they all follow the “old orthography”. I have devised an algorithm to take words using the old orthography and transform them to the new orthography, with manual correction where necessary. The algorithm is partly purely algorithmic, in that a number of possible translations from old to new forms are produced. It is also partly data-driven, in that the two Danish reference corpora, Korpus 2000 and Korpus 90 [Andersen et al., 2002], are used to check whether these possible translations are found in either of those corpora. In addition, if this process does not yield a match, the algorithm attempts to split the word in its constituent parts, in case it is a compound, and to check each part as to presence in the reference corpora. In addition, suffix addition and suffix stripping are used on the output of both the initial algorithmic translation and the split forms, in order to see whether these suffix-enhanced / suffix-stripped forms are found in the reference corpora. If none of these processes yield a match in the reference corpora, the form is written to a file for later manual translation to new orthography. These manually translated words are always checked for a match before any other if the above processes start. Once all forms in a text have been given a translation from old to new orthography, it is possible to transform the text, word-by-word, from old orthography to new orthography. This transformation is applied to the output of the previous process.

```
hvad var det dog , der skete ?
Mit vinterfrosne hjertes kvarts
```

**Sentence identification and tokenization:** In order to be able to perform part-of-speech tagging, the text needs to be split into sentences, and also to be “tokenized”. Tokenization is the process whereby a text is split into words, and words are split into “word-parts” and “non-word-parts” (most often punctuation). I have devised a tokenizer which works well for the Danish texts in the Kaj Munk corpus, based on

regular expressions.

The sentence identification is performed very simply, based on the following heuristic: Any question mark ("?") or exclamation mark ("!") marks the end of a sentence. Any period (".") marks the end of a sentence if and only if the next word starts with a capital letter. Of course, this heuristic fails to find the right sentence boundaries in some cases. For example: An abbreviation terminated with a period may be followed by a noun (nouns were capitalized in the old orthography), in which case the sentence-identification program will mark the period as the end of a sentence, even if it is not.

Our running example looks like this after this process:

```
Hvad var det dog , der skete ?
Mit vinterfrosne Hjertes Kvarts
```

**Encoding conversion:** The XML has been encoded in Unicode encoding (specifically, UTF-8). Some of the above output needs to be converted from Unicode to ISO-8859-1 (also known as "latin-1") encoding. This is done with a simple Python script which: a) translates characters with no equivalent in tha latin-1 character set, either to nothing (thereby eliminating the characters), or to some close latin-1 equivalent, and which b) translates characters with an equivalent in latin-1 to their latin-1 equivalent.

I will not show the latin-1 version of our running example, since there are no changes.

**Part-of-speech tagging:** The modernization of orthography mentioned above was motivated in part by a desire for end-users to be able to employ both new forms and old forms when searching. Another motivation was that Center for Language Technology (CST) at the University of Copenhagen kindly provided me with automated web-access to their part-of-speech tagger and lemmatizer. This part-of-speech tagger and lemmatizer needed to have "new ortography" as its input, and hence I needed to provide a method of translating the text to "new orthography". CST's tagger is based on that of Eric Brill [Brill, 1993].

The output from CST's tagger has then been used to train another tagger, namely the T3 tagger of Ingo Schröder [Schröder, 2002]. This tagger was inspired by the TnT tagger of Thorsten Brants [Brants, 2000]. It is my estimation that Ingo Schröder's T3 tagger yields slightly better results than that of CST, based on informal comparison of some texts tagged with both taggers. This is in line with Schröder's [2002] findings, as well as those of Brants [2000].

The CST-tagged version and the T3-tagged version can be seen in Figure 10.4.

**Stemming:** Stemming is the process whereby affixes are stripped from a word in order to obtain what is called a "stem". For example, the words "cool", "cools", "cooling",

```
[PRON_INTER_REL hvad hvad]
[V_PAST var være]
[PRON_PERS det det]
[ADV dog dog]
[TEGN , ,]
[UNIK der der]
[V_PAST skete ske]
[TEGN ? ?]
```

(a)

```
Hvad PRON var V det PRON dog ADV , PUNCT der UNIK skete V ?
PUNCT
```

(b)

Figure 10.4: The part-of-speech tagged version of the first verse of the first stanza of "The blue Anemone". Part (a) is the output from the CST pos-tagger and lemmatizer, while part (b) is the output from the T3 tagger.

"cooled", and "coolness" would all be stemmed to the stem "cool". In my work, I have used the Snowball stemmer[5] produced by Martin Porter and Richard Boulton [Porter, 1980]. The Snowball stemmer has a built-in stemmer for Danish.

**Stitching:** After all of the above processes have run, the results are all stitched together into files with MQL statements that will create the Emdros database. This is done using a fairly complex Python program which I have written. The program reads all of the above source files, and compares them in order to ascertain which words / new forms / tokens / parts of speech / lemmas / stems belong together, and where the sentence boundaries are. The XML form is also read, thereby creating objects such as "actor line", "stanza", "verse", "annotation", etc. For technical reasons, the annotations are written to a file which is separate from the other object types. The end result is two files per input XML file: One file with MQL statements which will create the annotations, and one file with MQL statements which will create the rest of the objects. Part of the MQL output can be seen in Figure 10.5 on the next page.

**Emdros database creation:** After the MQL has been created, it is run through the mql program, thereby creating the database. What can be done with the database after its creation includes: a) research, b) display of the documents, and c) annotation of the texts. I discuss these in later chapters.

The whole process can be seen in Figure 10.6 on page 134.

## 10.7 Conclusion

In this chapter, I have scratched the surface of the processes involved in digitizing and implementing the Kaj Munk corpus. Many details have been left out. I have first discussed the nature of the texts, followed by a defense of my choice of XML as the basis for encoding Kaj Munk's texts. I have then briefly discussed why I have not used the Text Encoding Initiative's (TEI's) schema. I have then given an overview of the digitization

[5] See http://snowball.tartarus.org/

```
CREATE OBJECTS WITH OBJECT TYPE [munktxt]
CREATE OBJECT FROM MONADS={1-6}[  kind:=poetry;
yearwritten:="1943";
basename:="Den-blaa-anemone";
title:="Den blaa Anemone";
yearpublished:="0";
]
GO
CREATE OBJECTS WITH OBJECT TYPE [lyrik]
CREATE OBJECT FROM MONADS={1-6}[]
GO
CREATE OBJECTS WITH OBJECT TYPE [strofe]
CREATE OBJECT FROM MONADS={1-6}[]
GO
CREATE OBJECTS WITH OBJECT TYPE [vers]
CREATE OBJECT FROM MONADS={1-6}[]
GO
CREATE OBJECTS WITH OBJECT TYPE [Token]
CREATE OBJECT FROM MONADS = {1}[
wholesurface:="Hvad ";
surface:="Hvad";
surface_lowcase:="hvad";
new_surface:="hvad";
new_surface_lowcase:="hvad";
lemma:="hvad";
t3pos:=PRON;
cstpos:=PRON;
cstwholepos:="PRON_INTER_REL";
morphology:=(INTER,REL);
surface_stem:="hvad";
new_surface_stem:="hvad";
prefix:="";
suffix:=" ";
break_occurs_after:=0;
]
CREATE OBJECT FROM MONADS = {2}[
wholesurface:="var ";
surface:="var";
surface_lowcase:="var";
new_surface:="var";
new_surface_lowcase:="var";
lemma:="v\xc3\xa6re";
t3pos:=V;
cstpos:=V;
cstwholepos:="V_PAST";
morphology:=(PAST);
surface_stem:="var";
new_surface_stem:="var";
prefix:="";
suffix:=" ";
break_occurs_after:=0;
]
... more Tokens follow ...
GO
```

Figure 10.5: Part of the MQL output for the first verse of the first stanza of “The Blue Anemone”.

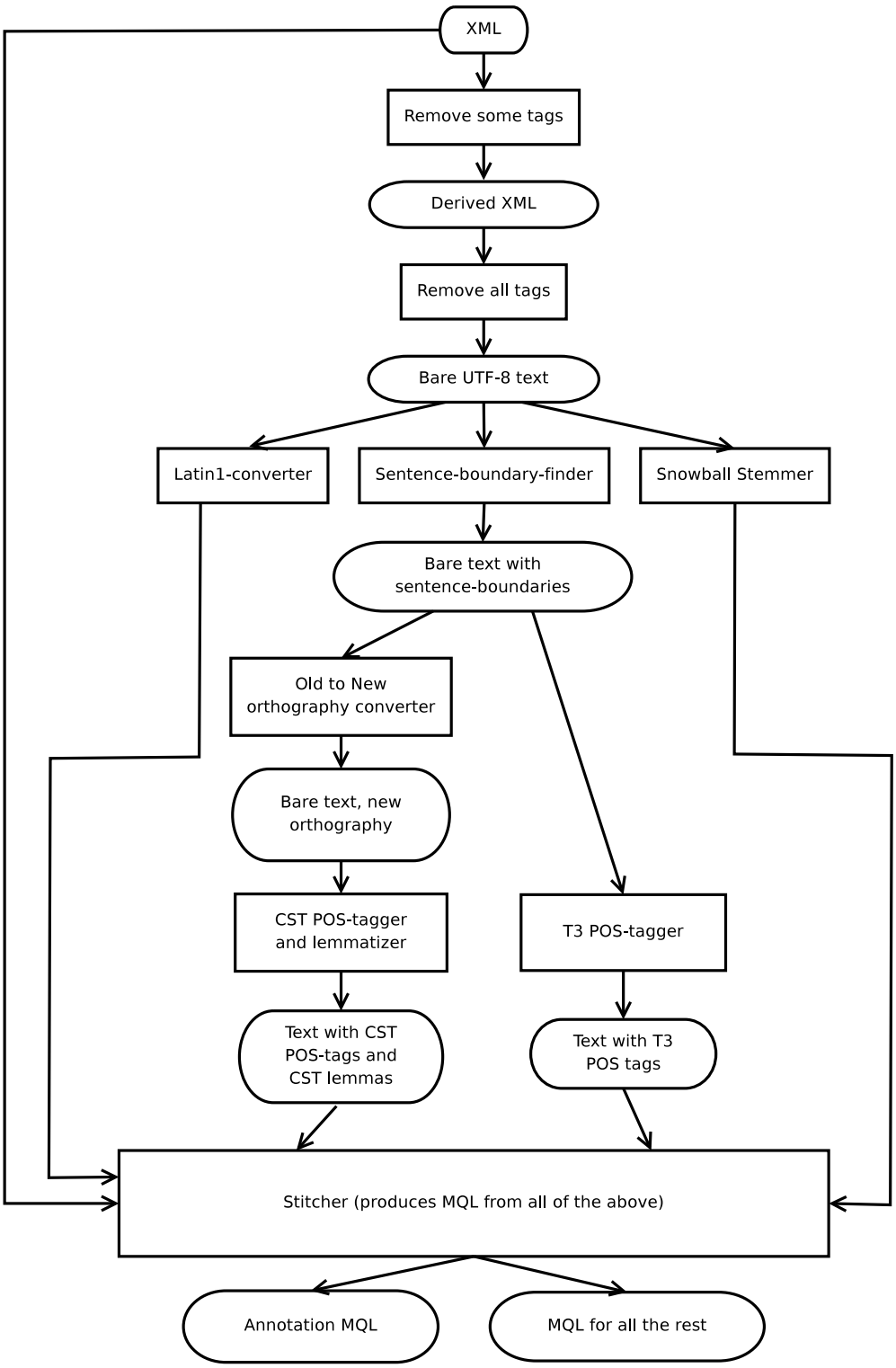



Figure 10.6: Overview of the “XML to Emdros MQL” process.

process. I have then discussed in some detail how a text from the Kaj Munk corpus becomes an Emdros database, from its XML form. The Emdros database can then be used for various purposes, some of which will be discussed in the following chapters.

# Chapter 11

# Principles of a collaborative annotation procedure

## 11.1 Introduction

As part of the tasks of the Kaj Munk Research Centre, it is our duty to make the works of Kaj Munk electronically available to the general public. In order to meet the needs of the end-users of such an electronic edition, it is a good idea to supply the text with annotations in the form of notes on certain words and phrases. We have employed a person to author these annotations, and I have written a web-based tool to aid this person in entering the annotations. Others are helping the main annotator, and so the tool is really a web-based, collaborative annotation tool.

The rest of the Chapter is laid out as follows. First, I discuss the general ideas and principles behind the annotation tool (11.2). Second, I discuss the actual implementation of these general ideas and principles (11.3). Finally, I conclude the chapter.

## 11.2 The general idea

The general idea is to provide a web-based, colloborative environment in which persons who are knowledgeable about the writings of Munk (or any other author) may enter annotations which are attached to specific words. The environment is based on Emdros, which stores the annotations on behalf of the annotators, as well as storing the texts to be annotated.

It is assumed that the text-establishment phase has already been carried out, that is, it is assumed that the texts are more or less stable as regards the specific wording. Some leeway is built into the system, however, for changes to the underlying texts.

The following principles apply:

1. All annotators are identifier to the system by means of a username and a password.

2. An annotation always carries the information that it was created by a certain annotator.

3. No annotation can be altered or deleted except by its own creator.

The reason for principle (2) is that we wish to be able to identify the person who is responsible for any given annotation. Assigning responsibility is a good idea in a project such as this, for several reasons. First, it is standard practice in encyclopedias and other reference works to specify after an article who has authored the article. Second, apart from being standard practice, it is also a good idea, seen from the perspective of research, to be able to go back and ask the person for more evidence or clarification.

The reason for principle (3) above is two-fold. First, we wish to maintain and retain the "responsibility-path" of any given annotation, as being assignable only to one person. This is the reason it is not possible for other users of the system to alter an annotation. Second, we wish to maintain a good working relationship among the annotators, and preventing others from deleting the annotations of their peers is one way of contributing towards that goal. Such deletion may occur either accidentally, or because one is unhappy with the contents of someone else's annotation. Preventing either of these was seen as a goal. Thus, if one annotator disagrees with another annotator, the only recourse the first annotator has is to add another annotation with a differing opinion.

This obviously has some relation to Wikipedia[1]. Wikipedia is a collaboratively edited encyclopedia, and has an ideal of "an upward trend of quality, and a growing consensus over a fair and balanced representation of information"[2]. This is also the goal of our annotation system. However, we do not allow users other than the creator to edit or delete an annotation. How is this to be resolved?

The answer is that it must be resolved at a level above the annotation itself: One user, let us call him or her "A", may write an annotation, *a*. Another user, let us call her or him "B", may disagree with *a*. The only recourse which B has is to write another annotation, *b*, of the same word, as mentioned above. Then, user A may choose to alter *a*, after which B can elect to delete *b*. If A and B cannot agree, the disagreement must stand. The key notion here is that any disputes must be resolved outside of the annotation-system, by the authors themsleves. This is a necessary consequence of the desire to maintain a clear "path of responsibility" back to precisely one author per annotation.

The system is based on Emdros, not surprisingly. The process around the system actually runs in a periodic cycle, as shown in Figure 11.1. The caption of this Figure has a lot of information, and the reader is advised to peruse both the Figure and its caption.

I now discuss how I have implemented these principles.

## 11.3 The implementation

### 11.3.1 Introduction

In this Section, I discuss the implementation of the principles laid out in Section 11.2. I do so in the following order: First, I discuss how annotations are represented in their XML form and in their Emdros form. Second, I discuss an overview of the processes which the annotator can go through as he or she does his or her job. Third, I discuss how the annotations, being stored in an Emdros database, are added back into the XML. Finally, I conclude the Section.

[1] http://www.wikipedia.org/

[2] See http://en.wikipedia.org/wiki/Wikipedia:About

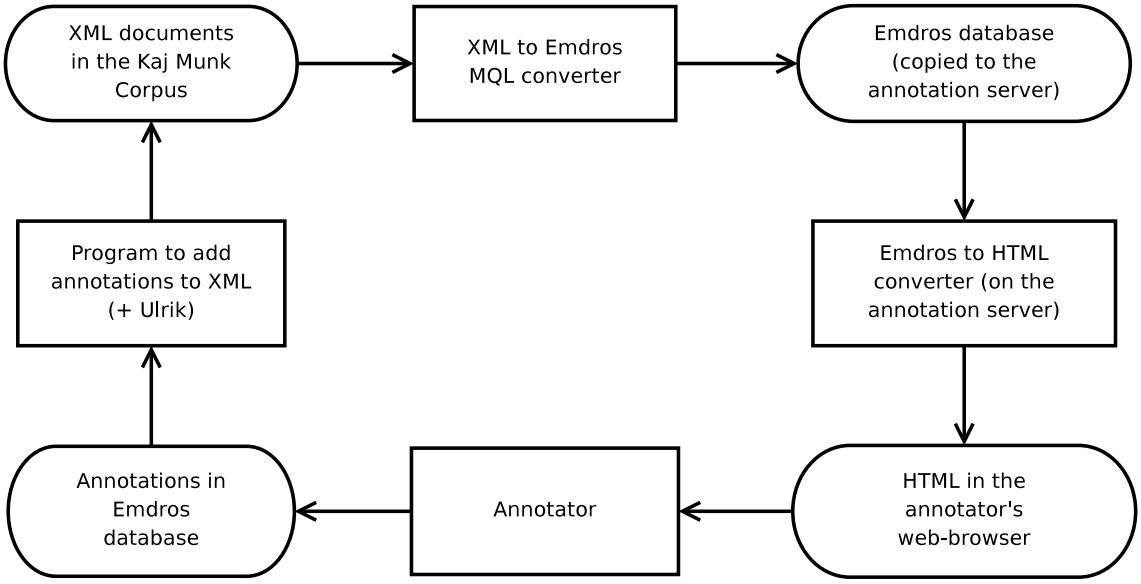



Figure 11.1: The annotation process cycle. The process starts in the upper left hand corner, with the XML version of the Munk texts. These are passed through an automatic conversion-step, producing an Emdros database. This Emdros database is then copied to the annotation server. On this server, the annotation tool uses the objects in the Emdros database to display the text as HTML to the annotator in the annotator's web-browser. This is then fed through the annotator, who adds or deletes annotations, which the tool then puts into the Emdros database. From time to time, the annotation tool is "taken down" for maintenance, and the annotations are added back into the XML using a Python script (mostly automatic), with manual addition where necessary. This XML can then be used to produce a new Emdros database, containing the new annotations, which is then copied to the annotation server, where the annotators can then add more annotations.

### 11.3.2 XML form and Emdros form

An "annotation" in the Munk XML DTD is an empty element which has the following attributes:

**id_d:** The id_d of the object in the Emdros database. This remains stable throughout the lifetime of the annotation, and is, of course, unique to any given annotation. These id_ds are never reused.

**annotator:** The username of the annotator who created and authored the annotation.

**subject:** A short string of words describing the contents. Usually, it is the word which is being annotated.

**datemodified:** A date and time in ISO 8601 format, namely the date and time at which the annotation was created, or last modified, whichever is later.

**content:** The contents of the annotation.

No HTML is allowed in the contents of the annotation. Instead, the following "light markup" is allowed:

- [*__this is bold__*]
- Two or more newlines constitute a paragraph break.

Each of the above attributes maps to a specific feature of the "annotation" object type:

```
CREATE OBJECT TYPE
WITH SINGLE MONAD OBJECTS
[annotation
   annotator : STRING FROM SET;
   subject : STRING;
   content : STRING;
   datemodified : STRING;
]
GO
```

The object type is declared "`WITH SINGLE MONAD OBJECTS`" in order to specify that they only occupy one monad, namely the monad of the last word of the word or phrase being annotated (or, put another way, the monad of the word being annotated directly after which the annotation must make its presence known to the user, e.g., with a link such as [*]). The object type is not declared as "`HAVING UNIQUE FIRST MONADS`", because any given word may be subject to more than one annotation.

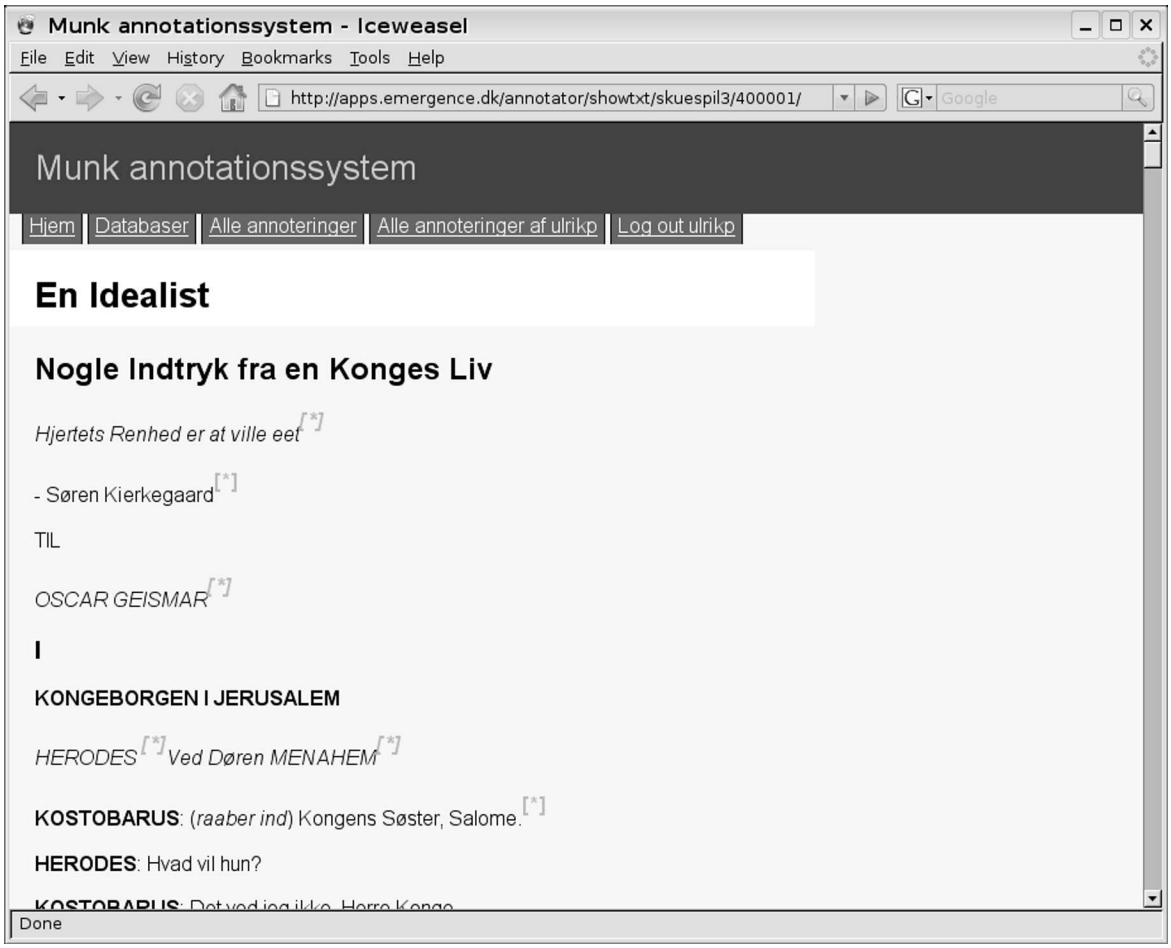



Figure 11.2: A sample window from the collaborative annotation tool. This shows the text of a play “An Idealist” (Danish, “En Idealist”).

### 11.3.3 Overview of annotation tool

The annotation tool starts with a “login” screen, which requires the user to enter their username and password. Nothing can be done with the tool unless the user is logged in with the right credentials.

Once logged in, the user is presented with a list of Emdros databases present on the annotation server. The user can choose one, e.g., “Plays”.

After having chosen a database, the user can then choose a munktext from the database in which to add or delete annotations. This brings up an HTML version of the text. For an explanation of how the Emdros objects are converted to HTML, please see Section 12.4. A sample window is shown in Figure 11.2.

Each word in the HTML version of the text is a hyperlink pointing to a page on which it is possible to create an annotation. The annotation must be given a subject and a content. The subject is pre-filled with the word on which the user clicked. The annotation is then created in the Emdros database, giving it the name of the user as the “annotator” feature. The annotation is, of course, created with the MQL statement `CREATE OBJECT`. An example is shown in Figure 11.3. An excerpt from the EMdF database may be seen in Figure 11.4.

Returning to the text, the user can now see that a little green mark has been added

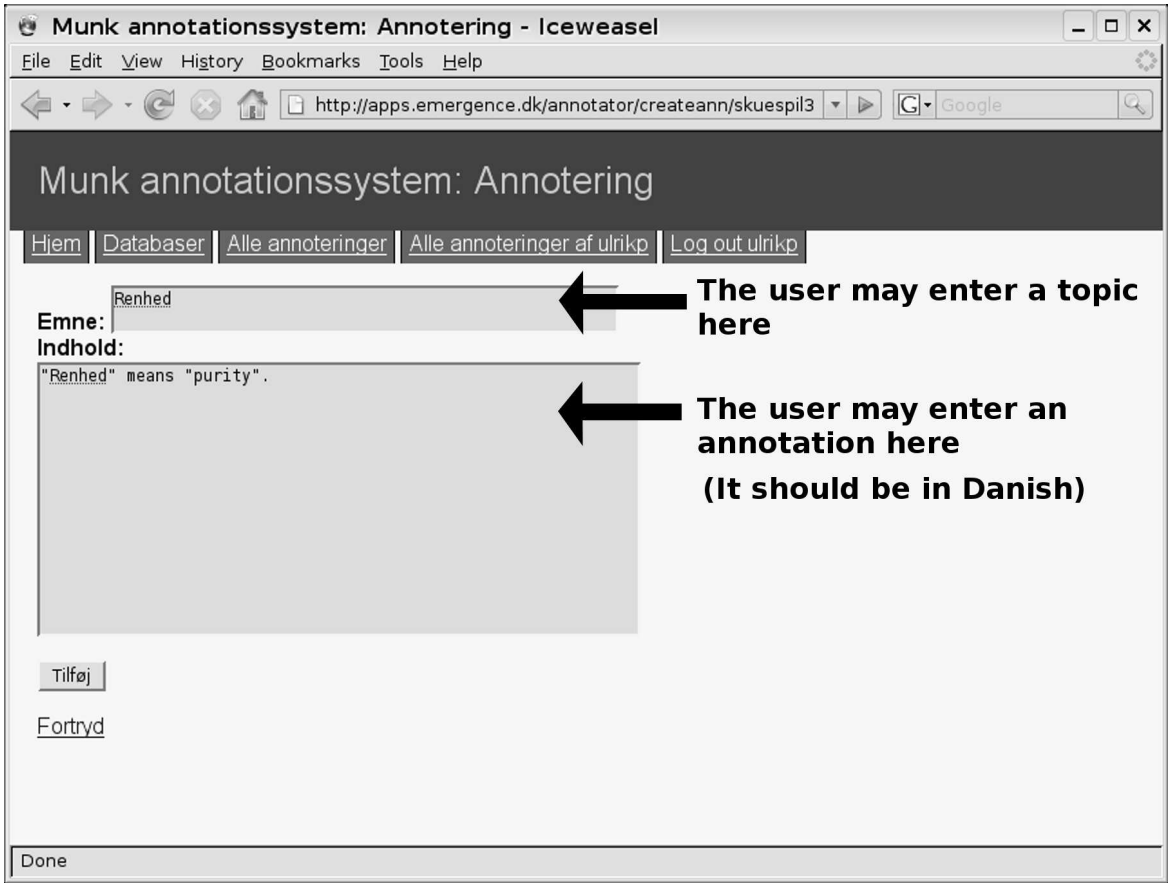



Figure 11.3: A sample window from the collaborative annotation tool. This shows an annotation being created. “Emne” means “Subject”, and “Indhold” means “Content”. “Tilføj” means “Add”, and “Fortryd” means “Cancel”.

| Monad | 80009 | 80010 | 80011 | 80012 |
|---|---|---|---|---|
| **Word** | 230121 | 230122 | 230123 | 230124 |
| surface | Hjertets | Renhed | er | at |
| **annotation** | | 10320001 | | |
| annotator | | ulrikp | | |
| subject | | [*Renhed*] | | |
| datemodified | | 2008-05-05 12:00:00 | | |
| content | | "Renhed" means "purity". | | |

Figure 11.4: An excerpt from a sample EMdF database containing an annotation of the word “Renhed”. Notice how the only object shown for “annotation” is the annotation with id_d 10320001. Notice also how all the features given in Section 11.3.2 (viz., self, annotator, subject, datemodified, and content) are present.

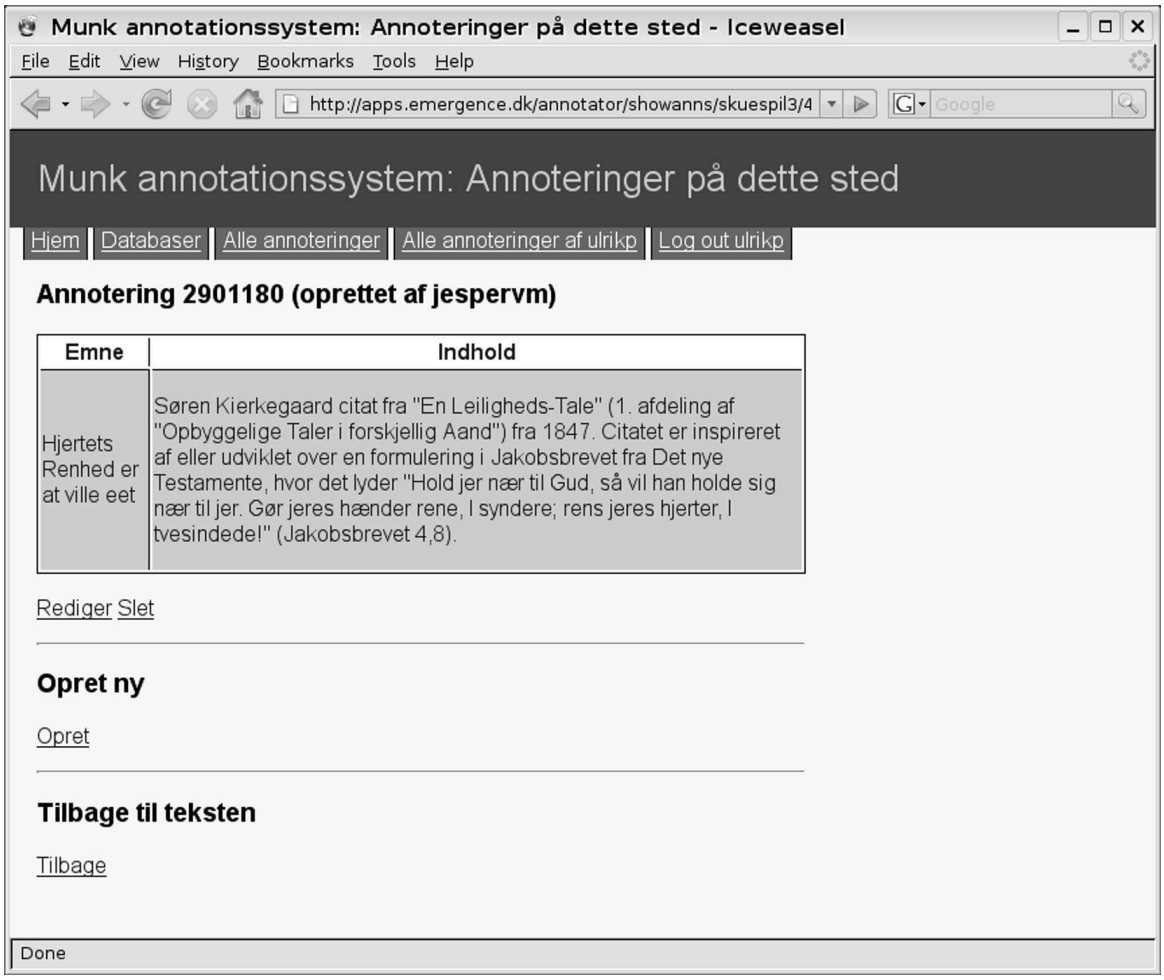



Figure 11.5: A sample window from the collaborative annotation tool. This shows an annotation created by “jespervm”. Note that, even though there are links to “Rediger” (in English, “Edit”) and “Slet” (in English, “Delete”), these links will not have any effect, since the user logged in is currently “ulrikp” in the picture.

after the word which was annotated. Clicking the word again will take the user to a page containing the annotations on this particular word. Each annotation is shown with the subject and content of the annotation. If the current user was also the user who created the annotation, they are free to update or delete the annotation. An update is done with the `UPDATE OBJECT` MQL statement, whereas a deletion is done with the `DELETE OBJECT` MQL statement. If the user is not identical to the creator of the annotation, they can only view the annotation. An example can be seen in Figure 11.5.

It is also possible to see a list of all annotations, sorted by user and by subject. Again, each subject is a hyperlink which will take the user to a page on which they can either modify or delete the annotation (if they created it themselves), or they can view the annotation (if they were not the original creator).

The whole tool has been implemented — by me — in Python, using the Django framework[3].

[3] See http://www.djangoproject.com/

### 11.3.4 Adding annotations back into the XML

As explained in the caption for Figure 11.1 on page 139, the process is cyclic, meaning that the Emdros database is created from the XML, the annotations are added to the Emdros database, and the annotations are then fed from the database back into the XML, at which point the cycle can start anew.

The annotations are "harvested" from the database in such a way that a program can later add the annotations back into the XML automatically — or with little supervision. This is done in the following manner.

1. The munktxt objects are retrieved.
2. Within the monads of each munktxt object, the annotation objects are retrieved.
3. Each annotation object is treated in the following way:
   (a) The database is queried using a topographic query, to see if this exact surface form occurs anywhere else in the same munktxt. If not, the annotation is written out to a file, together with information that identifies the word form which it annotates.
   (b) If, on the other hand, the word form is not unique within the munktxt from which the annotation came, then the database is queried to get the surface forms of the tokens right before and right after the annotation. This tri-gram is then checked for existence in the database. If only one such trigram exists, the process stops. Otherwise, the process keeps going in both directions, until a *n*-gram has been found which is unique within the munktxt. Of course, this process has to take into consideration factors such as: a) hitting the beginning or end of the munktxt, and b) hitting a boundary such as an "actor line" or "verse line". Of course, if a text only contains one word-form and one word-form only, for all tokens in the text, this strategy will fail. In practice, a unique *n*-gram is always found within a reasonable number of tokens measured from the annotation itself. The *n*-gram is then written to a file, along with the annotation, and information about which word in the *n*-gram is to carry the annotation.

This method produces, for each munktxt, a file containing pairs of (annotation, *n*-gram), where the *n*-gram (which may be a uni-gram) uniquely identifies the place in the munktxt where the annotation is to be inserted.

After these files have been "harvested", it is time to add the annotations back into the XML. This process can be mostly automated, and has been in a Python script which I have written. In some cases, however, the script cannot figure out where to place an annotation, perhaps because the n-gram spans some XML element boundary which it was either impossible to take into account in the "harvesting" process, or which the "harvesting process" simply failed to take into account. For example, the harvesting process currently does not take tags such as ... (for emphasis) into account, and so an emphasized string of words may prevent the "annotation-adding" Python script from working correctly. In such cases, the "residue" annotations are written to yet another file, which then must be processed manually in order for the annotations to be inserted. The majority

of annotations can be added automatically, while a small number of residual annotations need manual insertion.

Thus the annotations have been added back into the XML, and the process can start anew with XML-to-Emdros conversion, and renewed annotation.

One area of future development will be to delete annotations from the XML which have been deleted in the Emdros database. One way to do this would be to delete from the XML any annotation which was not present in the Emdros database. This could be done based on id_ds, since id_ds are never re-used.

### 11.3.5 Conclusion

In this Section, I have shown how the principles laid out in Section 11.2 have been implemented in a web-based collaborative annotation tool. The tool is written in Python, using the Django framework. It makes use of Emdros for keeping the Munk documents, and for keeping the annotations.

I have first shown what an annotation is, in this system, both in XML form and in Emdros form. I have then discussed what the annotation system looks like from a user's perspective. I have then discussed how the annotations can be added back into the XML. The reason for adding them back is that the annotations must be re-created each time the Emdros database is re-created, and the Emdros database is created from the XML form, which is the "canonical" form of the database.

## 11.4 Conclusion

In this Chapter, I have discussed a web-based, collaborative tool for annotating any collection of texts (in particular, the Kaj Munk Corpus) with notes which are tied to particular words. Other kinds of notes could be conceived of, but these are of the kind which we in the Kaj Munk Research Centre find most useful and promising.

I have first discussed the general ideas and principles behind the annotation tool, followed by a section on the actual implementation. The actual implementation has been described in terms of the annotations themselves, in terms of the tool seen from a user's perspective, and in terms of the process which adds the annotations back into the XML from the Emdros database.

The annotations are meant for consumption by the general public, and as such find a nature place in the "Munk Browser" software, to which we turn in the next chapter.

# Chapter 12

# Principles of the Munk Browser software

## 12.1 Introduction

One of the foremost tasks of the Kaj Munk Research Centre, at which I have been employed throughout my PhD studies, is to publish the works of Kaj Munk electronically. To this end, a large-scale digitization project has been carried out (as reported in Chapter 10) as a preliminary step. A further step is publication, and as part of this effort, I have written a piece of software, a "Munk Browser" desktop application. The Munk Browser is called "Kaj Munk Studieudgave" in Danish, or "Kaj Munk Study Edition" in English. I have written it in C++, using the wxWidgets[1] framework for cross-platform support. The program runs on Windows(R), Mac OS X(R), and Linux(R).

The purpose of the program is to give the user easy-to-use access to a subset of Kaj Munk's writings. We at the Kaj Munk Research Centre have defined five categories of potential users:

1. The pupil or teacher in an elementary school who wishes to read Kaj Munk's works.
2. The student or teacher in a high school who wishes to read Kaj Munk's works.
3. The pastor who wishes to read Kaj Munk's works (especially his sermons).
4. The scholar who wishes to study Kaj Munk's writings.
5. The "interested man on the street" who wishes to read Kaj Munk's works.

In designing the user interface and functionality, I have tried to meet the needs of most of these categories of users.

The rest of the chapter is structured as follows. First, I give a functional overview of the program (12.2), showing the capabilities of the program. Second, I give a modular overview of the program, discussing the various software modules involved (12.3). I then show how some of the theory developed in previous chapters has been used in the Munk Browser (12.4). Finally, I conclude the chapter (12.5).

[1] See Smart et al. [2005] and http://www.wxwidgets.org/

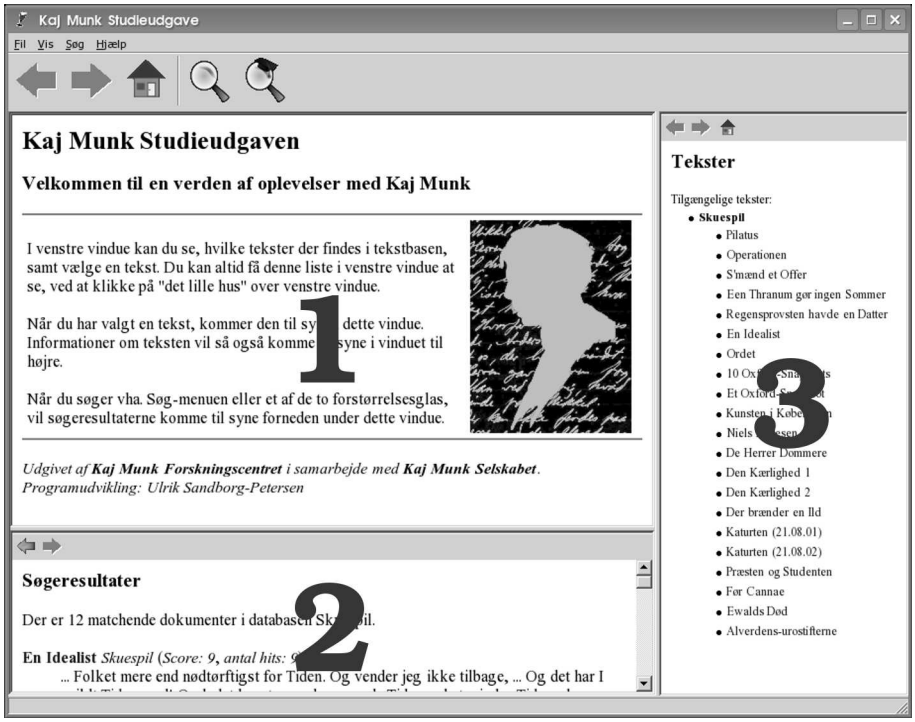



Figure 12.1: Screenshot of the “Munk Browser” software application.

## 12.2 Functional overview

A screenshot of the “Munk Browser” can be seen in Figure 12.1. The area of the screen labelled “1” is the “text window”. It is the main window, in which Kaj Munk’s works are displayed. The area of the screen labelled “2” is the “information window”, in which secondary information is displayed (usually not written by Kaj Munk, but by someone else). The area of the screen labelled “3” is the “search results window”, in which the results of a search will appear. The “toolbar” is the bar of buttons placed above the “text window” and “information window”. The “menu-bar” appears at the top, and the “status-bar” appears at the very bottom.

The software application is called a “Munk Browser” because the basic interface metaphor used is that of an Internet browser. That is, each window has a “history of navigation”, which can be navigated using “backwards” and “forwards” buttons (painted as backwards- and forwards-pointing arrows). In addition, there may be “links” within each window which, when clicked, will take the user to another text, either in the “text window”, or in the “information window”.

The screen with which the user is met when opening the program looks much like the screenshot in Figure 12.1. (The only difference between the initial view and Figure 12.1 is that in Figure 12.1, some search results from a previous search appear.) As can be seen, the “information window” contains a tree-like overview of the texts available (in this case, some of the plays of Kaj Munk). The “text window” contains a “welcome screen” with information on how to use the program. The titles of the works in the “information window” are links which, when clicked, will open that particular work in the “text window”. Figure 12.2 on the next page shows the screen after the work “Ordet” has ben opened. As can be seen, the “information window” now contains some “meta-information” about the work selected, such as “title” (“Titel”), “year of writing” (“Skriveår”), “year of publica-

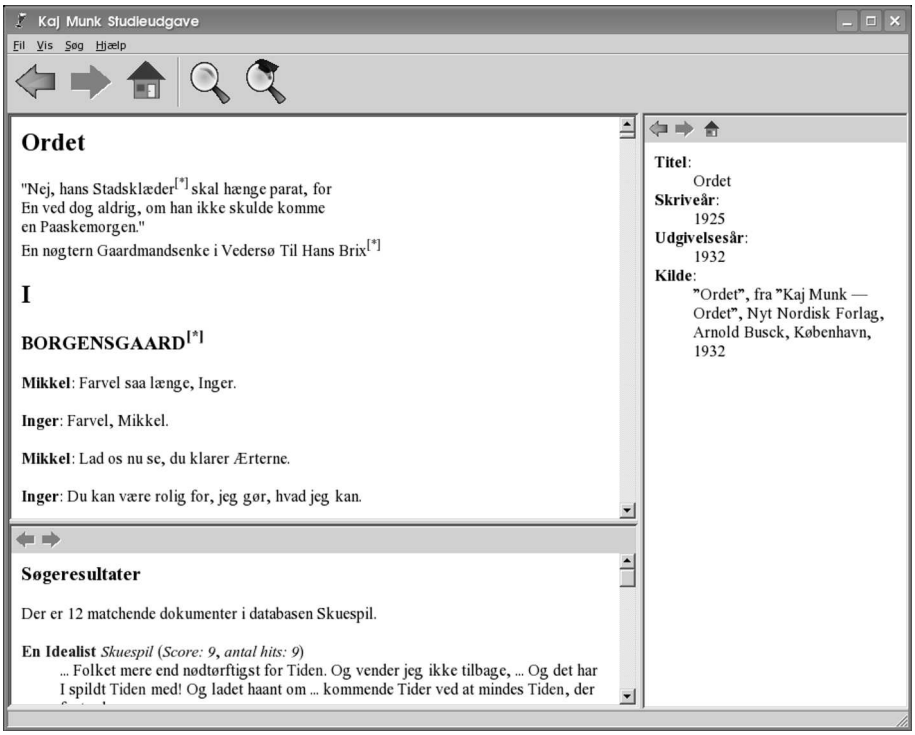



Figure 12.2: Screenshot of “Munk Browser” software application, after the play “Ordet” has been opened. Note the meta-information to the right in the “information window”.

tion” (“Udgivelsesår”), and “source” (“Kilde”).

As explained in Chapters 10 and 11, a work in progress at the Kaj Munk Research Centre is to annotate all of Kaj Munk’s plays with comments from scholars. The nature of these comments has been dealt with in Chapter 11, so suffice it here to say that whenever a word has been annotated, a link appears after that word, which, when clicked, will open that particular comment in the “information window”. An example can be seen in Figure 12.3 on the following page, where the link after “Brix” in Figure 12.2 has been clicked. The right-hand “information window” now displays the comment. It can also be seen that the left-arrow in the “information window” is no longer “grayed out”, but is active: Clicking it would take the “information window” back to the previous page, which in this case is the “meta-information” on the play “Ordet”.

The “magnifying glass” without the “academic hat” which appears in the toolbar can be used to launch the “simple search”. It is simply a dialog box in which the user can enter one or more words. When the user then clicks on the button that starts the search, a search will be performed, and the results will be shown in the “search results window”. These results are ordered after the ranking which each “document” (e.g., a “play” or a “sermon”) receives after the results have been harvested. The results are shown document-by-document, and each result set has an excerpt from the document from which it came. Within each result set, the words which were searched for are shown as “live links”, which, when clicked, will open that particular document in the “text window”. The link will take the user directly to the place in the document at which the words in question occur. The details of the search are described in Section 12.4.

The “magnifying glass” *with* the “academic hat” which appears in the toolbar should launch the “advanced search” dialog. Although this dialog has been implemented, it is

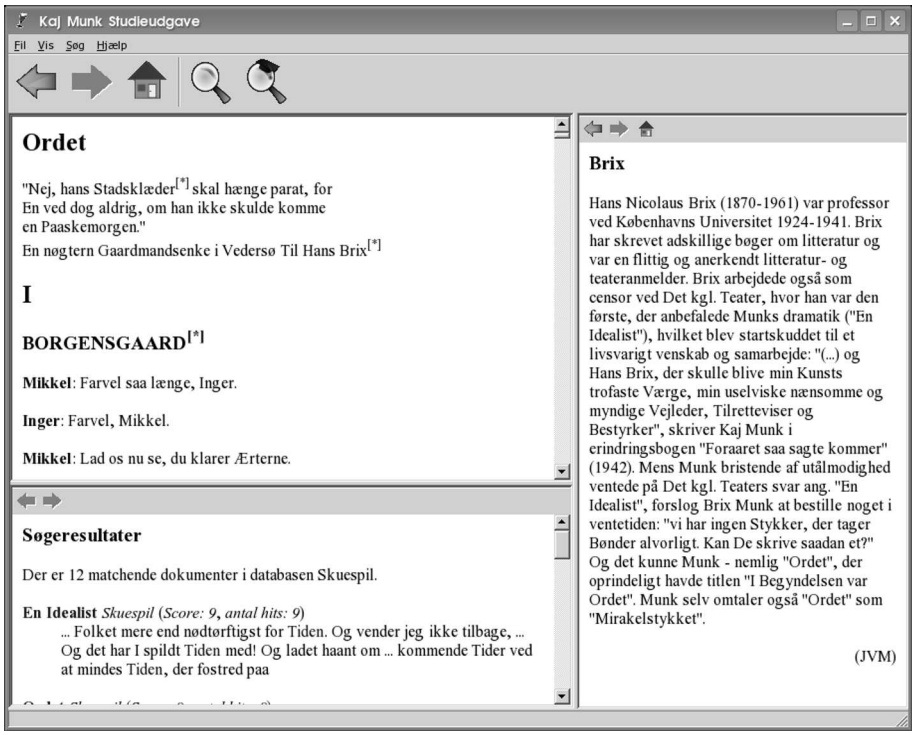



Figure 12.3: Screenshot of “Munk Browser” software application, after the play “Ordet” has been opened, and the link after “Brix” has been used to open the comment for “Brix” in the right-hand “information window”.

disabled at the moment, since it requires more research to make it simple enough to use. Some of the details of the already-implemented “advanced search” dialog are discussed in Section 12.4.

A “Find on this page” functionality is almost implemented, but has been disabled in the current version because it is not complete.

Pressing “F1” or using the “Help” (“Hjælp”) menu will bring up the User’s Guide. Finally, an “About” box can be reached from the “Help” (“Hjælp”) menu.

This concludes my functional overview of the Munk Browser.

## 12.3 Modular overview

An overview of the most important modules and their “calls” relations can be seen in Figure 12.4. Some of the “house-keeping” modules have been left out of the figure for simplicity. These house-keeping modules all have to do with either: a) Making sure that the “backwards and forwards” browsing history mechanism works, or b) providing a document hierarchy which is read from the Emdros databases which are installed alongside the program. The latter module simply reads all “munktxt” objects in a given database, harvests their “meta-data items”, and constructs a data structure which holds these “abstractions” of each munktxt document.

I now explain the nature of the arrows in Figure 12.4. A single-headed arrow from box A to box B means that module A calls module B. For example, the “Main GUI” calls the “Advanced Search” module, which in turn calls the “Harvester”. The Harvester, in turn,

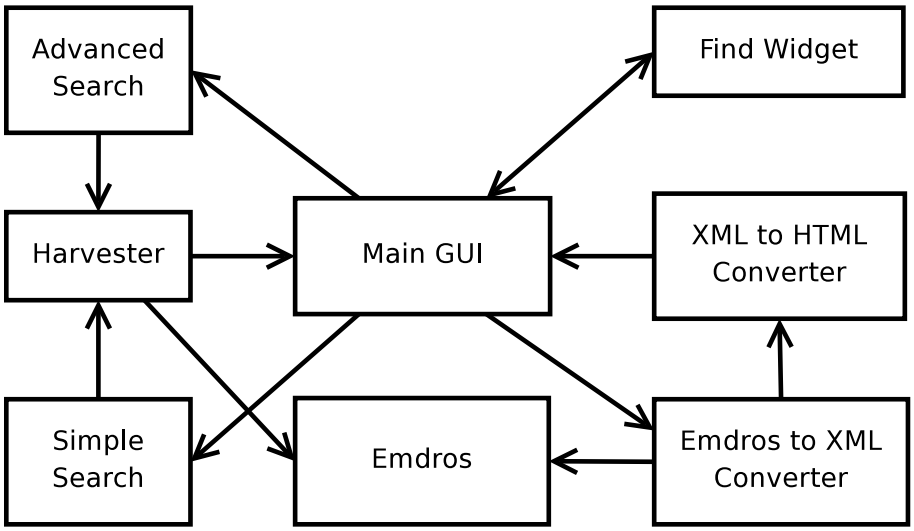



Figure 12.4: An overview of the most important software modules in the "Munk Browser". Some house-keeping modules have been left out.

calls "Emdros", as well as calling the "Main GUI". A double-headed arrow means that both modules at either end of the arrow call each other (in this diagram, the only example is the "Find Widget", which is called by the "Main GUI", and which may also call the "Main GUI" back).

As can be seen, the central module is the "Main GUI" (where "GUI" stands for "Graphical User Interface"). It is implemented using classes from wxWidgets. It is the module which responds to user input, and which displays the output in one of the three "browser windows" (labelled "1", "2", and "3" in Figure 12.1). Since it is the module that responds to user input, it is also the module which calls the other modules whenever an action is needed, such as starting a simple or advanced search, or opening a document.

There are four main call-chains:

1. In order to show a document, the "Main GUI" may call the "Emdros to XML Converter", which in turn calls both Emdros and the "XML to HTML Converter", which in turn calls the "Main GUI" to show the text desired in the appropriate window.

2. In order to perform a search, the "Main GUI" may call the "Simple Search" module, which calls the "Harvester", which in turn calls both Emdros (to obtain the results), and calls the "Main GUI" to display the "search results".

3. Something very similar to call-chain (2) happens when an "advanced search" is launched.

4. In order to search an already-opened document, the "Find Widget" window may be opened by the "Main GUI". This window then sits at the bottom of the screen and responds to user keystrokes and mouse-button clicks. This, in turn, causes the "Find Widget" to call the "Main GUI" in order to scroll to the desired, found positions in the document.

In the next section, I show in some detail how each of these are done.

## 12.4 Application of theory

### 12.4.1 Introduction

In this section, I show how the theory presented in Part I is brought to bear on the real-world problems inherent in the functionality of the Munk Browser. I first show how a "munktxt" document is "harvested" and then converted to XHTML for viewing. I then show how the "simple search" works, followed by a subsection on how "harvesting" works. I then show how the "advanced search" works. Finally, I show how the "Find Widget" will work once it is finished. This covers the main applications of the theory presented in Part I.

### 12.4.2 Showing a munktxt document

When the user clicks on a document link, the program will display that document in the "text window". In order to do so, the following chain of events occurs:

1. The "Emdros to XML Converter" is called. It is given information about the kind of link involved (e.g., munktxt document or comment within a munktxt). It is also handed at least one of the monads in the document to be retrieved.

2. The "Emdros to XML Converter" then uses the harvesting algorithms outlined in Chapter 7 to retrieve all of the objects from all of the object types which are needed in order to re-create the document which must be shown. This is done by calling Emdros. While the necessary objects are being retrieved, they are placed into an "in-memory EMdF database", as outlined in Section 4.6 on page 54. This is done inside the "Emdros to XML Converter". The algorithm is as follows:

   (a) From the monad handed to the "Emdros to XML Converter", use a `GET OBJECTS HAVING MONADS IN` statement to retrieve the munktxt object which is intended. The necessary meta-data are also retrieved. It is placed into the "in-memory EMdF database".

   (b) From this munktxt object, a monad range is extracted, starting at the first monad of the munktxt object, and ending at the last monad of the munktxt object.

   (c) For each object type to be retrieved, issue a `GET OBJECTS HAVING MONADS IN` statement, retrieving both the objects and the necessary features, using the monad-range obtained in Step (b). Place each object into the "in-memory EMdF database".

   The database depicted in Figure 12.5 will serve as the running example. For the remainder of this discussion, the reader is invited to imagine that GET OBJECTS HAVING MONADS IN { 20001-20010 } statements have been issued in order to retrieve all relevant object types, and any relevant features. The results have then been placed into an in-memory EMdF database.

3. Once all necessary objects have been retrieved into the "in-memory EMdF database", the document is re-assembled, monad-by-monad, into an XML representation. This

| Monad | 20001 | 20002 | 20003 | 20004 | 20005 | 20006 | 20007 | 20008 | 20009 | 20010 |
|---|---|---|---|---|---|---|---|---|---|---|
| **Word** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| surface | Hvad | var | det | dog, | der | skete? | Mit | vinterfrosne | Hjertes | Kvarts |
| **verse** | 11 | | | | | | 12 | | | |
| **stanza** | 13 | | | | | | | | | |
| **poetry** | 14 | | | | | | | | | |
| **munktxt** | 15 | | | | | | | | | |
| title | Den blaa Anemone | | | | | | | | | |

Figure 12.5: EMdF database of the first two verses of "The Blue Anemone". Notice that we are pretending that these two verses constitute the whole stanza, the whole "poetry" object, and the whole "munktxt" object.

is done by having an ordered list of object types, containing all of the object types whose objects were retrieved in Step 2. The order in which these object types occur in this list shows the order in which start-tags must be opened. The reverse list, of course, shows the order in which end-tags must be closed. The algorithm works as follows:

(a) Initialize a list of strings, intended to result in the XML output, and append the "XML declaration" to the list. The "XML declaration" simply says that this is an XML document, that it uses XML version 1.0, and that the character encoding is UTF-8.[2] The list will consist of "snippets" of XML, i.e., strings which, by themselves, are not full XML documents, but which taken together will constitute a full, validating XML document. At the end of the process, the strings in the list will be concatenated into one large string to produce the final XML document.

(b) For each monad *m* in the range starting at the first monad of the munktxt involved, and ending at the last monad of the munktxt involved:

i. For each object type *OT* in the list of object types:

A. Query the in-memory database to see if there is one or more objects of type *OT* at monad *m*. If yes, retrieve them from the in-memory database, convert them to an XML representation, and append them to the list of XML snippets to result in the output. Some object types will result in a start-tag (such as a "paragraph" or "actor line"), while others will result in an empty element tag (e.g., "comment" or "linebreak"). Tokens also result in an empty element, with attributes which indicate the surface of the token as well as the monad. An example could be: `<t m="20001" surface="Hvad"/>`, indicating that the surface of the token at monad 20001 is "Hvad".

ii. For each object type *OT* in the reverse of the list of object types:

[2] UTF-8 is an 8-bit encoding which can encode a Unicode character stream. See http://www.unicode.org for more information.

```
<?xml version='1.0' encoding='utf-8' ?>
<munktxt>
  <metadata>
    <metadataitem kind='title' value='Den blaa Anemone'/>
  </metadata>
  <poetry>
     <stanza>
       <verse>
          <t m="20001" surface="Hvad"/>
          <t m="20002" surface="var"/>
          <t m="20003" surface="det"/>
          <t m="20004" surface="dog,"/>
          <t m="20005" surface="der"/>
          <t m="20006" surface="skete?"/>
       </verse>
       <verse>
          <t m="20007" surface="Mit"/>
          <t m="20008" surface="vinterfrosne"/>
          <t m="20009" surface="Hjertes"/>
          <t m="20010" surface="Kvarts"/>
       </verse>
     </stanza>
  </poetry>
</munktxt>
```

Figure 12.6: XML document arising out of the "Emdros to XML Converter" for the database shown in Figure 12.5.

A. Query the in-memory database to see if there is one or more objects of type *OT* at monad *m*. If yes, retrieve the objects if they need to be closed with close-tags, and close the tags.

(c) Join the list of XML snippets with the empty string in between, resulting in one XML document. Return this XML document. The XML document looks something like that in Figure 12.6.

4. The resulting XML document is handed to the "XML to HTML converter". This consist of a simple SAX-parser[3] which I have written, which parses the XML document and converts each XML tag to a series of zero or more XHTML[4] tags, possibly with textual content. A token element, represented with the "`t`" empty element tag, is converted to the following string:
"`<a name="m##monad##">##surface##</a>`", where the "place-holder" called "`##monad##`" is replaced with the monad from the "`t`" element, and where the "`##surface##`" place-holder is replaced with the surface from the "`t`" element. For example, the XML element "`<t m="20001" surface="Hvad"/>`" will result in the XHTML string "`<a name="m20001">Hvad</a>`". The "`<a name="...">`"

[3] See http://www.saxproject.org/
[4] See [W3C contributors, 2002].

```
<?xml version='1.0' encoding='utf-8'?>
<html>
  <head>
    <title>Den blaa Anemone</title>
  </head>
  <body>
    <h1>Den blaa Anemone</h1>
    <p>
      <a name="m20001">Hvad</a> <a name="m20002">var</a>
      <a name="m20003">det</a> <a name="m20004">dog,</a>
      <a name="m20005">der</a> <a name="m20006">skete?</a>
      <br/>
      <a name="m20007">Mit</a> <a name="m20008">vinterfrosne</a>
      <a name="m20009>Hjertes</a> <a name="m20010">Kvarts</a>
    </p>
  </body>
</html>
```

Figure 12.7: XHTML document arising out of the XML document shown in Figure 12.6.

# Den blaa Anemone

Hvad var det dog, der skete?
Mit vinterfrosne Hjertes Kvarts

Figure 12.8: Rendering of the XHTML document in Figure 12.7.

tag, of course, starts a "name anchor", which "names" the surface text between the start- and end-tags of the "a" element. This "name anchor" can then be used later, when needing to scroll to a particular monad (see the following sections, on simple search and harvesting). Another example could be the end-tag "</verse>'', which results in a "forced line-break" tag, or "<br/>". Similarly, a "<stanza>" start-tag results in a "paragraph begin" tag, or " ", whereas a "</stanza>" end-tag results in a "paragraph end" tag, or " ".

The resulting XHTML document can be seen in Figure 12.7.

5. The resulting XHTML document is then handed to the "Main GUI" window, which displays it in the appropriate window ("text window", in the case of "munktxt" objects). The running example can be seen in its XHTML rendering in Figure 12.8.

A similar, yet simpler, method is applied when needing to retrieve a comment to be shown in the "information window".

I now show how "simple search" works.

### 12.4.3 Simple search

The "Simple Search" can be described from a high-level point of view as happening in four steps:

1. The user is asked, in a simple dialog, for one or more word. The dialog only contains a text-field, an "OK" button, and a "Cancel" button. If the user presses "Cancel", the process is aborted here. If the user presses "OK", the string of words entered is passed on to the next step.

   For the purposes of this discussion, let us assume that the user entered the string "Hvad Dog".

2. The string is tokenized (split into tokens, based on whitespace). This results in a list $L$ of tokens.[5] This list is turned into a topographic MQL query as follows:

   (a) Start a list of strings, to become the result. Call it "`result`". As with the XML snippets, these are "MQL snippets" which, alone, do not constitute a valid MQL query, but which, when concatenated, will constitute a valid MQL query.

   (b) Start a list of strings, called "`objectBlocks`".

   (c) For each token $t$ in $L$:

      i. Let `term` be the lower-case version of $t$, escaped with backslashes where appropriate, so that it can be fed into an MQL string in double quotes.

      ii. Append to "`objectBlocks`" the string which represents an object block with object type "`token`", where the following features of "token" are compared agains `term`:
      a) `surface_lowcase`, b) `new_surface_lowcase`, and c) `lemma`, with ''`OR`'' in between. For example, the term "hvad" ("what") would result in the following object block:
      "`[token surface_lowcase = "hvad" OR new_surface_lowcase = "hvad" OR lemma = "hvad"]`"[6]

   (d) Join the `objectBlocks` list into one string, with ".." in between. Prefix the string with "`SELECT ALL OBJECTS WHERE [munktxt GET title`", and append the string "`] GO`" to the string. This becomes the resulting topographic MQL query. In our running example, the full query is now as in Figure 12.9 on the facing page.

   Notice that every query has a surrounding "munktxt" object block, and that the title of the munktxt object is retrieved. This becomes important when describing the harvesting algorithm below.

[5] The list of tokens is ["Hvad", "Dog"].

[6] In our running example, the "`objectBlocks`" list now contains the following strings:

- `[token surface_lowcase = "hvad" OR new_surface_lowcase = "hvad" OR lemma = "hvad"]`
- `[token surface_lowcase = "dog" OR new_surface_lowcase = "dog" OR lemma = "dog"]`

```
SELECT ALL OBJECTS
WHERE
[munktxt GET title
   [token surface_lowcase="hvad"
          OR new_surface_lowcase="hvad"
          OR lemma="hvad"
   ]
   ..
   [token surface_lowcase="dog"
          OR new_surface_lowcase="dog"
          OR lemma="dog"
   ]
] GO
```

Figure 12.9: The MQL query resulting from the “simple search” query-building process.

3. This topographic MQL query is handed off to the harvesting module, which runs the query against Emdros, and returns an ordered list of “solutions”.

4. This solution is rendered as XHTML, and shown in the “search results” window of the “Main GUI”.

I now describe the harvesting algorithm in detail.

### 12.4.4 Harvesting

Before I describe the harvesting algorithm, I need to define a data structure, namely a “**Solution**”.

A Solution is a C++ object which describes one munktxt object which has at least one “hit” inside of it. It contains the following members[7]:

1. A database name (in order to be able to tell the “Emdros to XML Converter” from which database to load the document).

2. A munktxt title (which can be used to show the title when transforming to XHTML).

3. A monad (which will be set to some monad inside the munktxt; and thus can be used to identify the munktxt object in the database).

4. A list of strings (which will become the list of snippets to show together with the document title in the search results window).

5. An overall score.

6. A hit-count.

[7] In Object Oriented Programming terminology, a “member” is a variable associated with the state of an object (or class, in the case of static members). Thus a “member of an object” is a piece of the computer’s memory which holds a value of some specific type.

The Solution class has a number of methods,[8] two of which are important for the explanation of the harvesting algorithm below. I therefore mention them here:

- addWord(): Which adds a word to the snippets. This is done during harvesting. One of the parameters passed to this method is the monad associated with the word. This monad is assigned to the "monad" member of the Solution object. Thus, whichever word is added last, gets to define the "monad" member of the Solution object.
- getHTML(): This method is used when the list of Solution objects which results from a harvesting procedure needs to be transformed to XHTML. It assembles the information in the object into an XHTML snippet which represents the title, the score, the hit-count, and the snippets.

Given this data structure, we are now in a position to understand the the harvesting algorithm employed in the Munk Browser.

The general Harvesting procedure runs as follows:

1. Maintain a list of Solution objects.
2. For each database present:
   (a) Run the topographic input MQL query is run against the database.
   (b) If the result is an empty or failed sheaf, go to the next database.
   (c) Otherwise, if the result is a sheaf with content, run the "harvest" algorithm (shown below) on the sheaf. This results in a list of Solution objects, which are appended to the main list of Solution objects.
3. Once all databases have been processed, sort the list of Solution objects in reverse order of solution score, such that the solutions with the highest scores appear first in the list, and the solutions with the lowest score appear last.
4. If the list is empty, return with some XHTML to the effect that there are no "hits".
5. Otherwise, if the list is not empty, return with an XHTML document that contains the XHTML snippets of each Solution object, obtained by calling the "getHTML()" method on each Solution.

The "harvest" procedure mentioned above works as follows: The input is three-fold: A sheaf, a database name, and a length of monads to use for the context of each snippet (context_length).

1. Call harvest_sheaf() on the sheaf. (See below.) This results in: a) A set of munktxt titles and munktxt ranges of monads covering each munktxt involved, and b) A set of monads which contains the "focus monads" for this solution (i.e., the monads of the words which were "hits").

[8] In Object Oriented Programming terminology, a "method" is a piece of program code which is associated with an object (or class, in some programming languages, including C++). Generally, a method is for reading the internal state of the object, altering the state, or computing something on the basis of the state.

2. Traverse the set of focus monads, creating a new set of monads which consists of the ranges that arise when taking each focus monad, plus the stretch of "context_monads" on either side. This results in a set of monads corresponding to the words to retrieve.

3. If this set is empty, stop the procedure here.

4. If, on the other hand, the set is not empty, then retrieve all the words of the set, using a simple GET OBJECTS HAVING MONADS IN query. This results in a map data structure, mapping word-monads to objects representing the surface string of each word.

5. Traverse the set of munktxt titles obtained in Step (1) above. For each triple (first_monad, last_monad, munktxt_title):

   (a) Create a Solution object.

   (b) Add snippets of words and count "hits". Also calculate a score. (See the Excursus below for how the score is calculated.)

The "harvest_sheaf" method is a recursive function which takes as input a Sheaf, and traverses it, all the while gathering munktxt titles, munktxt first/last monads, and "focus" monads (from the monads of tokens). The details of the method have been left out of the thesis, since they are relatively trivial.

**Excursus: Calculation of score** The score is currently set to the sum of those distances (counted in monads) between each pair of "focus-monad" whose length is less than 50, subtracted from 50.

For example, if there are three "hits", each at the monads 1, 11, and 100, then the score will be 50 - (11 - 1) = 40. The interval from 11 to 100 will not be counted into the score, because it is too long (longer than 50). This method of calculating the score may seem somewhat arbitrary, and it is. Yet it is our experience that it is at least to some degree meaningful, in that it helps ordering (or "ranking") the search results in a way that makes sense. The intuition behind the score is that if hits "cluster" (that is, occur closely together), then a higher score should be awarded to the Solution. Given that the list of focus monads can be called "$x_1, x_2, \ldots, x_n$," the score $S$ can be formalized as:

$$S = \sum_{i=2}^{n} 50 - f(x_i - x_{i-1})$$

where the function $f(m)$ is defined as:

$$f(m) = \begin{cases} 50 & \text{if } m \geq 50 \\ m & \text{if } m < 50 \end{cases}$$

Of course, this is impossible when the list contains only one item. In such cases, the score is 0.

This formula is, of course, ad-hoc. There are many other ways published in the literature for calculating a relevance-score.

The number "50" is also somewhat arbitrary, but can be theoretically validated as follows: The average sentence-length in Kaj Munk's plays and sermons can be calculated

| | Plays | Sermons | Plays + Sermons |
|---|---|---|---|
| Sentence-count | 28595 | 1680 | 30275 |
| Word-count | 613641 | 109483 | 723124 |
| Average | 21.5 | 65.2 | 23.9 |

Table 12.1: Sentence-lengths in the Kaj Munk Corpus. Based on the plays and the sermons. Calculated using Emdros and the script shown in Figure 12.10.

to be 23.9 words, on the basis of the sentence-boundaries found by the method sketched in Chapter 10, using the Python script shown in Figure 12.10. In Van Valin and LaPolla [1997], the authors discuss Knud Lambrecht's theory of Information Structure. Lambrecht argues that there are certain states of a hearer's or reader's mind while hearing or reading, which may be labelled "Active", "Accessible", and "Inactive". Whether a referent in a text is labelled "Active", "Accessible", or "Inactive" depends on how easy it is for the hearer or reader to infer the identitiy of the referent. "Active", in particular, means that the referent is "active" in the hearer's or reader's short-term-memory, while "Accessible" means that some degree of mental work must be exercised in order to infer the referent's identity.

The more recent the placement of the referent's identity in the text, the more accessible the referent is to the hearer's or reader's mind. My guess is that in about the span of two sentences, a referent slips from "Active" to "Accessible". This is only a guess, not something empirically tested. But the span of 2 sentences is about 50 words, given that the average sentence-length is 23.8 words in the Kaj Munk Corpus.

The numbers used to calculate this average sentence-length are shown in Table 12.1.[9]

### 12.4.5 Advanced search

The Advanced Search Dialog is brought up by pressing the button on the toolbar which has a magnifying glass with an academic hat. The dialog looks something like in Figure 12.11.

In Figure 12.11, the area labelled "1" is where the user can enter one or more words to search for.[10]

The area labelled "2" contains a summary, which the program produces dynamically, of the current settings in the dialog. The summary is given in natural language, in order to aid the user in knowing what the options mean.

The area labelled "3" is where the user picks and chooses from a large array of options. The area has four tabs, of which only the first is currently not implemented.

[9] It is interesting to note that the average length of sentences in the sermons is 65.2, versus only 21.5 for the plays, which is more than a factor of three larger for the sermons. One conjecture for an explanation of this might be that the two are different genres: hortatory, edificative homilies on the one hand (sermons), versus spoken language on the other (plays). Further research is needed, however, in order to conclude anything about this. Further research should also include a comparative study of sentence lengths in other similarly genre-classified texts. Help might be found in, e.g., Longacre [1996].

[10] Note that it is a shortcoming of the current "Advanced Search" Dialog that the user has to enter at least one word. It is very conceivable that the user would wish to search, not on particular forms, but on other criteria, such as parts of speech, which would not be tied to any particular form. This is an area of possible future development.

```
import sys
import re

myre = re.compile(r'\{ ([^\}]+)\}')

myset = set()

# The sheaf from the following query must be passed through stdin:
#
# SELECT ALL OBJECTS
# WHERE
# [sentence]
# GO
#
for line in sys.stdin.readlines():
    if not "sentence" in line:
        pass
    else:
        monad_str = myre.findall(line)[0]
        if not "-" in monad_str:
            fm = lm = int(monad_str.strip())
        else:
            (fm_str, lm_str) = monad_str.strip().split("-")
            fm = int(fm_str)
            lm = int(lm_str)
        myset.add((fm,lm))

count = 0
mysum = 0
for (fm,lm) in sorted(myset):
    count += 1
    difference = lm - fm + 1
    mysum += difference

print "Count = %d" % count
print "mysum = %d" % mysum
print "verage = %f" % (mysum*1.0)/(count*1.0)
```

Figure 12.10: The Python script used to calculate average sentence length.

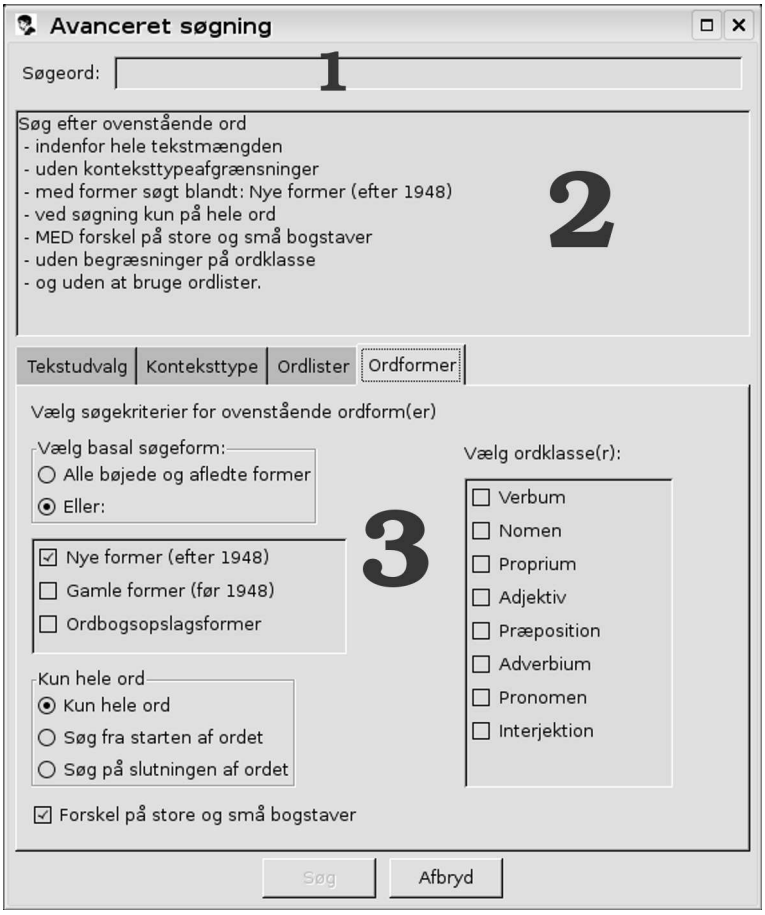



Figure 12.11: The Advanced Search Dialog of the Munk Browser Software. The area labelled “1” is where the user types in one or more words. The area labelled “2” is where the program shows a summary of the current options in natural language. The area labelled “3” is where the user picks the options. The latter area has four “tabs”, which each have options for one particular aspect or “axes” of the advanced search.

**Text delimitation:** Though currently not implemented, the idea is that this tab will provide the user with options to delimit the parts of the corpus to search, based on textual criteria such as "year of writing", "year of publication", "whether a play has been performed or not", etc. — based on the meta-data present for each Munk text. It should also be possible to choose a set of documents based on "genre" (in this case, whether it is a play, a sermon, a poem, a piece of prose, or a piece of journalism — the five "genres" within which Munk wrote).

**Context delimitation:** This tab allows the user optionally to select zero or more object types within which the query should be executed. The context object type (if there is one) will then be the surrounding context [object block] within which to search. If no context type is chosen, then "munktxt" is chosen as the default. If there is more than one, then each will get the appropriate inner blocks (as discussed below), and the list of "context object blocks" will then be concatenated, with "OR" in between.

**Word lists:** We have had a student, Jóannis Fonsdal, produce a list of lemmas, based on the lemmas in Kaj Munk's plays, where each lemma has been marked as to whether it is "religious by definition", or "religious by association", or "not religious". The precise definition of these terms is hard to get at, and the borders between the three categories are not hard and fast, but rather fluid in nature. One annotator might choose to categorize a particular lemma differently than another annotator. The important thing here is not the particular categorization, but the idea of how to use it, which I shall now describe.

In this tab, it is possible to choose whether the word(s) typed in the area labelled "1" should be found within another word (up to a specific distance away) which is either "religious by definition" or "religious by association", or both.

The way it is done is to take each word and turn it into an object block (see "Word forms" below, for how this is done).

Then, an object block is created for the particular kind of religious words chosen ("by definition" or "by association"). Currently, this is done by creating an [token] object block with a feature-restriction which says that the lemma must be one of the members of the chosen list. This is very inefficient, and so future work will include storing the "religious status" on the objects of the token object type directly.

Then, the "religious" [token] object block is placed before or after the "user-entered" [token] object block, with a power block between them. This is then wrapped in a [square brackets] for "grouping". For example:

```
[
[Token /* religious lemmas not shown */ ]
..  <= 5
[Token new_surface_lowcase="tidens"]
]
```

The reason it is wrapped in [grouping square brackets] is that a power block is

```
SELECT ALL OBJECTS
WHERE
[poetry
  [
    [Token /* religious lemma-restriction not shown */]
    .. <= 6
    [Token new_surface_lowcase = "tidens"]
  ]
  ..
  [
    [Token /* religious lemma-restriction not shown */]
    .. <= 6
    [Token new_surface_lowcase = "gang"]
  ]
]
GO
```

Figure 12.12: A sample query from the "Advanced Search" Dialog. The words entered were "tidens gang" (idiomatically, "the course of time"). Religious words should be found within a distance of 6 words before the word itself. The object type "poetry" should be used as context-delimitation.

placed between the groups, if the user has entered more than one words. A sample query can be seen in Figure 12.12.

**Word forms:** In this tab, it is possible to choose a wide range of options.

First, it is possible to choose whether to search based on the stemmed form of the word, or else a combination of "new form" (after the orthography reform of 1948), "old form" (before 1948), or "lemmas".

Second, it is possible to search on "exact words", "prefixes of words", or "suffixes of words".

Third, it is possible to choose a set of parts of speech within which to limit the search.

Finally, it is possible to choose whether to ignore the upper-case / lower-case distinction, or not.

The "Advanced Search" Dialog is a work in progress. Much could be said to its detriment. I shall refrain from commenting on its shortcomings, except to say that it is an area of future research, not only in database theory, but also in user interface design.

### 12.4.6 Find Widget functionality

The "Find Widget", it will be remembered, is supposed to pop up at the user's behest, and allow the user to search the currently open document for strings, and be taken to the

appropriate places immediately.

In order for this to work, I propose to implement the following general idea.

First, the idea is to build an "in-memory EMdF database" along the lines outlined in Section 4.6 on page 54, and populate it with objects of an object type (let us call it "display_object_range") which are "`WITH SINGLE RANGE OBJECTS`", and whose monads are character-indexes into a string. This string (let us call it "doc_text") represents the "bare" form of the XHTML document (i.e., a form without tags, and with whitespace normalized).

Second, the object type in question should have a "feature" which is an integer, which then points into a map mapping these integers to pointers to the right display objects inside the XHTML window. (The first monads of the "display_object_range" objects are not unique, and so cannot be used as a surrogate for the display object.) The reason for storing pointers to the display objects is that the XHTML window has a capability to scroll to any display object if only a pointer to the object is passed as a parameter to the scroll function.

Third, once this data structure is up and running, it should be possible to search the "doc_text" string, using a standard string-searching algorithm, such as Knuth-Morris-Pratt [Knuth et al., 1977]. This will yield zero or more character-indexes. If one or more are found, the right display objects can be found using the above data structure, and the XHTML window can then be scrolled to the right display object. It is also possible to traverse the list of "hits" up and down, using the "previous" and "next" buttons of the Find Widget.

## 12.5 Conclusion

In this Chapter, I have described the "Munk Browser" desktop application which I have developed. The purpose of the application is to provide access to the texts of Kaj Munk for a wide range of users.

I have first described the application from a functional point-of-view, followed by a description in terms of the modules involved. I have then shown how the theory presented in previous chapters has been brought to bear on the problem of producing the "Munk Browser", in showing a document using harvesting techniques, performing simple search, performing advanced search, harvesting, and a operating a "find widget".

Further research could include:

1. Searching within annotations. It is obvious that Emdros supports this, especially if the "~" regular expression operator is used. Therefore, it should probably be included in the "Advanced Search Dialog" as an option.

2. Making user surveys to see whether a "wizard-like" approach would be more intuitive than the current "tabbed dialog" approach. A "wizard" in this context is a series of dialogs in which the user is asked for more and more information, but in several steps, with "next" and "previous" buttons to guide along the way.

3. Making a facility for letting the user enter their own comments. Possibly, these comments should be uploadable to the Kaj Munk Centre's server, where someone could then sort through the users' comments and select the best ones for use.

4. Supporting better navigation of the databases, i.e., better selection of the document desired.

# Chapter 13

# Quotations from the Bible in Kaj Munk

## 13.1 Introduction

In this Chapter, I discuss a general approach to discovering quotations from one corpus in another corpus. In this case, the quotations are from the Bible, and must be found in the Kaj Munk Corpus. The method, however, is general enough that it could be applied to any suitably tagged pair of corpora. The research idea itself was inspired by van Wieringen [1993], who did something similar within the Book of Isaiah from the Old Testament, that is, finding quotations and analogies within the book itself by using a computer program and the WIVU Hebrew Database [Talstra and Sikkel, 2000].

This research has been carried out in collaboration with Peter Øhrstrøm. The division of labor was such that Peter Øhrstrøm came up with some of the research ideas, did some manual disambiguation, and provided general direction, while I came up with the rest of the research ideas, and implemented the ideas in concrete computer programs.

The rest of the Chapter is laid out as follows. First, I discuss how I obtained an electronic copy of the Danish Bible that is mostly error-free (13.2). Then, I discuss the general idea in more detail (13.3). I then discuss the process which we went through in our research (13.4), including results, followed by pointers to future research (13.5). Finally, I conclude the chapter.

## 13.2 Electronic edition of the Danish Bible (1931/1907)

There have been various translations of the Bible into Danish over the centuries. Of these, the most relevant ones for my purposes were the Old Testament from 1871, the New Testament from 1907, and the Old Testament from 1931, since they were the ones which Kaj Munk probably used.

- 1871: Old Testament
- 1907: New Testament
- 1931: Old Testament

The Old Testament from 1871 is available on the Internet in electronic form, but unfortunately, my comparison of forms from this OCR-scanned edition with my database of translations of old forms to new forms yields more than 8000 words which must be

"looked at", most of them errors in the electronic text. I have not yet had the time to undertake either a revision of this text, or a complete re-digitization.

The Old Testament from 1931 and the New Testament from 1907 have been available on the Internet at least since 1993, when Søren Hornstrup made it freely available on the ftp.dkuug.dk server. It was then picked up by Project Gutenberg[1] and project Runeberg[2].

In order to establish an "authoritative text" (Søren Hornstrup's text, unfortunately, had thousands of errors), I have gone through the following process.

First, I compared the texts from Project Runeberg and Project Gutenberg. The result was that the text from Project Runeberg was slightly better than that from Project Gutenberg. I then started correcting spelling mistakes and punctuation-mistakes in the edition from Project Runeberg. This was done using Emacs and regular expressions. I also ran the text through my "old to new" filter (explained in Chapter 10), thereby yielding around two thousand forms which must be "looked at", i.e., for which it was not possible to derive a modern form. I then hand-corrected any mistakes in found in this way.

I then found out that Sigve Bø from Norway, who runs a company called Sigve Saker[3], had done extensive proof-reading of the 1931/1907 Danish Bible. Sigve Bø was kind enough to send me his most current versions. I then converted Sigve Bø's files to a form that could be compared with my form, using the standard Unix(R) "diff" tool[4]. This yielded a few thousand errors, which I corrected by hand. I also corrected many errors in Sigve Bø's text, for I had found errors which Sigve Bø had not. Thus, in a sense, my form is based on Sigve Bø's form, and in a sense it is not — it is, in reality, a conflation of three sources, namely the Runeberg form, the Gutenberg form, and Sigve Bø's form, with numerous manual corrections on my part. I always checked against a printed edition of the respective Testament before making any corrections, except in a very small set of cases, where the correction was exceedingly obvious as per Danish grammar or orthography.

Naturally, I had to convert the orthography back to its pre-1948 form, since that was what Kaj Munk used whenever he quoted the Bible. Both the Runeberg edition, the Gutenberg edition, and Sigve Bø's edition had a strange combination of old and new orthography. For our purposes, this would not do.

In the end, the whole process had seen more than 7100 corrections to the Runeberg text, and 5037 corrections had been made to Sigve Bø's text.

Thus the end product was a Danish Bible which was mostly error-free. This was necessary in order to be able to find direct quotes. We now turn to "the general research idea".

## 13.3 The general idea

Suppose one has two corpora; let us call them "Corpus A" and "Corpus B" for generality. Corpus A is known to quote from Corpus B in various places, but it is unknown precisely where Corpus A quotes Corpus B. The challenge is to find the places in Corpus A where Corpus A quotes Corpus B.

[1] A project to digitize Public Domain textual resources. See http://www.gutenberg.org/

[2] Project Runeberg aims to be for the Scandinavian languages what Project Gutenberg is for the World's languages. See http://runeberg.org/

[3] See http://www.sigvesaker.no/

[4] For a description of diff, see the Unix manual page "diff(1)" and Hunt and McIlroy [1976].

There are many factors to consider in this problem. For example, what constitutes a quote? Especially quotes from the Bible may be hard to characterize completely, since some sequences of words that appear in the Bible would appear in other corpora as well, without necessarily being quotes from the Bible, simply because the sequence of words is so common. An example in English could be "and he said". This would be found both in many English Bibles, and in many other texts as well, yet we would not necessarily say that those three words were a quotation from the Bible, unless the context clearly indicated that this was so.

Another factor to consider in characterizing what constitutes a quote from the Bible is how to determine that something is actually a quote from the Bible. We finally settled on the following definition:

> A sequence of words found both in the Bible and in another corpus is a quote from the Bible if two or more persons who are knowledgeable about the contents of the Bible would agree that the sequence of words is a quote from the Bible.

The reason for settling on this definition is that we found out that the problem was so hard to characterize formally that we concluded that human intelligence is probably necessary in order to settle whether a specific sequence of words is, indeed, a quote from the Bible.

A third factor to consider is: Should one also look for "almost-quotations", or "allusions", or should the algorithm find only direct quotations? We settled on the latter, with the former two left for future research.

In the following, I discuss the process which we went through.

## 13.4 Process

The first thing we did was to choose a sub-corpus of Kaj Munk on which to develop the method. Other parts of the Kaj Munk Corpus could then serve as a testbed later, once the method had been developed. We chose the sermons as our "development corpus", since they were intuitively likely to to contain a lot of quotations from the Bible.

I then loaded the sermons into one Emdros database, and the Bible into another Emdros database. Both had been tokenized, and both had been tagged for part of speech using the CST tagger mentioned in Chapter 10.

I then wrote a Python script which did the following:

1. Load both the "development corpus" and the Bible into memory from the Emdros databases.
2. For window-lengths of length *n*, counting down from 40 down towards 2:
   (a) Create two sets of *n*-grams from both corpora, keeping track of where they occurred (using monads).
   (b) Take the set intersection of the two sets. This is the set of "direct quotations" of length *n*. (Actually, the strings in the intersection need not be quotes, especially if the strings are short; see below for a discussion of this.)
   (c) Further process the intersection, such as to display and count the "quotations".

(d) Make sure that the "quotations" found in step (b) above would not be counted in the next iteration, i.e., would not be counted as $(n-1)$-grams.

This produced, for each length of sequence $n$, a set of "quotations". For som $n$, the set was empty. The number 40 was chosen experimentally: We found that no string of length 40 or above was present in Kaj Munk's sermons (our "development corpus") which was also present in the Bible from 1931/1907.

For the sets which were not empty, we did some manual inspection, Peter Øhrstrøm and I. We found that sequences of length 7 or above were, in all cases considered, quotations in the sense defined in Section 13.3. For sequences of length 6, the overwhelming majority were quotations. For sequences of length 5, some were not quotations, whereas most were. For sequences of length 4, a lot of strings were not quotations, whereas a minority were. For sequences of length 3, the balance was even more in favor of non-quotations, and so on, down to 2. We did not try "sequences" of length 1, since they would intuitively be very unlikely to be quotes from the Bible.[5]

We then attempted to distinguish between quotations and non-quotations using statistics. This failed, mainly because we were unable to find a suitable metric with which to measure "quotation-likelihood".[6]

We then attempted to distinguish between quotations and non-quotations using grammatical patterns of part-of-speech tags. This worked to some extent for sequences of length 5, but still left some false positives in the mix, and also sifted out some false negatives. Furthermore, it failed for sequences shorter than 5, and did nothing for sequences of length 6. Since sequences of length 7 and above were found always to be quotations, this method also was not fruitful.

In the end, it was decided to leave sequences of length 6 and below for future research, and simply arrive at the result that sequences of length 7 and above were always quotations.

We then validated the method based on the play "The Word" (in Danish, "Ordet") by Kaj Munk. For this particular play, we had a set of quotations, found by Niels Nielsen [Munk, 1947], as a control. We produced a table of quotations based on the notes on the play made by Niels Nielsen. We then added to the table the exact text from the Bible being quoted, and the exact text from the play, and checked the output of our program. It was found that in every case where the quote was a direct quote, the program had indeed found the quote, and that there were no false positives. It was also found that many quotes were not found by the program, simply because they were not direct quotes. Thus, our method was validated against a control corpus which had been manually filtered for quotations, and the program fully met its requirements.

[5] However, some words are so improbable outside of the Bible, and have their sole origin in the Bible, that they must *de facto* be some kind of quotation from the Bible. One example would be "The Son of Man", which is an expression originating in the books of Daniel and Ezechiel in the Old Testament, and frequently used by Jesus about himself in the gospels. In Danish, "The Son of Man" is a single word, "Menneskesønnen". This word would appear to be some kind of quote from the Bible, although further philosophical considerations are needed in order to establish the precise nature of such quotations.

[6] Future research will likely reference the work of H.P. Luhn, who pioneered some statistical measures in Information Retrieval.

## 13.5 Future research

Future research could include:

- Finding a general way of distinguishing between quotations and non-quotations of lengths shorter than 7 words.
- Checking whether it would be fruitful to build up a database of quotations (or, indeed, non-quotations) shorter than 7 words to aid in the disambiguation process.
- Pursuing the statistical path in more detail.
- Being able to find "almost"-quotes, e.g., with words inserted or permutated, or with different tenses of verbs, or with different grammatical number and determination on nouns.
- Applying the method to other pairs of corpora than the Bible and the Kaj Munk Corpus. For example, it would be fruitful to be able to find quotes from the works of Søren Kierkegaard in other corpora. Doing so would require a lot of manual work in identifying a common orthography, if the two corpora were not contemporaneous, since the orthography of Danish has changed considerably since the times of Søren Kierkegaard. An "old to new" othography converter, such as the one sketched in Chapter 10, would be an invaluable tool in such an endeavor.

  Note that finding Kierkegaard quotations in Kaj Munk is not just interesting from a technical perspective: There is currently a debate in scholarly circles as to the degree of influence from Kierkegaard on Kaj Munk's works. Having empirical evidence would help settle this debate.

Thus this problem has far from been solved. In the end, we speculate that it may be impossible to achieve 100% precision and 100% recall[7] in finding quotations from the Bible in another corpus automatically. In a sense, this problem is a problem within "artificial intelligence", because it attempts to build an automatic method for finding quotations from one corpus in another corpus. This normally requires real, human intelligence, and the conclusion may well become (after more research) that we still have not found a way to solve this problem generally without recourse to real, human intelligence. That, at least, is the conclusion for the time being.

## 13.6 Conclusion

In this Chapter, I have discussed a method which I have developed together with Peter Øhrstrøm, the purpose of which is to find direct quotations from the Bible in some other corpus, in particular, the Kaj Munk Corpus.

I have first discussed how I obtained an electronic "authoritative" version of the Danish Bible from 1931 (Old Testament) and 1907 (New Testament), which formed part of Kaj Munk's Bible (the third part being the Danish Old Testament from 1871). I have

[7] For a definition of the terms "precision" and "recall", see Baeza-Yates and Ribeiro-Neto [1999].

then discussed the general research problem, and have defined what I mean by “quotation from the Bible”. I have then discussed the process which we went through, and the results. Finally, I have given pointers to further research.

The problem is far from solved, but is a very interesting problem, judging from the salivation produced in the mouthes of theologians when we present them with the prospect of having such an algorithm.

# Part III

# Perspectives

# Chapter 14

# Extending EMdF and MQL

## 14.1 Introduction

In this Chapter, I discuss a number of possible extensions of the EMdF model and the MQL query language, left for future research. In Section 14.2, I discuss extending the EMdF model and the MQL query language with support for declaring structural relations between object types. In Section 14.3, I discuss extending the EMdF model with support for inheritance between object types. Then, in Section 14.4, I discuss the need for supporting "typed id_ds", i.e., being able to retrieve the object type of the object which owns a given id_d. In Section 14.5, I talk about the need for supporting complex feature types in EMdF. I then turn to MQL, first specifying what parts of Doedens's QL still need to be implemented (14.6). I then discuss ways in which MQL could be extended beyond QL (14.7). Finally, I conclude the Chapter.

## 14.2 Declarative support for structural relations

Doedens defined the "covered_by" and "buildable_from" relations between object types. Briefly, if object type *B* is "covered_by" object type *A*, then all objects of type *B* consist of only of monads which can be expressed as the big-union of some set of objects of type *A*, and the set of objects of type from which the monads of each object of type *B* is constructed is disjoint from all other such sets for any other object of type *B*. Buildable_from is the same as covered_by, except for a conjoined requirement, namely that both type *A* and type *B* must be non-overlapping, i.e., all monad sets of all objects of type *A* must be disjoint, and the same must be true of all objects of type *B*.

Doedens states that covered_by and buildable_from constitute partial orders over object types. As I have shown elsewhere [Petersen, 1999], this is not true.

These should, however, still be supported by the EMdF model and the MQL query language, since they can be useful in query-optimization. For example, if it is known that an object type *C* is buildable_from another object type *W*, it might be possible to make assumptions about queries that use both *C* and *W*, which could lead to query optimizations. In fact, the current implementation assumes that all object types are neither covered_by nor buildable_from any other object type, and therefore does useless work in some cases, such as when looking for more than one object of a given type starting at a given monad. If object type *C* is buildable_from object type *W*, then clearly *W* will have

precisely one object for any given monad which is also part of a given object from object type *C*. This is because of the requirement on buildable_from that both object types must be non-overlapping. Hence, the implementation can assume that for any given monad which is also part of a *C* object, an object of type *W* does indeed exist which starts at this monad, and there is only one such object. This knowledge can be exploited in doing minimal work.

There is another kind of structural relationship which should be able to be stated, apart from buildable_from and covered_by: It should be possible to declare that a given object type *W* always stands in a structural relationship with one or more objects of types $S_1$, $S_2$, ..., $S_n$. This would allow for declarative support for trees and other directed acyclic graphs. In declaring these relationships, it should also be possible to specify which feature(s) on *W* would encode these structural relationships, presumably with id_ds.

Adding these capabilities would make EMdF and MQL meet demand D13.

## 14.3 Object type inheritance

It should be possible to implement object type inheritance along the lines outlined in Chapter 3. For example, an object type *Phrase* should be able to have subtypes *NP*, *VP*, etc., which each inherited the features of *Phrase*, and which might add other features of their own. Or an object type *Syntax_node* should be able to have subtypes *Phrase*, *Clause*, or *Sentence*. The benefits of such a scheme have already been outlined in Section 4.2.3.

Multiple inheritance should also be supported, meaning that an object type should be able to "inherit" from more than one object type.

This would meet requirement D5.1 explained in Section 4.2.3.

## 14.4 Typed id_ds

At the moment, an id_d is always typeless, i.e., no information is stored along with an id_d about the object type of the object which owns the id_d. If it were possible to ask, for any given id_d, to which object type it belonged, then topographic MQL queries could use this information both to speed up the queries (namely by making some checks that might cut short some otherwise-to-be-traversed query paths), and to allow greated expressivity of the MQL queries. For example, a "computed feature" might be invented, which took the id_d of another feature and returned the object type. For example:

```
SELECT ALL OBJECTS
WHERE
[Word type(parent) = phrase]
GO
```

This would find all phrases whose parent feature (which would be an id_d) would be of the type 'phrase'. The feature "type" would be a "computed feature".

## 14.5 Complex feature-types

The current implementation of the EMdF model only defines four atomic feature-types (integer, id_d, enumeration, and string), and three compound feature-types (list of inte-

ger, list of id_d, and list of enumeration). At the very least, the EMdF model should be extended with list of string as an option.

In addition, the EMdF model should be extended with the possibility of defining complex feature-types other than the simple ordered lists of atomic types currently available. For example, it should be possible to define features that were:

1. tuples with strongly typed members (any type),
2. sets of strongly typed members (any type), and
3. ordered lists of any other type.

It should be possible to declare these types compositionally, such that, for example it should be possible to define a pair (tuple) consisting of: a) a set of tuples of strings, and b) a list of sets of tuples of lists of strings ... and so on, to any (finite) depth of nesting. Needless to say, MQL would need to be extended for this to happen.

## 14.6 Full QL support

The full power of QL still awaits implementation, possibly in MQL. In particular, the following still need to be implemented:

- Variable support, or something that is on an equal footing with variables theoretically. The current Object Reference implementation has shaky theoretical foundations.
- Wrap blocks, i.e., being able to specify that some block_string should match a part of the database which might not be known directly as an object in the database.
- Automatic permutations of blocks, i.e., being able to specify sets of blocks that need to be permuted in all possible permutations.
- "AND" between block_strings (see Section 14.7 below).
- "Sifters", i.e., being able to specify, e.g., that only so many straws should be retrieved, or that the `blocks` should only match if there were precisely so many hits.

The list above is a complete list of things in QL not implemented in MQL. I now turn to the question of how to extend MQL beyond QL.

## 14.7 Beyond QL

### 14.7.1 Introduction

In this Section, I introduce two new ways of extending MQL beyond QL. First, I discuss variations over the theme of conjoining block_strings. Then, I discuss extending the capabilities of power blocks.

### 14.7.2 block_string conjunctions

The "**AND**" operator between block_strings in QL was specified by Doedens to mean that the query would match if there were two Straws, one from each of the two AND-conjoined block_strings, which extended over the same monads of the database, i.e., whose first and last monads were the same. There are, however, many variations over this theme. For any two block_strings *A* and *B*, the following operators can be defined:

1. *A* OVERLAPS WITH *B*, which means that the Straws from *A* and *B* must share at least one monad.

2. *A* OVERLAPS WITH AND STARTS BEFORE *B*, which means that the Straws from *A* and *B* must share at least one monad, and the straws of *A* must start before the straws of *B*.

3. *A* VIOLATES BORDERS OF *B*, which means that the Straws from *A* and *B* must "violate the borders" of each other, as defined in [ICCS-Suppl2008] and Chapter 8.

4. *A* STARTS AT *B*, which means that the Straws of *A* and *B* must have the same first monads.

5. *A* ENDS AT *B*, which means that the Straws of *A* and *B* must have the same last monads.

6. *A* STARTS AND ENDS AT *B*, which means that the Straws of *A* and *B* must have the same first *and* last monads. This would implement Doedens's AND.

7. *A* STARTS *n* MONADS BEFORE *B*, which means that the Straws of *A* must start *n* monads before the Straws of *B*.

8. *A* STARTS *n* MONADS AFTER *B*, which means that the Straws of *A* must start *n* monads after the Straws of *B*.

9. *A* ENDS *n* MONADS BEFORE *B*, which means that the Straws of *A* must end *n* monads before the ending of the Straws of *B*.

10. *A* ENDS *n* MONADS AFTER *B*, which means that the Straws of *A* must end *n* monads after the ending of the Straws of *B*.

The variations could probably go on, but I will stop here.

Especially the "OVERLAPS WITH" operators would be useful, e.g., when dealing with databases of annotated speech. As argued in [ICCS-Suppl2008] and in Chapter 8, speech databases often have speaker turns which overlap. Hence, these operators could be very useful for querying such databases.

### 14.7.3 power block limitor

I could also extend MQL's **power block** to be able to have a limitor which:

- Had a lower bound, but no upper bound:

  ```
  ..  >= 3
  ```

  would mean that there must be at least 3 monads between the block before and the block after the power block.

- Had a lower bound as well as an upper bound:

  ```
  ..  >= 3 and <= 10
  ```

  would mean that there must be at least 3 monads and at most 10 monads between the block before and the block after the power block.

Currently, it is only possible to specify an upper bound on the limitor of a power block, not a lower bound.

## 14.8 Conclusion

In this Chaper, I have discussed some possible future extensions of the EMdF model and/or the MQL query language. They include:

1. Support for declaring structural relationships between object types. This should, as a minimum, include the covered_by and buildable_from relations discussed by Doedens [1994]. I also suggest extending EMdF with declarative support for direct structural relationships which form a directed acyclic graph-structure. Doing so would meet Doedens's demand D13 on a "full access model".

2. Support for (multiple) inheritance between object types. Doing so would be extend EMdF such that it would be possible to specify ontologies of object types.

3. Support for typed id_ds. Adding this support would extend the expressivity of the MQL query language, and might lead to performance gains.

4. Support for types which are more complex than either atomic types or lists of atomic types, which is what EMdF supports today. Sets of arbitrary, tuples of arbitrary types, and ordered lists of arbitrary types were suggested. This would extend the number of possibilities for encoding annotations.

5. Supporting the full QL language in MQL.

6. Going beyond QL in expressivity for MQL.

Thus there are many ways of extending the EMdF model, some of which also have ramifications for the extension of the MQL query language.

# Chapter 15

# Conclusion

This dissertation has been about "annotated text databases", seen from a theoretical as well as a practical perspective. During the course of the dissertation, I have shown how text can be enriched with annotations of the text, and stored in a database for easy retrieval. I have shown how I have built on the work of Doedens [1994] in order to implement the EMdF model and the MQL query language. The EMdF model is at once a reduction and expansion of the MdF model introduced by Doedens [1994]. It provides the data structures needed in order to be able to represent "most" annotations that one would wish to express. The MQL query language, in turn, is also both a reduction and an expansion of the QL query language developed by Doedens. The "reduction" with respect to QL is that a few of QL's constructs have not been implemented. I have shown in Chapter 14 exactly what the very few constructs which have been left out are. The "expansion" of MQL with respect to QL is that it is now a "full access language", providing the full gamut of "create, retrieve, update, delete" on all of the data domains in the EMdF model. Thus Doedens's Demands D1–D12 have been fully met by EMdF and MQL, whereas Doedens's own MdF and QL only met Demands D1–D10, plus part of D12.

I have also shown how the monads sets of the EMdF model relate to time (Chapter 8). As it turns out, the relationship between a monad and the time of an utterance which it represents is that the monad abstracts away into an integer the time-span to which it corresponds. Thus, even though the linguistic unit to which the monad corresponds has a specific duration, which may be different from the duration of all other linguistic units in the text, the monad abstracts away this duration into a single integer. This is part of the genius of Doedens's work, for it is one of the cardinal reasons why Doedens's MdF model is so elegant mathematically. My EMdF model, being an extension of the MdF model, merely inherits this elegance.

In Part II, I have shown several applications of the theoretical work described in Part I. I have shown how the Kaj Munk Corpus has been implemented, first as XML, and then transformed to an Emdros database. Thus the EMdF model has been vindicated as being able to express at least the annotation required for storing the Kaj Munk Corpus. As argued many times in the articles which form an appendix to this dissertation, the EMdF model supports the free expression of almost any kind of linguistic annotation desirable. This has not been proven yet, but [RANLP2005] goes some of the way towards such a proof, by describing how the TIGER model of treebanks can be converted to the EMdF model.

In Chapter 11, I have shown how the EMdF model and the MQL query language can

be employed in order to support the collaborative annotation of a corpus by a team of annotators. The principles laid out in this Chapter only hint at what is possible for a collaborative annotation tool to support. I have argued the case for why I have chosen these principles, and leave it up to future research to find better ways of supporting collaborative annotation procedures.

In Chapter 12, I have shown how the theory laid out in Part I can be brought to bear on the Kaj Munk Corpus in a desktop software application, a so-called "Munk Browser". This Chapter is the central empirical chapter, bringing together most of the theory discussed in Part I.

In Chapter 13, I discuss a method of finding quotes from one corpus in another corpus, and how Peter Øhrstrøm and I have tested this method on the problem of finding quotations from the Bible in the Kaj Munk Corpus.

Finally, in Chapter 14, I have discussed ways in which the EMdF model and the MQL query language can be extended to support representation and querying of annotated text databases even better than the current implementation.

*The road goes ever on and on.*
(Bilbo Baggins)

# Bibliography

The TEI guidelines. http://www.tei-c.org/Guidelines/P5/ Access online April 12, 2008, November 2007.

XML Path Language (XPath) 2.0 – W3C recommendation 23 January 2007. Published on the web: http://www.w3.org/TR/xpath20/ Accessed March 2008, 2007.

Serge Abiteboul. Querying semi-structured data. In *Proceedings of the 6th International Conference on Database Theory*, volume 1186 of *Lecture Notes on Computer Science*, pages 1–18. Springer Verlag, Berlin, Heidelberg, New York, 1997. ISBN 3-540-62222-5.

Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques and Tools*. Addison-Wesley, Reading, Massachusetts, USA, 1985.

R. B. J. T. Allenby. *Rings, Fields and Groups: An Introduction to Abstract Algebra*. Edward Arnold, second edition, 1991.

Bernd Amann and Michel Scholl. Gram: a graph data model and query languages. In *ECHT '92: Proceedings of the ACM conference on Hypertext*, pages 201–211, New York, NY, USA, 1992. ACM. ISBN 0-89791-547-X. doi: http://doi.acm.org/10.1145/168466.168527.

Mette Skovgaard Andersen, Helle Asmussen, and Jørg Asmussen. The project of korpus 2000 going public. In Anna Braasch and Claus Povlsen, editors, *Proceedings of the 10th EURALEX International Congress, EURALEX 2002, Copenhagen, Denmark, August 13–17, 2002*, pages 291–299, 2002.

Galia Angelova, Kalina Bontcheva, Ruslan Mitkov, Nicolas Nicolov, and Nikolai Nikolov, editors. *International Conference Recent Advances in Natural Language Processing 2005, Proceedings, Borovets, Bulgaria, 21-23 September 2005*, Shoumen, Bulgaria, 2005. INCOMA Ltd. ISBN 954-91743-3-6.

Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40(1):1–39, February 2008.

Andrew W. Appel. *Modern Compiler Implementation in C: Basic Techniques*. Cambridge University Press, Cambridge, UK, 1997.

*Actes du Troisième Colloque International: "Bible et Informatique: Interprétation, Herméneutique, Compétence Informatique", Tübingen, 26-30 August, 1991*, number 49 in Travaux de linguistique quantitative, Paris and Genève, 1992. Association Internationale Bible et Informatique, Champion-Slatkine.

Niraj Aswani, Valentin Tablan, Kalina Bontcheva, and Hamish Cunningham. Indexing and querying linguistic metadata and document content. In Angelova et al. [2005], pages 74–81. ISBN 954-91743-3-6.

Ricardo Baeza-Yates and Gonzalo Navarro. XQL and Proximal Nodes. *Journal of the American Society for Information Science and Technology*, 53(6):504–514, May 2002. <http://dx.doi.org/10.1002/asi.10061>, <http://www.dcc.uchile.cl/~gnavarro/ps/jasis01.ps.gz>, Access Online August 2004.

Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

D. Barnard and N. Ide. The text encoding initiative: Flexible and extensible document encoding. *Journal of the American Society for Information Science*, 48(7):622–628, 1997.

Rudolph Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972. <http://www6.in.tum.de/info1/literatur/Bayer_hist.pdf> Access Online January 2005.

Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles+. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322–331. ACM, ACM Press, 1990.

J. Albert Bickford. *Tools for Analyzing the World's Languages – Morphology and Syntax*. Summer Institute of Linguistics, Dallas, Texas, 1998. Based on earlier work by John Daly, Larrt Lyman, and Mary Rhodes. ISBN 1-55671-046-x.

Steven Bird and Mark Liberman. A formal framework for linguistic annotation. *Speech Communication*, 33(1,2):23–60, 2001. <http://www.ldc.upenn.edu/sb/home/papers/0010033/0010033.pdf>, Access Online August 2004.

Steven Bird, Peter Buneman, and Wang-Chiew Tan. Towards a query language for annotation graphs. In *Proceedings of the Second International Conference on Language Resources and Evaluation*, pages 807–814. European Language Resources Association, Paris, 2000a. <http://arxiv.org/abs/cs/0007023> Access Online August 2004.

Steven Bird, David Day, John Garofolo, John Henderson, Christophe Laprun, and Mark Liberman. ATLAS: A flexible and extensible architecture for linguistic annotation. In *Proceedings of the Second International Conference on Language Resources and Evaluation, Paris*, pages 1699–1706. European Language Resources Association, 2000b. <http://arxiv.org/abs/cs/0007022>, Access Online August 2004.

Steven Bird, Yi Chen, Susan Davidson, Haejoong Lee, and Yifeng Zheng. Extending XPath to support linguistic queries. In *Proceedings of Programming Language Technologies for XML (PLANX) Long Beach, California. January 2005.*, pages 35–46, 2005.

G.E. Blake, M.P. Consens, P. Kilpeläinen, P.-Å. Larson, T. Snider, and F.W. Tompa. Text / relational database management systems: Harmonizing SQL and SGML. In *Proceedings of the First International Conference on Applications of Databases (ADB'94), Vadstena, Sweden*, pages 267–280, 1994.

Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language. W3C working draft 11 february 2005, 2005. <http://www.w3.org/TR/2005/WD-xquery-20050211/>, Access Online March 2005.

Sabine Brants and Silvia Hansen. Developments in the TIGER annotation scheme and their realization in the corpus. In *Proceedings of the 3rd International Conference on Language Resources and Evaluation (LREC), 2002* ELR [2002], pages 1643–1649. <http://www.ims.uni-stuttgart.de/projekte/TIGER/paper/lrec2002-brants-hansen.pdf> Access Online August 2004.

Sabine Brants, Stefanie Dipper, Silvia Hansen, Lezius Wolfgang, and George Smith. The TIGER treebank. In *Proceedings of the Workshop on Treebanks and Linguistic Theories, Sozopol, Bulgaria.*, pages 24–41, 2002.

Thorsten Brants. TnT – a statistical part-of-speech tagger. In *Proceedings of the Sixth Applied Natural Language Processing (ANLP-2000)*, Seattle, WA, 2000.

Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eva Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Third Edition). World Wide Web Consortium Recommendation 04 February 2004., 2004. <http://www.w3.org/TR/2004/REC-xml-20040204> Access Online August 2004.

Eric Brill. *A Corpus-Based Approach to Language Learning*. PhD thesis, Depart of Computer and Information Science, University of Pennsylvania, 1993.

J. Carletta, S. Evert, U. Heid, J. Kilgour, J. Robertson, and H. Voormann. The NITE XML toolkit: flexible annotation for multi-modal language data. *Behavior Research Methods, Instruments, and Computers, special issue on Measuring Behavior*, 35(3): 353–363, 2003a.

J. Carletta, J. Kilgour, T. O'Donnell, S. Evert, and H. Voormann. The NITE object model library for handling structured linguistic annotation on multimodal data sets. In *Proceedings of the EACL Workshop on Language Technology and the Semantic Web (3rd Workshop on NLP and XML, NLPXML-2003)*, 2003b.

J. Carletta, S. Dingare, M. Nissim, and T. Nikitina. Using the NITE XML toolkit on the Switchboard corpus to study syntactic choice: a case study. In *Proc. of Fourth Language Resources and Evaluation Conference, Lisbon, Portugal, May 2004*, 2004.

Jean Carletta, David McKelvie, Amy Isard, Andreas Mengel, Marion Klein, and Morten Baun Møller. A generic approach to software support for linguistic annotation using xml. In G. Sampson and D. McCarthy, editors, *Readings in Corpus Linguistics*. Continuum International, London and NY, 2002.

Steve Cassidy. Compiling multi-tiered speech databases into the relational model: Experiments with the Emu system. In *Proceedings of Eurospeech '99, Budapest, September 1999*, 1999.

Steve Cassidy. XQuery as an annotation query language: a use case analysis. In *Proceedings of the Third International Conference on Language Resources and Evaluation (LREC 2002), Las Palmas, Spain, May 2002* ELR [2002]. <http://wave.ldc.upenn.edu/Projects/QLDB/cassidy-lrec.pdf>, Access Online August 2004.

Steve Cassidy and Steven Bird. Querying databases of annotated speech. In M.E. Orlowska, editor, *Database Technologies: Proceedings of the Eleventh Australasian Database Conference, volume 22 of Australian Computer Science Communications, Canberra, Australia*, pages 12–20. IEEE Computer Society, 2000. <http://arxiv.org/abs/cs/0204026>, Access Online August 2004.

Steve Cassidy and Jonathan Harrington. Multi-level annotation in the Emu speech database management system. *Speech Communication*, 33(1,2):61–77, 2001.

Steve Cassidy, Pauline Welby, Julie McGory, and Mary Beckman. Testing the adequacy of query languages against annotated spoken dialog. In *Proceedings of the 8th Australian International Conference on Speech Science and Technology, Canberra, December 2000*, pages 428–433, 2000.

R.G.G. Cattell and Douglas K. Barry, editors. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, San Francisco, revised february 1998 edition, 1997. ISBN 1-55860-463-4.

Eugene Charniak, Don Blaheta, Niyu Ge, Keith Hall, John Hale, and Mark Johnson. BLLIP 1987–89 WSJ corpus release 1. Linguistic Data Consortium Catalog No LDC2000T43, http://www.ldc.upenn.edu., 2000.

Oliver Christ. A modular and flexible architecture for an integrated corpus query system. In *Proceedings of COMPLEX'94, 3rd Conference on Computational Lexicography and Text Research, Budapest, Hungary, July 7–10, 1994*, pages 23–32, 1994.

Oliver Christ. Linking WordNet to a corpus query system. In John Nerbonne, editor, *Linguistic Databases*, volume 77 of *CSLI Lecture Notes*, pages 189–202. CSLI Publications, Stanford, 1998. ISBN 1-57586-092-9 (PB), ISBN 1-57586-093-7 (HB).

Oliver Christ, Bruno M. Schulze, Anja Hofmann, and Esther König. The IMS corpus workbench: Corpus query processor CQP. Published at http://www.ims.uni-stuttgart.de/projekte/CorpusWorkbench/CQPUserManual/HTML/, 1999.

Vassilis Christophides, Serge Abiteboul, Sophie Cluet, and Michel Scholl. From structured documents to novel query facilities. In Richard T. Snodgrass and Marianne

Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 313–324. ACM Press, 1994. <http://doi.acm.org/10.1145/191839.191901> Access Online August 2004.

Charles L.A. Clarke, G.V. Cormack, and F.J. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 38(1):43–56, 1995.

E.F. Codd. Data models in database management. In *International Conference on Management of Data, Proceedings of the 1980 workshop on data abstraction, databases, and conceptual models, Pingree Park, Colorado, USA*, pages 112–114, 1980. ISBN 0-89791-031-1.

M. P. Consens and A. O. Mendelzon. Expressing structural hypertext queries in graphlog. In *HYPERTEXT '89: Proceedings of the second annual ACM conference on Hypertext*, pages 269–292, New York, NY, USA, 1989. ACM. ISBN 0-89791-339-6. doi: http://doi.acm.org/10.1145/74224.74247.

Johann Cook, editor. *Bible and Computer - the Stellenbosch AIBI-6 Conference: Proceedings of the Association Internationale Bible Et Informatique "From Alpha to Byte", University of Stellenbosch, 17-21 July, 2000*, Leiden, 2002. Association Internationale Bible et Informatique, Brill. ISBN 9004124950.

Scott Cotton and Steven Bird. An integrated framework for treebanks and multilayer annotations. In *Proceedings of the Third International Conference on Language Resources and Evaluation (LREC 2002), Las Palmas, Spain, May 2002* ELR [2002], pages 1670–1677. <http://arxiv.org/abs/cs/0204007>. Access Online August 2004.

Robin Cover. Cover pages – online resource for markup language technologies. http://xml.coverpages.org Accessed online April 12, 2008., 1986-2008.

Hamish Cunningham and Kalina Bontcheva. Software architectures for language engineering: a critical review. Technical report, Institute for Language, Speech and Hearing (ILASH) and Department of Computer Science, University of Sheffield, UK, 2003. <http://www.dcs.shef.ac.uk/research/resmes/papers/CS0309.pdf> Access Online August 2004.

Hamish Cunningham, Diana Maynard, Kalina Bontcheva, and Valentin Tablan. GATE: A framework and graphical development environment for robust NLP tools and applications. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02). Philadelphia, July 2002*, 2002. <http://gate.ac.uk/sale/acl02/acl-main.pdf> Access Online August 2004.

C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, sixth edition, 1995. ISBN 0-201-82458-2.

Mark Davies. Relational n-gram databases as a basis for unlimited annotation on large corpora. In *Proceedings of The Shallow Processing of Large Corpora Workshop (SProLaC 2003), held in conjunction with CORPUS LINGUISTICS 2003, Lancaster University (UK), 27 March, 2003*, 2003.

Alex de Joia and Adrian Stenton. *Terms in Systemic Linguistics: A Guide to Halliday*. Batsford Academic and Educational Ltd., London, 1980.

David DeHaan, David Toman, Mariano P. Consens, and M. Tamer Özsu. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *Proceedings of SIGMOD 2003, June 9-12, San Diego, CA*. Association for Computing Machinery, 2003.

Crist-Jan Doedens. *Text Databases: One Database Model and Several Retrieval Languages*. Number 14 in Language and Computers. Editions Rodopi Amsterdam, Amsterdam and Atlanta, GA, 1994. ISBN 90-5183-729-1.

Crist-Jan Doedens and Henk Harmsen. Quest retrieval language reference manual. unpublished reference manual, presented to AND software., September 1990.

Janet W. Dyk. *Participles in Context. A Computer-Assisted Study of Old Testament Hebrew.*, volume 12 of *APPLICATIO*. VU University Press, Amsterdam, 1994. Doctoral dissertation, Vrije Universiteit Amsterdam. Promotors:Prof. Dr. G. E. Booij and Prof. Dr. E. Talstra.

Janet W. Dyk and Eep Talstra. Computer-assisted study of syntactical change, the shift in the use of the participle in Biblical and Post-Biblical Hebrew texts. In Pieter van Reenen and Karin van Reenen-Stein, editors, *Spatial and Temporal Distributions, Manuscript Constellations – Studies in language variation offered to Anthonij Dees on the occasion of his 60th birthday*, pages 49–62, Amsterdam/Philadelphia, 1988. John Benjamins Publishing Co. ISBN 90-272-2062-X.

Suzanne Eggins. *An Introduction to Systemic Functional Linguistics*. Continuum, London and New York, 1994.

*Proceedings of the Third International Conference on Language Resources and Evaluation (LREC 2002), Las Palmas, Spain, May 2002*, 2002. ELRA, European Language Resources Association.

Stefan Evert, Jean Carletta, Timothy J. O'Donnell, Jonathan Kilgour, Andreas Vögele, and Holger Voorman. The NITE object model, version 2.1. Published at <http://www.ltg.ed.ac.uk/NITE/documents/NiteObjectModel.v2.1.pdf>., 24 March 2003.

Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, London, England and Cambridge, Massachusetts, 1998.

William B. Frakes and Ricardo Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice Hall, 1992.

Hideo Fujii and Bruce W. Croft. A comparison of indexing techniques for japanese text retrieval. In *Proceedings of SIGIR 1993, Pittsburgh, PA, USA.*, pages 237–246, 1993.

Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997. ISBN 3540627715. Translator-C. Franzke.

Gaston H. Gonnet and Frank Wm. Tompa. Mind your grammar: a new approach to modelling text. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 339–346. Morgan Kaufmann, 1987. ISBN 0-934613-46-X. <http://www.sigmod.org/sigmod/dblp/db/conf/vldb/GonnetT87.html> Access Online August 2004.

Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. Lexicographical indices for text: Inverted files vs. pat tree. In William B. Frakes and Ricardo A. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992. ISBN 0-13-463837-9.

Jacob Harold Greenlee. *Introduction to New Testament Textual Criticism*. Hendrickson, Peabody, MA, USA, revised edition, 1995.

Antonin Guttman. R-Trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of Data, Boston, Massachusetts, USA*, pages 47–57, 1984. <http://doi.acm.org/10.1145/602259.602266>, Access Online January 2005.

Marc Gyssens, Jan Paredaens, and Dirk Van Gucht. A grammar-based approach towards unifying hierarchical data models (extended abstract). In James Clifford, Bruce G. Lindsay, and David Maier, editors, *SIGMOD Conference*, pages 263–272. ACM Press, 1989.

Michael Alexander Kirkwood Halliday. Systemic grammar. In Gunther R. Kress, editor, *Halliday: System and Function in Language*, pages 3–6. Oxford University Press, London, 1976/1969. Original article from 1969.

Michael Alexander Kirkwood Halliday. *Introduction to Functional Grammar*. Edward Arnold, London and New York, 2nd edition, 1994.

Christof Hardmeier and Eep Talstra. Sprachgestalt und Sinngehalt: Wege zu neuen Instrumenten der computergestützten Textwahrnehmung. *Zeitschrift für due Alttestamentliche Wissenschaft*, 101:3:408–428, 1989.

Henk Harmsen. Software functions, quest-operating-system. unpublished report, Theological Faculty, Vrije Universiteit, Amsterdam., September 1988.

Henk Harmsen. QUEST: a query concept for text research. In *Proceedings of the 3rd Association Bible et Informatique Conference (AIBI3), Tübingen, 26–30 August, 1991* Ass [1992], pages 319–328.

Eliotte Rusty Harold and W. Scott Means. *XML In a Nutshell: A Desktop Quick Reference*. O'Reilly, third edition, 2004.

Geoffrey Horrocks. *Generative Grammar*. Longman, London and New York, 1987.

James W. Hunt and M. Doug McIlroy. An algorithm for differential file comparison. Technical Report CSTR 41, Bell Telephone Laboratories, Murray Hill, NJ, 1976.

Matt Insall and Eric W. Weisstein. “lattice.” from mathworld–a wolfram web resource. http://mathworld.wolfram.com/Lattice.html.

ISO. Information processing – text and office systems – standard generalized markup language (sgml). International ISO standard ISO 8879:1986, with corrigenda and amendments., 1986.

Jani Jaakkola and Pekka Kilpeläinen. Using sgrep for querying structured text files. Technical Report C-1996-83, Department of Computer Science, University of Helsinki, November 1996a. http://www.cs.helsinki.fi/TR/C-1996/83.

Jani Jaakkola and Pekka Kilpeläinen. Nested text-region algebra. Technical Report C-1999-2, Department of Computer Science, University of Helsinki, January 1996b. http://www.cs.helsinki.fi/TR/C-1992/2/.

Ray Jackendoff. *X-bar Syntax: A Study of Phrase Structure*. MIT Press, Cambridge, Massachusetts, 1977.

Laura Kallmeyer. A query tool for syntactically annotated corpora. In *Proceedings of Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora, Hong Kong, October 2000*, pages 190–198, 2000.

Vipul Kashyap and Marek Rusinkiewicz. Modeling and querying textual data using E-R models and SQL. In *Proceedings of the Workshop on Management of SemiStructured Data in conjunction with the 1997 ACM International Conference on the Management of Data (SIGMOD) Tucson, Arizona, May 1997*, 1997. <http://lsdis.cs.uga.edu/~kashyap/publications/SIGMOD-workshop.ps>, Access Online January 2005.

Stephan Kepser. Finite structure query: a tool for querying syntactically annotated corpora. In *EACL ’03: Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics*, pages 179–186, Morristown, NJ, USA, 2003. Association for Computational Linguistics. ISBN 1-333-56789-0. doi: http://dx.doi.org/10.3115/1067807.1067832.

Adam Kilgariff, Pavel Rychly, Pavel Smrz, and David Tugwell. The sketch engine. In *Proceedings of the 11th EURALEX International Congress, Loviat, France*, pages 105–116, 2004.

Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. doi: 10.1137/0206024.

Esther König and Wolfgang Lezius. The TIGER language. a description language for syntax graphs. formal definition. Technical report, Institut für Maschinelle Sprachverarbeitung (IMS) Unversity of Stuttgart,Germany, April 22 2003. www.ims.uni-stuttgart.de/projekte/TIGER.

Marcel Kornacker. High-performance extensible indexing. In *Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999*, pages 699–708. ACM, 1999.

Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. Managing intervals efficiently in object-relational databases. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 407–418, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. ISBN 1-55860-715-3.

Catherine Lai. A formal framework for linguistic tree query. Master's thesis, University of Melbourne, 2006. This is a "Master of Science by Research"-thesis. 170pp.

Per-Åke Larson. A method for speeding up text retrieval. *ACM SIGMIS Database*, 15: 19–23, 1984. <http://doi.acm.org/10.1145/1017712.1017717>, Access Online January 2005.

Mong Li Lee, Wynne Hsu, Christian S. Jensen, Bin Cui, and Keng Lik Teo. Supporting frequent updates in r-trees: A bottom-up approach. In *Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003*, 2003.

Fritz Lehmann and Rudolf Wille. A triadic approach to formal concept analysis. In Gerard Ellis, Robert Levinson, William Rich, and John F. Sowa, editors, *Conceptual Structures: Applications, Implementation and Theory – Third International Conference on Conceptual Structures, ICCS'95, Santa Cruz, CA, USA, August 1995, Proceedings*, volume 954 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 32–43, Berlin, 1995. Springer Verlag.

Wolfgang Lezius. *Ein Suchwerkzeug für syntaktisch annotierte Textkorpora*. PhD thesis, Institut für Maschinelle Sprachverarbeitung, University of Stuttgart, December 2002a. Arbeitspapiere des Instituts für Maschinelle Sprachverarbeitung (AIMS), volume 8, number 4. <http://www.ims.uni-stuttgart.de/projekte/corplex/paper/lezius/diss/>, Access Online August 2004.

Wolfgang. Lezius. TIGERSearch – ein Suchwerkzeug für Baumbanken. In Stephan Busemann, editor, *Proceedings der 6. Konferenz zur Verarbeitung natürlicher Sprache (KONVENS 2002), Saarbrücken*, pages 107–114, 2002b.

Arjan. Loeffen. Text databases: a survey of text models and systems. *ACM SIGMOD Record*, 23(1):97–106, March 1994. <http://doi.acm.org/10.1145/181550.181565> Access Online August 2004.

Robert E. Longacre. *The Grammar of Discourse*. Topics in Language and Linguistics. Kluwer Academic / Plenum Press, New York and London, 2nd edition, 1996. ISBN 0306452359.

H. P. Luhn. Keyword-in-context index for technical literature (KWIC index). *American Documentation*, 11(4):288–295, October 1960.

M. Marcus, G. Kim, M. Marcinkiewicz, R. MacIntyre, A. Bies, M. Ferguson, K. Katz, and B. Schasberger. The Penn treebank: Annotating predicate argument structure. In *ARPA Human Language Technology Workshop*, 1994a.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2): 313–330, 1994b.

John C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw-Hill, Singapore, international edition, 1991. ISBN 0-07-100851-9.

Ronald G. Matteson. *Introduction to Document Image Processing Techniques*. Artech House, Boston, London, 1995. ISBN 0-89006-492-X.

James D. McCawley. Parentheticals and discontinuous constituent structure. *Linguistic Inquiry*, 13(1):91–106, 1982.

David McKelvie, Amy Isard, Andreas Mengel, Morten Baun Møller, Michael Grosse, and Marion Klein. The MATE workbench — an annotation tool for XML coded speech corpora. *Speech Communication*, 33(1,2):97–112, 2001.

Andreas Mengel. MATE deliverable D3.1 – specification of coding workbench: 3.8 improved query language (Q4M). Technical report, Institut für Maschinelle Sprachverarbeitung, Stuttgart, 18. November, 1999. <http://www.ims.uni-stuttgart.de/projekte/mate/q4m/>.

Andreas Mengel and Wolfgang Lezius. An xml-based representation format for syntactically annotated corpora. In *In Proceedings of the International Conference on Language Resources and Evaluation (LREC), Athens, Greece, 2000*, pages 121–126, 2000.

Maël Benjamin Mettler. Parallel treebank search — the implementation of stockholm treealigner search. B.Sc. thesis in Computational Linguistics, Department of Linguistics, Stockholm University, Sweden., March 2007.

George A. Miller, Richard Beckwith, Christiane Fellbaum, Derek Gross, and Katherine J. Miller. Introduction to WordNet: an on-line lexical database. *International Journal of Lexicography*, 3(4):235–244, 1990.

Jiří Mírovský, Roman Ondruška, and Daniel Pruša. Searching through prague dependency treebank. In *Proceedings of the First Workshop on Treebanks and Linguistic Theories (TLT2002), Sozopol, Bulgaria*, pages 144–122, 2002.

Kaj Munk. *Ordet — Skoleudgave — Med Indledning og Oplysninger ved Niels Nielsen*. Nyt Nordisk Forlag, Arnold Busck, Kjøbenhavn, 1947.

Preslav Nakov, Ariel Schwartz, Brian Wolf, and Marti Hearst. Supporting annotation layers for natural language processing. In *Proceedings of the ACL 2005 on Interactive poster and demonstration sessions, Ann Arbor, Michigan*, pages 65–68, 2005.

Gonzales Navarro and Ricardo Baeza-Yates. A language for queries on structure and contents of textual databases. In Edward A. Fox, Peter Ingwersen, and Raya Fidel, editors, *SIGIR'95, Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. Seattle, Washington, USA, July 9-13, 1995 (Special Issue of the SIGIR Forum)*, pages 93–101. ACM Press, 1995. ISBN 0-89791-714-6.

Gonzales Navarro and Ricardo Baeza-Yates. Proximal nodes: A model to query document databases by content and structure. *ACM Transactions on Information Systems (TOIS)*,

15(4):400–435, October 1997. <http://doi.acm.org/10.1145/263479.263482>, Access Online August 2004.

Gavin Thomas Nicol. Core range algebra – toward a formal model of markup. Online article, http://www.mind-to-mind.com/library/papers/index.html Access online April 12, 2008, 2002.

Jórgen Fischer Nilsson. A logico-algebraic framework for ontologies: ONTOLOG. In Per Anker Jensen and Peter Skadhauge, editors, *Ontology-based Interpretation of Noun Phrases: Proceedings of the First International OntoQuery Workshop*, number 21/2001 in Skriftserie - Syddansk Universitet, Institut for Fagsprog, Kommunikation og Informationsvidenskab, pages 11–35, Kolding, 2001. Dept. of Business Communication and Information Science, University of Southern Denmark.

Peter Øhrstrøm and Per F.V. Hasle. *Temporal Logic — From Ancient Ideas to Artificial Intelligence*, volume 57 of *Studies in Linguistics and Philosophy*. Kluwer Academic Publisher, Dordrecht, 1995. ISBN 0-7923-3586-4.

Peter Øhrstrøm, Jan Andersen, and Henrik Schärfe. What has happened to ontology. In Frithjof Dau, Marie-Laure Mugnier, and Gerd Stumme, editors, *Conceptual Structures: Common Semantics for Sharing Knowledge: 13th International Conference on Conceptual Structures, ICCS 2005*, volume 3596 of *LNCS*, pages 425–438. Springer Verlag, 2005.

G. Petasis, V. Karkaletsis, G. Paliouras, I. Androutsopoulos, and C. D. Spyropoulos. Ellogon: A new text engineering platform. In *Proceedings of the Third International Conference on Language Resources and Evaluation (LREC 2002), Las Palmas, Spain, May 2002* ELR [2002], pages 72–78.

Ulrik Petersen. The Extended MdF model. Unpublished B.Sc. thesis in computer science, DAIMI, Aarhus University, Denmark. Available from http://ulrikp.org/., November 1999.

Ulrik Petersen. Emdros — a text database engine for analyzed or annotated text. In *Proceedings of COLING 2004*, 20th *International Conference on Computational Linguistics, August* 23rd *to* 27th*, 2004, Geneva*, pages 1190–1193. International Commitee on Computational Linguistics, 2004. http://emdros.org/petersen-emdros-COLING-2004.pdf.

Ulrik Petersen. Querying both parallel and treebank corpora: Evaluation of a corpus query system. In *Proceedings of LREC 2006*, 2006a. Available as http://ulrikp.org/˜ulrikp/pdf/LREC2006.pdf.

Ulrik Petersen. MQL programmer's reference guide. Published as part of every Emdros software release., 2007a.

Ulrik Petersen. Principles, implementation strategies, and evaluation of a corpus query system. In *Finite-State Methods and Natural Language Processing – 5th International Workshop, FSMNLP 2005, Helsinki, Finland, September 1-2, 2005. Revised Papers*, volume 4002 of *Lecture Notes in Artifical Intelligence (LNAI)*, pages 215–226, Berlin, Heidelberg, New York, 2006b. Springer Verlag.

Ulrik Petersen. Evaluating corpus query systems on functionality and speed: Tigersearch and emdros. In Angelova et al. [2005], pages 387–391. ISBN 954-91743-3-6.

Ulrik Petersen. Relational implementation of the emdf model. Published as part of every Emdros software release., 2007b.

Kenneth L. Pike and Evelyn Pike. *Grammatical Analysis*. Number 53 in Summer Institute of Linguistics, Publications in Linguistics. Summer Institute of Linguistics, 2nd revised edition, 1982. Reprint 1991. xxvi + 463 + 39 pages.

Martin F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–138, July 1980.

Eric Steven Raymond. *The Art of Unix Programming*. Addison-Wesley, Reading, Massachusetts, USA, 2003.

Philip Resnik and Aaron Elkiss. The linguist's search engine: An overview. In *Proceedings of ACL 2005 (Demonstration Section)*, 2005.

Douglas L. T. Rohde. Tgrep2 user manual, version 1.12. Available online <http://tedlab.mit.edu/~dr/Tgrep2/tgrep2.pdf>. Access Online April 2005, 2004.

Pavel Rychlý. *Korpusové manažery a jejich effektivní implementace*. PhD thesis, Faculty of Informatics, Masarykova Univerzita v Brně, Czech Republic, 2000. In Czech. English translation of title: "Corpus Managers and their effective implementation".

Ulrik Sandborg-Petersen. Emdros programmer's reference guide. http://emdros.org/progref/ Accessed March 2008, 2002-2008.

Ingo Schröder. A case study in part-of-speech tagging using the icopost toolkit. Technical Report Computer Science Memo 314/02, University of Hamburg, Germany, 2002.

Julian Smart, Kevin Hock, and Stefan Csomor. *Cross-Platform GUI Programming with wxWidgets*. Bruce Perens' Open Source Series. Prentice-Hall, 2005.

John F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Brooks/Cole Thomson Learning, Pacific Grove, CA, 2000.

Ilona Steiner and Laura Kallmeyer. VIQTORYA – a visual query tool for syntactically annotated corpora. In *Proceedings of the Third International Conference on Language Resources and Evaluation (LREC 2002), Las Palmas, Spain, May 2002* ELR [2002].

Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, USA, third edition, 1997.

Jean Tague, Airi Salminen, and Charles McClellan. Complete formal model for information retrieval systems. In *Proceedings of the 14th annual international ACM SIGIR conference on Research and Development in Information Retrieval, Chicago, Illinois, USA*, pages 14–20, 1991. <http://doi.acm.org/10.1145/122860.122862>, Access Online January 2005.

Eep Talstra. Text grammar and computer. The balance of interpretation and calculation. In *Actes du Troisième Colloque International: "Bible et Informatique: Interprétation, Herméneutique, Compétence Informatique", Tübingen, 26-30 August, 1991* Ass [1992], pages 135–149.

Eep Talstra. Grammar and prophetic texts – computer-assisted syntactical research in Isaiah. In Jacques Vermeylen, editor, *The Book of Isaiah*, volume LXXXI of *Bibliotheca Ephemeridum Theologicarum Lovaniensium*, pages 83–91, Leuven, Netherlands, 1989. Leuven University Press.

Eep Talstra. Computer-assisted linguistic analysis. The Hebrew database used in Quest.2. In Cook [2002], pages 3–22. ISBN 9004124950.

Eep Talstra. On scrolls and screens: Bible reading between history and industry. Unpublished manuscript of the Werkgroep Informatica, Vrije Universiteit, Amsterdam, n.d.

Eep Talstra. Signs, design, and signification. The example of I Kings 21. In Cook [2002], pages 147–166. ISBN 9004124950.

Eep Talstra. Phrases, clauses and clause connections in the Hebrew data base of the Werkgroep Informatica: Computer-assisted production of syntactically parsed textual data. Unpublished manuscript detailing the procedures used in the analysis-software developed at the Werkgroep Informatica, 3 February 1998.

Eep Talstra. A hierarchy of clauses in Biblical Hebrew narrative. In Ellen van Wolde, editor, *Narrative Syntax and the Hebrew Bible*, volume 29 of *Biblical Interpretation Series*, pages 85–118, Leiden, New York, Köln, 1997. Brill. ISBN 90-04-10787-8.

Eep Talstra and Ferenç Postma. On texts and tools. A short history of the Werkgroep Informatica (1977-1987). In Eep Talstra, editor, *Computer Assisted Analysis of Biblical Texts*, volume 7 of *APPLICATIO*, pages 9–27, Amsterdam, 1989. VU University Press.

Eep Talstra and Constantijn Sikkel. Genese und Kategorienentwicklung der WIVU-Datenbank. In Christof Hardmeier, Wolf-Dieter Syring, Jochen D. Range, and Eep Talstra, editors, *Ad Fontes! Quellen erfassen - lesen - deuten. Was ist Computerphilologie?*, volume 15 of *APPLICATIO*, pages 33–68, Amsterdam, 2000. VU University Press.

Eep Talstra and Christo H.J. van der Merwe. Analysis, retrieval and the demand for more data. Integrating the results of a formal textlinguistic and cognitive based pragmatic approach to the analysis of Deut 4:1-40. In Cook [2002], pages 43–78. ISBN 9004124950.

Eep Talstra and Archibald L.H.M. Van Wieringen, editors. *A Prophet on the Screen – Computerized Description and Literary Interpretation of Isaianic Texts*, volume 9 of *APPLICATIO*, Amsterdam, 1992. VU University Press.

Eep Talstra, Christof Hardmeier, and James Alan Groves. Quest. Electronic concordance application for the Hebrew Bible (database and retrieval software). Nederlands Bijbelgenootschap (NBG), Haarlem, Netherlands, 1992. Manual: J.A. Groves, H.J.

Bosman, J.H. Harmsen, E. Talstra, *User Manual Quest. Electronic Concordance Application for the Hebrew Bible, Haarlem, 1992*.

Henry S. Thompson and David McKelvie. Hyperlink semantics for standoff markup of read-only documents. In *Proceedings of SGML Europe '97, Barcelona, Spain, May 1997*, 1997. Available: <http://www.ltg.ed.ac.uk/~ht/sgmleu97.html>.

Frank Wm. Tompa. A data model for flexible hypertext database systems. *ACM Transactions on Information Systems*, 7(1):85–100, January 1989.

Jeffrey D. Ullman and Jennifer Widom. *A First Course in Database Systems*. Prentice-Hall International, London, 1997. ISBN 0-13-887647-9.

Robert D. Van Valin, Jr. *An introduction to Syntax*. Cambridge University Press, Cambridge, U.K., 2001.

Robert D. Van Valin, Jr. and Randy J. LaPolla. *Syntax – Structure, meaning, and function*. Cambridge University Press, Cambridge, U.K., 1997.

A.L.H.M. van Wieringen. *Analogies in Isaiah Volumes A + B*. Number 10 in Applicatio. Free University Press, Amsterdam, 1993.

Arian J.C. Verheij and Eep Talstra. Crunching participles. An aspect of computer assisted syntactical analysis demonstrated on Isaiah 1-12. In Talstra and Van Wieringen [1992], pages 21–33.

Holger Voormann, Stefan Evert, Jonathan Kilgour, and Jean Carletta. NXT search user's manual (draft). http://www.ims.uni-stuttgart.de/projekte/nite/manual/, 2003.

W3C contributors. XHTML 1.0 The extensible hypertext markup language: A reformulation of HTML 4 in XML 1.0, W3C Recommendation 26 january 2000, revised 1 august 2002, 2002.

Glynn Winskel. *The formal semantics of programming languages: An introduction*. MIT Press, Cambridge, Mass., 1993. ISBN 0-262-23169-7.

Nicolai Winther-Nielsen and Eep Talstra. *A Computational Display of Joshua. A Computer-assisted Analysis and Textual Interpretation*, volume 13 of *APPLICATIO*. VU University Press, Amsterdam, 1995.

Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, California, United States*, pages 425–436, 2001. ISBN:1-58113-332-4.

Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):1–56, July 2006.

Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490, December 1998.

# Appendix A

# Topographic MQL: Grammar

In this Appendix, I display the grammar of the topographic part of MQL, as it is today. This is done in something that looks like Backus-Naur form.

```
topograph ::= blocks.

blocks ::= block_string.

block_string ::= block_string2 .
block_string ::= block_string2 KEY_OR block_string .

block_string2 ::= block_string1 .
block_string2 ::= block_string1 block_string2 .
block_string2 ::= block_string1 KEY_EXCLAMATION block_string2 .

block_string1 ::= block_string0 .
block_string1 ::= block_string0 KEY_STAR star_monad_set.

block_string0 ::= block  .
block_string0 ::= KEY_OPEN_SQUARE_BRACKET block_string
                  KEY_CLOSE_SQUARE_BRACKET .

block ::= object_block.
block ::= power.
block ::= opt_gap_block.
block ::= gap_block.
block ::= notexist_object_block.

object_block ::=  KEY_OPEN_SQUARE_BRACKET  object_type_name
     mark_declaration
     object_reference_declaration
     retrieval  firstlast
     feature_constraints
     feature_retrieval
     opt_blocks  KEY_CLOSE_SQUARE_BRACKET.
```

```
notexist_object_block ::=  notexist
     KEY_OPEN_SQUARE_BRACKET  object_type_name
     mark_declaration
     object_reference_declaration
     retrieval  firstlast
     feature_constraints
     feature_retrieval
     opt_blocks  KEY_CLOSE_SQUARE_BRACKET.


notexist ::= KEY_NOTEXIST .
notexist ::= KEY_NOTEXISTS .

object_reference_declaration ::= . /* empty */
object_reference_declaration ::= KEY_AS object_reference.

mark_declaration ::= . /* empty */
mark_declaration ::= MARK .

object_reference ::= IDENTIFIER.

retrieval ::= . /* empty */
retrieval ::= KEY_NORETRIEVE.
retrieval ::= KEY_RETRIEVE.
retrieval ::= KEY_FOCUS.

firstlast ::= . /* empty */
firstlast ::= KEY_FIRST.
firstlast ::= KEY_LAST.
firstlast ::= KEY_FIRST  KEY_AND  KEY_LAST.

feature_constraints ::= .
feature_constraints ::= ffeatures.

ffeatures ::= fterm.
ffeatures ::= ffeatures KEY_OR fterm.

fterm ::= ffactor.
fterm ::= fterm KEY_AND ffactor.

ffactor ::=  KEY_NOT  ffactor.
ffactor ::= KEY_OPEN_BRACKET  ffeatures  KEY_CLOSE_BRACKET.
ffactor ::= feature_comparison.

feature_comparison ::= feature_name comparison_operator value.
feature_comparison ::= feature_name KEY_IN
        KEY_OPEN_BRACKET list_of_identifier KEY_CLOSE_BRACKET.
```

```
feature_comparison ::= feature_name KEY_IN
        KEY_OPEN_BRACKET list_of_integer KEY_CLOSE_BRACKET.
feature_comparison ::= feature_name KEY_IN object_reference_usage.

comparison_operator ::= KEY_EQUALS.
comparison_operator ::= KEY_LESS_THAN.
comparison_operator ::= KEY_GREATER_THAN.
comparison_operator ::= KEY_NOT_EQUAL.
comparison_operator ::= KEY_LESS_THAN_OR_EQUAL.
comparison_operator ::= KEY_GREATER_THAN_OR_EQUAL.
comparison_operator ::= KEY_TILDE.
comparison_operator ::= KEY_NOT_TILDE.
comparison_operator ::= KEY_HAS.


value ::= enum_const.
value ::= signed_integer.
value ::= STRING.
value ::= object_reference_usage.

enum_const ::=  IDENTIFIER.


object_reference_usage ::= object_reference KEY_DOT feature_name.


feature_retrieval ::= KEY_GET  feature_list.
feature_retrieval ::= . /* empty */

feature_list ::= feature_name.
feature_list ::= feature_list KEY_COMMA feature_name.

opt_blocks ::= . /* empty */
opt_blocks ::= blocks.

star_monad_set ::= .
star_monad_set ::= monad_set .

monad_set ::= KEY_OPEN_BRACE monad_set_element_list KEY_CLOSE_BRACE.

monad_set_element_list ::= monad_set_element.
monad_set_element_list ::=
    monad_set_element_list KEY_COMMA monad_set_element.

monad_set_element ::= INTEGER.
monad_set_element ::= INTEGER KEY_DASH INTEGER.
monad_set_element ::= INTEGER KEY_DASH .
```

```
opt_gap_block ::= KEY_OPEN_SQUARE_BRACKET  KEY_OPT_GAP
      mark_declaration gap_retrieval  opt_blocks
      KEY_CLOSE_SQUARE_BRACKET.

gap_retrieval ::= . /* empty */
gap_retrieval ::= KEY_NORETRIEVE.
gap_retrieval ::= KEY_RETRIEVE.
gap_retrieval ::= KEY_FOCUS.

gap_block ::= KEY_OPEN_SQUARE_BRACKET  KEY_GAP
      mark_declaration  gap_retrieval  opt_blocks
      KEY_CLOSE_SQUARE_BRACKET.

power ::=  KEY_POWER  restrictor.
power ::=  KEY_POWER KEY_BETWEEN limit KEY_AND limit.

restrictor ::= . /*  empty  */
restrictor ::= KEY_LESS_THAN  limit.
restrictor ::= KEY_LESS_THAN_OR_EQUAL  limit.

limit  ::=  INTEGER.  /* non-negative integer, may be 0. */
```

# Appendix B

# Published articles

The following pages contain the published articles which form part of the basis for evaluation of my PhD work.

# [COLING2004]

# Emdros — a text database engine for analyzed or annotated text

Ulrik Petersen

2004

**Published in:** Proceedings of COLING 2004, held August 23–27, 2004 in Geneva. International Committee on Computational Linguistics, pp. 1190–1193

This page left intentionally blank

# Emdros – a text database engine for analyzed or annotated text

**Ulrik Petersen**
Department of Communication
University of Aalborg
Kroghstræde 3
DK – 9220 Aalborg East
Denmark
ulrikp@hum.aau.dk

## Abstract

Emdros is a text database engine for linguistic analysis or annotation of text. It is applicable especially in corpus linguistics for storing and retrieving linguistic analyses of text, at any linguistic level. Emdros implements the EMdF text database model and the MQL query language. In this paper, I present both, and give an example of how Emdros can be useful in computational linguistics.

## 1 Introduction

As (Abeillé, 2003) points out, "corpus-based linguistics has been largely limited to phenomena that can be accessed via searches on particular words. Inquiries about subject inversion or agentless passives are impossible to perform on commonly available corpora" (p. xiii).

Emdros is a text database engine which attempts to remedy this situation in some measure. Emdros' query language is very powerful, allowing the kind of searches which Abeillé mentions to be formulated quickly and intuitively. Of course, this presupposes a database which is tagged with the data necessary for answering the query.

Work has been done on supporting complex queries, e.g., (Bird et al., 2000; Cassidy and Bird, 2000; Mengel, 1999; Clarke et al., 1995). Emdros complements these pieces of work, providing a working implementation of many of the features which these systems support.

In this paper, I present the EMdF text database model on which Emdros rests, and the MQL query language which it implements. In addition, I give an example of how Emdros can be useful in answering questions in computational linguistics.

## 2 History of Emdros

Emdros springs out of a reformulation and implementation of the work done by Crist-Jan Doedens in his 1994 PhD thesis (Doedens, 1994). Doedens defined the MdF (Monads-dot-Features) text database model, and the QL query language. Doedens gave a denotational semantics for QL and loaded QL with features, thus making it very difficult to implement. The present author later took Doedens' QL, scaled it down, and gave it an operational semantics, hence making it easier to implement, resulting in the MQL query language. I also took the MdF model and extended it slightly, resulting in the EMdF model. Later, I implemented both, resulting in the Emdros text database engine, which has been available as Open Source software since October 2001. The website[1] has full sourcecode and documentation.

Emdros is a general-purpose engine, not a specific application. This means that Emdros must be incorporated into a specific software application before it can be made useful.

## 3 The EMdF model

The EMdF model is an extension of the MdF model developed in (Doedens, 1994). The EMdF (Extended MdF) model is based on four concepts: Monad, object, object type, and feature. I describe each of these in turn, and give a small example of an EMdF database.

### 3.1 Monad

A monad is simply an integer. The sequence of the integers (1,2,3, etc.) dictates the sequence of the text. The monads do not impose a *reading-direction* (e.g., left-to-right, right-to-left), but merely a *logical text-order*.

### 3.2 Object

An object is simply a set of monads with an associated object type. The set is arbitrary in the sense that there are no restrictions on the set. E.g., $\{1\}$, $\{2\}$, $\{1,2\}$, $\{1,2,6,7\}$ are all valid objects. This allows for objects with gaps, or discontiguous objects (e.g., discontiguous clauses). In addition, an object always has a unique integer id, separate from the the object's monad set.

Objects are the building blocks of the text itself, as well as the annotations or analyses in the

[1] `http://emdros.org/`

database. To see how, we must introduce object types.

### 3.3 Object type

An object type groups a set of objects into such classes as “Word”, “Phrase”, “Clause”, “Sentence”, “Paragraph”, “Chapter”, “Book”, “Quotation”, “Report”, etc. Generally, when designing an Emdros database, one chooses a *monad-granularity* which dictates the smallest object in the database which corresponds to one monad. This smallest object is often “Word”, but could be “Morpheme”, “Phoneme” or even “Grapheme”. Thus, for example, Word number 1 might consist of the object set $\{1\}$, and Word number 2 might consist of the object set $\{2\}$, whereas the first Phrase in the database might consist of the set $\{1,2\}$.

### 3.4 Feature

An object type can have any number of *features*. A feature is an attribute of an object, and always has a type. The type can be a string, an integer, an enumeration, or an object id. The latter allows for complex interrelationships among objects, with objects pointing to each other, e.g., a dependent pointing to a head.

An enumeration is a set of labels with values. For example, one might define an enumeration “psp” (part of speech) with labels such as “noun”, “verb”, “adjective”, etc. Emdros supports arbitrary definition of enumeration label sets.

### 3.5 Example

Consider Figure 1. It shows an EMdF database corresponding to one possible analysis of the sentence “The door was blue.” There are three object types: Word, Phrase, and Clause. The Clause object type has no features. The Phrase object type has the feature “`phr_type`” (phrase type). The Word object type has the features “`surface`” and “`psp`”.

The monad-granularity is “Word”, i.e., each monad corresponds to one monad. Thus the word with id 10001 consists of the monad set $\{1\}$. The phrase with id 10005 consists of the monad set $\{1,2\}$. The single clause object consists of the monad set $\{1,2,3,4\}$.

The text is encoded by the “`surface`” feature on Word object type. One could add features such as “`lemma`”, “`number`”, “`gender`”, or any other feature relevant to the database under construction. The Phrase object type could be given features such as “`function`”, “`apposition_head`”, “`relative_head`”, etc. The Clause object type could be given features distinguishing such things as “VSO order”, “tense of verbal form”, “illocutionary force”, “nominal clause/verbless clause”, etc. It all depends on the theory used to describe the database, as well as the research goals.

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| **word** | **w: 10001**<br>surface: The<br>psp: article | **w: 10002**<br>surface: door<br>psp: noun | **w: 10003**<br>surface: was<br>psp: verb | **w: 10004**<br>surface: blue.<br>psp: adjective |
| **phrase** | **p: 10005**<br>phr_type: NP | | **p: 10006**<br>phr_type: VP | **p: 10007**<br>phr_type: AP |
| **clause** | **c: 10008** | | | |

Figure 1: A small EMdF database

## 4 The MQL query language

MQL is based on two properties of text which are universal: sequence and embedding. All texts have sequence, dictated by the constraints of time and the limitation of our human vocal tract to produce only one sequence of words at any given time. In addition, all texts have, when analyzed linguistically, some element of embedding, as embodied in the notions of phrase, clause, sentence, paragraph, etc.

MQL directly supports searching for sequence and embedding by means of the notion of *topographicity*. Originally invented in (Doedens, 1994), a (formal) language is topographic if and only if there is an isomorphism between the structure of an expression in the language and the objects which the expression denotes.

MQL’s basic building block is the *object block*. An object block searches for objects in the database of a given type, e.g., Word, Phrase or Clause. If two object blocks are adjacent, then the objects which they find must also be adjacent in the database. If an object block is embedded inside another object block, then the inner object must be embedded in the outer object in the database.

Consider Figure 2. It shows two adjacent object blocks, with feature constraints. This would find two Phrase objects in the database where the first is an NP and the second is a VP. The objects must be adjacent in the database because the object blocks are adjacent.

```
[Phrase phrase_type = NP]
[Phrase phrase_type = VP]
```

Figure 2: Two adjacent object blocks

Now consider Figure 3. This query would find a clause, with the restriction that embedded inside the clause must be two phrases, a subject NP and

a predicate VP, in that order. The "`..`" operator means that space is allowed between the NP and the VP, but the space must be inside the limits of the surrounding clause. All of this presupposes an appropriately tagged database, of course.

```
[Clause
  [Phrase phrase_type = NP
          and function = Subj]
  ..
  [Phrase phrase_type = VP
          and function = Pred]
]
```

Figure 3: Examples of embedding

The restrictions of type "`phrase_type = NP`" refer to features (or attributes) of the objects in the database. The restriction expressions can be any Boolean expression (and/or/not/parentheses), allowing very complex restrictions at the object-level.

Consider Figure 4. It shows how one can look for objects inside "gaps" in other objects. In some linguistic theories, the sentence "The door, which opened towards the East, was blue" would consist of one discontiguous clause ("The door ... was blue") with an intervening nonrestrictive relative clause, not part of the surrounding clause. For a sustained argument in favor of this interpretation, see (McCawley, 1982). The query in Figure 4 searches for structures of this kind. The surrounding context is a Sentence. Inside of this sentence, one must find a Clause. The first object in this clause must be a subject NP. Directly adjacent to this subject NP must be a *gap* in the surrounding context (the Clause). Inside of this gap must be a Clause whose clause type is "`nonrestr_rel`". Directly after the close of the gap, one must find a VP whose function is predicate. Mapping this structure to the example sentence is left as an exercise for the reader.

```
[Sentence
  [Clause
    [Phrase FIRST phrase_type = NP
       and function = Subj]
    [gap
      [Clause cl_type = nonrestr_rel]
    ]
    [Phrase phrase_type = VP
            and function = Pred]
  ]
]
```

Figure 4: An example with a gap

Lastly, objects can refer to each other in the query. This is useful for specifying such things as agreement and heads/dependents. In Figure 5, the "`AS`" keyword gives a name ("`w1`") to the noun inside the NP, and this name can then be used inside the adjective in the AdjP to specify agreement.

```
[Phrase phrase_type = NP
  [Word AS w1 psp = noun]
]
[Phrase phrase_type = AdjP
  [Word psp = adjective
      and number = w1.number
      and gender = w1.gender]
]
```

Figure 5: Example with agreement

MQL provides a number of features not covered in this paper. For full documentation, see the website.

The real power of MQL lies in its ability to express complex search restrictions both at the level of structure (sequence and embedding) and at the object-level.

## 5 Application

One prominent example of an Emdros database in use is the Werkgroep Informatica (WI) database of the Hebrew Bible developed under Prof. Dr. Eep Talstra at the Free University of Amsterdam. The WI database is a large text database comprising a syntactic analysis of the Hebrew Bible (also called the Old Testament in Hebrew and Aramaic). This is a 420,000 word corpus with about 1.4 million syntactic objects. The database has been analyzed up to clause level all the way through, and has been analyzed up to sentence level for large portions of the material. A complete description of the database and the underlying linguistic model can be found in (Talstra and Sikkel, 2000).

In the book of Judges chapter 5 verse 1, we are told that "Deborah and Barak sang" a song. Deborah and Barak are clearly a plural entity, yet in Hebrew the verb is feminine singular. Was this an instance of bad grammar? Did only Deborah sing? Why is the verb not plural?

In Hebrew, the rule seems to be that the verb agrees in number and gender with the first item in a compound subject, when the verb precedes the subject. This has been known at least since the 19th century, as evidenced by the Gesenius-Kautzsch grammar of Hebrew, paragraph 146g.

With Emdros and the WI database, we can validate the rule above. The query in Figure 6 finds

234 instances, showing that the pattern was not uncommon, and inspection of the results show that the verb most often agrees with the first member of the compound subject. The 234 “hits” are the bare results returned from the query engine. It is up to the researcher to actually look at the data and verify or falsify their hypothesis. Also, one would have to look for counterexamples with another query.

```
[Clause
  [Phrase function = Pred
    [Word AS w1 psp = verb
          and number = singular]
  ]
  ..
  [Phrase function = Subj
    [Word (psp = noun
       or psp = proper_noun
       or psp = demonstrative_pronoun
       or psp = interrogative_pronoun
       or psp = personal_pronoun)
      and number = singular
      and gender = w1.gender]
    ..
    [Word psp = conjunction]
  ]
]
```

Figure 6: Hebrew example

The query finds clauses within which there are two phrases, the first being a predicate and the second being a subject. The phrases need not be adjacent. The predicate must contain a verb in the singular. The subject must first contain a noun, proper noun, or pronoun which agrees with the verb in number and gender. Then a conjunction must follow the noun, still inside the subject, but not necessarily adjacent to the noun.

The WI database is the primary example of an Emdros database. Other databases stored in Emdros include the morphologically encoded Hebrew Bible produced at the Westminster Hebrew Institute in Philadelphia, Pennsylvania, and a corpus of 67 million words in use at the University of Illinois at Urbana-Champaign.

## 6 Conclusion

In this paper, I have presented the EMdF model and the MQL query language as implemented in the Emdros text database engine. I have shown how MQL supports the formulation of complex linguistic queries on tagged corpora. I have also given an example of a specific problem in Hebrew linguistics which is nicely answered by an Emdros query. Thus Emdros provides a solid platform on which to build applications in corpus linguistics, capable of answering linguistic questions of a complexity higher than what most systems can offer today.

## Acknowledgements

My thanks go to Constantijn Sikkel of the Werkgroep Informatica for coming up with the problem for the Hebrew query example.

## References

Anne Abeillé. 2003. Introduction. In Anne Abeillé, editor, *Treebanks – Building and Using Parsed Corpora*, volume 20 of *Text, Speech and Language Technology*, pages xiii–xxvi. Kluwer Academic Publishers, Dordrecht, Boston, London.

Steven Bird, Peter Buneman, and Wang-Chiew Tan. 2000. Towards a query language for annotation graphs. In *Proceedings of the Second International Conference on Language Resources and Evaluation*, pages 807–814. European Language Resources Association, Paris. http://arxiv.org/abs/cs/0007023.

Steve Cassidy and Steven Bird. 2000. Querying databases of annotated speech. In *Database technologies: Proceedings of the Eleventh Australasian Database Conference*, pages 12–20. IEEE Computer Society.

Charles L. A. Clarke, G. V. Cormack, and F. J. Burkowski. 1995. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 38(1):43–56.

Christianus Franciscus Joannes Doedens. 1994. *Text Databases: One Database Model and Several Retrieval Languages*. Number 14 in Language and Computers. Editions Rodopi Amsterdam, Amsterdam and Atlanta, GA.

James D. McCawley. 1982. Parentheticals and discontinuous constituent structure. *Linguistic Inquiry*, 13(1):91–106.

Andreas Mengel. 1999. MATE deliverable D3.1 – specification of coding workbench: 3.8 improved query language (Q4M). Technical report, Institut für Maschinelle Sprachverarbeitung, Stuttgart, 18 Nov. http://www.ims.uni-stuttgart.de/projekte/mate/q4m/.

Eep Talstra and Constantijn Sikkel. 2000. Genese und Kategorienentwicklung der WIVU-Datenbank. In Christof Hardmeier, Wolf-Dieter Syring, Jochen D. Range, and Eep Talstra, editors, *Ad Fontes! Quellen erfassen - lesen - deuten. Was ist Computerphilologie?*, volume 15 of *APPLICATIO*, pages 33–68, Amsterdam. VU University Press.

# [RANLP2005]

# Evaluating Corpus Query Systems on Functionality and Speed: TIGERSearch and Emdros

Ulrik Petersen

2005

**Published in:** Angelova, G., Bontcheva, K., Mitkov, R., Nicolov, N. and Nikolov, N. (Eds): *International Conference Recent Advances in Natural Language Processing 2005, Proceedings, Borovets, Bulgaria, 21-23 September 2005*, pp. 387–391.

This page left intentionally blank

# Evaluating corpus query systems on functionality and speed: TIGERSearch and Emdros

**Ulrik Petersen**
Department of Communication, University of Aalborg
Kroghstræde 3
9220 Aalborg East, Denmark
ulrikp@hum.aau.dk
**http://emdros.org/**

## Abstract

In this paper, we evaluate two corpus query systems with respect to search functionality and query speed. One corpus query system is TIGERSearch from IMS Stuttgart and the other is our own Emdros corpus query system. First, we show how the database model underlying TIGERSearch can be mapped into the database model of Emdros. Second, the comparison is made based on a set of standard linguistic queries culled from the literature. We show that by mapping a TIGERSearch corpus into the Emdros database model, new query possibilities arise.

## 1 Introduction

The last decade has seen a growth in the number of available corpus query systems. Some query systems which have seen their debut since the mid-1990ies include MATE Q4M (Mengel 99), the Emu query language (Cassidy & Bird 00), the Annotation Graph query language (Bird *et al.* 00), TGrep2 (Rohde 04), TIGERSearch (Lezius 02b), NXT Search (Heid *et al.* 04), Emdros (Petersen 04), and LPath (Bird *et al.* 05). In this paper, we have chosen to evaluate and compare two of these, namely TIGERSearch and Emdros.

TIGERSearch is a corpus query system made at the Institut für Maschinelle Sprachverarbeitung at the University of Stuttgart (Lezius 02a; Lezius 02b). It is a general corpus query system over so-called *syntax graphs* (König & Lezius 03), utilizing the TIGER-XML format for import (Mengel & Lezius 00). Converters have been implemented for the Penn Treebank, NeGRA, Susanne, and Christine formats, among others. It is available free of charge for research purposes.[1]

Emdros is also a general corpus query system, developed at the University of Aalborg, Denmark. It is applicable to a wide variety of linguistic corpora supporting a wide variety of linguistic theories, and is not limited to treebanks. It implements the EMdF model and the MQL query language described in (Petersen 04). Importers for the TIGER-XML and other corpus formats have been implemented, and more are under development. It is available free of charge as Open Source software from the address specified at the beginning of the paper.

The layout of the rest of the paper is as follows. First, we briefly introduce the EMdF database model underlying Emdros. Second, we introduce the database model underlying TIGERSearch. Next, we show how to map the TIGERSearch database model into the EMdF model. The next section explores how the TIGERCorpus (Brants & Hansen 02), now in Emdros format, can be queried with – in some instances – greater functionality and speed by Emdros than by TIGERSearch. Finally, we conclude the paper.

[1] See http://www.tigersearch.de/

## 2 The EMdF model of Emdros

The EMdF text database model underlying Emdros is a descendant of the MdF model described in (Doedens 94). At the backbone of an EMdF database is a string of *monads*. A monad is simply an integer. The sequence of the integers dictates the logical reading sequence of the text. An *object* is an arbitrary (possibly discontiguous) set of monads which belongs to exactly one *object type*. An object type (e.g., Word, Phrase, Clause, Sentence, Paragraph, Article, Line, etc.) determines what *features* an object has. That is, a set of attribute-value pairs are associated with each object, and the attributes are determined by the object type of the object. All attributes are strongly typed. Every object has a database-widely unique ID called its *id_d*, and the feature *self* of an object denotes its id_d. The notation $O.f$ is used to denote the value of feature $f$ on an object $O$. Thus, for example, $O_1.self$ denotes the id_d of object $O_1$. An id_d feature can have the value NIL, meaning it points to no object. No object can have NIL as its id_d.

The sample tree in Figure 1 shows a discontiguous element, and is adapted from (McCawley 82, p. 95). The tree can be visualized as an EMdF database as in Figure 2. This figure exemplifies a useful technique used for representing tree-structures in Emdros: Since, in a tree, a child node always has at most one parent, we can represent the tree by means of id_d features pointing upwards from the child to its parent. If a node has no parent (i.e., is a root node), we can represent this with the value NIL. This technique will be used later when describing the mapping from TIGERSearch to EMdF.

## 3 The TIGERSearch database model

The database model underlying TIGERSearch has been formally described in (Lezius 02a) and (König & Lezius 03). The following description has been adapted from the former, and is a slight reformalization of the database model with respect to edge-labels.

**Definition 1** A *feature record* $F$ is a relation over $FN \times C$ where $FN$ is a set of feature-names and $C$ is a set of

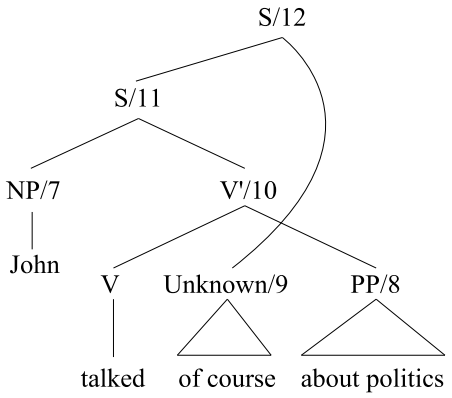



Figure 1: A tree with a discontiguous clause, adapted from (McCawley 82, p. 95).

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| **Word** | id_d: 1<br>surf.: John<br>pos: NProp<br>parent: 7 | id_d: 2<br>surf.: talked<br>pos: V<br>parent: 10 | id_d: 3<br>surf.: of<br>pos: P<br>parent: 9 | id_d: 4<br>surf.: course<br>pos: N<br>parent: 9 | id_d: 5<br>surf.: about<br>pos: P<br>parent: 8 | id_d: 6<br>surf.: politics<br>pos: N<br>parent: 8 |
| **Phrase** | id_d: 7<br>type: NP<br>parent: 11 | | id_d: 9<br>type: Unknown<br>parent: 12 | | id_d: 8<br>type: PP<br>parent: 10 | |
| **Phrase** | | id_d: 10<br>type: V'<br>parent: 11 | | | id_d: 10<br>type: V'<br>parent: 11 | |
| **Clause** | id_d: 11<br>type=S<br>parent: 12 | | | | id_d: 11<br>type=S<br>parent: 12 | |
| **Clause** | id_d: 12<br>type=S | | | | | |

Figure 2: An EMdF representation of the tree in Figure 1.

constants. The relation is defined such that for any $l_i = \langle f_i, c_i \rangle$ and any $l_j = \langle f_j, c_j \rangle$, $l_i \neq l_j \Rightarrow f_i \neq f_j$. That is, all $f_i$ within a feature-record are distinct. The set of all feature-records over FN and C is denoted $\mathcal{F}$.

**Definition 2** The set of all node ids is called $ID$ and the relation $ID \subset C$ holds.

**Definition 3** A *node* is a two-tuple $v \in ID \times \mathcal{F}$. That is, a node consists of a node id $\nu$ and a feature-record $F$.

**Definition 4** A *syntax graph* $G$ in the universe of graphs $\mathcal{G}$ is a six-tuple $G = (V_{NT}, V_T, L_G, E_G, O_G, R_G)$ with the following properties:

1. $V_{NT}$ is the (possibly empty) set of non-terminals.
2. $V_T$ is the non-empty set of terminals.
3. $L_G$ is a set of edge labels where $L_G \subset C$.[2]
4. $E_G$ is the set of labeled, directed edges of $G$. $E_G$ is a set of two-tuples from $V_{NT} \times (V_{NT} \cup V_T)$. If $L_G$ is non-empty, there exists an assignment of edge-labels $el$ which is a total function $el : E_G \rightarrow L_G$ which need be neither surjective nor injective.[3]
5. $O_G$ is a bijective function $O_G : V_T \rightarrow \{1, 2, \ldots, |V_T|\}$ which orders the terminal nodes. That the function is bijective guarantees that all terminal nodes can be ordered totally by $O_G$.
6. $R_G \in V_{NT}$ is the single root node of $G$, and has no incoming edges.

$G$ is a graph with the following characteristics:

**G1:** $G$ is a DAG with exactly one root node $R_G$.

**G2:** All nodes $v \in ((V_{NT} \cup V_T) \setminus R_G)$ have exactly one incoming edge in $E_G$.

**G3:** All nonterminals $v \in V_{NT}$ must have at least one outgoing edge. That is, $\forall v \in V_{NT} \exists v' \in (V_{NT} \cup V_T) : \langle v, v' \rangle \in E_G$.[4]

Thus syntax graphs are not strict trees in the traditional sense, since crossing edges are not prohibited. Nevertheless, syntax graphs are not arbitrary DAGs, since by **G2**, every node has at most one parent, and in this respect they do resemble trees.

This brief reformulation does not do justice to the full description available in (Lezius 02a) and (König & Lezius 03). For more information on the syntax graph formalism, see the cited publications.

## 4 Mapping syntax graphs to EMdF

TIGERSearch was developed specifically for use with the TIGERCorpus (Brants & Hansen 02), though it is applicable to other corpora as well (Lezius 02a, p. 136). In order to compare TIGERSearch with Emdros, we had to import a corpus available for TIGERSearch into Emdros. The TIGERCorpus was chosen because it represents the primary example of a TIGERSearch database, and because it has a reasonably large size, furnishing a basis for speed-comparisons.

We have developed an algorithm to transform any database encoded in the syntax graph formalism into an EMdF database. This section describes the algorithm. First, we give some definitions, after which we show the four algorithms involved.

**Definition A1:** For any syntax graph $G$, $Obj_G$ is the set of EMdF objects which $G$ gives rise to, and $IDD_G$ is the set of id_d's of the objects in $Obj_G$. Note, however, that $IDD_G$ may be defined before $Obj_G$, since there is no causality in the direction from $Obj_G$ to $IDD_G$; in fact it is the other way around in the algorithms below.

[2] The latter restriction is not mentioned by (Lezius 02a) directly on page 103 where this is defined, but is inferred from the rest of the dissertation.

[3] This is where our reformulation differs in meaning from (Lezius 02a). We think our formalization is slightly clearer than Lezius', but we may, of course, have misunderstood something.

[4] Again, my reformulation differs slightly from Lezius' formulation, due to my reinterpretation of $E_G$.

**Definition A2:** For any syntax graph $G$, $NOB_G$ is a bijective function from syntax graph nodes in $G$ to $Obj_G$. That is, $NOB_G : (V_{NT} \cup V_T) \rightarrow Obj_G$.

**Definition A3:** For any syntax graph $G$ and $v \in (V_{NT} \cup V_T)$, $parent(v)$ is the parent node of $v$ if $v$ is not $R_G$, or $\emptyset$ if $v$ is $R_G$.

**Definition A4:** For any syntax graph $G$ and its concomitant $Obj_G$, $id\_d_G$ is a bijective function $id\_d_G : (V_{NT} \cup V_T) \rightarrow IDD_G$ with the definition $id\_d(v) ::= NOB_G(v).\text{self}$. Note, however, that this definition only holds *after* the algorithms have all been applied; in fact $id\_d_G$ is defined by construction rather than by the given intensional, after-the-fact definition.

With this apparatus, we can define four algorithms which use each other. Algorithm 0 merely creates an empty object with a unique EMdF id_d corresponding to each node in a syntax graph $G$. Algorithm 1 adds monads to all objects corresponding to a nonterminal (i.e., all syntax-level nodes). Algorithm 2 constructs a set of EMdF objects for a given syntax graph $G$, and uses Algorithm 0 and 1. Algorithm 3 constructs an EMdF database from a set $\mathcal{G}$ of syntax graphs, and uses Algorithm 2

**Algorithm 0:** *Purpose*: Create empty objects in $Obj_G$ and assign id_ds to each object and to the $id\_d_G$ function and $IDD_G$.
*Input*: A syntax graph $G$ and a starting id_d $d$.
*Output*: A four-tuple consisting of the function $id\_d_G$, the set $IDD\_G$, the set $Obj_G$, the set $NOB_G$ and an ending id_d $d_e$.

1. let $id\_d_G := \emptyset$, and let $Obj_G := \emptyset$
2. For all nodes $v \in (V_{NT} \cup V_T)$ (the ordering does not matter, so long as each node is treated only once):
   (a) let $id\_d_G(v) := d$
   (b) Create an EMdF object $O_d$ being an empty set of monads and let $O_d$.self := $d$
   (c) let $Obj_G := Obj_G \cup \{O_d\}$
   (d) let $IDD_G := IDD_G \cup \{d\}$
   (e) let $NOB_G := NOB_G \cup \langle v, O_d \rangle$
   (f) let $d := d + 1$
3. Return $\langle id\_d_G, IDD_G, Obj_G, NOB_G, d \rangle$.

**Algorithm 1:** *Purpose*: To add monads to all objects corresponding to a non-terminal.
*Input*: A non-terminal $p$, the set $IDD_G$, and the set $Obj_G$.
*Output*: Nothing, but $Obj_G$ is changed. ($Obj_G$ is call-by-value here, so it is changed as a side-effect and not returned.)

1. Let $Ch := \{c | parent(c) = p\}$ (all immediate children of $p$.
2. For all $c \in Ch$:
   (a) If $c \in V_T$: Let $IDD_G(parent(c)) := IDD_G(parent(c)) \cup IDD_G(c)$ (Add terminals' monad-set to parent.)
   (b) Else:
      i. Call ourselves recursively with the parameters $langlec, IDD_G, Obj_G \rangle$.
      ii. Let $IDD_G(parent(c)) := IDD_G(parent(c)) \cup IDD_G(c)$ (Add $c$'s monad-set to parent.)

**Algorithm 2:** *Purpose*: To construct a set of EMdF objects from a syntax graph $G$.
*Input*: A syntax graph $G$, a starting id_d $d$, and a starting monad $m$.
*Output*: A three-tuple consisting of a set of EMdF objects $Obj_G$, an incremented id_d $d_e$ and an ending monad $m_e$.

1. Call Algorithm 0 on $\langle G, d \rangle$ to obtain $\langle id\_d_G, IDD_G, Obj_G, NOB_G, d_e \rangle$.
2. For all terminals $t \in V_T$:
   (a) let $O_t := NOB_G(t) \cup \{m_t\}$ where $m_t = O_G(t) + m - 1$. (Remember that an object is a set of monads, so we are adding a singleton monad set here.)
   (b) Let $O_t.\text{parent} := id\_d_G(parent(t))$ if $t$ is not $R_G$, and NIL if $t$ is $R_G$.
   (c) Assign other features of $O_t$ according to the feature-record $F$ in $t = \langle \nu, F \rangle$.[5]
   (d) if $L_G$ is non-empty, let $O_t.\text{edge} := el(\langle parent(t), t \rangle)$
3. Call Algorithm 1 with the parameters $\langle R_G, IDD_G, Obj_G \rangle$. This assigns monad sets to all objects.
4. For all $v$ in $V_{NT}$:
   (a) Let $O_v := Obj_G(v)$.
   (b) Let $O_v.\text{parent} := id\_d_G(parent(v_v))$ if $v$ is not $R_G$, and NIL if $v$ is $R_G$.
   (c) Assign other features of $O_v$ according to the feature-record $F$ in $v = \langle \nu, F \rangle$.
   (d) if $L_G$ is non-empty, let $O_t.\text{edge} := el(\langle parent(t), t \rangle)$
5. Return $\langle Obj_G, d, m_t \rangle$ where $m_t \equiv O_G(v_t) + m - 1$ where $v_t$ is the rightmost terminal node, i.e., $\exists v_t \in V_T : \forall v_j \in V_T : v_j \neq v_t \Rightarrow O_G(v_t) > O_G(v_j)$

**Algorithm 3:** *Purpose*: To construct a set of EMdF objects from a universe of syntax graphs $\mathcal{G}$.
*Input*: A set of syntax graphs $\mathcal{G}$, a starting id_d $d$, and a starting monad $m$.
*Output*: A two-tuple consisting of an incremented id_d $d_e$ and an ending monad $m_e$.

[5] It is assumed, though the formalisation does not say so, that the feature-records of all $V_T$ in all $G \in \mathcal{G}$ have the same "signature", i.e., have the same set of feature-names that are assigned a value in each $F$ in each $v \in V_T$. A similar assumption is made for the signatures of all feature-records of all $V_{NT}$. This is certainly the case with the TIGERCorpus. Therefore, the object type `Terminal` is well-defined with respect to its features. Similarly for the object type `Nonterminal` used below.

```
Q1. Find sentences that include the word 'saw'.
Q2. Find sentences that do not include the word 'saw'.
Q3. Find noun phrases whose rightmost child is a noun.
Q4. Find verb phrases that contain a verb immediately
    followed by a noun phrase that is immediately
    followed by a prepositional phrase.
Q5. Find the first common ancestor of sequences of a
    noun phrase followed by a verb phrase.
Q6. Not relevant to TIGER Corpus.
Q7. Find a noun phrase dominated by a verb phrase.
    Return the subtree dominated by that noun phrase.
```

Figure 3: The test queries from (Lai & Bird 04), Fig. 1.

```
Q1  #s:[cat="S"] & #l:[word="sehen"] & #s >* #l
Q2* #s:[cat="S"] & #l:[word="sehen"] & #s !>* #l
Q3  #n1:[cat="NP"] & #n2:[pos="NN"]  & (#n1 >@r #n2)
Q4  #vp:[cat="VP"] & #v:[pos="VVFIN"] & #np:[cat="NP"]
    & #pp:[cat="PP"]& #vp >* #v & #vp >* #np
    & #vp >* #pp & #v >@r #vr & #np >@l #npl
    & #vr .1 #npl & #np >@r #npr & #pp >@l #ppl
    & #npr .1 #ppl
Q5* #vp:[cat="VP"] & #np:[cat="NP"] & (#np .* #vp)
    & (#x >* #vp) & (#x >* #np)
Q7* #vp:[cat="VP"] & #np:[cat="NP"] & (#vp >* #np)
```

Figure 4: The test queries of Figure 3 attempted implemented in TIGERSearch. Adapted from (Lai & Bird 04), Fig. 4. The queries marked with a * may not produce the correct results.

1. For all graphs $G$ in $\mathcal{G}$ (if an ordering is intended, i.e., this is not a quotation corpus, then that order should be applied; otherwise, the order is undefined):
   (a) Let $\langle Obj_G, d_e, m_e \rangle$ be the result of calling Algorithm 2 on $\langle G, d, m \rangle$
   (b) Add $Obj_G$ to the EMdF database.
   (c) Let $d := d_e$ and let $m := m_e + 1$
2. Return $\langle d, m \rangle$

## 5 Comparing TIGERSearch and Emdros

Using a variant of this algorithm, we have imported the TIGERCorpus into Emdros. This gives us a common basis for comparing TIGERSearch and Emdros.

The paper (Lai & Bird 04) sets out to specify some requirements on corpus query systems for treebanks that the authors perceive to be essential. Among other criteria, Lai and Bird set up a set of standard queries which are reproduced in Figure 3.

Lai and Bird show how some of the queries can be expressed in TIGERSearch, though they find that not all queries can be expressed. I have attempted to reformulate Lai and Bird's TIGERSearch queries in therms of the TIGERCorpus (see Figure 4).

Query Q2 cannot be formulated correctly in TIGERSearch. This is because what is being negated is the *existence* of the word "sehen", and in TIGERSearch, all nodes are implicitly existentially quantified. Negated existence would require a forall-quantification, as mentioned e.g. in (König & Lezius 03).

Query Q5 is probably not expressible in TIGERSearch, and the given query fails to find the *first* common ancestor only. The currect syntax graphs are returned, but with a number of subgraphs which are not rooted in the first common ancestor.

```
Q1  [Sentence [Word surface="sehen"] ]
Q2  [Sentence NOTEXIST [Word surface="sehen"] ]
Q3  [Phrase tag="NP" [Word last postag="NN"] ]
Q4  [Phrase tag="VP"
       [Word postag="VVFIN"]!
       [Phrase tag="NP"]!
       [Phrase tag="PP"]
    ]
Q5* [Phrase
       [Phrase tag="NP"][Phrase tag="VP"]
    ]
Q7* [Phrase tag="VP"  [Phrase tag="NP"]  ]
```

Figure 5: Emdros queries for Q1-Q7

```
Find all NPs which is a subject, inside of which there
is a relative clause whose parent is the NP.  Inside
the relative clause, there must be a phrase p2, inside
of which there must be a word which is a cardinal.  At
the end of the relative clause must be a finite verb
whose parent is the same as that of p2.  No PP may
intervene between p2 and the verb.

[Phrase as p1 tag="NP" AND edge="SB"
   [Phrase edge="RC" and parent=p1.self
      [Phrase as p2  [Word postag="CARD"] ]
      ..
      NOTEXIST [Phrase tag="PP"]
      ..
      [Word last postag="VVFIN"
            AND parent=p2.parent]
   ]
]
```

Figure 6: Emdros query for Q8

Query Q7 again finds the correct syntax graphs, but fails to retrieve exactly the subtree dominated by the NP. In TIGERSearch, what parts of a matched syntax-graph to retrieve is, in a sense, an irrelevant question, since the main result is the syntax graph itself. Thus the assumption of Lai and Bird that only parts of the matched tree is returned does not hold for TIGERSearch.

Emdros fares slightly better as regards functionality, as can be seen in Figure 5. Query Q2 is correctly expressed in Emdros using the NOTEXIST operator at object-level, which gives Emdros a slight edge over TIGERSearch in this comparison. However, queries Q5 and Q7 fail to give correct results on Emdros as they did on TIGERSearch. Query Q5 fails because, while it returns the correct syntax graphs, it fails to find only the first common ancestor. This is the same situation as with TIGERSearch. As in TIGERSearch, the requirement to find the "first common ancestor" is difficult to express in Emdros. Query Q7 fails because Emdros, like TIGERSearch, was not designed to retrieve subgraphs as part of the query results – subgraphs are to be retrieved later, e.g., for viewing purposes. Like TIGERSearch, Emdros returns the correct syntax graphs, and thus works as designed.

Query Q8 can be seen in Figure 6 along with the Emdros equivalent. It cannot be expressed in TIGERSearch because of the negated existence-operator on the intervening PP.

The queries were all timed, except for Q2 and Q6, which were not expressible in either or both of the corpos query systems. The hardware was an AMD Athlon 64 3200+ with

| Query | Emdros | TIGERSearch |
|---|---|---|
| Q1 | 0.199; 0.202; 0.179 | 0.5; 0.3; 0.3 |
| Q3 | 1.575; 1.584; 1.527 | 10.1; 9.9; 9.9 |
| Q4 | 1.604; 1.585; 1.615 | 9.9; 9.9; 9.9 |
| Q5 | 3.449; 3.319; 3.494 | 5.5; 6.6; 5.5 |
| Q7 | 0.856; 0.932; 0.862 | 1.1; 1.1; 1.1 |
| Q8 | 3.877; 3.934; 4.022 | N/A |

Table 1: Execution times in seconds

1GB of RAM and a 7200RPM harddrive running Linux Fedora Core 4. Three measurements were taken for each query. In the case of TIGERSearch, the timings reported by the program's status bar were used. For Emdros, the standard Unix command `time` was used. The results can be seen in Table 1.

As can be seen, Emdros is faster than TIGERSearch on every query that they can both handle. (Lezius 02a) mentions that the complexity is exponential in the number of query terms. It is very difficult to assess the complexity of an Emdros query, since it depends on a handful of factors such as the number of query items, the number of objects that match each query item, and the number of possible combinations of these.

Probably Emdros is faster in part because it takes a different algorithmic approach to query resolution that TIGERSearch: Instead of using proof-theory, it uses a more linear approach of first retrieving all possible object-"hits", then iteratively walking the query, combining the objects in monad-order as appropriate. Part of the speed increase may stem from its being written in C++ rather than Java, but for queries such as Q3 and Q4, the algorithm rather than the language seems to be the decisive factor, since such a large difference in execution time, relative to the other increases, cannot be accounted for by language differences alone.

## 6 Conclusion

In this paper, we have compared two corpus query systems, namely TIGERSearch on the one hand and our own Emdros on the other. We have briefly introduced the EMdF model underlying Emdros. The EMdF model is based on the MdF model described in (Doedens 94). We have also given a reformalization of the syntax graph formalism underlying TIGERSearch, based on the presentation given in (Lezius 02a). We have then presented an algorithm for converting the syntax graph formalism into the EMdF model.

Having done this, we have compared the two corpus query systems with respect to query functionality and speed. The queries were mostly culled from the literature. It was found that Emdros was able to handle all the test queries that TIGERSearch was able to handle, in addition to a few that TIGERSearch was not able to express. The latter involved the negation of the existence of an object; it is a limitation in the current TIGERSearch that all objects are implicitly existentially quantified, which means that negating the existence of an object is not possible. Negation at the feature-level is, however, possible in both corpus query systems. In both systems, the semantics of feature-level negation is the same as the $\neg$ operator in First Order Logic.

Finally, the test queries which both systems were able to handle were executed on the same machine over the same corpus, namely the TIGERCorpus, and it was found that Emdros was faster than TIGERSearch on every query, and that the algorithm of Emdros seems to scale better than that of TIGERSearch.

## References


(Bird *et al.* 00) Steven Bird, Peter Buneman, and Tan Wang-Chiew. Towards a query language for annotation graphs. In *Proceedings of the Second International Conference on Language Resources and Evaluation*, pages 807–814. European Language Resources Association, Paris, 2000. http://arxiv.org/abs/cs/0007023 Access Online August 2004.

(Bird *et al.* 05) Steven Bird, Yi Chen, Susan Davidson, Haejoong Lee, and Yifeng Zheng. Extending XPath to support linguistic queries. In *Proceedings of Programming Language Technologies for XML (PLANX) Long Beach, California. January 2005.*, pages 35–46, 2005.

(Brants & Hansen 02) Sabine Brants and Silvia Hansen. Developments in the TIGER annotation scheme and their realization in the corpus. In *Proceedings of the Third International Conference on Language Resources and Evaluation (LREC 2002), Las Palmas, Spain, May 2002*, pages 1643–1649, 2002. http://www.ims.uni-stuttgart.de/projekte/TIGER/paper/lrec2002-brants-hansen.pdf Access Online August 2004.

(Cassidy & Bird 00) Steve Cassidy and Steven Bird. Querying databases of annotated speech. In M.E. Orlowska, editor, *Database Technologies: Proceedings of the Eleventh Australasian Database Conference, volume 22 of Australian Computer Science Communications, Canberra, Australia*, pages 12–20. IEEE Computer Society, 2000. http://arxiv.org/abs/cs/0204026, Access Online August 2004.

(Doedens 94) Christianus Franciscus Joannes Doedens. *Text Databases: One Database Model and Several Retrieval Languages*. Number 14 in Language and Computers. Editions Rodopi, Amsterdam and Atlanta, GA., 1994.

(Heid *et al.* 04) U. Heid, H. Voormann, J-T Milde, U. Gut, K. Erk, and S. Pado. Querying both time-aligned and hierarchical corpora with NXT Search. In *Fourth Language Resources and Evaluation Conference, Lisbon, Portugal, May 2004*, 2004.

(König & Lezius 03) Esther König and Wolfgang Lezius. The TIGER language. a description language for syntax graphs. formal definition. Technical report, Institut für Maschinelle Sprachverarbeitung (IMS), University of Stuttgart, Germany, April 22 2003.

(Lai & Bird 04) Catherine Lai and Steven Bird. Querying and updating treebanks: A critical survey and requirements analysis. In *Proceedings of the Australasian Language Technology Workshop, December 2004*, pages 139–146, 2004.

(Lezius 02a) Wolfgang Lezius. *Ein Suchwerkzeug für syntaktisch annotierte Textkorpora*. Unpublished PhD thesis, Institut für Maschinelle Sprachverarbeitung, University of Stuttgart, December 2002. Arbeitspapiere des Instituts für Maschinelle Sprachverarbeitung (AIMS), volume 8, number 4. http://www.ims.uni-stuttgart.de/projekte/corplex/paper/lezius/diss/, Access Online August 2004.

(Lezius 02b) Wolfgang. Lezius. TIGERSearch – ein Suchwerkzeug für Baumbanken. In Stephan Busemann, editor, *Proceedings der 6. Konferenz zur Verarbeitung natürlicher Sprache (KONVENS 2002), Saarbrücken*, pages 107–114, 2002.

(McCawley 82) James D. McCawley. Parentheticals and discontinuous constituent structure. *Linguistic Inquiry*, 13(1):91–106, 1982.

(Mengel & Lezius 00) Andreas Mengel and Wolfgang Lezius. An XML-based encoding format for syntactically analyzed corpora. In *Proceedings of the Second International Conference on Language Resources and Evaluation (LREC 2000), Athens, Greece, 31 May – 2 June 2000*, pages 121–126, 2000.

(Mengel 99) Andreas Mengel. MATE deliverable D3.1 – specification of coding workbench: 3.8 improved query language (Q4M). Technical report, Institut für Maschinelle Sprachverarbeitung, Stuttgart, 18. November, 1999. http://www.ims.uni-stuttgart.de/projekte/mate/q4m/.

(Petersen 04) Ulrik Petersen. Emdros — a text database engine for analyzed or annotated text. In *Proceedings of COLING 2004, held August 23-27 in Geneva*. International Commitee on Computational Linguistics, 2004. http://www.hum.aau.dk/~ulrikp/pdf/petersen-emdros-COLING-2004.pdf, Access online August 2004.

(Rohde 04) Douglas L. T. Rohde. Tgrep2 user manual, version 1.12. Available online http://tedlab.mit.edu/~dr/Tgrep2/tgrep2.pdf. Access Online April 2005, 2004.

(Voormann & Lezius 02) Holger Voormann and Wolfgang Lezius. TIGERin - Grafische Eingabe von Benutzeranfragen für ein Baumbank-Anfragewerkzeug. In Stephan Busemann, editor, *Proceedings der 6. Konferenz zur Verarbeitung natürlicher Sprache (KONVENS 2002)*, pages 231–234, Saarbrücken, 2002.

This page left intentionally blank

# [FSMNLP2005]

# Principles, Implementation Strategies, and Evaluation of a Corpus Query System

Ulrik Petersen

2006

**Published in:** Yli-Jyrä, Anssi, Karttunen, Lauri and Karhumäki, Juhani (Eds.): *Finite-State Methods in Natural Language Processing; 5th International Workshop, FSMNL 2005, Helsinki, Finland, September 1–2, 2005, Revised Papers*, Lecture Notes in Computer Science, Volume 4002/2006, Springer-Verlag, Heidelberg, New York, pp. 215–226

This page left intentionally blank

# Principles, Implementation Strategies, and Evaluation of a Corpus Query System

Ulrik Petersen

University of Aalborg
Department of Communication and Psychology
Kroghstræde 3
DK — 9220 Aalborg East, Denmark
ulrikp@hum.aau.dk
**http://emdros.org/**

**Abstract.** The last decade has seen an increase in the number of available corpus query systems. These systems generally implement a query language as well as a database model. We report on one such corpus query system, and evaluate its query language against a range of queries and criteria quoted from the literature. We show some important principles of the design of the query language, and argue for the strategy of separating what is retrieved by a linguistic query from the data retrieved in order to display or otherwise process the results, stating the needs for generality, simplicity, and modularity as reasons to prefer this strategy.

## 1 Introduction

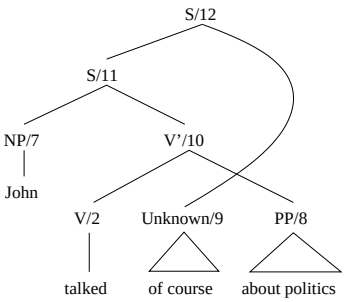
The last decade has seen a growth in the number of available corpus query systems. Newcomers since the mid-1990ies include MATE Q4M [1], the Emu query language [2], the Annotation Graph query language [3], TIGERSearch [4], NXT Search [5], TGrep2 [6], and LPath [7].

Our own corpus query system, Emdros [8,9], has been in development since 1999. It is based on ideas from the PhD thesis by Crist-Jan Doedens [10]. It implements a database model and a query language which are very general in their applicability: Our system can be applied to almost any linguistic theory, almost any linguistic domain (e.g., syntax, phonology, discourse) and almost any method of linguistic tagging. Thus our system can be used as a basis for implementing a variety of linguistic applications. We have implemented a number of linguistic applications such as a generic query tool, a HAL[1] space, and a number of import tools for existing corpus formats. As the system is Open Source, others are free to implement applications for their linguistic problem domains using our system, just as we plan to continue to extend the range of available applications.

The rest of the paper is laid out as follows: First, we briefly describe the EMdF database model underlying Emdros, and give an example of a database

[1] HAL here stands for "Hyperspace Analogue to Language," and is a statistical method based on lexical co-occurrence invented by Dr. Curt Burgess and his colleagues [11].

A. Yli-Jyrä, L. Karttunen, and J. Karhumäki (Eds.): FSMNLP 2005, LNAI 4002, pp. 215–226, 2006




**a.** A tree with a discontiguous clause, adapted from [12, p. 95]

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| **Word** | id: 1<br>surf.: John<br>pos: NProp<br>parent: 7 | id: 2<br>surf.: talked<br>pos: V<br>parent: 10 | id: 3<br>surf.: of<br>pos: P<br>parent: 9 | id: 4<br>surf.: course<br>pos: N<br>parent: 9 | id: 5<br>surf.: about<br>pos: P<br>parent: 8 | id: 6<br>surf.: politics<br>pos: N<br>parent: 8 |
| **Phrase** | id: 7<br>type: NP<br>parent: 11 | | id: 9<br>type: Unknown<br>parent: 12 | | id: 8<br>type: PP<br>parent: 10 | |
| **Phrase** | | id: 10<br>type: V'<br>parent: 11 | | | id: 10<br>type: V'<br>parent: 11 | |
| **Clause** | id: 11<br>type=S<br>parent: 12 | | | | id: 11<br>type=S<br>parent: 12 | |
| **Clause** | id: 12<br>type=S | | | | | |

**b.** A EMdF representation of the tree

**Fig. 1.** Two representation of a tree with a discontiguous clause

expressed in EMdF. Second, we describe the MQL query language of Emdros and its principles. Third, we argue for the strategy of separating the process of retrieving linguistic query results from the process of retrieving linguistic objects based on such results for application-specific purposes. Fourth, we evaluate MQL against a set of standard queries and criteria for corpus query languages culled from the literature. Finally, we conclude the paper.

## 2 The EMdF Database Model

To illustrate how data can be stored in Emdros, consider Fig. 1. It shows an example of a discontiguous clause, taken from [12, p. 95], represented both as a tree and as a database expressed in the EMdF database model.

At the top of Fig. 1.b. are the *monads*. A monad is simply an integer, and the sequence of the monads defines the logical reading order. An object is a (possibly discontiguous) set of monads belonging to an object type (such as "Word", "Phrase", "Clause"), and having a set of associated attribute-values. The object type of an object determines what attributes it has. For example, the "Word" object type in Fig. 1.b has attributes "id", "surface", "pos" (part of speech), and "parent". The id is a database-widely unique number that identifies that object. In the above database, this has been used by the "parent" attribute to point to the immediately dominating node in the tree.

In the EMdF database model, object attributes are strongly typed. The model supports strings, integers, ids, and enumerations as types for attributes, as well as lists of integers, ids, and enumeration labels. Enumerations are simply sets of labels, and have been used for the Word.pos, Phrase.type, and Clause.type

attributes in the figure.[2] Real-number values are under implementation, and will be useful for, e.g., acoustic-signal timelines.

## 3 The MQL Query Language

The MQL query language of Emdros is a descendant of the QL query language described in [10]. Like QL, it is centered around the concept of "blocks", of which there are three kinds: "Object blocks", "gap blocks", and "power blocks".

An "Object block" finds objects in the database (such as phonemes, words, phrases, clauses, paragraphs, etc.) and is enclosed in [square brackets]. For example, the query `[Word surface="saw"]` will find `Word` objects whose `surface` attribute is "saw", whereas the query `[Phrase type = NP and function = Subj]` will find phrases whose phrase type is NP and whose function is Subject. Of course, this presupposes an appropriately tagged database. The attribute-restrictions on the object are arbitrary boolean expressions providing the primitives "AND", "OR", "NOT", and "grouping (parentheses)". A range of comparison-operators are also provided, including equality, inequality, greater-than (or equal to), less than (or equal to), regular expressions (optionally negated), and `IN` a disjoined list of values. For lists, the `HAS` operator looks for a specific value in the list.

A "gap block" finds "gaps" in a certain context, and can be used to look for (or ignore) things like embedded relative clauses, postpositive conjunctions, and other material which is not part of the surrounding element. A gap block is specified as `[gap ...]` when obligatory, and as `[gap? ...]` when optional.

A "power block" is denoted by two dots ("`..`"), and signifies that there can be arbitrary space between the two surrounding blocks. However, this is always confined to be within the limits of any context block.

The power block can optionally have a restriction such as "`.. <= 5`" or "`.. BETWEEN 3 AND 6`" meaning respectively that the "space" can be between zero and five "least units" long, or that it must be between 3 and 6 "least units" long. Precisely what the "least unit" is, is database-dependent, but is usually "Word" or "Phoneme".[3]

The MQL query language implements the important principle of *topographicity* described in [10], meaning that there is an isomorphism between the structure of the query and the structure of the objects found. The principle of topographicity works with respect to two important textual principles, namely embedding and sequence.

As an example of topographicity with respect to embedding, consider the query Q1 in Fig. 3 on page 8. This query finds sentences within which there is at least one word whose surface is "saw". The "`[Word surface="saw"]`" object

[2] The "dot-notation" used here is well known to programmers, and is basically a possessive: "Word.pos" means "the pos attribute **of** the Word object-type".

[3] This is an example of the generality of the EMdF database model, in that it supports many different linguistic paradigms and methods of analysis.

block is *embedded in* the "`[Sentence ...]`" object block. Because of the principle of topographicity, any Word objects found must also be *embedded in* the Sentence objects found.

Similarly, in Query Q5 in Fig. 3, the two inner `[Syntax level=Phrase ...]` object blocks find Syntax objects that immediately follow each other in sequential order, because the object blocks are adjacent. "Being adjacent" here means "not being separated by other blocks" (including a power block). There is a caveat, however. The default behavior is to treat objects in the database as "being adjacent" even if they are separated by a gap in the surrounding context. For examle, in Query Q5, if the surrounding Sentence object has a gap between the NP and the VP[4], then that query will find such a sentence due to the default behavior. If this is not the desired behavior (i.e., gaps are not allowed), one can put the "`!`" (bang) operator in between the object blocks, as in Query Q4 in Fig. 3. This will require the objects found by the object blocks surrounding the bang to be strictly sequential.

An object block can be given the restriction that it must be `first`, `last`, or `first and last` in its surrounding context. An example using the `last` keyword can be seen in Query Q3 in Fig. 3.

The object retrieved by an object block can be given a name with the `AS` keyword. Subsequent object blocks can then refer back to the named object. An example can be seen in Query Q5 in Fig. 3, where the dominating Syntax object is named `AS S1`. The dominated phrase-level Syntax object blocks then refer back to the dominating object by means of the "possessive dot notation" mentioned previously. Obviously, this facility can be used to specify both agreement, (immediate) dominance, and other inter-object relationships.

The NOTEXIST operator operates on an object block to specify that it must not exist in a given context. An example can be seen in Query Q2 in Fig. 3, where the existence of a word with the surface "saw" is negated. That is, the query finds sentences in which the word "saw" does not occur.

Notice that this is different from finding sentences with words whose *surface* is not "saw", as the query `[Sentence [Word surface<>"saw"]]` would find. Relating this to First Order Logic, the NOTEXIST operator is a negated existential quantifier $\neg\exists$ at *object* level, whereas the `<>` operator is a negated equality operator $\neq$ at object *attribute* level. If the NOTEXIST operator is applied to an object block, the object block must be the only block in its context.

The Kleene Star operator also operates on an object block, and has the usual meaning of repeating the object block zero or more times, always restricted to being within the boundaries of any surrounding context block. For example, the query

```
[Sentence
   [Word pos=preposition]
   [Word pos IN (article,noun,adjective,conjunction)]*
]
```

[4] As argued by [12], the sentence "John, of course, talked about politics" is an example of an element with a gap, since "of course" is not part of the surrounding clause.

would find the words of many prepositional phrases, and could be used in a stage of initial syntactic markup of a corpus. The Kleene Star also supports restricting the number of repetitions with an arbitrary set of integers. For example: `[Phrase]*{0,1}` means that the Phrase object may be repeated 0 or 1 times;[5] `[Clause]*{2-4}` means that the Clause object may be repeated 2, 3, or 4 times; and any set of integers can be used, even discontiguous ones, such as `[Phrase]*{0-3,7-9,20-}`. The notation "`20-`" signifies "from 20 to infinity".

An `OR` operator operating on strings of blocks is available. It means that one or both strings may occur in a given context. An example is given in Query Q7 in Fig. 3.

MQL has some shortcomings, some of which will be detailed later. Here we will just mention four shortcomings which we are working to fix, but which time has not allowed us to fix yet. We have worked out an operational semantics for the following four constructs: AND between strings of blocks (meaning that both strings must occur, and that they must overlap);[6] Grouping of strings of blocks; and general Kleene Star on strings of blocks (the current Kleene Star is only applicable to one object block). A fourth operator can easily be derived from the existing OR construct on strings of blocks, namely permutations of objects.

## 4 Retrieval of Results

When querying linguistic data, there are often three distinct kinds of results involved:

1. The "**meat**", or the particular linguistic construction of interest.
2. The **context**, which is not exactly what the user is interested in, but helps delimit, restrict, or facilitate the search in some way. For example, the user may be interested in subject inversion or agentless passives, but both require the context of a sentence. Similarly, the user may be interested in objects expressed by relative pronouns combined with a repeated pronoun in the next clause, which might require the presence of intervening, specified, but otherwise non-interesting material such as a complementizer.[7] In both cases, the user is interested in a specific construction, but a certain context (either surrounding or intervening) needs to be present. The context is thus necessary for the query to return the desired results, but is otherwise not a part of the desired results.
3. The **postprocessing** results which are necessary for purposes which are outside the scope of the search.

To illustrate, consider the query Q2 in Fig. 3. For display purposes, what should be retrieved for this query? The answer depends, among other things,

[5] Notice that this supports optionality in the language; that the phrase object appears 0 or 1 times is equivalent to saying that it is optional.

[6] This is precisely what is needed for querying overlapping structures such as those found in speech data with more than one speaker, where the speaker turns overlap.

[7] E.g., "He gave me a ring, which, I really don't like that it is emerald."

on the linguistic domain under consideration (syntax, phonology, etc.), the linguistic categories stored in the database, the purposes for which the display is made, and the sophistication of the user. For the domain of syntax, trees might be appropriate, which would require retrieval of all nodes dominated by the sentence. For the domain of phonology, intonational phrases, tones, pauses, etc. as well as the phonemes dominated by the sentence would probably have to be retrieved. As to purpose, if the user only needed a concordance, then only the words dominated by the sentence need be retrieved, whereas for purposes requiring a full-fledged tree, more elements would have to be retrieved. The level of sophistication of the user also has a role to play, since an untrained user might balk at trees, whereas keywords in context may be more understandable.

Similarly, for statistical purposes, it is often important to retrieve frequency counts over the entire corpus to compare against the current result set. These frequency counts have nothing to do with the answer to the original query, but instead are only needed after the results have been retrieved. They are, in a very real sense, outside the scope of the query itself: The user is looking for a particular linguistic construction, and the corpus query system should find those constructions. That the post-query purpose of running the query is statistical calculations is outside the scope of the query, and is very application-specific.

Thus what is asked for in a linguistic query is often very different from what needs to be retrieved eventually, given differences in linguistic domain, categories in the database, purpose of display, and sophistication of the user. Therefore, in our view, it is advantageous to split the two operations into separate query language constructs. The subset of the query language supporting linguistic querying would thus be concerned with returning results based on what is asked for in a linguistic query, whereas other subsets of the query language would be concerned with retrieving objects based on those results for display- or other purposes.

This separation, because it is general, supports a multiplicity of linguistic applications, since the concern of linguistic querying (which is common to all linguistic query applications) is separated from the concern of querying for display-, statistical, or other purposes (which are specific to a given application). Moreover, it shifts the burden of what to retrieve based on a given query (other than what is being asked for) off the user's mind, and onto the application, thus making the query language simpler both for the user and for the corpus query system implementor. Finally, this strategy lends itself well to modularization of the query language. That modularization is good, even necessary for correct software implementation has long been a credo of software engineering.[8]

## 5 Evaluation

Lai and Bird [13] formulate some requirements for query languages for treebanks. They do so on the backdrop of a survey of a number of query languages, including

[8] Emdros adheres to this modular principle of separation of concerns between corpus query system and a particular linguistic application on top of it.

Q1. Find sentences that include the word 'saw'.
Q2. Find sentences that do not include the word 'saw'.
Q3. Find noun phrases whose rightmost child is a noun.
Q4. Find verb phrases that contain a verb immediately followed by a noun phrase that is immediately followed by a prepositional phrase.
Q5. Find the first common ancestor of sequences of a noun phrase followed by a verb phrase.
Q6. Find a noun phrase which dominates a word *dark* that is dominated by an intermediate phrase that bears an L-tone.
Q7. Find a noun phrase dominated by a verb phrase. Return the subtree dominated by that noun phrase.

**Fig. 2.** The test queries from [13], Fig. 1

TGrep2, TIGERSearch, the Emu query language, CorpusSearch, NXT Search, and LPath. Lai and Bird set up a number of test queries (see Fig. 2) which are then expressed (or attempted expressed) in each of the surveyed query languages. For all query languages surveyed, it is the case that at least one query cannot be correctly expressed.

The queries are attempted expressed in MQL as in Fig. 3. Query Q1 is trivial, and performs as expected. Query Q2 has already been explained above, and deserves no further comment. The constraint of query Q3 that the noun must be the rightmost child is elegantly expressed by the "`last`" operator on the noun.

In query Q4, the verb, the NP, and the PP are not separated by power blocks ("`..`") and so must immediately follow each other. As mentioned above, gaps are ignored unless the "bang" operator ("`!`") is applied in between the object blocks. Since the query specification explicitly mentions "immediately followed by", we have chosen to insert this operator. Of course, if the default behavior is desired, the bang operator can simply be left out.

Query Q5 fails to yield the correct results in some cases because it presupposes that the "first common ancestor" is the immediate parent, which it need not be. Had the "`parent=S1.id`" terms been left out of the conjunctions, the query would have found all ancestors, not just the immediate ancestor. It is a shortcoming of the current MQL that it is not easy to express other relationships than "general ancestry" and "immediate ancestry".

Query Q5 also presupposes a different database structure than the other queries: In the database behind Q5, all syntax-level objects have been lumped together into one "Syntax" type. This "Syntax" type has a "level" attribute specifying the linguistic level at which the element occurs (Phrase, Clause, etc.), as well as other attributes.

This reorganization of the database is necessary for Q5 because it does not specify what level the dominating node should be at (Phrase, Clause, or Sentence). It is a limitation in Emdros that it can only handle one, explicit type for each object block.

```
Q1. [Sentence
       [Word surface="saw"]
    ]
Q2. [Sentence
       NOTEXIST [Word
             surface="saw"]
    ]
Q3. [Phrase type=NP
       [Word last pos=noun]
    ]
Q4. [Phrase type=VP
       [Word pos=verb]!
       [Phrase type=NP]!
       [Phrase type=PP]
    ]
```

```
Q5.? [Syntax AS S1
        [Syntax level=Phrase AND type=NP
           AND parent=S1.id]
        [Syntax level=Phrase AND type=VP
           AND parent=S1.id]
     ]
Q6.? [Intermediate tone="L-"
        [Phrase type=NP
           [Word surface="dark"]
        ]
     ]
Q7.  [Phrase type=VP
        [Phrase type=NP AS np1
           [Phrase parents HAS np1.id
              [Word]
           ] OR
           [Word parent=np1.id]
        ]
     ]
```

**Fig. 3.** MQL queries for Q1-Q7

For some lingusitic databases, query Q6 would fail to retrieve all possible instances because it assumes that the NP is wholly contained in the Intermediate Phrase. But as [14, p. 176] reports, this is not always true.[9]

Query Q7 not only needs to specify context, but also to retrieve the subtree, presumably for display- or other purposes, since it is not part of what is being asked for (i.e., the "meat"). As mentioned in Sect. 4, Emdros adheres to a different philosophy of implementation. While it is possible in MQL to retrieve exactly whatever the user wants, the algorithm for doing so would in most cases be split between retrieving linguistic results and using other parts of the query language for retrieving objects for display-purposes.

The Q7 query nevertheless fulfills its purpose by retrieving all phrases dominated by the NP together with the words they contain, `OR` all words immediately dominated by the NP. Thus, Emdros is able to fulfill the purpose of the query even though Emdros was not designed for such use.

Lai and Bird go on from their survey to listing a number of requirements on linguistic query languages. The first requirement listed is "accurate specification of the query tree". Lai and Bird give eight subtree-matching queries, all of which can be expressed in MQL (see Fig. 4). Query number 5 would require the employment of the technique used for query Q5 in Fig. 3 of using a single object type for all syntax objects, using an attribute for the syntactic level, then leaving out the level from the query.

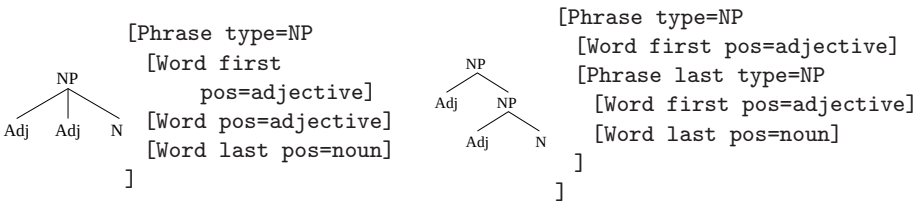
[9] The example given there is an intermediate phrase boundary between adjectives and nouns in Japanese — presumably the adjective and the noun belong in the same NP, yet the intermediate phrase-boundary occurs in the middle of the NP.

| | |
|---|---|
| 1. Immediate dominance: A dominates B, A may dominate other nodes. | `[A AS a1 [B parent=A1.id]]` |
| 2. Positional constraint: A dominates B, and B is the first (last) child of A. | `[A [B first] ]` or: `[A [B last] ]` |
| 3. Positional constraint with respect to a label: A dominates B, and B is the last B child of A. | `[A [B last] ]` |
| 4. Multiple Dominance: A dominates both B and C, but the order of B and C is unspecified. | `[A [B]..[C] OR [C]..[B]]` |
| 5. Sibling precedence: A dominates both B and C, B precedes C; A dominates both B and C, B immediately precedes C, and C is unspecified. | precedes: `[A [B]..[C]]` immediately precedes: `[A [B][C]]` or `[A [B]![C]]`. |
| 6. Complete description: A dominates B and C, in that order, and nothing else. | `[A as a1`<br>`  [B first parent=a1.id]!`<br>`  [B last parent=a1.id]`<br>`]` |
| 7. Multiple copies: A dominates B and B, and the two Bs are different instances. | `[A [B]..[B]]` |
| 8. Negation: A does not dominate node with label B. | `[A NOTEXIST [B] ]` |

**Fig. 4.** Subtree queries in the MQL query language, after Lai and Bird's Fig. 9

Another requirement specified by Lai and Bird is that of reverse navigation, i.e., the need to specify context in any direction. MQL handles this gracefully, in our opinion, by the principle of topographicity with respect to embedding and sequence. Using this principle, any context can be specified in both vertical directions, as well as along the horizontal axis.

Lai and Bird then mention non-tree navigation as a requirement. They give the example of an NP being specified either as "[NP Adj Adj N]" or as "[NP Adj [NP Adj N]]", the latter with a Chomsky-adjoined NP inside the larger NP. MQL handles querying both structures with ease, as seen in Fig. 5. Note that the query in Fig. 5.a. would also find the tree in Fig. 5.b. Thus non-tree navigation is well supported.




**a.** Flat structure **b.** Chomsky-adjoined structure

**Fig. 5.** Queries on NP structure

Furthermore, Lai and Bird mention specification of precedence and immediate precedence as a requirement. MQL handles both with ease because of the principle of topographicity of sequence. General precedence is signified by the power block ("`..`"), whereas immediate precedence is signified by the absence of the power block, optionally with the bang operator ("`!`").

Lai and Bird then discuss closures of various kinds. MQL is closed both under dominance (by means of topographicity of embedding) and under precedence and sibling precedence (by means of topographicity of sequence, as well as the power block and the `AS` keyword, which separately or in combination can be used to specify closures under both relationships). MQL is also closed under atomic queries involving one object (by means of the Kleene Star).[10]

Lai and Bird discuss the need for querying above sentence-level. Since the EMdF database model is abstract and general, the option exists of using ordered forests as mentioned by Lai and Bird. The MQL query language was designed to complement the EMdF model in its generality, and thus querying over ordered forests is well supported using the principle of topographicity of sequence combined with the `AS` construct. Thus the MQL language is not restricted to querying sentence-trees alone, but supports querying above sentence-level.

Another requirement mentioned by Lai and Bird is that of integration of several types of lingusitic data, in particular using intersecting hierarchies and lookup of data from other sources. The EMdF model supports intersecting hierarchies well. MQL, however, because of the principle of topographicity of embedding and the lack of an AND construct between strings of blocks, does not currently support querying of intersecting hierarchies very well, as illustrated by the failure of Query Q6 in Fig. 3 to be correct. Thus Emdros currently falls short on this account, though an AND construct is planned.

There is also currently a lack of support for querying data from other sources. However, this can be implemented by the application using Emdros, provided the data from other sources can be known before query-time and can thus be written into the query. This would, of course, presuppose that the application does some kind of rewriting of the query made by the user.

The final requirement mentioned by [13] is the need to query non-tree structure. For example, the TIGER Corpus [15] includes secondary, crossing edges, and the Penn Treebank includes edges for WH-movement and topicalization [16]. MQL handles querying these constructions by means of the `AS` keyword and referencing the ID of the thus named object, as in Query Q5 in Fig. 3.

## 6 Conclusion and Further Work

We have presented the EMdF database model and the MQL query language of our corpus query system, Emdros. We have shown how the data to be retrieved for display-, statistical, or other purposes can often be different from what is asked for in a linguistic query, differentiating between "meat", "context", and

[10] Once we have implemented the general Kleene Star on strings of blocks, MQL will be closed under atomic queries involving more than one block.

"postprocessing results". On the basis of this distinction, we have argued for the strategy of separating the process of lingusitic querying from the process of retrieval of data for display- or other purposes. This implementation strategy of separation of concerns gives rise to the benefits of generality of the language (and thus its applicability to a wide variety of linguistic applications), simplicity of the language (and thus ease of use for the user), and modularity (and thus ease of implementation, maintainability, and attainment of the goal of correctness for the system implementor). Finally, we have evaluated MQL against the queries and requirements of [13], and have shown MQL to be able to express most of the queries, and to meet most of the requirements that [13] puts forth.

However, Emdros falls short on a number of grounds. First, although its database model is able to handle intersecting hierarchies, its query language does not currently handle querying these intersecting hierarchies very well. This can be fixed by the inclusion of an AND operator between strings of object blocks. Second, a general Kleene Star is lacking that can operate on groups of (optionally embedded) objects. Third, the query language currently only supports one, explicit object type for any given object block. This can be fixed, e.g., by introducing true object orientation with inheritance between object types. Fourth, the system currently does not support real numbers as values of attributes of objects, which would be very useful for phonological databases. Fifth, it is currently not easy to express other, more specific dominance relationships than immediate dominance and general dominance. As has been described above, the removal of most of these shortcomings is planned.

Thus Emdros is able to meet most of the requirements being placed on today's linguistic query systems. We have not here fully explored its applicability to phonological or discourse-level databases, since [13] concentrated on treebanks, but that is a topic for a future paper.

## References

1. Mengel, A.: MATE deliverable D3.1 – specification of coding workbench: 3.8 improved query language (Q4M). Technical report, Institut für Maschinelle Sprachverarbeitung, Stuttgart, 18. November (1999)
2. Cassidy, S., Bird, S.: Querying databases of annotated speech. In Orlowska, M., ed.: Database Technologies: Proceedings of the Eleventh Australasian Database Conference, volume 22 of Australian Computer Science Communications, Canberra, Australia. IEEE Computer Society (2000) 12–20
3. Bird, S., Buneman, P., Tan, W.C.: Towards a query language for annotation graphs. In: Proceedings of the Second International Conference on Language Resources and Evaluation. European Language Resources Association, Paris (2000) 807–814
4. Lezius, W.: TIGERSearch – ein Suchwerkzeug für Baumbanken. In Busemann, S., ed.: Proceedings der 6. Konferenz zur Verarbeitung natürlicher Sprache (KONVENS 2002), Saarbrücken. (2002) 107–114
5. Heid, U., Voormann, H., Milde, J.T., Gut, U., Erk, K., Pado, S.: Querying both time-aligned and hierarchical corpora with NXT Search. In: Fourth Language Resources and Evaluation Conference, Lisbon, Portugal, May 2004. (2004)

6. Rohde, D.L.T.: TGrep2 user manual, version 1.12. Available for download online http://tedlab.mit.edu/~dr/Tgrep2/tgrep2.pdf. Access Online April 2005 (2004)
7. Bird, S., Chen, Y., Davidson, S., Lee, H., Zheng, Y.: Extending XPath to support linguistic queries. In: Proceedings of Programming Language Technologies for XML (PLANX) Long Beach, California. January 2005. (2005) 35–46
8. Petersen, U.: Emdros — A text database engine for analyzed or annotated text. In: Proceedings of COLING 2004, 20th International Conference on Computational Linguistics, August 23rd to 27th, 2004, Geneva, International Commitee on Computational Linguistics (2004) 1190–1193 http://emdros.org/petersen-emdros-COLING-2004.pdf.
9. Petersen, U.: Evaluating corpus query systems on functionality and speed: Tigersearch and emdros. In Angelova, G., Bontcheva, K., Mitkov, R., Nicolov, N., Nikolov, N., eds.: International Conference Recent Advances in Natural Language Processing 2005, Proceedings, Borovets, Bulgaria, 21-23 September 2005, Shoumen, Bulgaria, INCOMA Ltd. (2005) 387–391 ISBN 954-91743-3-6.
10. Doedens, C.J.: Text Databases: One Database Model and Several Retrieval Languages. Number 14 in Language and Computers. Editions Rodopi, Amsterdam and Atlanta, GA. (1994)
11. Lund, K., Burgess, C.: Producing high-dimensional semantic spaces from lexical co-occurrence. Behavior Research Methods, Instruments and Computers **28** (1996) 203–208
12. McCawley, J.D.: Parentheticals and discontinuous constituent structure. Linguistic Inquiry **13** (1982) 91–106
13. Lai, C., Bird, S.: Querying and updating treebanks: A critical survey and requirements analysis. In: Proceedings of the Australasian Language Technology Workshop, December 2004. (2004) 139–146
14. Beckman, M.E., Pierrehumbert, J.B.: Japanese prosodic phrasing and intonation synthesis. In: Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics. ACL (1986) 173–180
15. Brants, S., Hansen, S.: Developments in the TIGER annotation scheme and their realization in the corpus I. In: Proceedings of the Third International Conference on Language Resources and Evaluation (LREC 2002), Las Palmas, Spain, May 2002. (2002) 1643–1649
16. Taylor, A., Marcus, M., Santorini, B.: The Penn treebank: An overview. In Abeillé, A., ed.: Treebanks — Building and Using Parsed Corpora. Volume 20 of Text, Speech and Language Technology. Kluwer Academic Publishers, Dordrecht, Boston, London (2003) 5–22

# [LREC2006]

# Querying both Parallel and Treebank Corpora: Evaluation of a Corpus Query System

Ulrik Petersen

2006

**Published in:** Proceedings of International Language Resources and Evaluation Conference, LREC 2006

This page left intentionally blank

# Querying Both Parallel And Treebank Corpora: Evaluation Of A Corpus Query System

**Ulrik Petersen**

Department of Communication and Psychology
University of Aalborg, Kroghstræde 3
9220 Aalborg East, Denmark
ulrikp@hum.aau.dk

**Abstract**

The last decade has seen a large increase in the number of available corpus query systems. Some of these are optimized for a particular kind of linguistic annotation (e.g., time-aligned, treebank, word-oriented, etc.). In this paper, we report on our own corpus query system, called Emdros. Emdros is very generic, and can be applied to almost any kind of linguistic annotation using almost any linguistic theory. We describe Emdros and its query language, showing some of the benefits that linguists can derive from using Emdros for their corpora. We then describe the underlying database model of Emdros, and show how two corpora can be imported into the system. One of the two is a parallel corpus of Hungarian and English (the Hunglish corpus), while the other is a treebank of German (the TIGER Corpus). In order to evaluate the performance of Emdros, we then run some performance tests. It is shown that Emdros has extremely good performance on "small" corpora (less than 1 million words), and that it scales well to corpora of many millions of words.

## 1. Introduction

The last decade has seen a large increase in the number of available corpus query systems. Systems such as TGrep2 (Rohde, 2005), Emu (Cassidy and Harrington, 2001), TIGERSearch (Lezius, 2002a; Lezius, 2002b), NXT Search (Heid et al., 2004), Viqtoria, Xaira, Emdros, and others have been implemented during this time. Often, these corpus query systems will specialize in one or two kinds of corpora, such as time-aligned, treebank, parallel, or word-oriented corpora; others are optimized for a particular size of corpus.

The value of a corpus query system lies in its two-fold ability to store and retrieve corpora — both the text and its linguistic annotation. The query capability is important for researchers in both theoretical and computational linguistics. Theoretical linguists might be enabled to answer theoretical questions and back up their claims with actual usage rather than introspective intuitions about language. Computational linguists are given a repository in which to store their data in the short- or long-term, and are also being given query capabilities which might help them, e.g., test the accuracy of a parser or pull up a list of all words with specific properties.

In this paper, we present our own Corpus Query System, called Emdros[1]. Emdros is very generic, and can be applied to almost any kind of linguistic annotation from almost any linguistic theory. We show that when applied to parallel corpora, many millions of words are easily supported with quick execution times. When applied to treebanks, Emdros performs extremely well for "small" corpora (less than 1 million words; see (Petersen, 2005)), but performance is also good for "large" corpora (many millions of words).

The rest of the paper is laid out as follows. First, we briefly describe Emdros and the benefits a researcher might reap from using the software. Second, we describe the EMdF database model underlying Emdros. This sets the stage, then, for describing how the EMdF model has been applied to two corpora, namely the Hunglish corpus (Varga et al., 2005), and the TIGER Corpus (Brants and Hansen, 2002; Brants et al., 1999). We then describe some experiments used to evaluate the speed of Emdros based on these two corpora, followed by the results of the experiments and an evaluation of the results. Finally, we conclude the paper.

## 2. Benefits of Emdros

In this section, we briefly describe some of the characteristics and features of Emdros, as well as describing some of the query language of Emdros.

Emdros has a four-layer architecture (see Fig. 1): At the bottom, a relational DBMS lays the foundation, with backends for PostgreSQL, MySQL, and SQLite currently implemented. On top of that, a layer implementing the EMdF database model is found. The EMdF model is a particular model of text which lends itself extremely well to linguistic annotation, and is described in more detail in the next section. On top of the EMdF layer, a layer implementing the MQL query language is found. MQL is a "full access language", featuring statements for create/retrieve/update/delete on the full range of the data types made available in the EMdF model. The EMdF model and the MQL query language are descendants of the MdF model and the QL query language described in (Doedens, 1994).

On top of the MQL layer, any number of linguistic applications can be built. For example, the standard Emdros distribution comes with: a) a generic graphical query application; b) importers from Penn Treebank and NeGRA format (with more importers to come); c) exporters to Annotation Graph XML format and MQL; d) a console application for accessing the features of MQL from the command-line; e) a graphical "chunking-tool" for exemplifying how to use Emdros; f) and a number of toy applications showing linguistic use, among other tools.

Emdros has been deployed successfully in a number of research projects, e.g., at the Free University of Amster-

[1] See `http://emdros.org/` and (Petersen, 2004; Petersen, 2005; Petersen, 2006 to appear)

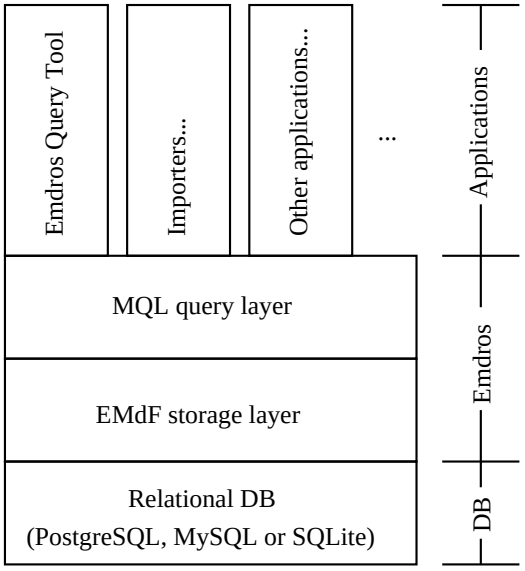



Figure 1: Emdros architecture

dam (for a database of Hebrew), and at the *Institut de Recherche en Informatique de Toulouse* (for a concordancer-application), among others. Two companies have licensed Emdros for inclusion in their software products, one of which is Logos Research Systems, using Emdros to query a number of Biblical Greek and Hebrew databases.

Emdros runs on Windows, Mac OS X, Linux, FreeBSD, NetBSD, Sun Solaris, and other operating systems, and has been implemented in a portable subset of C++. Language bindings are available for Java, Perl, Python, Ruby, and PHP. It is being made available under the GNU General Public License, but other licensing can be negotiated with the author.

The retrieval-capabilities of the MQL query language are particularly powerful, and can be very useful to linguists. Examples are given in Fig. 2 and Fig. 3.

MQL is centered around "blocks" enclosed in "[square brackets]". There are three kinds of blocks: *Object blocks* (which match objects in the database); *Gap blocks* (which match "gaps" in the database, e.g., embedded relative clauses); and *power blocks* (which match "arbitrary stretches of monads"). The examples given in this paper only use object blocks; for more examples, please see the website.

The overruling principle of MQL is: "The structure of the query mirrors the structure of the objects found", i.e., there is an isomorphism between the structure of the query and the inter-object structure of the objects found. This is with respect to two key principles of text, both of which are very familiar to linguists, namely "sequence" and "embedding". For example, query Q1 in Fig. 3 simply finds "Root" objects (i.e., "Sentence" objects) *embedded within which* there is a "Token" object whose attribute "surface" is equal to "sehen".

Similarly, query Q4 finds "Nonterminal" objects of type "NP" embedded within which we find: a) first a token of type "VVFIN", then b) a Nonterminal of type "NP", and then c) a Nonterminal of type "PP". The fact that these three are placed after each other implies (because of the overruling principle of MQL) that the objects found must occur in that sequence.

Query Q9 shows how to use references between objects — the surrounding NP Nonterminal is labelled "`AS p1`", which effectively gives the object a name which can be used further down in the query. This is used in query Q9 to ensure that the NP is the immediate parent of the objects found embedded inside of it (the automatically generated "self" attribute of any object gives the ID of that object).

Query Q3 and Q9 show the "first" and "last" keywords — meaning that the object that bears such a designation must be either "first" or "last" in its surrounding context.

Queries Q2 and Q8 show the "NOTEXIST" operator. As it is currently implemented, the NOTEXIST operator means that the following object must not exist in the surrounding context from the point at which it is found on to the end of the surrounding context. For example, in query Q8, once the Token of type "NN" has been found, there must not exist a Token of type "ADJA" or type "ADJD" after the "NN" token, up to the end of the surrounding NP. Note that this is *existential* negation at *object-level* ($\neg\exists$) — not negation of *equality* at the object *attribute* level ($\neq$).

Various attribute-comparison operators are available, including "`=`", "`<>`" (inequality)", "`<`", "`>`", "`<=`", "`>=`", `IN` a list, regular expressions "`~`", and negated regular expressions "`!~`", among others. Queries H1-H4 in Fig. 2 illustrate the regular expression operator "`~`" for simple queries. These examples, however do not show the full range of capabilities in MQL. For example, Kleene Star is not shown, nor is the `OR` operator between strings of objects shown. The latter supports searches for permutations of positions of objects using one query rather than several queries. MQL is able to handle queries of any complexity, and the queries shown here are all on the low end of the scale of complexity which MQL can handle. For more information, consult either the documentation on the website[2] or (Petersen, 2004; Petersen, 2005; Petersen, 2006 to appear).

## 3. The EMdF model

The EMdF (Extended MdF) model derives from the MdF (Monads dot Features) model described in (Doedens, 1994). There are four basic concepts in the EMdF model, which all derive from Doedens' work: Monad, Object, Object Type, and Feature. A monad is simply an integer — no more, no less. An object *is* a set of monads, and *belongs to* an Object Type. The Object Type groups objects with similar characteristics, e.g., Words, Phrases, Clauses, Sentences, Documents, etc. The model is generic in that it does not dictate *what* Object Types to instantiate in any database schema. Thus the database designer is free to design their linguistic database in ways that fit the particular linguistic problems at hand. The Object Type of an Object determines what *features* (or attributes) it has. Thus a database designer might choose to let the "Word" object type have features called "surface", "part_of_speech", "lemma", "gloss", etc. Or the database designer might choose to let the "Phrase" object type have features called "phrase_type", "function", "parent", etc.

[2] `http://emdros.org`

```
H1:  [Sentence english ~ " is "]
H2:  [Sentence english ~ " is " AND
        english ~ " was "
     ]
H3:  [Sentence english ~ " is " AND
        english ~ " necessary "
     ]
H4:  [Sentence english ~ " [Ii]s "
        AND english ~ " [Ww]as "
     ]
```

Figure 2: Queries on the Hunglish corpus

The backbone of the database is the string of monads (i.e., the integers: 1,2,3,... etc.). As mentioned, an object *is* a set of monads. The set is completely arbitrary, in that it need not be contiguous, but can have arbitrarily many "gaps". This supports things like embedded clauses with a surrounding clause of which it is not a part, discontiguous phrases, or other discontiguous elements.

Thus far, we have described the MdF model. The Extended MdF (EMdF) model that Emdros implements adds some additional concepts.

First, each object has an *id_d*, which is simply a database-wide unique integer that uniquely identifies the object.

Second, the datatypes that a feature can take on includes: strings, integers, id_ds, and enumerations (sets of labels), along with lists of integers, lists of id_ds, and lists of enumerations.

Third, an object type can be declared to be one of three *range-classes*. The range-classes are: a) "WITH SINGLE MONAD OBJECTS", b) "WITH SINGLE RANGE OBJECTS", and c) "WITH MULTIPLE RANGE OBJECTS". The "SINGLE MONAD" range-class is for object types that will only ever have objects that consist of a single monad, e.g., Word-object types. The "SINGLE RANGE" range-class is for object types that will only ever have contiguous objects, never objects with gaps. Finally, the "MULTIPLE RANGE" range-class is for object types that will have objects that *may* (but need not) have gaps in them. These range-classes are used for optimizations in the way the data is stored, and can lead to large performance gains when used properly.

In the next section, we show how we have applied the EMdF model to the design of two Emdros databases for two corpora.

## 4. Application

For the purposes of this evaluation, two corpora have been imported into Emdros. One is the Hunglish corpus (Varga et al., 2005), while the other is the TIGER Corpus (Brants and Hansen, 2002; Brants et al., 1999).

The TIGER Corpus has been imported from its instantiation in the Penn Treebank format, rather than its native NeGRA format. That is, the secondary edges have been left out, leaving only "normal" tree edges and labels. Coreference labels have, however, been imported.

Each root tree gets imported into an object of type "Root". This has been declared "WITH SINGLE RANGE OBJECTS".

```
Q1:  [Root
         [Token surface="sehen"]
     ]
Q2:  [Root
         NOTEXIST [Token surface="sehen"]
     ]
Q3:  [Nonterminal mytype="NP"
         [Token last mytype="NP"]
     ]
Q4:  [Nonterminal mytype="VP"
         [Token mytype="VVFIN"]!
         [Nonterminal mytype="NP"]!
         [Nonterminal mytype="PP"]
     ]
Q8:  [Nonterminal mytype="NP"
         [Token mytype="NN"]
         NOTEXIST [Token mytype="ADJA"
              OR mytype="ADJD"]
     ]
Q9:  [Nonterminal AS p1 mytype="NP"
         [Token FIRST mytype="ART"
              AND parent = p1.self
         ]
         [Token mytype="ADJA"
              AND parent = p1.self
         ]
         [Token LAST mytype="NN"
              AND parent = p1.self
         ]
     ]
```

Figure 3: Queries on the TIGER Corpus

Likewise, each Nonterminal (whether it be an S or a Phrase) gets imported into an object of type "Nonterminal". This object type has the features "mytype" (for the edge label, such as "NP"), function (for the function, such as "SUBJ"), and "coref" (a list of id_ds pointing to coreferent nodes), as well as a "parent" feature (pointing to the id_d of the parent).

Finally, each terminal (whether it be a word or punctuation) is imported as an object of type "Token". This object type has the same features as the "Nonterminal" object type, with the addition of a "surface" feature of type STRING, showing the surface text of the token. The "Token" object type has been declared "WITH SINGLE MONAD OBJECTS".

The Hunglish corpus has been imported in a very simple manner: Each sentence has been imported as a single object, belonging to the object type "Sentence". This object type has only two features: "English" and "Hungarian", both of which are of type "STRING". For each sentence, punctuation has been stripped, and each word surrounded by a space on both sides. This makes for easy searching using regular expressions. Since there is no syntactic markup for the Hunglish corpus, having only sentence-boundaries, it seemed natural to gather all words into a single string rather than splitting them out into separate objects. As it turns out, this leads to a huge increase in performance, simply because there are fewer rows to query in the backend. Each object occupies exactly one monad in the monad-stream, and so the object type has been declared "WITH

| 1000 Tokens | H1 | H2 | H3 | H4 |
|---|---|---|---|---|
| 16531 | 6.53 | 7.625 | 7.74 | 6.2 |
| 33063 | 13.345 | 16.095 | 16.085 | 11.91 |
| 49595 | 21.08 | 23.705 | 23.58 | 18.565 |
| 66127 | 26.99 | 30.49 | 32.375 | 24.785 |
| 82659 | 33.98 | 42.485 | 40.245 | 31.275 |

Table 1: Average times in seconds for SQLite on the Hunglish corpus

| 1000 Tokens | Q1 | Q2 | Q3 | Q4 | Q8 | Q9 |
|---|---|---|---|---|---|---|
| 712 | 0.47 | 0.80 | 1.91 | 1.17 | 3.37 | 2.40 |
| 2849 | 1.80 | 3.00 | 7.54 | 4.39 | 12.55 | 9.03 |
| 8547 | 5.37 | 9.16 | 22.97 | 12.75 | 36.56 | 27.64 |
| 17095 | 11.09 | 17.56 | 45.52 | 26.77 | 77.66 | 54.48 |
| 25643 | 16.97 | 26.83 | 72.64 | 43.68 | 117.72 | 84.76 |
| 34191 | 25.62 | 36.52 | 105.63 | 71.35 | 175.80 | 129.78 |

Table 2: Average times in seconds for SQLite on the TIGER corpus

SINGLE MONAD OBJECTS".

## 5. Experiments

In order to test the scalability of Emdros, both corpora have been concatenated a number of times: The Hunglish corpus has been concatenated so as to yield the corpus 1-5 times (i.e., with 0-4 concatenation operations), while the TIGER Corpus has been concatenated so as to yield the corpus 4, 12, 24, 36, and 48 times. There are 712,332 tokens and 337,881 syntactic objects on top in the TIGER corpus, yielding 34.19 million tokens and 16.22 million syntactic objects in the case where the corpus has been concatenated 47 times. For the Hunglish corpus, there are 852,334 sentences in two languages totalling 16,531,968 tokens. For the case where the corpus has been concatenated 4 times, this yields 81.09 million tokens and 4.26 million sentences. A number of queries have been run on either corpus. They are shown in Fig. 2 for the Hunglish corpus and in Fig. 3 for the TIGER Corpus. For the TIGER Corpus, queries Q1-Q4 have been adapted from (Lai and Bird, 2004).

The performance of Emdros has been tested by running all queries in sequence, twice in a row each (i.e., Q1, Q1, Q2, Q2, etc.). The queries have been run twice so as to guard against bias from other system processes. This has been done on a Linux workstation running Fedora Core 4 with 3GB of RAM, a 7200 RPM ATA-100 harddrive, and an AMD Athlon64 3200+ processor. The queries have been run against each of the concatenated databases.

For each database, a number of queries have been run against the database before speed measurements have taken place, in order to prime any file system caches and thus get uniform results.[3] In a production environment, the databases would not be queried "cold", but would be at least partially cached in memory, thus this step ensures production-like conditions.

## 6. Results

The results of the experiments can be seen in Figures 4–5. Fig. 4 shows the time for queries H1-H4 added together on the Hunglish corpus. Fig. 5 shows the same for the queries on the TIGER Corpus. Figures 6, 7, and 8 show the times of the individual queries on the TIGER Corpus for SQLite, MySQL, and PostgreSQL respectively. The average times for each query can be seen for SQLite on the Hunglish corpus in Table 1, and for SQLite on the TIGER Corpus in Table 2. The distribution of times is similar for PostgreSQL and MySQL, and so these times are not shown as tables, only as graphs.

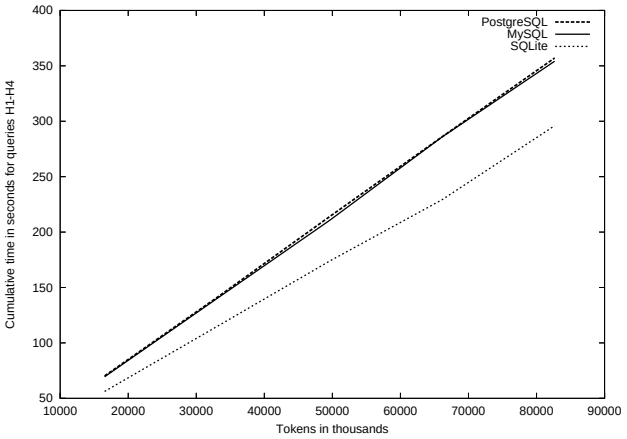



Figure 4: Times for all queries added together on the Hunglish corpus

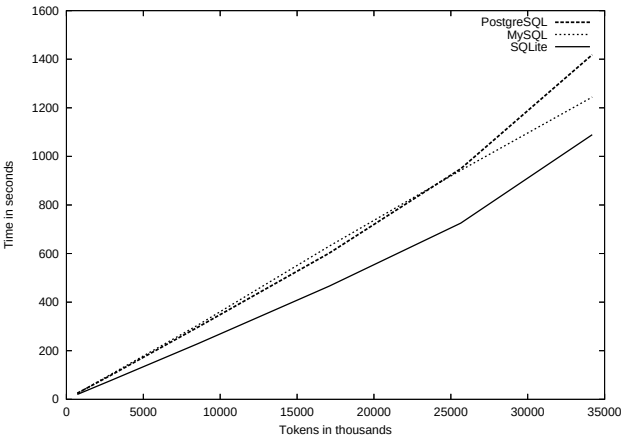



Figure 5: Times for all queries added together on the TIGER corpus

## 7. Evaluation

As can be seen from Table 2, Emdros performs extremely well on the single instance of the TIGER corpus ($712 \times 10^3$ words), running the most complex query, Q8, in less than 3.5 seconds. This is typical of Emdros' performance on "small" corpora of less than a million words. For further details, please see (Petersen, 2005).

As can be seen from a comparison of Table 1 and Table 2, the query times for the Hunglish corpus are significantly lower per token queried than for the TIGER corpus. This is because of the differences in the way the EMdF databases for the two corpora have been designed: The Hunglish corpus has been gathered into far fewer RDBMS rows than the

[3] The queries used for "priming" were: H1 for the Hunglish corpus; and Q2, Q4, and Q8 for the TIGER Corpus.

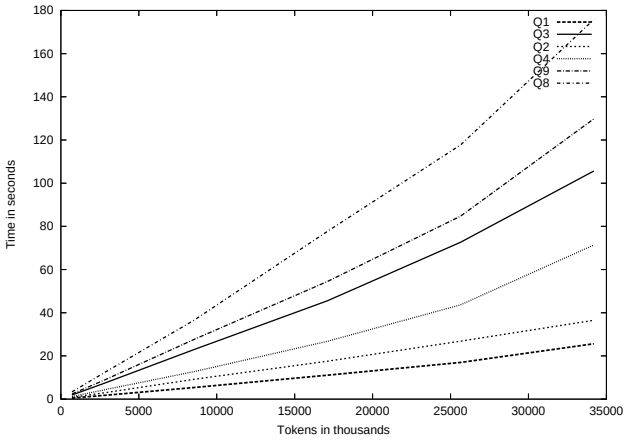



Figure 6: TIGER SQLite execution times

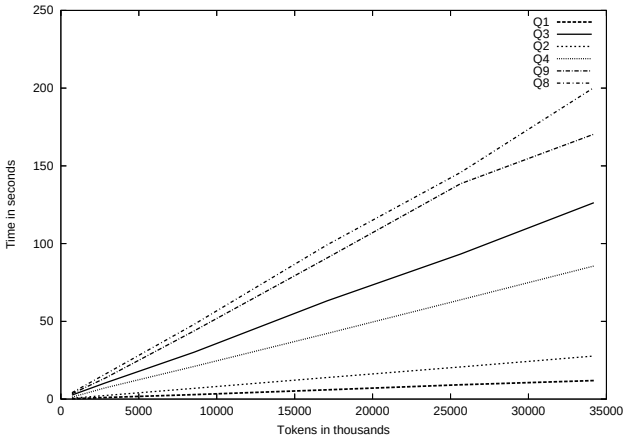



Figure 7: TIGER MySQL execution times

TIGER Corpus, in that each sentence becomes one row as is the case for the Hunglish corpus, rather than one token becoming one row as is the case for the TIGER corpus. In addition, there is no linguistic information associated with each word in the Hunglish corpus. These two factors mean that the storage overhead per token is significantly less for the Hunglish corpus. This is the reason for the dramatical difference in query times between the two corpora.

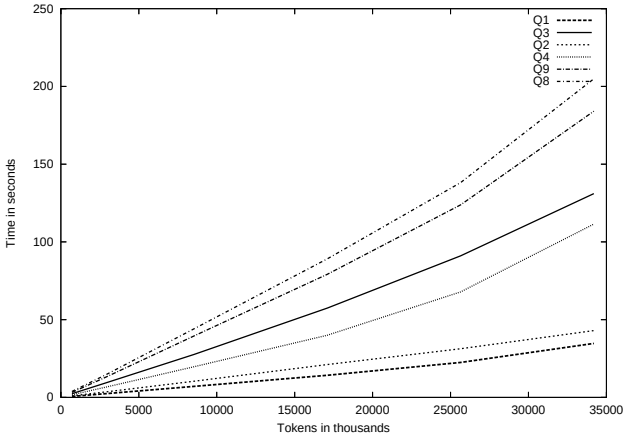



Figure 8: TIGER PostgreSQL execution times

It will be noted, however, that the TIGER corpus, because it is a treebank, supports significantly more advanced queries than the Hunglish corpus. Also, query Q1 on the TIGER corpus is only marginally more advanced than query H1 on the Hunglish corpus, in that both queries query for the existence of a single word, the only difference being that query Q1 also retrieves the structurally enclosing Root (i.e., Sentence) object. Moreover, if we extrapolate the SQLite query time for query Q1 linearly (see Table 2) up to the size of the biggest concatenation of the Hunglish corpus (82 million), we get an execution time of $25.62 \times \frac{82659}{34191} = 61.93$, which is only roughly twice the execution time of H1 (33.98).[4] Thus the added complexity of the TIGER corpus only lowers performance by a factor of roughly 2, while adding many complex query capabilities, as exemplified by query Q9.

As can be seen from Fig. 4, which shows the times of all queries added together for the Hunglish corpus, performance on the Hunglish corpus is very linear in the number of tokens queried.

The same is almost true for the TIGER corpus, as can be seen from Fig. 5, which shows the times of all queries added together for the TIGER corpus. However, here the curves suffer a bend after 25 million tokens — at least on PostgreSQL and SQLite, while MySQL stays linear even up to 34 million words. It is our estimation that tuning PostgreSQL's memory usage, and increasing the amount of RAM available to SQLite, would change this picture back to linear for these two databases, even beyond 25 million tokens queried.

As can be seen from Fig. 6, which shows the time taken for individual queries on SQLite, it is the case that the curve suffers a bend on all queries after 25 million tokens queried. The same is true for PostgreSQL, as can be seen from Fig. 8. On MySQL, however, all queries are linear even beyond 25 million, except for query Q9, which strangely shows better-than-linear performance after 25 million words, as can be seen in Fig. 7. We have no explanation for this phenomenon at this point.

It is curious that query Q8 is uniformly slower than query Q9 across the three backend databases, even though query Q8 is less complex than query Q9 in the number of query terms. This is probably because query Q8 finds more than 12.58 times the number of "hits" than query Q9[5], and so has to do more memory-house-keeping, as well as dumping more results afterwards.

## 8. Conclusion and further work

Corpus query systems are of great value to the Language Resources community. In this paper, we have presented our own corpus query system, called Emdros, and have described its architecture, its MQL query language, and its underlying EMdF database model. We have then shown how one can apply the EMdF database model to two kinds

[4] As Fig. 6 shows, we are not completely justified in extrapolating linearly, since query Q1 (as well as the other queries) show a small but significant non-linear bend in the curve after 25 million words queried. However, this bend is very small for query Q1.

[5] 3,843,312 for Q8 vs. 305,472 for Q9 on the 34 million-word corpus.

of corpora, one being a parallel corpus (the Hunglish corpus) and the other being a treebank (the TIGER corpus).

We have then described some experiments on the two corpora, in which we have measured the execution time of Emdros against the two corpora on a number of queries. The corpora have been concatenated a number of times so as to get more data to query. This has resulted in databases of different sizes, up to 82 million words for the Hunglish corpus and up to 34 million tokens for the TIGER corpus. The execution times have been plotted as graphs, which have been shown, and selected times have been shown as tables.

We have then discussed and evaluated the results. It has been shown that execution time is linear in the number of tokens queried for the Hunglish corpus, and nearly linear for the TIGER Corpus. It has also been shown that execution times are extremely good for "small" corpora of less than a million words, while execution time remains good for "large" corpora of many millions of words.

We plan to extend Emdros in variuos ways. For example: Adding importers for more corpus formats; Adding an AND operator between strings of object blocks; Adding automatically generated permutations of blocks; Adding support for Kleene Star on groups of blocks rather than single blocks; Extending the underlying EMdF model to scale even better; Adding ngram support directly into the underlying EMdF model; Adding lists of strings as a feature-type; Adding caching features which would support web-based applications better; and adding a graphical management tool in addition to the existing graphical query tool.

The good execution times, coupled with a query language that is easy to read, easy to learn, and easy to understand while supporting very complex queries, makes Emdros a good choice as a tool for researchers working with linguistic corpora.

## 9. References


Galia Angelova, Kalina Bontcheva, Ruslan Mitkov, Nicolas Nicolov, and Nikolai Nikolov, editors. 2005. *International Conference Recent Advances in Natural Language Processing 2005, Proceedings, Borovets, Bulgaria, 21-23 September 2005*, Shoumen, Bulgaria. INCOMA Ltd. ISBN 954-91743-3-6.

Sabine Brants and Silvia Hansen. 2002. Developments in the TIGER annotation scheme and their realization in the corpus I. In *Proceedings of the Third International Conference on Language Resources and Evaluation (LREC 2002), Las Palmas, Spain, May 2002*, pages 1643–1649. ELRA, European Language Resources Association.

Thorsten Brants, Wojciech Skut, and Hans Uszkoreit. 1999. Syntactic annotation of a German newspaper corpus. In *Proceedings of the ATALA Treebank Workshop*, pages 69–76, Paris, France.

Steve Cassidy and Jonathan Harrington. 2001. Multi-level annotation in the Emu speech database management system. *Speech Communication*, 33(1,2):61–77.

Crist-Jan Doedens. 1994. *Text Databases: One Database Model and Several Retrieval Languages*. Number 14 in Language and Computers. Editions Rodopi Amsterdam, Amsterdam and Atlanta, GA. ISBN 90-5183-729-1.

U. Heid, H. Voormann, J-T Milde, U. Gut, K. Erk, and S. Pado. 2004. Querying both time-aligned and hierarchical corpora with NXT Search. In *Fourth Language Resources and Evaluation Conference, Lisbon, Portugal, May 2004*.

Catherine Lai and Steven Bird. 2004. Querying and updating treebanks: A critical survey and requirements analysis. In *Proceedings of the Australasian Language Technology Workshop, December 2004*, pages 139–146.

Wolfgang Lezius. 2002a. *Ein Suchwerkzeug für syntaktisch annotierte Textkorpora*. Ph.D. thesis, Institut für Maschinelle Sprachverarbeitung, University of Stuttgart, December. Arbeitspapiere des Instituts für Maschinelle Sprachverarbeitung (AIMS), volume 8, number 4.

Wolfgang. Lezius. 2002b. TIGERSearch – ein Suchwerkzeug für Baumbanken. In Stephan Busemann, editor, *Proceedings der 6. Konferenz zur Verarbeitung natürlicher Sprache (KONVENS 2002), Saarbrücken*, pages 107–114.

Ulrik Petersen. 2004. Emdros — a text database engine for analyzed or annotated text. In *Proceedings of COLING 2004, 20$^{\text{th}}$ International Conference on Computational Linguistics, August 23$^{\text{rd}}$ to 27$^{\text{th}}$, 2004, Geneva*, pages 1190–1193. International Commitee on Computational Linguistics. http://emdros.org/petersen-emdros-COLING-2004.pdf.

Ulrik Petersen. 2005. Evaluating corpus query systems on functionality and speed: Tigersearch and emdros. In Angelova et al. (Angelova et al., 2005), pages 387–391. ISBN 954-91743-3-6.

Ulrik Petersen. 2006; to appear. Principles, implementation strategies, and evaluation of a corpus query system. In *Proceedings of the FSMNLP 2005 workshop*, Lecture Notes in Artifical Intelligence, Berlin, Heidelberg, New York. Springer Verlag. Accepted for publication.

Douglas L. T. Rohde. 2005. Tgrep2 user manual, version 1.15. Available online http://tedlab.mit.edu/~dr/Tgrep2/tgrep2.pdf.

Dániel Varga, Peter Hálacsy, András Kornai, Viktor Nagy, Lázló Németh, and Viktor Trón. 2005. Parallel corpora for medium density languages. In Angelova et al. (Angelova et al., 2005), pages 590–596. ISBN 954-91743-3-6.

# [CS-TIW2006]

# Prolog+CG: A Maintainer's Perspective

Ulrik Petersen

2006

**Published in:** de Moor, Aldo, Polovina, Simon and Delugach, Harry (Eds.): *First Conceptual Structures Interoperability Workshop (CS-TIW 2006). Proceedings.* Aalborg University Press, pp. 58–71.

This page left intentionally blank

# Prolog+CG: A maintainer's perspective

Ulrik Petersen

Department of Communication and Psychology
Aalborg University
Kroghstræde 3
DK – 9220 Aalborg East
Denmark
ulrikp@hum.aau.dk
http://prologpluscg.sourceforge.net

**Abstract.** Prolog+CG is an implementation of Prolog with Conceptual Graphs as first-class datastructures, on a par with terms. As such, it lends itself well to applications in which reasoning with Conceptual Graphs and/or ontologies plays a role. It was originally developed by Prof. Dr. Adil Kabbaj, who in 2004 turned over maintainership of Prolog+CG to the present author. In this paper, Prolog+CG is described in terms of its history, evolution, and maintenance. A special web-enabled version of Prolog+CG is also described. Possible interoperability with CGIF and the CharGer tool are explored. Finally, we offer some general observations about the tenets that make Prolog+CG a success.

## 1 Introduction

Prolog+CG is an implementation of the Prolog programming language [1, 2], with extensions for handling the Conceptual Graphs of John Sowa [3–5] as well as object-oriented extensions. It was first developed by Prof. Dr. Adil Kabbaj as part of his doctoral studies at the University of Montreal in Canada. In 2004, Dr. Kabbaj graciously turned over maintainership of Prolog+CG to the present author. Since then, Prolog+CG has had its home on the web at the SourceForge.Net software-development collaboration site.[1] Prolog+CG is being used around the world both in teaching-environments and in research. The software has, at the time of writing, undergone 12 releases since maintainership was handed over to the present author, and has enjoyed more than 1800 downloads in total.

The purpose of this paper is to offer insights from the current maintainer's perspective on the history, maintenance, development, and future of Prolog+CG. The rest of the paper is laid out as follows. We first provide a bit of background on the history of Prolog+CG, followed by a description of the current version. We then offer a description of and reflection on the maintenance of Prolog+CG since 2004. We then describe a web-enabled version of Prolog+CG which the current maintainer has added to Prolog+CG as part of the development of the software. We then explore how, in the future, Prolog+CG might interoperate

[1] See http://prologpluscg.sourceforge.net

with other software through the CGIF standard, and also how Prolog+CG might interoperate with the CharGer tool. We then offer some general observations on some of the tenets which make Prolog+CG a success. Finally, we conclude the paper and describe future work.

## 2 History

Prolog+CG evolved out of work on the PROLOG++ system by Adil Kabbaj et al. [6]. Dr. Kabbaj then in his PhD thesis [7] developed the system further. This led to version 1.5, described in [8], further improved in [9]. The history thus far can be traced in the publications cited and is thus not described further here.

After 2001, development almost halted, then at version 2.0. At the University of Aalborg, the present author and his colleagues, Henrik Schärfe and Peter Øhrstrøm, became interested in using Prolog+CG as a basis for teaching formalization of meaning as well as logic programming to 2nd and 3rd year students of humanistic informatics. We therefore developed some teaching materials based on Prolog+CG[2] and its later successor, the Amine Platform[3], also written by Prof. Dr. Kabbaj.[4] In the spring of 2004, Aalborg University successfully attracted Dr. Kabbaj to come to Aalborg to give a PhD course on Artificial Intelligence. During Dr. Kabbaj's time in Aalborg, he graciously agreed that he would relicense Prolog+CG under an Open Source and Free Software license, the GNU Lesser General Public License version 2.1 [12], and that he would turn over maintainership of Prolog+CG to the present author.

The first release of Prolog+CG under the current maintainer was dubbed version 2.0.1, and was released on July 5, 2004. At the time of writing, the current version is 2.0.12, and version 2.0.13 is being prepared.

## 3 Description

In the following, we describe Prolog+CG as of version 2.0.12.

Consider Fig. 1. The screen is divided into five regions: From top to bottom: The standard menu-bar and toolbar, the Program Area (in which the Prolog program is written or loaded), the console (in which queries are entered and answers given), and the standard status bar.

The "File" menu supports the standard "New", "Open", "Save", "Save as", "Close", and "Exit" operations. In addition, two operations are available which enable the use of Prolog+CG as a HTML-embedded Java applet over the web (see Sect. 5).

The "Edit" menu supports the standard copy-paste-cut operations, as well as "go to line" (in the Program Area). The latter is useful when the compiler flags an error, giving a particular line at which the error occurs.

[2] See [10].
[3] See `http://amine-platform.sourceforge.net`
[4] We have recorded some of our experiences with the teaching materials in [11].

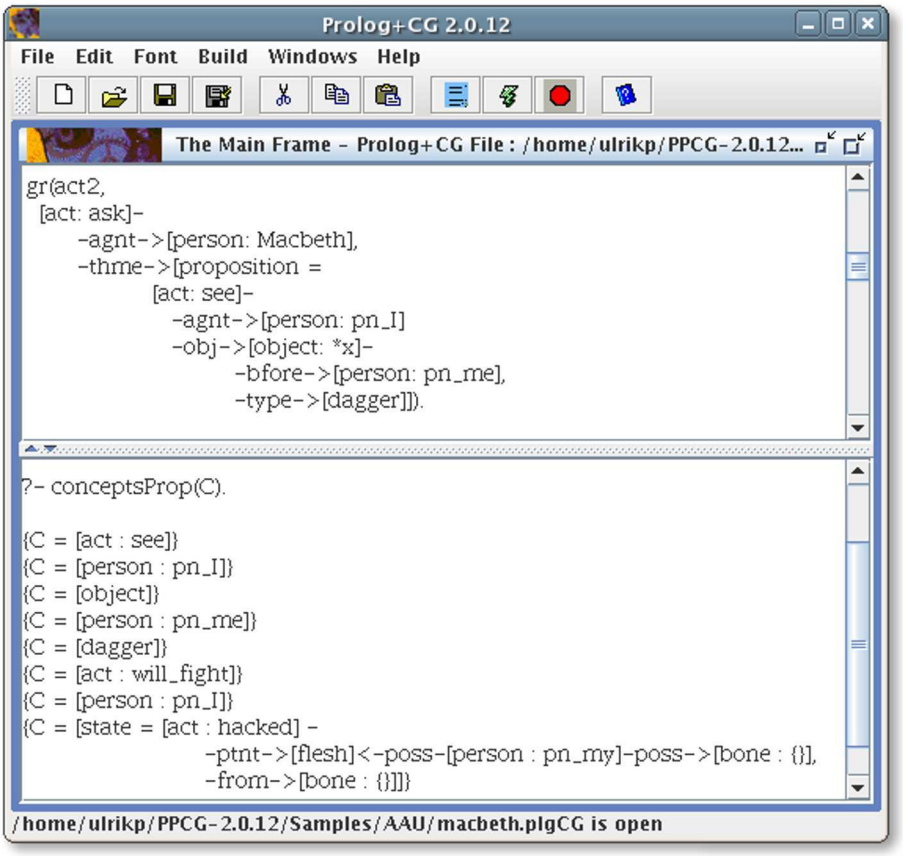



**Fig. 1.** Screenshot of Prolog+CG.

The "Font" menu supports changing the size and bold/normal status of the font of the current pane (either the Program Area or the console).

The "Build" menu supports compilation of the Prolog program, answering the current query (in the console), starting the debugger, stopping execution, and entering expert system mode.

The "Windows" menu supports opening the "primitives" window, which briefly summarizes the built-in Prolog primitives (see Fig. 2).

The "Help" menu also supports opening the "primitives" window, as well as showing the user's manual and the "About" box.

## 4 Maintenance

The present author received the Java source files for Prolog+CG from Dr. Kabbaj in April 2004. The present author then spent some time cleaning up the

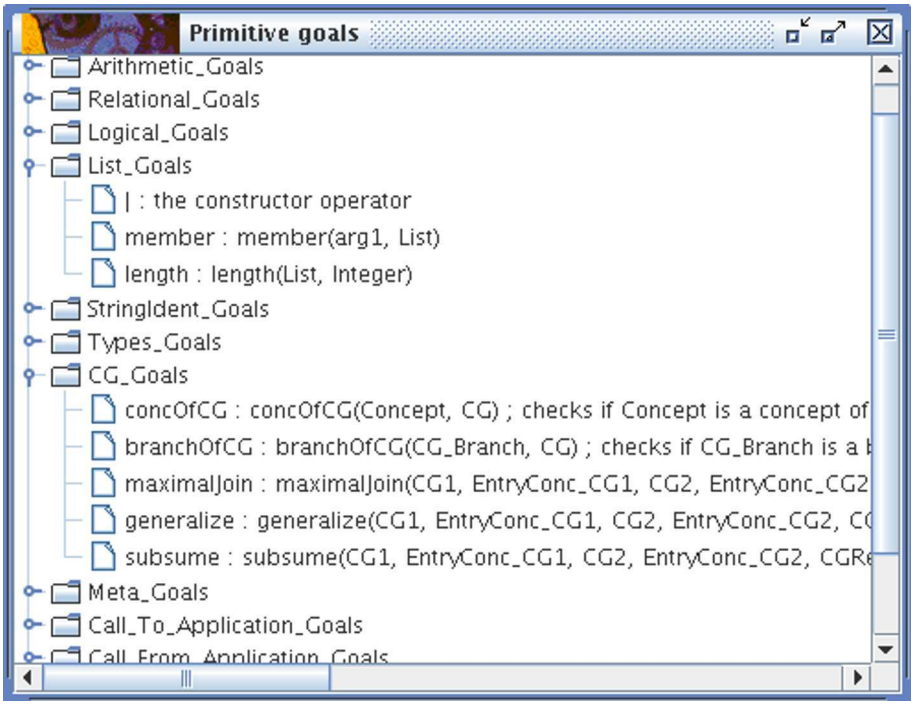



**Fig. 2.** The "primitives" window in Prolog+CG.

code and getting to know it, fixing a few bugs that had been annoying us in our teaching. This led to version 2.0.1, which was released on July 5, 2004.

Development has taken place via SourceForge.Net, which is an Internet-based collaborative platform for Open Source software development and distribution. SourceForge.Net provides gratis resources to the Open Source community, in the form of (among other services) web-space, download-hosting, CVS repositories, bug-tracking, and fora for discussion.

A website was put together, and was hosted on SourceForge.Net from the time of the first release (2.0.1).[5] Since the website is simple, the current maintainer has not had cause to alter the website drastically during the course of his maintainership.

From the beginning of the presence of Prolog+CG on SourceForge.Net,[6] development has taken place using the CVS repository facilities of SourceForge.Net. Not only has this helped the maintainer in the face of changing local workstations; it has also facilitated easy tracking of code-changes and versions. This has

[5] See `http://prologpluscg.sourceforge.net`
[6] The project was registered on the site on July 1, 2004.

proven crucial more than once, for example when needing to track exactly what had happened to the sourcecode since a given release.

The manual has been reorganized from its original one-page HTML document into a more highly structured LaTeX document, which then becomes the source for both the printable manual in PDF format and the HTML version.[7]

Throughout the process of maintenance, code-comments in French have been translated into English, so as to better serve the current maintainer. Likewise, many identifiers have been translated from French into English, to improve legibility and intelligibility for people who, like the current maintainer, are less than strong in French. Likewise, the code has been reindented to reach similar goals.

Gradually, features have been added. Some examples follow.

New predicates have been added, such as nl/0 (emit a newline on the console), writenl/1 (emit a term on the console, followed by newline), clearConsole/0 (erase or clear the contents of the console).[8] Another predicate, concat/3 was added to concatenate strings.[9] Two other predicates, minComSuperTypes/3 and maxComSubTypes/3 were added for those situation in which the type hierarchy in a Prolog+CG program is not a lattice, but in which there is more than one minimum common supertype or maximum common subtype.[10] Other new predicates include seed/1 and rnd/3 for doing pseudo-random number generation.[11]

Functionality has also been removed: For example, the generation of object files containing the compiled versions of Prolog+CG programs was removed in version 2.0.9; it was no longer needed, since we have fast enough machines today that loading a compiled object file was no quicker than compiling the program in-memory.

In version 2.0.10, the code was heavily refactored, changing almost everything from French into English, also yielding a new package structure instead of everything being in the same Java package. This helped the current maintainer understand the code even better, as the various parts were now cleanly separated into CG operations, Prolog implementation, GUI implementation, and top-level entry points.

This refactoring also paved the way for another development: The STARlab laboratory in Belgium, with which Dr. Aldo de Moor is associated, wanted to run Prolog+CG as part of an Enterprise Java Beans application. The problems in doing so included:

1. Prolog+CG required the presence of a GUI, in particular, an X server had to be running on the server on which STARlab wanted to run Prolog+CG. This was a problem on a headless server.
2. Prolog+CG was implemented with a single class, PrologPlusCGFrame, being the centre of communication between the various parts. This would not have

[7] The latex2html project is used for the conversion from LaTeX to HTML. See `http://www.latex2html.org/`

[8] All these were added in version 2.0.6.

[9] This was done in version 2.0.7.

[10] These were added in version 2.0.8.

[11] These were added in version 2.0.12.

been a problem, were it not for the fact that the fields of this class were almost all declared `static`, meaning that only one instance could be present in a Java Virtual Machine. This meant that STARlab had to implement a wrapper around Prolog+CG which serialized access to Prolog+CG, thereby slowing down the many-threaded application.

The solution turned out to be to separate the GUI from the Prolog and CG engines, and to make the internal communication happen around a class having only non-static member variables, and then passing an instance of this class around inside of Prolog+CG.

A number of bugfixes have been made, both ancient bugs and bugs introduced by the current maintainer. For example, backtracking was not always done properly, leading to crashes and "hanging" of the program. To guard against future bugs, a regression test suite was introduced in version 2.0.11, as was a command-line version of Prolog+CG (to facilitate scripting of the test suite). In addition, the "findbugs"[12] program was run on the code which resulted in version 2.0.12, and over 100 potential trouble spots were fixed.

The present author has attempted to apply standard software development practices to the maintenance and development of Prolog+CG. For example: Using source code control (CVS); Making regression tests; Using code analysis tools (findbugs); Indenting code as per the structure of the flow of control [13]. Open Source practices have also been followed, such as: Release early, release often; Value feedback from users [14].

No software is developed in a vacuum. The input of users like Dr. Aldo de Moor and Prof. Dr. Peter Øhrstrøm and others, both as to bug-reports and as to feature-requests, is what has really driven Prolog+CG development.[13]

## 5 An application

Prof. Dr. Peter Øhrstrøm deserves the credit for coming up with the idea of using Prolog+CG over the web. In January of 2005, he prompted the current maintainer to implement support for such a usage scenario. The primary goal was to enhance the usefulness of Prolog+CG in teaching environments.

In version 2.0.6,[14] an applet version of Prolog+CG was introduced, running locally in the user's browser. The applet attempts to follow the principle of simplicity in design and usage. A screenshot of the applet can be seen in Fig. 3.

In Fig. 3, it can be seen that there is one input-field ("Name") and two buttons ("Run" and "Clear"). This is configurable such that up to five input

[12] Findbugs has been developed by the Computer Science Department of the University of Maryland. See `http://findbugs.sourceforge.net/`

[13] This is also the experience of the present author in his various other Open Source projects, including the Emdros corpus query system (`http://emdros.org/`). This social process around Open Source software has been described and dissected at length by Eric S. Raymond in [14].

[14] Released February 2, 2005.

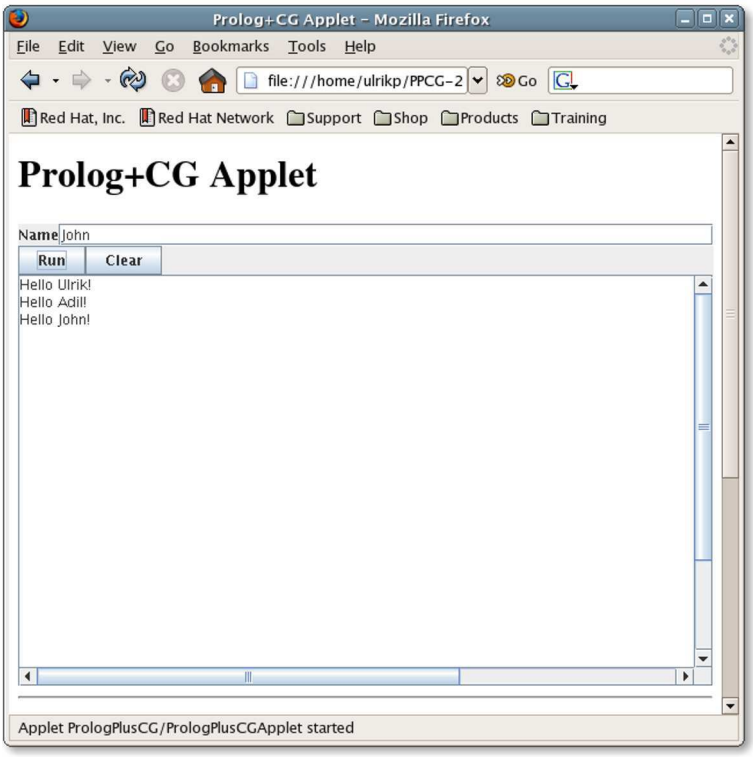



**Fig. 3.** An example applet written with Prolog+CG.

fields and five buttons can be added. The buttons execute arbitary Prolog+CG goals, either built-in or in the program underlying the applet. The input-fields are used to pass arguments to the goals executed by pressing the buttons.

The Prolog program underlying the applet shown in Fig. 3 is the following simple example:

```
main(X) :-
     write("Hello"), write(X), write("!"), nl.
```

Figure 4 shows the Applet Deployment dialog configuring the same applet as in Fig. 3. This dialog is accessible from within the "File" menu in the main Prolog+CG program. It can be seen that the number of input fields ("boxes") is configurable, as are the names of the input fields. Also, the buttons are configurable as to their number, and each button can be given a label as well as specifying the goal to execute when the button is pressed. In the figure, the "Run" button is configured to execute the "main/1" goal of the underlying example Prolog+CG program, passing the contents of input field 1 (signified by

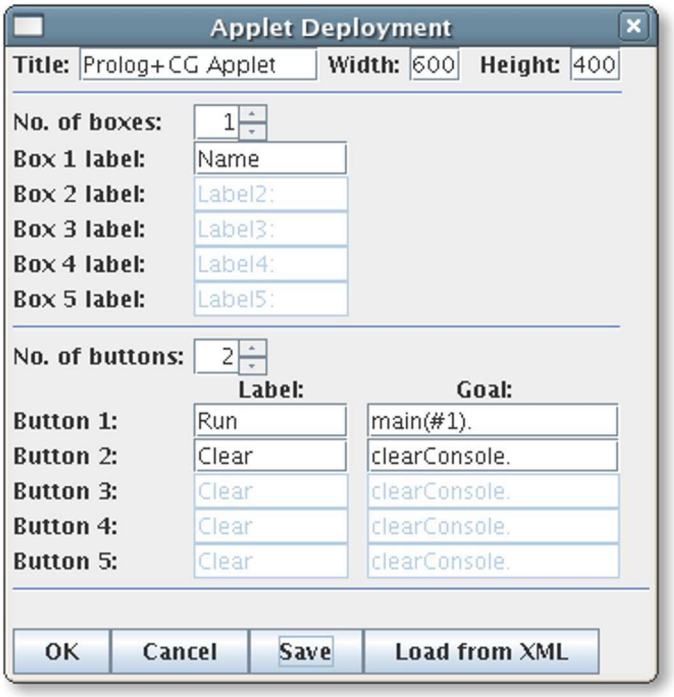



**Fig. 4.** The applet deployment dialog.

"#1") as the sole parameter to main/1. The parameters of the applet can be saved to XML for easy loading later.

It is easy to see that, although this simple example only demonstrates "hello world" functionality, the full scale of Prolog capabilities are potentially behind any Prolog+CG applet. To be precise: Any goal can be called with any parameters, and the output can be shown in a user-friendly way through the "write" family of built-in primitives. Because the applet is running locally and because it is stateful (not stateless like some CGI applications), Prolog asserts can also be made, thus growing the database behind the interface. This supports "intelligent" interactivity to as high a degree as almost any other Prolog system can afford. The only limitations are the limit on the number of buttons available (and thus the number of goals which can be run), and the limit on the number of input-fields available (and thus the number of unique parameters available to any goal). The former can even be overcome by judicious usage of the "exec" built-in primitive, which executes any goal, the idea being to have a button which uses the "exec" primitive to execute a goal whose name is stated in one of the input fields. The latter limitation can easily be overcome by extending the source code, which is, of course, open source and thus available to anyone for modification.

The applet version of Prolog+CG is very useful in a teaching context in the field of humanistic informatics: Students are able to experiment not only with writing a Prolog program, but also with the deeper applications of logic to real-world problems, with embedding the applet in a HTML page of their own design, and with an interactive system, simple to use yet capable of a degree of reasoning. In addition, they are able to host it on their own web-page in a simple-to-deploy manner, which makes it easy for them to show their work to their parents and peers, thus enabling socialization of their studies in their out-of-school relationships.

## 6 Interoperability

Interoperability is important in almost any software: Almost all software uses at least one library, for example,[15] which is a very old example of interoperability requirements between different parts of a software solution. Furthermore, consider the facts that: a) all computations by their nature have a relation between input and output, and b) computations can be serialized, and c) software is often compartmentalized into modules[16]. These facts together imply that different pieces of software often need to interoperate as to their inputs and outputs. This, in turn, requires not only agreed-upon formats, but also agreed-upon semantics of those formats.

In the following, we describe some problems involved in adding more interoperability capabilities to Prolog+CG, including some potential solutions to some of the problems. We first explore interoperability with the CGIF standard, after which we explore how Prolog+CG might interoperate with CharGer. We then conclude this section.

[15] E.g., the standard C library.

[16] The reasons for this compartmentalization are deep and multifarious, so I shall restrict myself here to mentioning only one: First and foremost, human beings have limited capabilities of abstraction, memory, and ability to keep great amounts of detail in focus at the same time. These limitations necessarily leads to methodologies and practices of software engineering which encourage modularity in programming. As Alfred North Whitehead famously said in his 1911 *Introduction to Mathematics*, "Civilization advances by extending the number of important operations which we can perform without thinking about them." One might paraphrase this in software engineering terms by saying that the quality and usefulness of software advances by extending the number of important computational operations which we can have the computer perform without thinking about how they are done. This requires important functionality to be encapsulated in "black boxes" with clean and — ideally — lean interfaces. Thus our human limitations of "intelligence" by their very nature lead to methodologies and practices of software engineering which compartmentalizes software.

### 6.1 CGIF

CGIF[17] is one format designed for interoperability between software components. In this section, we describe some problems involved in supporting import from and export to CGIF in Prolog+CG.

It is an important limitation in Prolog+CG that the software only supports a subset of the general CG theory. In particular, relations are always binary, meaning that a relation always has exactly two concepts attached. This limitation leads to the fact that, while it would be relatively easy to export CGs from Prolog+CG to CGIF, it would be more difficult to import "general" CGs from CGIF into Prolog+CG. The reason for the difficulty of the latter operation is that "general" CGs might have unary relations or ternary relations, or relations with even higher arity, which would require some massaging in order to constrict them into the binary relation paradigm. Exactly how this massaging is to be done algorithmically is unclear at this stage. However, provided the particular CGs to be imported were algorithmically mappable to the subset of the CG theory supported by Prolog+CG, there is no fundamental reason why CGs in CGIF could not be (at least partially) imported.

Exporting to CGIF, however, might proceed as follows: Some kind of built-in primitive would have to be added, having two parameters, namely: a) The name of a file to which to append the output, and b) A CG. Prolog's backtracking mechanism could be used to call this built-in primitive repeatedly for all the CGs which one wished to export. For example:

```
graph(cg1, [Man]<-agnt-[Love]-benf->[Woman]).
graph(cg2, [Woman]<-rcpt-[Flower]<-thme-[Give]-agnt->[Man]).

writeCGs :- graph(X, G), writeCGIF("/home/joe/love.cgif.txt", G).
```

The writeCGs predicate would first write graph `cg1`, then backtrack and write graph `cg2`, then stop.

Various problems exist in exporting CGs from Prolog+CG to CGIF. For example:

1. Coreferents are marked in Prolog+CG by identical variable names. This would have to be identified and translated to the defining-label/bound-label notation of CGIF.
2. Multi-referents are marked in Prolog+CG by concepts with the same type, having a referent of the form "*DIGIT" (e.g., "*1", "*2", etc.). This notation means that the two concepts are really the same concept. This would have to be mapped to only one concept in CGIF.
3. Relations can be (free) variables in Prolog+CG, which (to the best of our knowledge) is not possible in CGIF. The export of CGs with variables as relations names would have to fail if the variables were not bound variables at the time of the export.

[17] Conceptual Graph Interchange Format.

Thus several problems exist in exporting CGs from Prolog+CG to CGIF, and the above list is not even exhaustive. Solutions can be found for most of them, of course, so it is not a completely impossible task.

### 6.2 CharGer

CharGer is a tool for drawing conceptual graphs, maintained by Dr. Harry Delugach.[18] CharGer supports export to, among other formats, CGIF. Provided the CGs thus exported met the requirement mentioned above of being algorithmically mappable to the subset of the CG theory implemented in Prolog+CG, such graphs could be imported into Prolog+CG.

Export to CharGer format would be difficult, but not impossible. CharGer currently supports saving and loading an XML-based format. This format has all of the information necessary for maintaining the CGs in a datastructure (e.g., concepts, concept types, referents, relations, concept attachments to relations, etc.). This information could easily be exported to something that looks like CharGer's XML-based format. However, the XML also has (when saved from CharGer) some visual information, such as: Width, height, and location of concepts and relations, location and path of arrows, etc. This information seems to be necessary for loading the CGs correctly into CharGer. This information could be obtained automatically from within Prolog+CG by using a graph layout engine such as graphviz[19].

### 6.3 Conclusion

Interoperability between different software components often involves exchange formats. We have looked at two formats, namely CGIF and the XML-based format of CharGer. We have identified some problems involved in importing into and exporting CGs from Prolog+CG into each of these formats. In particular, for importing, the CGs have to be mappable to the binary-relation-only paradigm of Prolog+CG, and for exporting, certain problems have to be overcome, such as the differences in syntax between CGIF and Prolog+CG, or the need for visual layout in CharGer's XML format.

## 7 General Observations

Prolog+CG owes most of its success to the insights had by Dr. Adil Kabbaj while developing the first versions. Here the present author wishes to elaborate on his views on what makes Prolog+CG a success, both in research and in teaching.

First, the twin foundations of Prolog+CG are — and remain — two well-studied languages, namely Prolog and Conceptual Graphs. Prolog is based on Horn Clause logic [2], another well-studied topic, and Conceptual Graphs take

[18] See `http://charger.sourceforge.net`
[19] See `http://www.graphviz.org/`

their point of departure in the Existential Graphs of Charles Sanders Peirce [3]. These twin foundations of Prolog+CG are the core of what makes Prolog+CG useful. Separately, they remain useful. The insight of Dr. Kabbaj was that their combination could prove to be potentially even more useful.

Prolog is useful for reasoning about atoms and other terms. Conceptual Graphs are a version of logic which, like any logic, is useful for reasoning. In addition, Conceptual Graphs are able to easily express richer meaning than what is easy to express using Prolog terms.[20] By making Conceptual Graphs first-class datastructures on a par with terms, Dr. Kabbaj has enabled much easier integration of knowledge-bases with the reasoning powers of Prolog.[21]

Second, the integration of not only bare conceptual graphs, but also ontologies containing both a type hierarchy and a catalog of instances, increases the level of usefulness of Prolog+CG. At the University of Aalborg, we have used Prolog+CG in our teaching of the subject of formalization of meaning, and have sometimes made use of only the ontology-part of Prolog+CG. We have been able to apply the built-in ontology primitives to enable students to reason with ontologies, thereby increasing their level of understanding and insight into the subject. Ontologies are an inseparable part of Conceptual Graphs [3, 5] if one wishes to reason with them, and as such belong in any system dealing with the use of Conceptual Graphs. This insight has been applied in Prolog+CG by Dr. Kabbaj, and contributes to its success.

Third, the integration of Conceptual Graphs into the Prolog programming language has been implemented such that the tightness of the integration enables full support of Conceptual Graphs within the Prolog paradigm. For example, variables may appear anywhere an identifier may appear in a Conceptual Graph, including relation-names and concept types, thereby enabling unification at all levels, including variables being present at all levels. This is especially useful in such predicates as branchOfCG, concOfCG, and subsume, as demonstrated in [15, 16].

Thus there are at least three tenets of Prolog+CG which contribute to its success. First, it is founded upon two well-studied languages, namely Prolog and Conceptual Graphs. Separately, they are useful, but in combination, they can potentially become even more useful. Second, the integration of ontologies and catalogs of instances into Prolog+CG enables useful reasoning over type hierarchies, thus enhancing the usefulness of Prolog+CG. And third, the tight integration of Conceptual Graphs into the Prolog language enables easier development of knowledge-based systems than would have been possible with standard Prolog alone, or with a lesser integration of Conceptual Graphs than what has, in fact, been implemented.

[20] It is possible to express Conceptual Graphs entirely within the paradigm of standard Prolog. Yet such expressions would remain cumbersome to write and not easy to read.

[21] Henrik Schärfe has shown at length how this combination can lead to not only empirically pleasing results, but also theoretically profound insights in the field of computer aided narrative analysis [15, 16].

## 8 Conclusion and further work

We have described the history, maintenance, and development of Prolog+CG, an implementation of Prolog supporting Conceptual Graphs as first-class datastructures. We have also reported on one application of Prolog+CG, namely a web-enabled version running as a Java applet in the user's local browser. This version of Prolog+CG is especially useful in a teaching environment. In addition, we have elaborated on some of the problems involved in adding more interoperability capabilities to Prolog+CG, including potential solutions to some of the problems. Finally, we have offered some general observations about the tenets which make Prolog+CG a success.

As already mentioned, version 2.0.13 is under development at the time of writing. The single largest planned change is the ability of Prolog+CG to be embedded in a Java Servlet, serving up HTML via a Tomcat server.[22] It is planned that this version will be able to run Prolog+CG programs written in Prolog, stored on a server, and able to answer HTTP/1.1 requests, including GET and POST methods of communication with the Prolog+CG program. Thus Prolog+CG will become fully web-enabled, able to act both on the client side and on the server side.

It is hoped that some of the work which the present author has exercised on Prolog+CG will work its way into the successor to Prolog+CG, namely the Amine platform already mentioned. In particular, the web-enablement features would be very useful in an Amine context, especially in a teaching-environment, for reasons similar to those already mentioned.

Prolog+CG has already proven useful to the Conceptual Graphs community over its long history. In order to ensure the future success of Prolog+CG, the University of Aalborg is in the process of bringing in and funding an additional maintainer, namely cand.scient. Jørgen Albretsen. It is planned that the maintenance of Prolog+CG will continue for the foreseeable future.

Prolog+CG's success has largely depended on user support in the form of feedback. It is hoped that this feedback will continue to be given.

## Acknowledgements

The present author wishes to thank the Department of Communication and Psychology and the Study Committee for Humanistic Informatics at the University of Aalborg for their financial support, and SourceForge.Net for their generous support of the Open Source community. Most of all, the present author wishes to thank Prof. Dr. Adil Kabbaj for writing Prolog+CG in the first place, and for having the foresight, wisdom, and willingness to hand over maintainership of Prolog+CG to the present author.

[22] See `http://tomcat.apache.org/`

## References


1. Clocksin, W.F., Mellish, C.: Programming in Prolog. 2nd edn. Springer Verlag, Berlin (1984)
2. Rogers, J.B.: A Prolog Primer. Addison-Wesley (1986)
3. Sowa, J.F.: Conceptual Structures: Information Processing in Mind and Machine. Addison-Wesley, Reading, MA. (1984)
4. Sowa, J.F.: Conceptual graphs summary. In Nagle, T.E., Nagle, J.A., Gerholz, L.L., Eklund, P.W., eds.: Conceptual Structures: Current Research and Practice. Ellis Horwood, New York (1992) 3–51 ISBN: 0-13-175878-0.
5. Sowa, J.F.: Knowledge Representation: Logical, Philosophical, and Computational Foundations. Brooks/Cole Thomson Learning, Pacific Grove, CA (2000)
6. Kabbaj, A., Frasson, C., Kaltenbach, M., Djamen, J.Y.: A conceptual and contextual object-oriented logic programming: The PROLOG++ language. In Tepfenhart, W.M., Dick, J.P., Sowa, J.F., eds.: Conceptual Structures: Current Practices – Second International Conference on Conceptual Structures, ICCS'94, College Park, Maryland, USA, August 1994, Proceedings. Volume 835 of Lecture Notes in Artificial Intelligence (LNAI)., Berlin, Springer Verlag (1994) 251–274
7. Kabbaj, A.: Un systeme multi-paradigme pour la manipulation des connaissances utilisant la theorie des graphes conceptuels. PhD thesis, Univ. De Montreal, Canada (1996)
8. Kabbaj, A., Janta-Polczynski, M.: From PROLOG++ to PROLOG+CG : A CG object-oriented logic programming language. In Ganter, B., Mineau, G.W., eds.: Proceedings of ICCS 2000. Volume 1867 of Lecture Notes in Artificial Intelligence (LNAI)., Berlin, Springer Verlag (2000) 540–554
9. Kabbaj, A., Moulin, B., Gancet, J., Nadeau, D., Rouleau, O.: Uses, improvements, and extensions of Prolog+CG : Case studies. In Delugach, H., Stumme, G., eds.: Conceptual Structures: 9th International Conference on Conceptual Structures, ICCS 2001, Stanford, CA, USA, July/August 2001, Proceedings. Volume 2120 of Lecture Notes in Artificial Intelligence (LNAI)., Berlin, Springer Verlag (2001) 346–359
10. Petersen, U., Schärfe, H., Øhrstrøm, P.: Online course in knowledge representation using conceptual graphs. On the web: `http://www.huminf.aau.dk/cg/` (2001-2006)
11. Schärfe, H., Petersen, U., Øhrstrøm, P.: On teaching conceptual graphs. In Priss, U., Corbett, D., Angelova, G., eds.: Proceedings of ICCS 2002. Volume 2393 of Lecture Notes in Artificial Intelligence (LNAI). Springer Verlag, Berlin (2002) 285–298
12. Stallman, R.M.: GNU lesser general public license, version 2.1. On the web: `http://www.gnu.org/copyleft/lesser.html` (1999)
13. Kernighan, B.W., Pike, R.: The Practice of Programming. Addison-Wesley (1999)
14. Raymond, E.S.: The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. 1st edn. O'Reilly and Associates (2001)
15. Schärfe, H.: Reasoning with narratives. Master's thesis, Department of Communication, Aalborg University (2001) Available on the web from `http://www.hum.aau.dk/~scharfe`.
16. Schärfe, H.: Computer Aided Narrative Analysis. PhD thesis, Faculty of Humanities, Aalborg University, Denmark (2004)

# [CS-TIW2007]

# Using interoperating conceptual tools to improve searches in Kaj Munk

Ulrik Petersen

2007

**Published in:** Pfeiffer, Heather D., Kabbaj, Adil and Benn, David (Eds.), *Second Conceptual Structures Tool Interoperability Workshop (CS-TIW 2007). Held July 22, 2007 in Sheffield, UK, in conjunction with International Conference on Conceptual Structures (ICCS) 2007.* Research Press International, Bristol, UK. ISBN: 1-897851-16-2, pp. 45–55

This page left intentionally blank

# Using interoperating conceptual tools to improve searches in Kaj Munk

Ulrik Petersen

Department of Communication and Psychology
Kroghstræde 3
DK – 9220 Aalborg East
Denmark
ulrikp@hum.aau.dk
http://www.kajmunk.hum.aau.dk

**Abstract.** Query Expansion is a technique whereby a query in an information retrieval system is expanded with more terms, thus most likely increasing the number of relevant documents retrieved. In this paper, we describe a prototype of a system built around a number of interoperating conceptual structures tools, and how it uses Query Expansion to retrieve greater numbers of relevant documents. Thus the system is an example of how interoperating conceptual structures tools can be used to implement an information retrieval system.

## 1 Introduction

In what ways is it possible to query a corpus of natural language text conceptually? That is the motivating question behind the research presented in this paper. In order to answer this question partially, we have built a prototype system which incorporates three technologies, namely the Amine Platform [1–6], the Emdros corpus query system [7–10], and some natural language processing software in the form of a lemmatizer and a part of speech tagger[1]. The system is able to increase the recall of queries for a given corpus of text, by expanding the query with lemmas taken from an Amine ontology. The system could not have been built without the integration of the three key technologies mentioned. In this paper, we show how the system works in terms of its architecture, and how it is able to achieve greater recall.

The organizational context of the present research is the Kaj Munk Research Centre at Aalborg University, Denmark. Kaj Munk (1898-1944) was a Danish playwright, pastor, poet, and author, who was very influential both in Danish cultural life and outside of Denmark in the period between the two World Wars. He was killed by the Germans in 1944 for his resistance stance.

The Kaj Munk Research Centre has bought the *nachlass* of Kaj Munk, and is in the process of digitizing the material for electronic publication on the web and

[1] The lemmatizer and part of speech tagger employed in this research are the ones developed by Centre for Language Technology (CST), Copenhagen, Denmark. See http://www.cst.dk.

in other ways. The envisaged website will feature advanced search capabilities that go beyond mere matching of text strings into the realm of semantics. In this endeavour, conceptual structures play a key role, and software tools that deal with conceptual structures become critical in the development of the underlying database technology.

The rest of the paper is laid out as follows. First, we introduce the literature behind our system. Second, we give an overview of our system. Third, we offer a more detailed look at the query-process that leads to semantic querying. Fourth, we give an example of the query process. Fifth, we give an analysis of the functionality in terms of the precision and recall of the system. Sixth, we report on the method of achieving interoperability between the various parts of the system. Finally, we conclude the paper.

## 2 Literature review

Within the field of information retrieval, the notions of precision and recall are often used to describe how well a search system performs. Briefly, recall is a percentage showing how many documents out of all relevant documents were retrieved, while precision is a percentage showing how many of the retrieved documents are in fact relevant. For more information, see [11] and [12].

Query Expansion refers to a class of techniques in Information Retrieval in which a query given by the user is expanded with more query terms. The intent is always either to increase the recall, or to increase the precision, or both. Query Expansion is an old technique, but as demonstrated by the literature, a very useful technique. See for example, [13–15]. In so far as WordNet [16] can be considered an ontology, [13, 17, 15] among many others show that ontologies can prove valuable in the process of Query Expansion. The article [18] shows how compound forms in Danish can be split into their respective lemmas, then used as a basis for Query Expansion using a thesaurus.

The present author has built a corpus query system called Emdros. The Emdros software is a generic query system for "analyzed or annotated text." As such, the software accommodates "text plus information about that text."[2] In the present research, Emdros is the component that stores and queries both the text corpus to be queried and the part of speech and lemma information with which each token is annotated. Additional information such as sentence boundaries, paragraph boundaries, and noun phrase boundaries are also present, but are not used in the present research. Document boundaries, however, are used.

Emdros was written in C++, but has language bindings for several programming languages including Java. These language bindings are provided through SWIG.[3] For more information on Emdros, the reader is invited to consult both

[2] This phrase is taken from [19], which is the PhD thesis of Crist-Jan Doedens. Emdros implements an extension of Doedens' database model, and a subset of Doedens' query language. As such, Emdros can be seen as a derivate of the labours of Dr. Doedens.

[3] See `http://www.swig.org`. See also [20].

the Emdros website[4] and [7–10], all of which can be downloaded from the author's website[5].

The Amine Platform is a platform intended for development of intelligent systems and multi-agent systems [5]. It implements a large array of the technology components needed for building Knowledge Systems, including an ontology builder, a CG layer with concomitant CG operations, and a logic inference engine built around the integration of Prolog and CGs.[6] The latter component is called Prolog+CG, and is the software hub in the prototype which we have developed.

## 3 System Overview

An overview of the system is given in Fig. 1. It is given in the form of a conceptual graph, with an implied ontology of concept types such as the one given in Fig. 2, and an implied relation hierarchy such as the one given in Fig. 3.

As can be seen from the ontology of concept types in Fig. 2, there are essentially two kinds of concepts in Fig. 1: Software and Data. Indeed, the relation types reflect this, as can be seen in Fig. 3, in which all subtypes of DataSoftwareRole have the signature (Data,Software), and all subtypes of SoftwareSoftwareRole have the signature (Software,Software). Consequently, the signature of Role in our small conceptual graph of Fig. 1 must be (Bits,Software), indicating that the outgoing arrow on every relation always is attached to a concept of type Software (cf. [21–24]).

There are three related but distinct flows of data in Fig. 1. The first flow starts with the TextCorpus at the left edge of the leftmost row. This TextCorpus (which, in our case, is the corpus of published sermons of Kaj Munk) is read by CST's part of speech tagger and lemmatizer to produce a pos-tagged and lemmatized corpus. This corpus is then read by a program which converts the corpus to a number of `CREATE OBJECT` statements in the MQL query language of Emdros. This produces the MQLCorpus, which is read by Emdros to produce the EmdrosDatabase at the bottom right hand corner of Fig. 1.

The second flow starts with the Amine Ontology Builder in the middle of the second row of Fig. 1, in which a domain expert creates an ontology of the domain which one would like to query. This produces the AmineOntology, which again is read by Amine's Prolog+CG engine. Notice that the method of production of the ontology is irrelevant for our prototype: It might just as well have been produced automatically. In our case, for simplicity and accuracy, we produced our own ontology "by hand."

The third, and main, flow starts with the user query in the top right hand corner of Fig. 1. This query is read by the Prolog+CG programs written for our

[4] `http://emdros.org`

[5] `http://ulrikp.org`

[6] However, Amine is much more than the few components listed here. Interested readers are invited to consult [5, 6] and the Amine Website: `http://amine-platform.sourceforge.net`

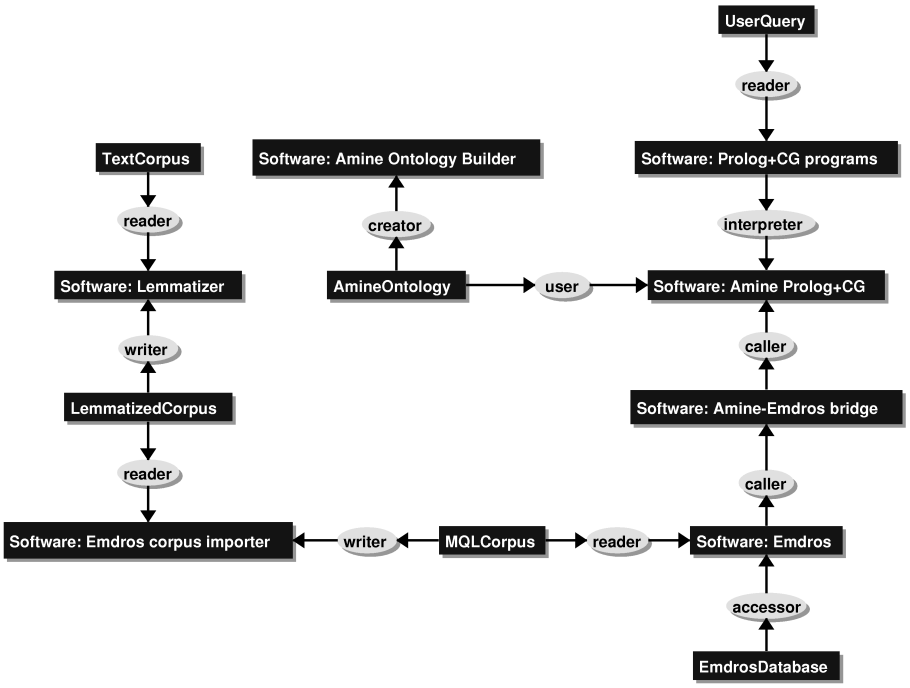



**Fig. 1.** Overview of our system

prototype, and is transformed, inside of Prolog, to an Emdros query based on the type(s) from the AmineOntology given in the query. This Emdros query is then fed to the Amine-Emdros bridge (through the interpreter-nature of Amine Prolog+CG), which takes care of calling Emdros to query the EmdrosDatabase in the bottom right hand corner of Fig. 1. An answer comes back from Emdros to the Amine-Emdros bridge. This answer is then further processed by the Amine-Emdros bridge in order to obtain not only the words found, but also their context.[7] This result is then passed back to the Prolog+CG programs (through the interpreter-nature of Amine Prolog+CG), which then displays the results to the user.

[7] Emdros's query language is designed to be very generic. As such, it uses a generic method of returning query results, which must then be interpreted in the context of a given database. This interpretation usually involves retrieval of more objects from the database, such as whole sentences and their constituent tokens, and/or the titles of the document(s) in which the results are found.

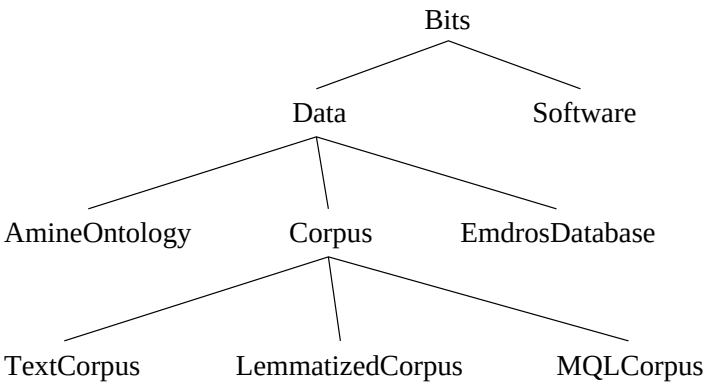



**Fig. 2.** A possible ontology for the concept types in our system

## 4 Querying

In our prototype, the user enters a query in the form of a set $Q$ of types from the Amine ontology, along with an indication of whether supertypes or subtypes are wanted. If supertypes are wanted, a set $E_i$ are constructed containing $n$ levels of supertypes of each term $t_i$ from $Q$. Similarly for subtypes, if subtypes are wanted.

An Emdros query is then constructed from the sets $E_i$, as follows. For each set $E_i$, a single query fragment (technically, an "object block" retrieving an object with certain characteristics) is created, finding all objects of type `token` whose lemma is one of the terms $e_j$ in $E_i$, with Boolean disjunction being the operator between the comparison. This could look as follows:

```
[token lemma='party' or lemma='organization' or lemma='group']
```

If there is more than one set $E_i$, then all permutations of all sequential orders of the query fragments arising from each $E_i$ is constructed, allowing for arbitrary space in between each query fragment, and using an OR construct on strings of blocks between each permutation. This results in a string of OR-separated strings of blocks, where each string of blocks represents one possible order of the query terms. Finally, this string of OR-separated strings is wrapped in a block indicating that the search is to be done within document boundaries.

Variations over this theme abound. For example, the number of levels $n$ to go up or down in the ontology can be varied; sibling nodes may be considered; various measures of semantic distance may be employed in determining which concepts to include in the search; word-sense disambiguation may be performed based on either the query terms and their cooccurrence in the query, or on the documents actually stored in the database, or both; the context may be changed from "Document" to a more restrictive "paragraph" or even "sentence" textual unit, thus increasing the likelihood that the query terms do in fact have

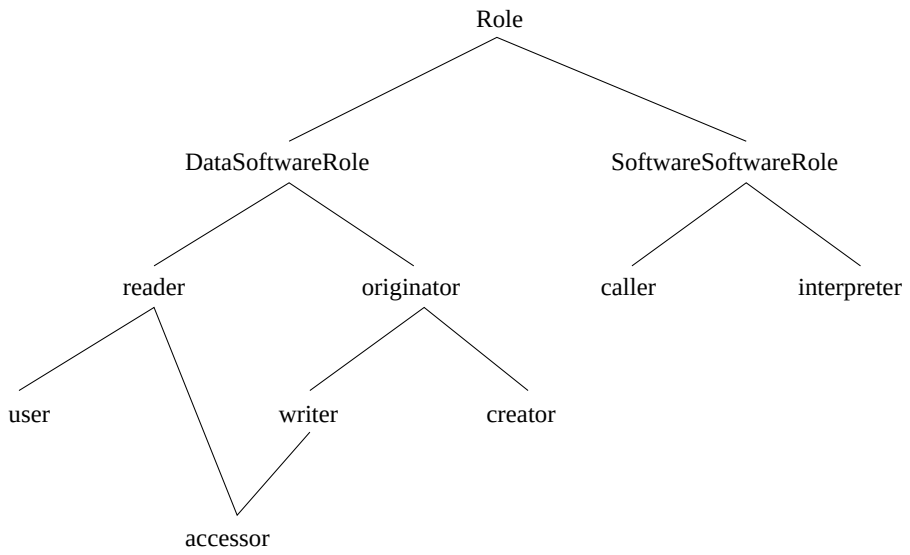



**Fig. 3.** A possible ontology for the relation types in our system

something to do with each other; named entity recognition may be performed, and named entities may be classified as per the ontology, thus aiding in increasing recall; compounds may be split and used as the basis of further Query Expansion, as described in [18]; parsing or chunking of the texts may be performed so as to aid in identifying noun phrases that could aid in identifying more precisely where to search for given kinds of entries from the ontology; the ontology may be enriched with part of speech information, such that this information can be taken into account when searching. Many other techniques have been tried over the years, all built up around the single, simple idea of Query Expansion.

## 5 Query Example

In this section, we give an example of how the query-process works.

Consider the small ontology in Fig. 4. It is a sample ontology of concepts from the political domain. Some of the concepts are annotated underneath with zero or more lemmas, separated by a vertical bar if more than one are present. Where no lemma corresponds to the type, the number of lemmas is zero.

Suppose the user enters a query in which the set $Q$ of query types is { PartyMember, PoliticalEmployee }, and suppose that the user specifies that 3 levels of subtypes are to be used for query expansion. In this case, two sets $E_0 =$ { partymember, minister, primeminister, MP, parliamentmember } and $E_1 =$ { spindoctor } are constructed.

From these sets, the two object blocks:

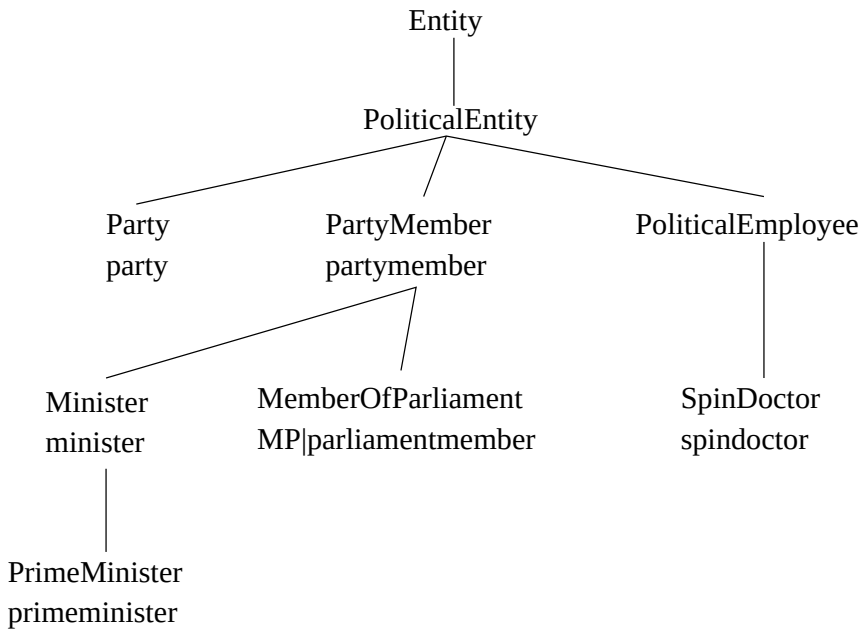



**Fig. 4.** A sample domain ontology of political language

```
[token lemma='partymember' OR lemma='minister'
       OR lemma='primeminister' OR lemma='MP'
       OR lemma='parliamentmember']
```

and

```
[token lemma='spindoctor']
```

are constructed. There are only two possible permutations of the order in which two objects can occur (2 factorial), so these object blocks give rise to the query shown in Fig. 5.

Briefly, the query means that, within the context of a document, two tokens must be found, in either order, where the lemma of each token is either drawn from the sets $E_0$ and $E_1$. The "`..`" between each `[token]` object block means that the tokens need not be adjacent, but may be separated by arbitrary space, within the scope of the context Document.

This query is executed by the Amine-Emdros bridge, and the results post-processed in order to get the context of the "hits", to be shown to the user by Prolog+CG.

```
[Document
  [token lemma='partymember' OR lemma='minister'
         OR lemma='primeminister' OR lemma='MP'
         OR lemma='parliamentmember'
  ]
  ..
  [token lemma='spindoctor']

  OR

  // Now the other order is tried...
  [token lemma='spindoctor']
  ..
  [token lemma='partymember' OR lemma='minister'
         OR lemma='primeminister' OR lemma='MP'
         OR lemma='parliamentmember'
  ]
]
```

**Fig. 5.** Example Emdros query

## 6 Precision and Recall

As mentioned in Sect. 2, "recall" is a measure used within Information Retrieval to describe how well a system performs; in particular, it shows how many documents were retrieved, divided by the total number of relevant documents for any given query. "Precision," on the other hand, is the number of relevant documents returned, divided by the number of documents returned [12].

As confirmed by the research reported in [13–15, 17], our system improves recall, and for the same reason that any Query Expansion technique in general improves recall: Since semantically similar terms are added to the query, more documents that contain semantically similar terms will be found. Since relevant documents may contain terms semantically similar to the original query terms, yet may not contain the actual query terms, increasing the number of documents retrieved with semantically similar terms will most likely increase recall.

We have not evaluated our system formally on either precision or recall measures, but this is something for future research.

## 7 Interoperability

This being a practical rather than theoretical paper, a number of comments on the interoperability of the various system components are in order.

Both Amine's ontology builder, Amine's Prolog+CG, and Emdros can be viewed as tools for dealing with conceptual structures; Amine's tools more so than Emdros. Amine's treatment of conceptual structures goes right to the core

of the very purpose for which Amine was created [5, 6]; thus a large part of Amine's codebase is centered around conceptual structures. Emdros, on the other hand, has a different focus, namely that of storage and retrieval of annotated text. Given that lemmas represent a unified form for all forms of a given word, and given that this simplifies the task of assigning meaning to any given word, and given that lemmas play an important role in the selection of labels for the concept types in many kinds of ontologies, and given that Emdros can store lemmas just as well as any other annotation, Emdros can be seen to be able to deal with conceptual structures.

The interoperability of Amine with Emdros was achieved through the use of a "bridge" written in Java. This bridge is simply a Java class which instantiates a connection to an Emdros datbase, receives queries, and "harvests" the results. The latter task involves processing the results of a query, then retrieving as much context as necessary for the purposes at hand. This usually involves things like retrieving document titles, all the words of the sentence surrounding a "hit", retrieval of other objects necessary for display of the hits, etc.

Amine's Prolog+CG supports calling arbitrary Java methods and instantiating arbitrary Java objects from within Prolog+CG. This is the method used in our prototype system, where Prolog+CG instantiates an "Amine-Emdros bridge" as a Java object, then calls methods on this bridge both to retrieve query results and to postprocess them as described above.

The present author found that Amine's Java-integration made it easy to call both the Amine API and the Emdros bridge. The ability to call arbitrary methods in the host language (Java, in this case) was key in making the interoperability work.

## 8 Conclusion

We have described a prototype system that enables a user to query a collection of documents semantically rather than just by keyword. This is done through the use of three key technologies: The Amine Platform, the Emdros Corpus Query System, and a lemmatizer and part of speech tagger for the target language. An ontology is used to guide the process of query expansion, leading to a greater number of relevant documents being returned than would have been the case, had the program only found documents containing the original query terms.

Pointers to further research have already been given.

## Acknowledgements

Thanks are due to cand.scient. Jørgen Albretsen, who provided the ontology used in this prototype. Prof. dr.scient., PhD Peter Øhrstrøm provided many of the research ideas used in this research. The Danish Centre for Language Technology (CST) provided the part of speech tagger and lemmatizer used. Figure 1 was

drawn with the CharGer software written by Harry Delugach.[8] The SWIG team[9] made the integration of Emdros with Java possible. Finally, many thanks to Prof. Dr. Adil Kabbaj, who wrote the Amine-Platform, without which this research would have been much more difficult to carry out.

## References

1. Kabbaj, A., Frasson, C., Kaltenbach, M., Djamen, J.Y.: A conceptual and contextual object-oriented logic programming: The PROLOG++ language. In Tepfenhart, W.M., Dick, J.P., Sowa, J.F., eds.: Conceptual Structures: Current Practices – Second International Conference on Conceptual Structures, ICCS'94, College Park, Maryland, USA, August 1994, Proceedings. Volume 835 of Lecture Notes in Artificial Intelligence (LNAI)., Berlin, Springer Verlag (1994) 251–274
2. Kabbaj, A.: Un systeme multi-paradigme pour la manipulation des connaissances utilisant la theorie des graphes conceptuels. PhD thesis, Univ. De Montreal, Canada (1996)
3. Kabbaj, A., Janta-Polczynski, M.: From PROLOG++ to PROLOG+CG : A CG object-oriented logic programming language. In Ganter, B., Mineau, G.W., eds.: Proceedings of ICCS 2000. Volume 1867 of Lecture Notes in Artificial Intelligence (LNAI)., Berlin, Springer Verlag (2000) 540–554
4. Kabbaj, A., Moulin, B., Gancet, J., Nadeau, D., Rouleau, O.: Uses, improvements, and extensions of Prolog+CG : Case studies. In Delugach, H., Stumme, G., eds.: Conceptual Structures: 9th International Conference on Conceptual Structures, ICCS 2001, Stanford, CA, USA, July/August 2001, Proceedings. Volume 2120 of Lecture Notes in Artificial Intelligence (LNAI)., Berlin, Springer Verlag (2001) 346–359
5. Kabbaj, A.: Development of intelligent systems and multi-agents systems with amine platform. [25] 286–299
6. Kabbaj, A., Bouzouba, K., El Hachimi, K., Ourdani, N.: Ontologies in Amine Platform: Structures and processes. [25] 300–313
7. Petersen, U.: Emdros — a text database engine for analyzed or annotated text. In: Proceedings of COLING 2004. (2004) 1190–1193 http://emdros.org/petersen-emdros-COLING-2004.pdf.
8. Petersen, U.: Evaluating corpus query systems on functionality and speed: TIGERSearch and Emdros. In Angelova, G., Bontcheva, K., Mitkov, R., Nicolov, N., Nikolov, N., eds.: International Conference Recent Advances in Natural Language Processing 2005, Proceedings, Borovets, Bulgaria, 21-23 September 2005, Shoumen, Bulgaria, INCOMA Ltd. (2005) 387–391
9. Petersen, U.: Principles, implementation strategies, and evaluation of a corpus query system. In: Proceedings of the FSMNLP 2005. Volume 4002 of Lecture Notes in Artifical Intelligence., Berlin, Heidelberg, New York, Springer Verlag (2006)
10. Petersen, U.: Querying both parallel and treebank corpora: Evaluation of a corpus query system. In: Proceedings of LREC 2006. (2006) Available as http://ulrikp.org/pdf/LREC2006.pdf.
11. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley (1999)

[8] `http://charger.sourceforge.net`
[9] `http://www.swig.org`, led by David Beazley.

12. Frakes, W.B., Baeza-Yates, R.: Information Retrieval: Data Structures and Algorithms. Prentice Hall (1992)
13. Voorhees, E.M.: Query expansion using lexical-semantic relations. In: SIGIR '94: Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval, New York, NY, USA, Springer-Verlag New York, Inc. (1994) 61–69
14. Mitra, M., Singhal, A., Buckley, C.: Improving automatic query expansion. In: SIGIR '98: Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval, New York, NY, USA, ACM Press (1998) 206–214
15. Moldovan, D.I., Mihalcea, R.: Using WordNet and lexical operators to improve internet searches. IEEE Internet Computing **4**(1) (2000) 34–43
16. Fellbaum, C., ed.: WordNet: An Electronic Lexical Database. MIT Press, London, England and Cambridge, Massachusetts (1998)
17. Smeaton, A.F., Quigley, I.: Experiments on using semantic distances between words in image caption retrieval. In: Research and Development in Information Retrieval. (1996) 174–180
18. Pedersen, B.S.: Using shallow linguistic analysis to improve search on Danish compounds. Nat. Lang. Eng. **13**(1) (2007) 75–90
19. Doedens, C.J.: Text Databases: One Database Model and Several Retrieval Languages. Number 14 in Language and Computers. Editions Rodopi Amsterdam, Amsterdam and Atlanta, GA (1994) ISBN 90-5183-729-1.
20. Beazley, D.M., Fletcher, D., Dumont, D.: Perl extension building with SWIG (1998) Presented at the O'Reilly Perl Conference 2.0, August 17-20, 1998, San Jose, California. Access online 2007-04-22: `http://www.swig.org/papers/Perl98/swigperl.htm`.
21. Sowa, J.F.: Conceptual Structures: Information Processing in Mind and Machine. Addison-Wesley, Reading, MA. (1984)
22. Sowa, J.F.: Conceptual graphs summary. In Nagle, T.E., Nagle, J.A., Gerholz, L.L., Eklund, P.W., eds.: Conceptual Structures: Current Research and Practice. Ellis Horwood, New York (1992) 3–51
23. Sowa, J.F.: Knowledge Representation: Logical, Philosophical, and Computational Foundations. Brooks/Cole Thomson Learning, Pacific Grove, CA (2000)
24. Petersen, U., Schärfe, H., Øhrstrøm, P.: Online course in knowledge representation using conceptual graphs. On the web: `http://www.huminf.aau.dk/cg/` (2001-2007)
25. Henrik Schärfe, Pascal Hitzler, P.Ø., ed.: Conceptual Structures: Inspiration and Application. 14th International Conference on Conceptual Structures, ICCS 2006, Aalborg, Denmark, July 2006, Proceedings. In Henrik Schärfe, Pascal Hitzler, P.Ø., ed.: Conceptual Structures: Inspiration and Application. 14th International Conference on Conceptual Structures, ICCS 2006, Aalborg, Denmark, July 2006, Proceedings. Volume 4068 of Lecture Notes in Artificial Intelligence (LNAI)., Berlin, Heidelberg, Springer-Verlag (2006)

This page left intentionally blank

# [ICCSSuppl2008]

# An FCA classification of durations of time for textual databases

Ulrik Sandborg-Petersen

2008

**Published in:** Eklund, Peter and Haemmerlé, Olivier, *Supplementary Proceedings of ICCS 2008*, CEUR-WS, http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/

This page left intentionally blank

# An FCA classification of durations of time for textual databases

Ulrik Sandborg-Petersen

Department of Communication and Psychology
Kroghstræde 3, DK – 9220 Aalborg East, Denmark
ulrikp@hum.aau.dk

**Abstract.** Formal Concept Analysis (FCA) is useful in many applications, not least in data analysis. In this paper, we apply the FCA approach to the problem of classifying sets of sets of durations of time, for the purposes of storing them in a database. The database system in question is, in fact, an object-oriented text database system, in which all objects are seen as arbitrary sets of integers. These sets need to be classified in textually relevant ways in order to speed up search. We present an FCA classification of these sets of sets of durations, based on linguistically motivated criteria, and show how its results can be applied to a text database system.

## 1 Introduction

Formal Concept Analysis (FCA)[1, 2] has many applications, not least of which is aiding a human analyst in making sense of large or otherwise incomprehensible data sets. In this paper, we present an application of FCA to the problem of classifying classes of linguistic objects that meet certain linguistically motivated criteria, with the purpose of storing them in a text database system.

We have developed a text database system, called Emdros[1], capable of storing and retrieving not only text, but also *annotations* of that text [3, 4]. Emdros implements the EMdF model, in which all textual objects are seen as sets of sets of durations of time with certain attributes.

The rest of the paper is laid out as follows. In Sect. 2, I describe four properties of language as it relates to time. In Sect. 3, I describe the EMdF model. In Sect. 4, I mathematically define a set of criteria which may or may not hold for a given object type. This results in a Formal Context of possible classes of objects, having or not having these criteria. In Sect. 5, I use FCA to arrive at a set of criteria which should be used as indexing mechanisms in Emdros in order to speed up search. In Sect. 6, I discuss the implementation of the criteria arrived at in the previous section, and evaluate the performance gains obtained by using them. Finally, I conclude the paper and give pointers to further research.

[1] http://emdros.org

## 2 Language as durations of time

Language is always heard or read in time. That is, it is a basic human condition that whenever we wish to communicate in verbal language, it takes time for us to decode the message. A word, for example, may be seen as a duration of time during which a linguistic event occurs, viz., a word is heard or read. This takes time to occur, and thus a message or text occurs in time.

In this section, we describe four properties of language which have consequences for how we may model linguistic objects such as words or sentences.

First, given that words occur in time, and given that words rarely stand alone, but are structured into sentences, and given that sentences are (at one level of analysis) sequences of words, it appears obvious that *sequence* is a basic property of language. We will therefore not comment further on this property of language.

Second, language always carries some level of structure; for example, the total duration of time which a message fills may be broken down into shorter durations which map to words. Intermediate between the word-level and the message-level, we usually find sentences, clauses, and phrases. Thus, linguistic units *embed* within each other. For a lucid discussion of the linguistic terms involved, please see [5, 6].

Third, language carries the property of being *resumptive*. By this we mean that linguistic units are not always contiguous, i.e., they may occupy multiple, disjoint durations of time. For one such opinion, see [7].

A fourth important property of linguistic units is that they may "violate each other's borders." By this we mean that, while unit $A$ may start at time $a$ and end at time $c$, unit $B$ may start at time $b$ and end at time $d$, where $a < b < c < d$. Thus, while $A$ overlaps with $B$, they cannot be placed into a strict hierarchy.

## 3 The EMdF model

In his PhD thesis from 1994 [8], Crist-Jan Doedens formulated a model of text which meets the four criteria outlined in the previous section. Doedens called his model the "Monads dot Features" (MdF) model. We have taken Doedens' MdF model and extended it in various ways, thus arriving at the Extended MdF (EMdF) model. In this section, we describe the EMdF model.

Central to the EMdF model is the notion that textual units (such as books, paragraphs, sentences, and even words) can be viewed as *sets of monads*. A monad *is* simply an integer, but may be viewed as an indivisible duration of time.[2]

*Objects* in the EMdF model are pairs $(M, F)$ where $M$ is a set of monads, and $F$ is a set of pairs $(f_i, v_i)$ where $f_i$ is the $i^{\text{th}}$ *feature* (or attribute), and $v_i$ is the value of $f_i$ for this particular object. A special feature, "self" is always

[2] Please note that we use the term "monad", *not* in the well-established algebraic sense, but as a synonym for "integer in the context of the EMdF model, meaning an indivisible duration of time".

present in any $F$ belonging to any object, and provides an integer ID which is unique across the whole database. The inequality $M \neq \emptyset$ holds for all objects in an EMdF database.

Since textual objects can often be classified into similar kinds of objects with the same attributes (such as words, paragraphs, sections, etc.), the EMdF model provides *object types* for grouping objects.

## 4 Criteria

In this section, we introduce some linguistically motivated criteria that may or may not hold for the objects of a given object type $T$. This will be done with reference to the properties inherent in language as described in Sect. 2.

In the following, let $\text{Inst}(T)$ denote the set of objects of a given object type $T$. Let $a$ and $b$ denote objects of a given object type. Let $\mu$ denote a function which, given an object, produces the set of monads $M$ being the first part of the pair $(M, F)$ for that object. Let $m$ denote a monad. Let $f(a)$ denote $\mu(a)$'s first (i.e., lowest) monad, and let $l(a)$ denote $\mu(a)$'s last (i.e., highest) monad. Let $[m_1 : m_2]$ denote the set of monads consisting of all the monads from $m_1$ to $m_2$, both inclusive.

**Range types:**

**single monad($T$):** means that all objects are precisely 1 monad long.
$\forall a \in \text{Inst}(T) : f(a) = l(a)$

**single range($T$):** means that all objects have no gaps (i.e., the set of monads constituting each object is a contiguous stretch of monads).
$\forall a \in \text{Inst}(T) : \forall m \in [f(a) : l(a)] : m \in \mu(a)$

**multiple range($T$):** is the negation of "single range($T$)", meaning that there exists at least one object in $\text{Inst}(T)$ whose set of monads is discontiguous. Notice that the requirement is not that all objects be discontiguous; only that there exists at least one which is discontiguous.
$\exists a \in \text{Inst}(T) : \exists m \in [f(a) : l(a)] : m \notin \mu(a)$
$\equiv \neg(\forall a \in \text{Inst}(T) : \forall m \in [f(a) : l(a)] : m \in \mu(a))$
$\equiv \neg(\text{single range(T)})$

**Uniqueness constraints:**

**unique first monad($T$):** means that no two objects share the same starting monad.
$\forall a, b \in \text{Inst}(T) : a \neq b \leftrightarrow f(a) \neq f(b)$
$\equiv \forall a, b \in \text{Inst}(T) : f(a) = f(b) \leftrightarrow a = b$

**unique last monad($T$):** means that no two objects share the same ending monad.
$\forall a, b \in \text{Inst}(T) : a \neq b \leftrightarrow l(a) \neq l(b)$
$\equiv \forall a, b \in \text{Inst}(T) : l(a) = l(b) \leftrightarrow a = b$

Notice that the two need not hold at the same time.

**Table 1.** All the possible classes of object types. Legend: sm = single monad, sr = single range, mr = multiple range, ufm = unique first monad, ulm = unique last monad, ds = distinct, ol = overlapping, vb = violates borders.

| Class name | sm | sr | mr | ufm | ulm | ds | ol | vb |
|---|---|---|---|---|---|---|---|---|
| 1.000 | X | X | | | | | X | |
| 1.300 | X | X | | X | X | X | | |
| 2.000 | | X | | | | | X | |
| 2.001 | | X | | | | | X | X |
| 2.100 | | X | | | X | | X | |
| 2.101 | | X | | | X | | X | X |
| 2.200 | | X | | X | | | X | |
| 2.201 | | X | | X | | | X | X |
| 2.300 | | X | | X | X | | X | |
| 2.301 | | X | | X | X | | X | X |
| 2.310 | | X | | X | X | X | | |

| Class name | sm | sr | mr | ufm | ulm | ds | ol | vb |
|---|---|---|---|---|---|---|---|---|
| 3.000 | | | X | | | | X | |
| 3.001 | | | X | | | | X | X |
| 3.100 | | | X | | X | | X | |
| 3.101 | | | X | | X | | X | X |
| 3.200 | | | X | X | | | X | |
| 3.201 | | | X | X | | | X | X |
| 3.300 | | | X | X | X | | X | |
| 3.301 | | | X | X | X | | X | X |
| 3.310 | | | X | X | X | X | | |

**Linguistic properties:**

**distinct($T$):** means that all pairs of objects have no monads in common.
$\forall a, b \in \text{Inst}(T) : a \neq b \rightarrow \mu(a) \cap \mu(b) = \emptyset$
$\equiv \forall a, b \in \text{Inst}(T) : \mu(a) \cap \mu(b) \neq \emptyset \rightarrow a = b$

**overlapping($T$):** is the negation of distinct($T$).
$\neg(\text{distinct}(T))$
$\equiv \exists a, b \in \text{Inst}(T) : a \neq b \wedge \mu(a) \cap \mu(b) \neq \emptyset$

**violates borders($T$):** $\exists a, b \in \text{Inst}(T) : a \neq b \wedge \mu(a) \cap \mu(b) \neq \emptyset \wedge ((f(a) < f(b)) \wedge (l(a) \geq f(b)) \wedge (l(a) < l(b)))$

Notice that violates borders($T$) $\rightarrow$ overlapping($T$), since violates borders($T$) *is* overlapping($T$), with an extra, conjoined term.

It is possible to derive the precise set of possible classes of objects, based on logical analysis of the criteria presented in this section. For details, please see [9]. The possible classes are listed in Table 1.

The context resulting from these tables is then processed by the Concept Explorer software (ConExp)[3]. This produces a lattice as in Fig. 1.

## 5 Application

It is immediately noticeable from looking at Fig. 1 that "ds" is quite far down the lattice, with several parents in the lattice. It is also noticeable that "ol" is quite far up in the lattice, with only the top node as its parent. Therefore, "ds" may not be as good a candidate for a criterion on which to index as "ol". Hence, we decided to experiment with the lattice by removing the "ds" attribute.

[3] See http://conexp.sourceforge.net. Also see [10].

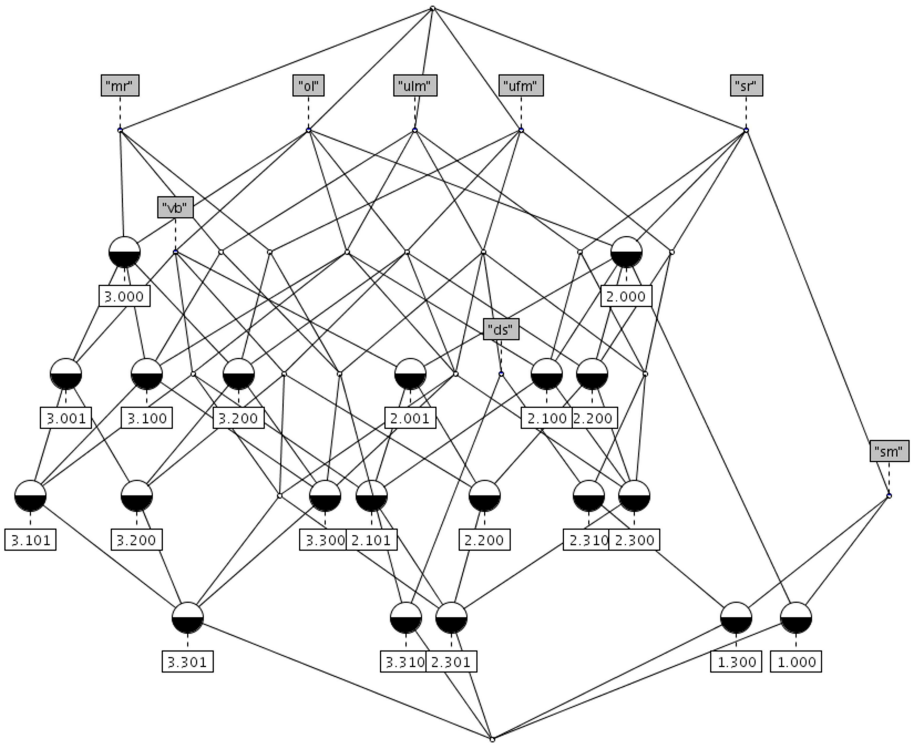



**Fig. 1.** The lattice drawn by ConExp for the whole context.

By drawing this new lattice with ConExp, it is noticeable that the only dependent attributes are “sm” and “vb”: All other attributes are at the very top of the lattice, with only the top node as their parent. This means we are getting closer to a set of criteria based on which to index sets of monads.

The three range types should definitely be accommodated in any indexing scheme. The reasons are: First, “single monad” can be stored very efficiently, namely just by storing the single monad in the monad set. Second, “single range” is also very easy to store: It is sufficient to store the first and the last monad. Third, “multiple range”, as we have argued in Sect. 2, is necessary to support in order to be able to store resumptive (discontiguous) linguistic units. It can be stored by storing the monad set itself in marshalled form, perhaps along with the first and last monads.

This leaves us with the following criteria: “unique first monad”, “unique last monad”, “overlapping”, and “violates borders” to decide upon.

In real-life linguistic databases, “unique first monads” and “unique last monads” are equally likely to be true of any given object type, in the sense that if one is true, then the other is likely also to be true, while if one is false, then the other is likely also to be false. This is because of the embedding nature of

language explained in Sect. 2: If embedding occurs at all within a single object type, then it is likely that both first and last monads are not going to be unique.

Therefore, we decided to see what happens to the lattice if we remove one of the two uniqueness criteria from the list of attributes. The criterion chosen for removal was "unique last monads". Once this is done, ConExp reports that "unique first monads" subsumes 11 objects, or 55%. This means that "unique first monads" should probably be included in the set of criteria on which to index.

Similarly, still removing "ds" and "ulm", and selecting "overlapping", we get the lattice drawn in Fig. 2. ConExp reports that "overlapping" subsumes 17 objects, or 85%, leaving only 3 objects out of 20 not subsumed by "overlapping". This indicates that "overlapping" is probably too general to be a good candidate for treating specially.

It is also noticeable that "violates borders" only subsumes 4 objects. Hence it may not be such a good candidate for a criterion to handle specially, since it is too specific in its scope.

Thus, we arrive at the following list of criteria to handle specially in the database: a) single monad; b) single range; c) multiple range; and d) unique first monads.

## 6 Implementation and evaluation

The three range types can be easily implemented in a relational database system along the lines outlined in the previous section.

The "unique first monads" criterion can be implemented in a relational database system by a "unique" constraint on the "first monad" column of a table holding the objects of a given object type. Notice that for multiple range, if we store the first monad of the monad set in a separate column from the monad set itself, this is possible for all three range types. Notice also that, if we use one row to store each object, the "first monad" column can be used as a primary key if "unique first monads" holds for the object type.

We have run some evaluation tests of 124 diverse Emdros queries against two versions of the same linguistic database, each loaded into four backends (SQLite 3, SQLite 2, PostgreSQL, and MySQL). One version of the database did not have the indexing optimizations arrived at in the previous section, whereas the other version of the database did. The version of Emdros used was 3.0.1. The hardware was a PC with an Intel Dual Core 2, 2.4GHz CPU, 7200RPM SATA-II disks, and 3GB of RAM, running Fedora Core Linux 8. The 124 queries were run twice on each database, and an average obtained by dividing by 2 the sum of the "wall time" (i.e., real time) used for all $2 \times 124$ queries. The results can be seen in Table 2.

As can be seen, the gain obtained for MySQL and PostgreSQL is almost negligible, while it is significant for the two versions of SQLite.

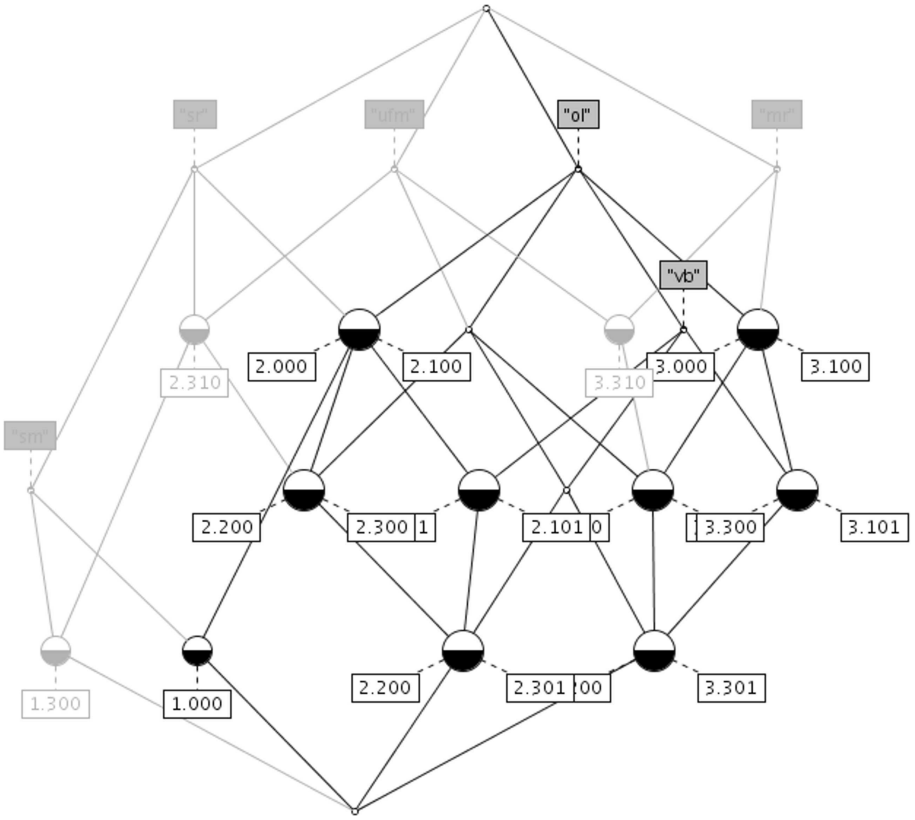



**Fig. 2.** The lattice drawn without the "ds" and "ulm" attributes, and with "ol" selected.

## 7 Conclusion

We have presented four properties that natural language possesses, namely sequence, embedding, resumption, and non-hierarchic overlap, and we have seen how these properties can be modeled as sets of durations of time.

We have presented the EMdF model of text, in which indivisible units of time (heard or read) are represented by integers, called "monads". Textual units are then seen as objects, represented by pairs $(M, F)$, where $M$ is a set of monads, and $F$ is a set of attribute-value assignments. An object type then gathers all objects with like attributes.

We have then presented some criteria which are derived from some of the four properties of language outlined above. We have formally defined these in terms of objects and their monads. We have then derived an FCA context from these criteria, which we have then converted to a lattice using the Concept Explorer Software (ConExp).

**Table 2.** Evaluation results on an Emdros database, in seconds.

| Backend | SQLite 3 | SQLite 2 | PostgreSQL | MySQL |
|---|---|---|---|---|
| Avg. time for DB **without** optimizations | 153.92 | 130.99 | 281.56 | 139.41 |
| Avg. time for DB **with** optimizations | 132.40 | 120.00 | 274.20 | 136.65 |
| Performace gain | 13.98% | 8.39% | 2.61% | 1.98% |

We have then analyzed the lattice, and have arrived at four criteria which should be treated specially in an implementation.

We have then suggested how these four criteria can be implemented in a relational database system. They are, in fact, implemented in ways similar to these suggestions in the Emdros corpus query system. We have also evaluated the performance gains obtained by implementing the four criteria.

Thus FCA has been used as a tool for reasoned selection of a number of criteria which should be treated specially in an implementation of a database system for annotated text.

Future work could also include: a) Derivation of more, pertinent criteria from the four properties of language; b) Exploration of these criteria using FCA; c) Implementation of such criteria; and d) Evaluation of any performance gains.

## References


1. Lehmann, F., Wille, R.: A triadic approach to formal concept analysis. In Ellis, G., Levinson, R., Rich, W., Sowa, J.F., eds.: Proceedings of ICCS'95. Volume 954 of LNAI., Springer Verlag (1995) 32–43
2. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1997) Translator-C. Franzke.
3. Petersen, U.: Emdros — a text database engine for analyzed or annotated text. In: Proceedings of COLING 2004. (2004) 1190–1193 http://emdros.org/petersen-emdros-COLING-2004.pdf.
4. Petersen, U.: Principles, implementation strategies, and evaluation of a corpus query system. In: Proceedings of the FSMNLP 2005. Volume 4002 of LNAI., Springer Verlag (2006)
5. Van Valin, Jr., R.D.: An introduction to Syntax. Cambridge University Press, Cambridge, U.K. (2001)
6. Horrocks, G.: Generative Grammar. Longman, London and New York (1987)
7. McCawley, J.D.: Parentheticals and discontinuous constituent structure. Linguistic Inquiry **13**(1) (1982) 91–106
8. Doedens, C.J.: Text Databases: One Database Model and Several Retrieval Languages. Editions Rodopi Amsterdam (1994) ISBN 90-5183-729-1.
9. Sandborg-Petersen, U.: Annotated Text Databases in the Context of the Kaj Munk Corpus: One database model, one query language, and several applications. PhD thesis, Aalborg University, Denmark (2008)
10. Yevtushenko, S.A.: System of data analysis "concept explorer". (in russian). In: Proceedings of the 7th national conference on Artificial Intelligence KII-2000, Russia. (2000) 127–134