Integrating Deep and Shallow Natural Language Processing Components – Representations and Hybrid Architectures

Dissertation zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften der
Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Eingereicht von Dipl.-Inform. Ulrich Schäfer

Saarbrücken, 10. Dezember 2006

Datum des Promotionskolloquiums: 29. Juni 2007

Dekan der Naturwissenschaftlich-Technischen Fakultät I (Mathematik und Informatik): Prof. Dr.-Ing. Thorsten Herfet

Vorsitzender: Prof. Dr. Andreas Zeller

Berichterstattende: Prof. Dr. Hans Uszkoreit, Prof. Dr. Wolfgang Wahlster

Akad. Mitarbeiter: Dr. Stephan Busemann

Abstract

We describe basic concepts and software architectures for the integration of shallow and deep (linguistics-based, semantics-oriented) natural language processing (NLP) components. The main goal of this novel, hybrid integration paradigm is improving robustness of deep processing. After an introduction to constraint-based natural language parsing, we give an overview of typical shallow processing tasks. We introduce XML standoff markup as an additional abstraction layer that eases integration of NLP components, and propose the use of XSLT as a standardized and efficient transformation language for online NLP integration.

In the main part of the thesis, we describe our contributions to three hybrid architecture frameworks that make use of these fundamentals. *SProUT* is a shallow system that uses elements of deep constraint-based processing, namely type hierarchy and typed feature structures. WHITEBOARD is the first hybrid architecture to integrate not only part-of-speech tagging, but also named entity recognition and topological parsing, with deep parsing. Finally, we present Heart of Gold, a middleware architecture that generalizes WHITEBOARD into various dimensions such as configurability, multilinguality and flexible processing strategies.

We describe various applications that have been implemented using the hybrid frameworks such as structured named entity recognition, information extraction, creative document authoring support, deep question analysis, as well as evaluations. In WHITEBOARD, e.g., it could be shown that shallow pre-processing increases both coverage and efficiency of deep parsing by a factor of more than two.

Heart of Gold not only forms the basis for applications that utilize semantics-oriented natural language analysis, but also constitutes a complex research instrument for experimenting with novel processing strategies combining deep and shallow methods, and eases replication and comparability of results.

Zusammenfassung (kurz)

Diese Arbeit beschreibt Grundlagen und Software-Architekturen für die Integration von flachen mit tiefen (linguistikbasierten und semantikorientierten) Verarbeitungskomponenten für natürliche Sprache. Das Hauptziel dieses neuartigen, hybriden Integrationparadigmas ist die Verbesserung der Robustheit der tiefen Verarbeitung. Nach einer Einführung in constraintbasierte Analyse natürlicher Sprache geben wir einen Überblick über typische Aufgaben flacher Sprachverarbeitungskomponenten. Wir führen XML Standoff-Markup als zusätzliche Abstraktionsebene ein, mit deren Hilfe sich Sprachverarbeitungskomponenten einfacher integrieren lassen. Ferner schlagen wir XSLT als standardisierte und effiziente Transformationssprache für die Online-Integration vor.

Im Hauptteil der Arbeit stellen wir unsere Beiträge zu drei hybriden Architekturen vor, welche auf den beschriebenen Grundlagen aufbauen. *SProUT* ist ein flaches System, das Elemente tiefer Verarbeitung wie Typhierarchie und getypte Merkmalsstrukturen nutzt. WHITEBOARD ist das erste System, welches nicht nur Part-of-speech-Tagging, sondern auch Eigennamenerkennung und flaches topologisches Parsing mit tiefer Verarbeitung kombiniert. Schließlich wird Heart of Gold vorgestellt, eine Middleware-Architektur, welche WHITEBOARD hinsichtlich verschiedener Dimensionen wie Konfigurierbarkeit, Mehrsprachigkeit und Unterstützung flexibler Verarbeitungsstrategien generalisiert.

Wir beschreiben verschiedene, mit Hilfe der hybriden Architekturen implementierte Anwendungen wie strukturierte Eigennamenerkennung, Informationsextraktion, Kreativitätsunterstützung bei der Dokumenterstellung, tiefe Frageanalyse, sowie Evaluationen. So konnte z.B. in WHITEBOARD gezeigt werden, dass durch flache Vorverarbeitung sowohl Abdeckung als auch Effizienz des tiefen Parsers mehr als verdoppelt werden.

Heart of Gold bildet nicht nur Grundlage für semantikorientierte Sprachanwendungen, sondern stellt auch eine wissenschaftliche Experimentierplattform für weitere, neuartige Kombinationsstrategien dar, welche zudem die Replizierbarkeit und Vergleichbarkeit von Ergebnissen erleichtert.

Zusammenfassung (ausführlich)

Diese Arbeit beschreibt Grundlagen und Software-Architekturen für die Integration von flachen mit tiefen, linguistikbasierten und semantikorientierten Verarbeitungskomponenten für natürliche Sprache. Das Hauptziel dieser hybriden Integration ist die Verbesserung der Robustheit der tiefen Verarbeitung.

Nach einer Übersicht in Kapitel 1 führen wir in Kapitel 2 allgemeine Begriffe wie tiefe und flache Analyse ein und geben eine Motivation für die vorliegende Arbeit. In Kapitel 3 führen wir kurz in tiefe constraintbasierte Grammatikformalismen für natürliche Sprache ein und stellen die kopfbasierte Phrasenstrukturgrammatik (head-driven phrase structure grammar; HPSG) vor. Wir geben Definitionen für getypte Merkmalsstrukturen und Unifikation an, und beschreiben informell die Arbeitsweise und Ergebnisse (semantische Analyse) eines HPSG-Parsers.

Typische Aufgaben flacher Sprachverarbeitungskomponenten wie Tokenisierung, Chunking und Eigennamenerkennung werden in Kapitel 4 beschrieben, um dann auf die Beziehung zwischen flacher Verarbeitung und Dokumentauszeichnungssprachen eingehen zu können. Wir geben einen kurzen Abriss der Geschichte von XML und SGML sowie darauf basierender linguistischer Auszeichnungsstandards wie TEI und (X)CES. Schließlich führen wir den Begriff des Standoff-Markup ein.

Kapitel 5 beginnen wir mit einer eingehenden Analyse des Flach-Tief-Integrationsproblems, um dann technische Lösungswege mit Hilfe von Markup-Anfragesprachen wie XPath, XSLT, XQuery, aber auch in der Literatur beschriebenen Anfragesprachen für linguistisch annotierte Korpora aufzuzeigen. Wir begründen unsere Wahl von XSLT als standardisierter und effizienter Transformationssprache für die Online-Integration von Sprachverarbeitungskomponenten und zeigen beispielhaft die Transformation von getypten Merkmalsstrukturen.

In Kapitel 6 motivieren wir die Notwendigkeit von Architekturen für Flach-Tief-Integration und leiten zu den drei in den Folgekapiteln beschriebenen Architektur-Frameworks über. Im Hauptteil der Arbeit stellen wir unsere Beiträge zu drei hybriden Architekturen für die Flach-Tief-Integration vor, welche auf den zuvor beschriebenen Grundlagen aufbauen.

SProUT (Kapitel 7) ist ein flaches System, das Elemente tiefer Verarbeitung wie Typhierarchie und getypte Merkmalsstrukturen nutzt. Hauptvorteil des regelbasierten Systems ist neben der flexiblen Konfigurierbeit die strukturierte Ausgabe, welche sich in Anwendungen wie Eigennamenerkennung und Informationsextraktion als vorteilhaft herausstellt.

Wir beschränken uns nach einer Einführung in *SProUT* auf vom Autor entwickelte Teile des Systems wie Formalismus-Typüberprüfung und automatische Evaluation, *SProUT* ist jedoch auch als (optionaler) Bestandteil des dritten beschriebenen Frameworks, Heart of Gold, von Bedeutung. Wir gehen näher auf die mit *SProUT* realisierte mehrsprachige Eigennamenerkennung ein und geben eine Evaluation an, welche auf dem MUC-Annotationsschema beruht und state-of-the-art-Ergebnisse zeigt, wobei die realisierten *SProUT*-Grammatiken durch ihre

strukturierte Ausgabe mehr Information bereitstellen, als durch das MUC-Schema abgebildet wird (z.B. innere Struktur von Personennamen, Zeit- und Ortsangaben).

Das Kapitel schließt mit einer Beschreibung der zahlreicher Anwendungen und Projekte in den Bereichen Informationsextraktion, Eigennamenerkennung und opionion mining, in welchen *SProUT* erfolgreich eingesetzt wurde.

WHITEBOARD (Kapitel 8) ist die erste hybride Architektur, welche nicht nur Part-of-speech-Tagging, sondern auch Eigennamenerkennung und flaches topologisches Parsing mit tiefem HPSG-Parsing kombiniert. Wir beschreiben ausführlich die beiden Ausbaustufen der Integration (zunächst part-of-speech tagging und Eigennamenerkennung, später flacher topologischer Parser mit Hilfe einer XSLT-Kaskade). In der Evaluation von WHITEBOARD konnte gezeigt werden, dass durch flache Vorverarbeitung sowohl Abdeckung als auch Effizienz des tiefen Parsers mehr als verdoppelt werden. Eine Anwendung der Architektur im Bereich hybrider Informationsextraktion wird kurz skizziert.

In Kapitel 9 schließlich wird Heart of Gold vorgestellt, eine Middleware-Architektur, welche WHITEBOARD hinsichtlich verschiedener Dimensionen wie Konfigurierbarkeit, Mehrsprachigkeit und Unterstützung flexibler Verarbeitungsstrategien generalisiert. Wir beschreiben neben der Middleware selbst auch die wichtigsten integrierten Komponenten für verschiedene Sprachen und beispielhaft Konfigurationen für robustes Parsen von deutschen, englischen und japanischen Texten.

Einen besonderen Stellenwert nimmt die neuartige Integration auf Basis des robusten Semantikformalismus RMRS ein, welcher es erlaubt, auch nach dem tiefen Parsen noch auf semantischer Ebene Informationen verschiedener Sprachverarbeitungskomponenten zu einer einheitlichen Struktur zusammen zu fügen. Heart of Gold unterstützt die RMRS-Integration optional, für *SProUT*-Eigennamen-Grammatiken wird ein Codegenerierungsverfahren vorgestellt, welches automatisch aus den deklarativen Typbeschreibungen XSLT-Code für die Laufzeit-Transformationen nach RMRS erzeugt.

Ein weiterer Abschnitt des Kapitels beschäftigt sich mit der Integration von Ontologie-Information in tiefe Satzanalysen. Hier wird ein implementierter Ansatz vorgestellt, welcher mittels XSLT in einem offline-Verfahren aus OWL-Ontologien Lingware-Resourcen für *SProUT* dergestalt erzeugt, dass im tiefen Parseergebnis Ontologieinformation (bzw. Referenzen darauf) enthalten sind.

Wir beschreiben schließlich verschiedene, mit Hilfe von Heart of Gold realisierte Anwendungen wie Kreativitätsunterstützung bei der Dokumenterstellung, automatische Email-Beantwortung im Bereich des customer relationship management und tiefe Frageanalyse bei automatischer Fragebeantwortung auf strukturierten Wissenquellen, sowie entsprechende, anwendungsbezogene Evaluationen.

Wir fassen zusammen, dass Heart of Gold aufgrund der generischen Architektur nicht nur Grundlage für semantikorientierte Sprachanwendungen bilden kann, sondern auch eine wissenschaftliche Experimentierplattform für weitere, neuartige Kombinationsstrategien darstellt, welche zudem Replizierbarkeit und Vergleichbarkeit von erzielten Ergebnissen erleichtert. Wir schließen die Arbeit mit einer Zusammenfassung in Kapitel 10 ab.

Contents

List of Tables

List of Figures

Acknowledgments

First of all, I would like to thank Hans Uszkoreit for giving me the opportunity to work in the exciting DFKI environment and the chance to continuously pursuing the topic of my thesis in challenging projects, and for contributing valuable hints and advice. I would like to thank him and Wolfgang Wahlster for their willingness to review the thesis, despite their enormous workloads.

While this thesis focuses on the computational and software aspects of linguistic component integration architectures, the integration is not limited to a software problem. Many people have worked on related aspects such as extending the deep grammars, building efficient shallow systems and extending the deep parser.

An integration task such as this at the core interface between linguistic resource development, component software development and system engineering is inherently collaborative and would have been impossible without contributions from many people. Thus, I am indebted to my present and former colleagues Markus Becker, Paul Buitelaar, Stephan Busemann, Ulrich Callmeier, Berthold Crysmann, Witold Drożdżyński, Andreas Eisele, Anette Frank, Gregory Gulrajani, Bernd Kiefer, Hans-Ulrich Krieger, Günter Neumann, Jakub Piskorski, Stefania Racioppa, Oliver Scherf, Melanie Siegel, Jörg Steffen, and Feiyu Xu for fruitful cooperation, helpful discussions and implementation work.

I would also like to thank our 'Hiwis' Robert Barbey, Daniel Beck, Özgür Demir, Andreas Freier, Thomas Klöcker, and Kathrin Spreyer for their efficient and intelligent contributions.

I am grateful to several members of the international DELPH-IN collaboration, most notably Stephan Oepen, Dan Flickinger, Ann Copestake, Ben Waldron, and Francis Bond. I am grateful for valuable comments and interesting discussions at various meetings in Saarbrücken, Edinburgh, Lisbon, Jerez de la Frontera, and Fefor (Norway).

I would also like to thank Thierry Declerck for letting me dive a bit into the world of the ISO standardization processes, and related discussions.

Many anonymous reviewers and the audiences at the following workshops and conferences have contributed valuable comments and feedback on papers I have presented and on which parts of this thesis are based: HLT-NAACL-2003 workshop on Software Engineering and Architectures for Language Technology Systems, Edmonton, Canada. ACL-2003 conference, Sapporo, Japan. LREC-2004 conference and workshop on a Registry of Linguistic Data Categories within

an Integrated Language Resources Repository Area, Lisbon, Portugal. ESSLLI-2004 workshop on Combining Shallow and Deep Processing for Natural Language Processing, Nancy, France. AAAI-2005 workshop on Question Answering in Restricted Domains, Pittsburgh, Pennsylvania. EACL-2006 workshop on Multi-Dimensional Markup in Natural Language Processing, Trento, Italy. LREC-2006 main conference and workshop on Quality Assurance and Quality Measurement for Language and Speech Resources, Genoa, Italy. ELSNET-2006 Summer School on Information Fusion in Natural Language Processing Systems, Hamburg, Germany.

Finally, many thanks go to my parents whom I owe so much and to my family for their patience.

Chapter 1

Introduction

In this thesis, we describe our contributions to a new paradigm for hybrid natural language processing (NLP) architectures. The idea is to combine pre-existing shallow language technology components and a deep linguistic parser by using XML technology. The main goal is to increase robustness of application-oriented deep linguistic parsing. Further possible advantages of exploiting synergy gained through NLP component combination is reduced ambiguity and increased performance of deep parsing.

The resulting overall systems can form the basis for a new generation of applications that build on machine understanding of human language which can – to a certain extent and by means of a formal semantics representation language – be provided by deep linguistic parsing.

The novel applications may comprise basic techniques such as textual entailment, natural language question interpretation and answering, advanced information extraction, but also more complex tasks such as opinion mining, creative authoring support, business intelligence and further Semantic Web-related challenges.

Our thesis focuses on three architecture frameworks. The first one, *SProUT*, is shallow with respect to its basic processing model, but integrates deep and shallow elements on the formalism level. It provides generic XML input and output interfaces and can be used for many different (shallow) NLP tasks.

The second framework, WHITEBOARD, is an API-based sequential architecture that uses XML and XSLT technology at numerous levels of shallow processing that are input to a deep parser. It has been mainly used to research, evaluate and demonstrate the feasibility and benefits of intensely interleaved shallow processing and deep parsing.

The third framework, Heart of Gold, is a generalization of WHITEBOARD with respect to many dimensions such as processing models, multilinguality, configurability and networkability. Moreover, a new integration layer, based on a robust semantics formalism that allows for underspecified representations from various deep and shallow NLP components, is supported.

Heart of Gold forms a middleware architecture for a wide range of applications related to the Semantic Web, and also (optionally) incorporates and builds on the first framework, *SProUT*, in many of the realized multilingual integration scenarios.

Finally, Heart of Gold not only forms the basis for applications that make use semantics-oriented natural language analysis, but also constitutes a complex research instrument for experimenting with novel processing strategies combining deep and shallow methods that at the same time eases replication and comparability of results.

After thorough introduction of the processing and representation paradigms for deep and shallow analysis, we motivate our idea of using XML transformation for flexible integration of NLP components. Then, we describe each of the three frameworks, their use in implemented applications, and evaluations in application contexts such as information extraction, creative authoring support and deep question analysis for question answering.

The overall result of our thesis is that the implemented hybrid architectures for combining deep and shallow processing help to drastically increase the robustness of deep linguistic parsing for use in NLP application contexts, and that utilizing XML and XSLT technology eases flexible integration and combination of NLP components.

The thesis is structured as follows.

In Chapter 2, we begin with some general definitions and present the motivations for the thesis. Chapter 3 briefly and informally introduces the HPSG grammar formalism for deep natural language processing. Chapter 4 introduces shallow processing, linguistic XML representations and the relation between them. In Chapter 5, we introduce general problems of hybrid deep-shallow integration and propose a solution based on XML transformation. Chapter 6 explains and motivates the need for architectures for application-oriented hybrid processing.

In Chapter 7, we describe our contributions to the *SProUT* platform that combines elements from deep and shallow NLP paradigms on the formalism level. Chapter 8 discusses WHITEBOARD, an architecture for hybrid parsing that has been developed to research and evaluate the possible benefits of interleaved processing at multiple levels of language technology. In Chapter 9, the Heart of Gold middleware architecture for robust, multilingual application-oriented deep-shallow integration as well implemented applications are presented. Finally, we conclude in Chapter 10.

Appendix A contains some of the most important document type definitions discussed in the thesis (Chapters 4 through 9). Appendix B displays sample XSLT stylesheets discussed in Chapter 9.

All HTTP links referred to in this thesis (including those mentioned in the bibliography) have been verified on September 6, 2006.

Chapter 2

Definitions and Motivation

In this chapter, we briefly introduce deep and shallow natural language processing and present an application-oriented description of the tasks and problems that are addressed in our thesis. We then motivate the integration of both processing paradigms and the possible benefits of this combination.

2.1 Deep and Shallow Natural Language Processing

2.1.1 Deep Natural Language Processing

Deep natural language processing (DNLP) systems try to apply as much linguistic knowledge as possible to analyze natural language utterances. The linguistic knowledge is declaratively encoded. The general term used in computer science is information-based, knowledge representation-based or constraint-based processing, as the knowledge about natural language is neither encoded in algorithms nor in simple databases. Instead, the language knowledge is separated from the (at least in principle) quite simple algorithms[1] in a formal grammar with underlying (type) theory and well-defined information fusing and consistency checking operations.

The analysis result of the natural languages utterances (typically sentence-wise) contains a collection of the knowledge that successfully contributed to the analysis. The result often consists of many possible analyses per sentence reflecting the uncertainty which of the syntactically possible readings was intended – or a rejection (failure) of the input if the linguistic knowledge was contradictory or insufficient with respect to the input. DNLP systems generally are rule-based[2].

Rules describe constraints on the correct composition of linguistic entities (syntax) based on a linguistic grammar theory, but abstract from concrete words which

[1] However, a considerable amount of the complexity of the implemented systems comes from sophisticated methods making the processing efficient using e.g. compilation, optimization and statistical methods *etc.*

[2] This does not mean that deep is the opposite of statistical, because statistical methods can well be and are applied successfully to deep grammars and systems.

are encoded in a lexicon. Using syntactic analysis of a deep grammar alone (without semantics construction) can form the basis for applications such as *grammar checking*, where correct use of natural language syntax is either approved or a sentence is rejected as ungrammatical[3].

On the basis of the syntactic analysis, rules can also describe the *compositional construction of a natural language semantics representation* of the meaning of a sentence. Throughout this thesis, we always mean by *semantics representation* a natural language semantics representation, an abstract, simplified language describing in a *logical form* the meaning of a syntactically represented natural language utterance, e.g. based on first order predicate logic such as the following clause for the sentence *John gave Mary a book*.

past(give(John, Mary, book))

The term *deep structure* was coined by Chomsky (1965), describing the theoretical construct underlying several possibly similar *surface forms*, to abstract e.g. from syntactic variants such as 'John gave Mary the book' or 'The book was given to Mary by John' bearing essentially the same meaning. Chomsky later replaced the term *deep structure* by *logical form* and *surface form* by *phonetic form*. However, one could think of *deep analysis* as of an algorithm that computes a *deep structure*.

The constructed semantics representation can then be used in application-specific actions such as relation extraction, text summarization, question answering, opinion detection, command interpretation[4] *etc*. Moreover, as semantics representation ideally would be language-independent (a kind of neutral *interlingua*, the pseudo-English relation names starting with c_ in the example), it could also form the basis for machine translation, i.e., the generation of a syntactic surface in the target language.

In an ideal multilingual DNLP setting, syntax and lexicon are language-specific, while the semantics representation abstracts from the syntactic surface up to a certain extent. A simple example may illustrate the idea for the sentence

The farmer reads the newspaper.

is e.g.

$$\text{c\_farmer}(y) \wedge \text{c\_newspaper}(z) \wedge \text{c\_transitive\_read}(x,y,z) \wedge \text{c\_tense\_present}(x)$$

The same semantic representation (modulo language-specific concepts) would also hold for the French sentence

Le paysan lit le journal.

and the German sentence

[3]However, also for grammar checking, semantics can be beneficial, e.g. for (at least partial) disambiguation.

[4]e.g. translation of natural language commands or questions into SQL queries on databases.

Der Bauer liest die Zeitung.

However, this is a gross simplification as the isomorphism may neither hold for lexical semantics[5] nor for e.g. more complex verbal constructions as shown in Copestake *et al.* (1995a). By *semantic transfer* (translating concepts and more complex constructions *etc*), an equivalent representation in the target language would have to be computed in the machine translation scenario.

Ideally, the same grammar for a specific language can be used for analysis and generation, cf. Shieber (1988); Shieber *et al.* (1990); van Noord (1993); Neumann (1994). In any case, due to ambiguity and richness of human language, multiple semantic representations may be valid for a sentence and *vice versa* – there may be syntactic variants for the same semantic representation.

The knowledge-intensive approach of DNLP requires considerable computational power, and has in the past sometimes been judged as being intractable[6]. However, research on improving efficiency of deep processing has made considerable advances during the last years (Callmeier, 2000; Uszkoreit, 2002), and today, efficiency is no longer a major problem for applications using deep processing.

2.1.2 Shallow Natural Language Processing

Unlike DNLP, shallow natural language processing (SNLP[7]) systems do not attempt to achieve an exhaustive linguistic analysis. They are designed for specific tasks ignoring many details in input and linguistic (grammar) framework.

Utilizing rule-based (e.g. regular grammars) or statistics-based approaches, they are in general faster than DNLP, but only deliver flat, simple, partial, non-exhaustive representations. Examples for dedicated shallow processing stages (in the order of increasing complexity) are e.g. tokenization, part-of-speech tagging, chunking, named entity recognition, and shallow sentence parsing. The purpose of these shallow tasks will be described below (Section 2.3).

Due to the initial lack of efficiency and robustness of DNLP systems, the trend in application-oriented language technology in the last years was to improve SNLP systems. They are now capable of analyzing megabytes of texts within seconds, but precision and quality barriers are so obvious (especially on domains the systems where not designed for or trained on) that a need for 'deeper' systems re-emerged (Uszkoreit, 2002). Moreover, semantics construction of SNLP from an input sentence is quite poor and imprecise in typical shallow systems, and in many cases insufficient for the above mentioned NLP applications[8].

[5] The standard example being the German word *Bank* which has multiple meanings, e.g. financial institution, but also bench.

[6] Cf. Chapter 3 for a discussion of theoretical results.

[7] The abbreviation SNLP is sometimes also used for *statistical natural language processing*, e.g. in Callison-Burch and Osborne (2003). Although there is a big overlap, namely *shallow statistical language processing*, the term statistical NLP excludes e.g. rule-based approaches that are comprised by shallow NLP such as finite-state processing.

[8] Therefore, the somewhat derogative term sometimes used for the result of shallow processing is

2.2 Integration of the Paradigms

A promising approach to improve the quality of natural language text analysis is the combination of deep and shallow processing technologies. Deep processing could benefit from specialized and fast shallow analysis results and fill its 'knowledge gaps', e.g. in the lexicon, to increase robustness.

When DNLP returns too many readings for a sentence, statistics-based SNLP components could help to select the most probable reading(s). Moreover, a combinatorial explosion of the search space in long sentences that is intrinsic for DNLP could be avoided by filtering with the help of shallow analyses.

The integration of deep and shallow NLP systems is also economically motivated. Because of expensive, time-consuming grammar development, most deep systems tend to be domain-independent (describing common properties, phenomena and syntactic constructions of a language), re-usable, and modular at least in parts (cf. Bender *et al.* 2003; Pollard and Sag 1994) across languages. However, lexical and semantic coverage on specific domains (e.g. the medical domain) is limited by the (hand-crafted) lexicon.

Shallow components could help to bridge this gap and establish the following task sharing. Domain-specific extensions could be contributed by shallow systems in conjunction with generic entries in the deep lexicon that are filled according to the input text, while the core linguistic and grammar theory-based knowledge stems from DNLP.

Because human language is productive (compound words, proper names, new word formation, product names) and mutating, and words may also be misspelled, there may be always in a general natural language text words that can not even be found in a huge lexicon. Shallow taggers can guess the word types of such unknown words in a text, and assign an appropriate generic lexicon entry with default values as fall-back information.

Deep linguistic analysis is only one stage in the process of language understanding. Full ambiguity resolution further requires discourse or context information, world and domain-specific knowledge, and also semantic inference. These tasks are left to application-specific processing and are out of the scope of this thesis.

The prospective added value of integrating deep and shallow processing is a more robust semantic analysis of text than is possible with standalone deep parsers or shallow processors. The thesis centers around the representation languages and architectures that are necessary to perform the integration task.

Table 2.1 coarsely sums up *typical* properties of deep and shallow NLP. Of course, the classifications are simplifying, and there are exceptions or cases where the distinctions are fuzzy.

Resulting from this table, one can identify various challenges for integration architectures of deep and shallow systems, most of which we will describe in more

low-level markup.

Property	Deep NLP	Shallow NLP
Processing cost	high	low
Coverage	low	high
Behavior on ill-formed input	not robust	robust
Ambiguity rate	high	low
Syntactic analysis	fine-grained	coarse-grained
Generated semantics representation	rich	poor
Precision	high	low
Partial analysis	(infrequent)	frequent

Table 2.1: Typical properties of deep and shallow NLP

detail in Chapter 5.

The challenges are on the one hand linguistic, such as the different paradigms (partial vs. exhaustive analysis, filtering vs. monotonic enrichment, different granularity, namings, types, spans for a recognized entity, and the question how to resolve ambiguity). On the other hand, there are technical challenges for the integration such as the fact that one would necessarily have to build on existing components implemented in various programming languages, the problem of online vs. offline integration, and finally the questions whether experimental or application-oriented use and flexibility or efficiency should be in the foreground.

In search of an architecture that integrates deep and shallow NLP, reading and citing the seminal work of Cunningham (2000) about *Software Architecture for Language Engineering* is a must.

However, although his approach claims to be somewhat universal and to cover any kind of NLP, it only covers shallow language technology and mentions deep processing, typed feature structures and HPSG on a single page (108) and in no way attacks the specific problem for this important class of NLP.

Therefore, our thesis is at large parts complementary to Cunningham (2000) and tries to attack the problems left unmentioned and unsolved there for deep-shallow integration.

2.3 The Benefits of Robust Deep Processing Enhanced by Shallow Processing

Nowadays, shallow natural language processing (SNLP) tasks are performed through statistical or simple rule-based, typically finite-state methods, often supported by table or database lookups, with sufficient precision and recall. These tasks are e.g.

- *Tokenization* is the task of separating words and e.g. punctuation symbols in a way appropriate for the subsequent tasks below. It is typically handled by

regular expressions, and can range from very simple distinctions (e.g. word vs. punctuation symbol) up to quite fine-grained differentiation of token classes. It may also include the task of *sentence boundary detection* which potentially is a hard problem (e.g. abbreviations with dots within or at the end of a sentence), therefore requiring NER (cf. below) or even parsing as well.

- *Part-of-speech tagging (PoS tagging)* is the task of determining the class of a word, such as noun, pronoun, verb, adjective, adverb *etc*. Again, more or less fine-grained classifications exist.

- *Morphological analysis/stemming* is the task of decomposing a inflected word into its stem and suffix and/or prefix, even if the stem's surface is modified (e.g. to be - I am - I was - they were), and enumerating the inflection feature value for e.g. case, gender, number, person. This task may also include decomposition of compound words like in German or Dutch.

- *Named entity recognition* (NER) is the task of determining the type and boundary of name words such as proper names, location names, time expressions, possibly including subtypes such as surname, given name, country, city name. As these classes of words are very productive, and cannot be captured by lexical enumerations, they are also called *open class words*.

- *Phrase chunking* is the task of segmenting a text into information units larger than a word and possibly comprising other chunks, e.g. 'the white horse' forms a noun phrase.

- *Shallow parsing* is the task of analyzing sentence structure by e.g. a probabilistic context-free model (PCFG parsing). This task is similar to chunking, but on sentence level, and may build on chunking.

Some of the state-of-the-art language technology components of this 'shallow' kind are included in every-day applications such as word processors, word-based search engines, full text document indexing *etc*.

Common to the performance of the above mentioned tasks is their imperfection that is mainly due to insufficient resources (unknown words, domain-specific expressions *etc*), ambiguity in natural language, unawareness of context, and the inability to draw inferences and to use world knowledge.

However, despite these deficiencies, the described tasks are performed typically with precision and recall between 80 and 99 % which is considered acceptable and comparable to human performance which is also not perfect indeed (and ignoring the fact that humans need much longer to perform the tasks).

The judgment on quality of computerized NLP deteriorates when more complex tasks are considered. Such tasks include

- *Information extraction*: from texts such a newspaper articles on management position changes (succession) fill in expected template slots, e.g. name, company, former position, new position *etc*.
- *Text summarization*: produce in a couple of well-formed sentences an outline of the relevant information that a larger text contains.
- *Document or information retrieval*: return from a document collection those documents that contain relevant information according to a description (keywords in the trivial case).
- *Text mining*: try to discover e.g. trends from text. Although the similarity to the term 'data mining' suggests that new facts (factoids) are discovered, the widely accepted scope of text mining is restricted to simpler, more specific and specified tasks, cf. Hearst (1999).
- *Question answering*: find a correct answer (e.g. a single sentence, or summarizing in a few sentences) in a local text repository or on the world-wide web to a question a user formulates in natural language. QA is typically be based on document retrieval.
- *Opinion mining*: extract meanings uttered by people, e.g. judgments on a product in a web forum or a newsgroup.
- *Machine translation*: translate a text into a different language while preserving the meaning and style of the original.
- *Textual entailment recognition*: this is the somewhat artificial, but nevertheless very interesting task of deciding, given two text fragments, whether the meaning of one text is entailed (can be inferred) from another text. It has applications in information extraction, information retrieval, question answering and machine translation.

Further advanced applications may use combinations of these basic tasks, e.g. *Scam seeking*, the task of finding documents that contain faked information such as illegal investment proposals in financial news (Patrick, 2004).

While humans still need a considerable amount of time to solve these tasks, they can (approximately) do it with precision and recall comparable to those on the the above mentioned low-level tasks.

However, the fraction of acceptable solutions delivered by current NLP systems is *much* lower than for the low-level tasks. One reason is that e.g. simple language models such as trigrams or simple finite-state models are obviously inadequate (as more than just local information has to be considered) and cannot capture the complexity of the task and the necessarily involved data, e.g. information extraction can be very hard, depending on the specific task formulation.

Another reason is that training data and corpora are not available for many domains and applications, and are expensive to acquire. But even if huge data

collections would be used, as was the case for brute-force table-based machine translation experiments in recent years, the results would still be unacceptable.

A further reason why the shallow methods fail is the lack of 'understanding' a text. While humans can easily detect the agent of an action described in a sentence even if it is not mentioned explicitly, or the scope of a negation intended by the author, this is not possible so reliably with shallow methods.

Consider only the probably easiest of the above mentioned complex tasks, namely information extraction. In a sentence such as[9]

Things would be different if Microsoft was located in Georgia,

a shallow IE system could – wrongly – infer the information that *Microsoft's corporate headquarters* were *located in Georgia*.
In the example

The National Institute for Psychobiology in Israel was established in May 1971 as the Israel Center for Psychobiology by Prof. Joel,

a shallow IE system using patterns of regular expressions could – wrongly – infer that Israel was established in May 1971.

Many problems of this kind could be overcome by thorough linguistic analysis. Although for further disambiguation, world knowledge and inference is necessary, deep linguistic processing is quite reliably able to syntactically rule out impossible interpretations, to identify the arguments of a verb, to determine the scope of negation, and – most importantly – to compute a semantic representation of the meaning of a sentence.

However, this does not imply that deep linguistic processing always computes 'the' meaning of a sentence. In general, it computes a set of possible meanings that are licensed by the syntax of the language, and that are constrained by word-specific information from a lexicon.

This set is thus a superset of the interpretations a human would draw, because humans make use of world knowledge, and context (discourse) information, that a linguistics-based parser may not have. For a further improved language analysis, it is therefore necessary to adjoin to the (after the) linguistic analysis a model of world knowledge (usually called 'ontology') and further information resources such as discourse model or context.

It has to be pointed out that this scenario in no way claims to be psycholinguistically adequate nor does it model how language is processed in human brain. It is only a model of language and reflects linguistic knowledge and how it could be made processable for natural language applications.

Finally, although we will in this thesis describe some interesting and successful applications of integrated deep and shallow processing (also in conjunction with

[9]The two examples are taken from the Recognizing Textual Entailment Challenge (`http://www.pascal-network.org/Challenges/RTE/`).

ontologies, especially domain-specific ontologies), we do not try to model and implement the *whole* process just described. In particular, we will not say anything about modeling discourse, context or world knowledge, let alone semantic inference. These will be subsequent steps in the processing chain not covered by this thesis. In this sense, integrated deep and shallow processing is only a–however important–initial building block for linguistics-oriented natural language analysis that has mainly increased robustness in focus.

Chapter 3

Deep Linguistic Processing with Head-Driven Phrase Structure Grammar

In this chapter, we give a short introduction to deep linguistic processing on the basis of the Head-driven Phrase Structure Grammar (HPSG). We briefly introduce the main concepts such as type system, typed feature structures, subsumption and unification, phrase structure rules, lexicon entries and semantics output. We conclude with a discussion of the robustness problem of deep processing.

3.1 A Short Introduction to HPSG

The two most important and most elaborated grammar theories that play a major role in computational linguistics today are Lexical Functional Grammar (LFG; Kaplan and Bresnan 1982) and Head-Driven Phrase Structure Grammar (HPSG; Pollard and Sag 1987, 1994). Although many of the solutions described in this thesis could probably be applied to other grammar theories as well, we will concentrate on HPSG here because it is probably the most elegant, popular and predominant grammar theory (Richter, 2000; Kirby, 1996).

HPSG is a blend of logic, inheritance-based *knowledge representation* (Brachman, 1979), type theory and linguistic theory that not only makes it suitable for computer implementation but also gives the computational linguist (and grammar writer) a powerful, well-defined and uniform representation language for the encoding of linguistic knowledge. Several development platforms exist that allow to write and test HPSG grammars (Copestake, 2002; Uszkoreit *et al.*, 1994).

However, HPSG is not a fixed theory. Its development and research is still in progress, but mostly only details are changed or added, not the general setup. Hence, HPSG implementations are also playground for research on and experimentation with formalized linguistic theories or hypotheses of language modeling (e.g. in natural language syntax, syntax-semantic interface).

HPSG is a *lexicalized* grammar theory, i.e., most linguistic knowledge is encoded in lexicon entries. Only a relatively small number of very general and partly even language-independent rules and principles exist that define general information flow and combination within a sentence.

Inheritance is used to ensure that the knowledge is not encoded redundantly. HPSG is *sign-based* following the ideas of de Saussure (1916), i.e., (syntactic, phonetic) form and meaning are represented as *ensemble*. HPSG is *monostratal* in the sense that the same (data) structures are use to describe phonology, syntax and semantics.

HPSG deliberately incorporates ideas from Categorial Grammar (CG; Wood 1993), Discourse Representation Theory (DRT; Kamp and Reyle 1993), Lexical-Functional Grammar (LFG; Kaplan and Bresnan 1982), Generalized Phrase Structure Grammar (GPSG; Gazdar *et al.* 1985), Government and Binding Theory (GB; Chomsky 1981). However, it supersedes them in elegantly using solely typed feature structures to implement linguistic concepts originally developed in its predecessors rather than (as they did) presenting informal accounts of how linguistic phenomena could be modeled within the theory.

In HPSG, both rules and lexicon are encoded in a uniform, well-defined data structure called *typed feature structure* which is based on an type hierarchy with inheritance (Figure 3.1), with a monotonic information-combining and consistency-checking operation called unification. An HPSG parser program basically performs unification of constraints encoded in typed feature structures imposed by the grammar and the lexicon entries triggered by the input sentence (the primary domain of an HPSG parser is a sentence).

3.1.1 Excursus: Typed Feature Structures

Typed feature structures are commonly considered an appropriate, declarative vehicle with which ‘many technical problems in language description and computer manipulation of language can be solved’ (Shieber, 1992).

Feature structures are partial descriptions of (linguistic) objects. The original term *feature* is related to an attribute with a binary value for a linguistic property such as VOICED: +. Later, the binary values have been generalized to arbitrary atomic values, types and to recursively embedded values, i.e. attributes in a typed feature structure may have typed feature structures as values and so forth. This is why *attribute-value pairs* may be considered the correct term, although *typed feature structure* is the term that is established in the literature.

The idea and logic of typed feature structures are presented and clarified e.g. in Pereira and Shieber (1984); Shieber (1986); Carpenter (1992). Viewing conjunctive sets of feature-value pairs as *constraints* immediately reveals similarity with constraint logic programming (CLP, Jaffar and Lassez 1987). In fact, numerous insights and also implementation techniques have been influenced by and are shared with CLP.

Many valuable insights on complexity, relation to other knowledge representation languages *etc* came from Smolka (1989); Kasper and Rounds (1990). Rounds and Manaster-Ramer (1987) were the first to prove that feature logic with recursive types is undecidable. Smolka (1989) showed that this is due to the coreference constraints.

Aït-Kaci and Nasr (1986) and Nebel and Smolka (1990) have also clarified the relation to other knowledge representation formalisms such as KL-ONE (Brachman and Schmolze, 1985). In particular, they showed that a set-theoretical semantics can be given to typed feature structures in analogy to the set-theoretical semantics of KL-ONE.

There are a couple of different characterizations of typed feature structures. Krieger (1995) gives a good overview. The following definitions are close to those presented in Carpenter (1992), though slightly simplified and abbreviated.

In the sample of an HPSG type hierarchy shown in Figure 3.1, *sign* is the top type that divides into *word*s and *phrase*s which themselves fall into structures with and without a linguistic *head*. *Head*s will be explained in Section 3.1.2.4.

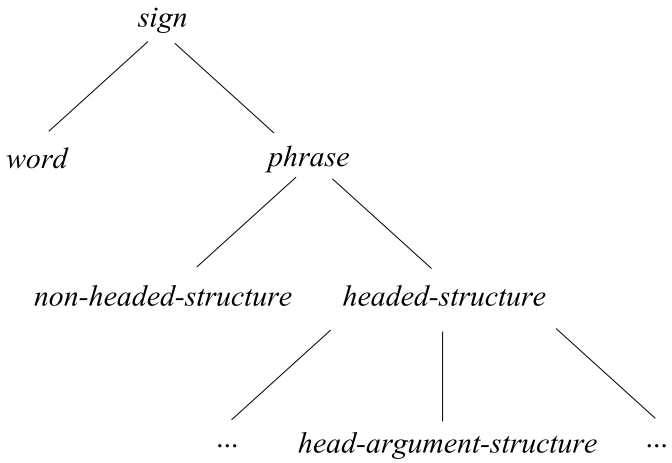


Figure 3.1: A small extract of the HPSG type hierarchy

3.1.1.1 Definition Typed Feature Structure Grammar

An *HPSG grammar* is a tuple $\langle T, \sqsupseteq, F, \Theta \rangle$ with

- $\langle T, \sqsupseteq \rangle$ constituting an inheritance hierarchy which can be characterized as a finite Bounded Complete Partial Order (BCPO, or finite semi-lattice)
- F, a set of feature symbols
- a set of typed feature structures Θ being partitioned into lexical entries and rules

3.1.1.2 Definition Typed Feature Structure

A typed feature structure $\theta \in \Theta$ over T and F is a rooted, directed, labeled graph. It is defined by a tuple $\theta = \langle Q, q_0, \tau, \delta \rangle$, where

- Q is a finite set of nodes rooted at q_0
- $q_0 \in Q$ is the root node
- $\tau : Q \rightarrow T$ is a total typing function assigning a type to each node in the feature structure
- $\delta : F \times Q \rightarrow Q$ is a partial feature value function assigning a value to a feature

A node without outgoing feature arcs is called an *atom*. An example of a feature structure depicted as graph is shown in Figure 3.2.

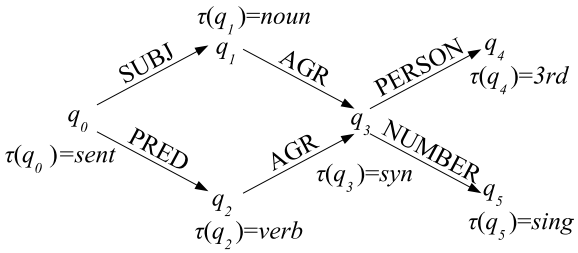


Figure 3.2: A typed feature structure as graph

A more convenient and better readable representation of typed feature structures is called an *AVM*, an *attribute-value matrix* (being *attribute* a synonym for a *feature*). A bunch of feature arcs outgoing from a typed node is visualized in stretched brackets, the type is written above the features:

$$\left[\begin{array}{ll} \textit{type0} & \\ \textsc{feature}1 & \textit{type1} \\ \textsc{feature}2 & \left[\begin{array}{ll} \textit{type2} & \\ \textsc{feature}3 & \textit{type3} \end{array}\right] \end{array}\right]$$

Features within typed feature structures may share values. This structure sharing (or reentrancy) is indicated by a labeled (or numbered) box, also called a coreference, the shared value is printed next to the first occurrence of the coreference box:

$$\begin{bmatrix} \textit{sent} & & \\ \text{SUBJ} & \begin{bmatrix} \textit{noun} & & \\ \text{AGR} & \boxed{1} & \begin{bmatrix} \textit{syn} & \\ \text{PERSON} & \textit{3rd} \\ \text{NUMBER} & \textit{sing} \end{bmatrix} \end{bmatrix} \\ \text{PRED} & \begin{bmatrix} \textit{verb} & \\ \text{AGR} & \boxed{1} \end{bmatrix} \end{bmatrix}$$

In the above figure, the value under the feature *path* SUBJ.AGR is shared with the value under PRED.AGR.

Throughout this thesis, we may additionally use the $\langle \theta_1, \theta_2, \ldots, \theta_n \rangle$ notation for list-valued feature nodes. A list-valued feature node can be conceived as an abbreviation for a feature structure of type **cons** (non-empty list) with attributes FIRST and REST, where the value of FIRST contains a list element, and REST contains the rest of the list. The featureless type **null** indicates an empty list (and hence list end under feature REST). Thus $\langle \theta_1, \theta_2, \theta_3 \rangle$ is an abbreviation for

$$\begin{bmatrix} \textit{*cons*} & & \\ \text{FIRST} & \theta_1 & \\ \text{REST} & \begin{bmatrix} \textit{*cons*} & \\ \text{FIRST} & \theta_2 \\ \text{REST} & \begin{bmatrix} \textit{*cons*} & \\ \text{FIRST} & \theta_3 \\ \text{REST} & \textit{*null*} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

3.1.1.3 Definition Subsumption

Subsumption ($\sqsupseteq$) between typed feature structures is defined as a transitive, anti-symmetric and reflexive relation between two typed feature structures that states whether one feature structure is more general than the other.

$\theta = \langle Q, q_0, \tau, \delta \rangle$ subsumes $\theta' = \langle Q', q'_0, \tau', \delta' \rangle$, short $\theta \sqsupseteq \theta'$ if and only if there is a total function $h : Q \rightarrow Q'$ such that:

- $h(q_0) = q'_0$
- $\tau(q) \sqsupseteq \tau'(h(q)) \; \forall q \in Q$
- $h(\delta(f,q)) = \delta'(f, h(q)) \; \forall q \in Q$ and for every feature $f \in F$ for which $\delta(f,q)$ is defined

Informally, $\theta \sqsupseteq \theta'$ iff θ is more general than θ'.

Set-theoretically, the denotation of subsumption of two typed feature structures corresponds to the subset relation of the denotations of the typed feature structures.

Examples for proper subsumptions are:

$$\begin{bmatrix} agr \\ \text{PERSON} \end{bmatrix} \sqsupseteq \begin{bmatrix} agr \\ \text{PERSON} \quad 1st \end{bmatrix}$$

$$\begin{bmatrix} agr \\ \text{PERSON} \quad 1st \end{bmatrix} \sqsupseteq \begin{bmatrix} agr \\ \text{PERSON} \quad 1st \\ \text{NUMBER} \quad sing \end{bmatrix}$$

$$\begin{bmatrix} sign \\ \text{AGR} \quad \begin{bmatrix} agr \\ \text{PERSON} \quad 1st \end{bmatrix} \end{bmatrix} \sqsupseteq \begin{bmatrix} phrase \\ \text{AGR} \quad \begin{bmatrix} agr \\ \text{PERSON} \quad 1st \\ \text{NUMBER} \quad sing \end{bmatrix} \end{bmatrix}, \text{ if } sign \sqsupseteq phrase$$

$$\begin{bmatrix} sent \\ \text{SUBJ} \quad \begin{bmatrix} agr \\ \text{PERSON} \quad 1st \end{bmatrix} \\ \text{PRED} \quad \begin{bmatrix} agr \\ \text{PERSON} \quad 1st \end{bmatrix} \end{bmatrix} \sqsupseteq \begin{bmatrix} sent \\ \text{SUBJ} \quad \boxed{1} \begin{bmatrix} agr \\ \text{PERSON} \quad 1st \end{bmatrix} \\ \text{PRED} \quad \boxed{1} \end{bmatrix}$$

3.1.1.4 Definition Unification

Unification of two typed feature structures $\theta \sqcap \theta'$ is then defined as the operation determining the most general feature structure that is subsumed by both θ and θ'. The result of unification is a failure ($\perp$) if θ and θ' contain contradictory information, e.g. by incompatible types under the same feature paths.

In other words, unification is both a satisfiability-checking and structure building operation. The former is important to rule out impossible (contradictory) descriptions, the latter is e.g. used to build sentence structure and semantics output compositionally from a syntactic analysis during parsing. Formal definitions for unification are presented e.g. in Carpenter (1992), Chapter 3[1].

Set-theoretically, the denotation of unification of two feature structures corresponds to the set intersection of the denotations of the feature structures.

$$\begin{bmatrix} agr \\ \text{PERSON} \quad 1st \end{bmatrix} \sqcap \begin{bmatrix} agr \\ \text{NUMBER} \quad sing \end{bmatrix} = \begin{bmatrix} agr \\ \text{PERSON} \quad 1st \\ \text{NUMBER} \quad sing \end{bmatrix}$$

[1] Kay (1979) first introduced feature structure unification.

$$\begin{bmatrix} sent \\ \text{SUBJ} \ \boxed{1} \\ \text{PRED} \ \boxed{1} \end{bmatrix} \sqcap \begin{bmatrix} sign \\ \text{PRED} \ \begin{bmatrix} agr \\ \text{PERSON} \ 1st \end{bmatrix} \end{bmatrix} = \begin{bmatrix} sent \\ \text{SUBJ} \ \boxed{1} \begin{bmatrix} agr \\ \text{PERSON} \ 1st \end{bmatrix} \\ \text{PRED} \ \boxed{1} \end{bmatrix},$$

if $sign \sqsupseteq sent$

$$\begin{bmatrix} agr \\ \text{PERSON} \ 1st \\ \text{NUMBER} \ sing \end{bmatrix} \sqcap \begin{bmatrix} agr \\ \text{PERSON} \ 2nd \\ \text{NUMBER} \ sing \end{bmatrix} = \bot,$$

if *1st* and *2nd* are incompatible (atomic) types

Unification algorithms have been presented e.g. in Aït-Kaci (1984) and Karttunen and Kay (1985), many improvements and variations have been published, e.g. by Kogure (1990); Godden (1990); Emele (1991); Tomabechi (1991, 1992); Wroblewski (1987); Kiefer *et al.* (1999); Malouf *et al.* (2000); van Lohuizen (2000). A good overview (of the graph-based ones) is presented in Callmeier (2001).

3.1.1.5 Further Constraints on Typed Feature Structure Definitions

There are a few restrictions to the above definitions that allow very efficient processing without sacrificing the general spirit of the grammar theory. The fastest HPSG implementations agree with and share these additional constraints (again, formal definitions presented e.g. in Carpenter 1992).

- *Bounded Complete Partial Order (BCPO)* The BCPO type hierarchy condition already mentioned implies a *closed type world*, i.e., every pair of types which has no explicitly defined common subtype is incompatible. The type inheritance hierarchy is defined by the grammar writer as graph which can be automatically completed to a BCPO in polynomial time at compile time (broad-coverage HPSG grammars currently typically contain 50000–100000 types).

- *Strong typing* requires that every feature structure node in a typed feature structure has a type.

- *Appropriateness* requires each feature in a grammar to be introduced only by a single (unique) type. This is a strict notion of appropriateness that eases (efficient) implementation and type inference.

- *Well-typedness*: This condition states that for every outgoing feature arc in a feature structure that it is appropriate for its type, and the value type of the feature is subsumed by its *Approp* type.

To precisely define appropriateness and welltypedness, we define a function called *Approp*:

3.1.1.6 Definition Appropriateness Function

For an inheritance hierarchy $\langle T, \sqsupseteq \rangle$ and the feature set F of the grammar, we define $\mathit{Approp} : F \times T \rightarrow T$ to be a partial function, and

- $\forall f \in F$, there is a most general type $\mathit{Intro}(f) \in T$ such that $\mathit{Approp}(f, \mathit{Intro}(f))$ is defined.
- furthermore, we define that if $\mathit{Approp}(f, \sigma)$ is defined and $\sigma \sqsupseteq \tau$ (with $\sigma, \tau \in T$), then $\mathit{Approp}(f, \tau)$ is also defined and $\mathit{Approp}(f, \sigma) \sqsupseteq \mathit{Approp}(f, \tau)$.

3.1.1.7 Definition Welltypedness

A typed feature structure $\theta = \langle Q, q_0, \tau, \delta \rangle$ is well-typed if whenever $\delta(f, q)$ is defined, then $\mathit{Approp}(f, \tau(q))$ is defined, and such that $\mathit{Approp}(f, \tau(q)) \sqsupseteq \tau(\delta(f, q))$.

A typed feature structure is *totally well-typed* if every node in it is well-typed, and for every type occurring at a node, its appropriate features are explicit.

These above definitions do not only enable *type inference*, i.e., an algorithm to uniquely determine the type of an untyped or 'under-typed' feature structure node (node with a less specific type than the maximally possible according to the outgoing feature arcs), but also allow to implement type checking at definition time (e.g. to prevent errors in hand-crafted definitions, cf. also Schäfer 1995), and – even more importantly – efficient packing, unfolding and unification algorithms, cf. Callmeier (2001, Chapter 9). The latter is indispensable for fast HPSG parsing.

3.1.2 HPSG and HPSG Parsing

Having introduced the basic concepts for efficient typed feature structure unification, we can turn back to the essentials of HPSG (partly referring to an HPSG introduction by Müller 2007).

An HPSG parser is a computer program similar to a context-free parser. It takes an input sentence, looks up the lexicon entries (typed feature structures) in the HPSG lexicon for each occurring word, puts them on a chart, and applies general rules, consisting of typed feature structures as well (some defined as 'principles'), to the chart items. Instead of using symbol equality for comparison of chart items such as in context-free parsing, the unification operation is applied to the typed feature structures.

The result of parsing is then either a failure, signaling inconsistency or ungrammaticality of the input with respect to the grammar, or one or more typed feature structures remaining on the chart, containing the complete parsing result (or fragments if no analysis could be computed spanning the complete input). Ambiguity is introduced e.g. through multiple lexicon entries for an input word.

3.1.2.1 Phrase Structure

As already mentioned, HPSG at its backbone relies on phrase structure (the 'PS' in HPSG). Only grammatically correct words can combine to a phrase, which in turn only combine to a sentence if licensed by the grammar. Ungrammatical sequences of words should be rejected.

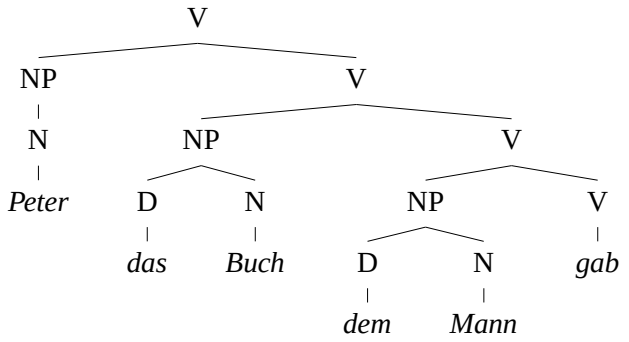


Figure 3.3: Phrase structure tree

What makes HPSG different from other (previous) grammar theories is that also phrase structure is encoded in typed feature structures. I.e., phrase structure is not defined separately by context-free rules like in LFG's c-structures (Kaplan and Bresnan, 1982), but within typed feature structures and together (in conjunction) with additional feature constraints. Dominance information is encoded in DTR attributes (DTR for daughter). Cf. the phrase structure for the sentence fragment '... Peter das Buch dem Mann gab' in Figure 3.4.

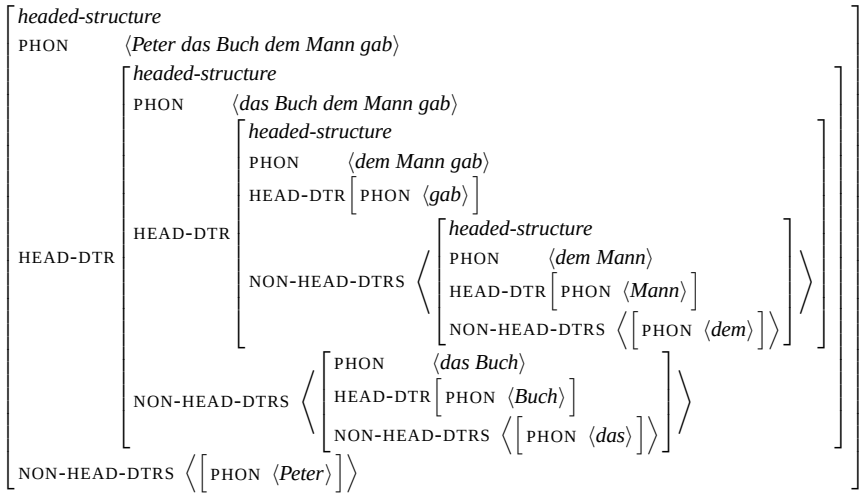


Figure 3.4: Phrase structure encoded in a typed feature structure

The corresponding tree structure with the phrase categories as node labels is depicted in Figure 3.3. What rules out grammatically incorrect structures such as

**Peter gab das Buch.*

where an indirect object is missing, or

**Peter gab.*

where both direct and indirect object are missing, comes from the lexicon entry of the verb 'geben' (to give). Here, a list under the feature SUBCAT specifies the *subcategorization* objects, in this case the subject NP[*nom*], the direct object NP[*acc*] and the indirect object NP[*dat*] (with the NP[*case*] notation abbreviating a feature structure with embedded category feature NP and case feature *nom*/*dat*/*acc*).

A general *Subcategorization Principle*, encoded as type constraint in the inheritance hierarchy of the HPSG grammar, states that the subcategorization list must be saturated element by element at each *head-argument-structure*:

$$\begin{bmatrix} \textit{head-argument-structure} & \\ \textsc{subcat} & \boxed{1} \\ \textsc{head-dtr} & \begin{bmatrix} \textit{head-argument-structure} \\ \textsc{subcat} \quad \textit{append}(\boxed{1}, \langle \boxed{2} \rangle) \end{bmatrix} \\ \textsc{non-head-dtrs} & \langle \boxed{2} \rangle \end{bmatrix}$$

This (much simplified) principle defines a binary branching structure, and the SUBCAT attribute which is omitted in the example in Figure 3.4 is processed by means of the Subcategorization Principle as sketched in Figure 3.5.

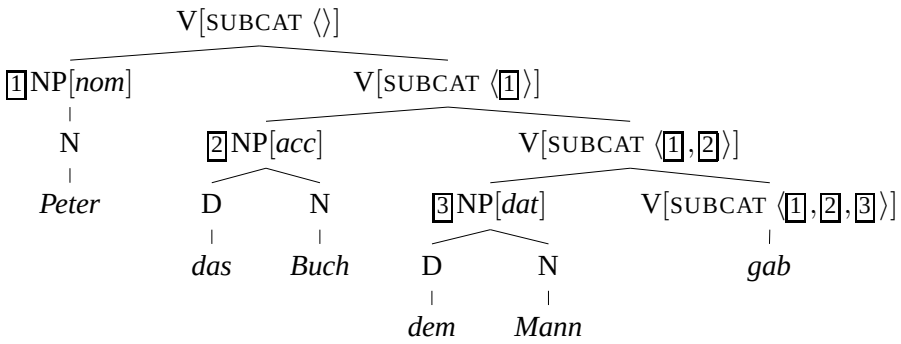


Figure 3.5: Subcategorization Principle in Phrase Structure

If no lexicon entry licenses the

**Peter gab.*

reading where the verb 'geben' would accept empty objects, then this would render the sentence unacceptable by the grammar, which is appropriate for this verb having no intransitive reading.

3.1.2.2 Generating a Semantics Description from Syntax

On the other hand, the lexicon entry with direct and indirect object for 'geben' specified in the SUBCAT list would not only make sentences providing candidates for the objects acceptable, but would also at the same time construct (by additional features and principles not shown here) a semantic representation for the sentence assigning 'geben' as action, the subject as agent of the action, and the direct and indirect object as arguments – the *predicate–argument structure*.

In the same way, verb control is marked in the lexicon, e.g. equi verbs such as 'promise' and 'persuade' may take an embedded verb phrase (VP) and assign the object of the controlling verb via a specification in the SUBCAT list to the subject (in case of 'promise') or the object (in case of 'persuade') of the VP (similar for auxiliary verbs). In other words, the SUBCAT list also plays an important role in the syntax-semantics interface of a grammar – and this is one of they key strengths of HPSG over shallow approaches.

3.1.2.3 The Role of the Lexicon

A main difference of the HPSG paradigm compared to the usual context-free grammar approach is that the role of the CFG rules is generalized to very generic 'meta-rules'. While a pure CFG has to move specific information to (as many as possible) rule symbols, this is avoided in HPSG.

Instead, information that would be encoded in a CFG by blowing up the symbol repository, is moved as much as possible to the lexicon in HPSG. In other words, if one would expand (or 'fill') lexicon entries, quite complex structures such as the one shown in Figure 3.6 from the English Resource Grammar (Flickinger, 2002) would appear (including syntax and semantics features) – which still is not fully filled for space reasons here.

It is important to understand that complexity of lexicon entries is not a shortcoming, but an advantage for the structure of a grammar, and is greatly facilitated because redundancy is reduced by moving information to efficiently encoded type inheritance in the lexicon. This is in full analogy to the well-known and researched advantages of inheritance in knowledge representation and programming languages. The main advantages are that it is possible to (1) easily encode idiosyncrasies in the lexicon without sacrificing the monotonicity of the base formalism, (2) provide fine-grained word-specific semantics representations, but inherit all other information from more general types.

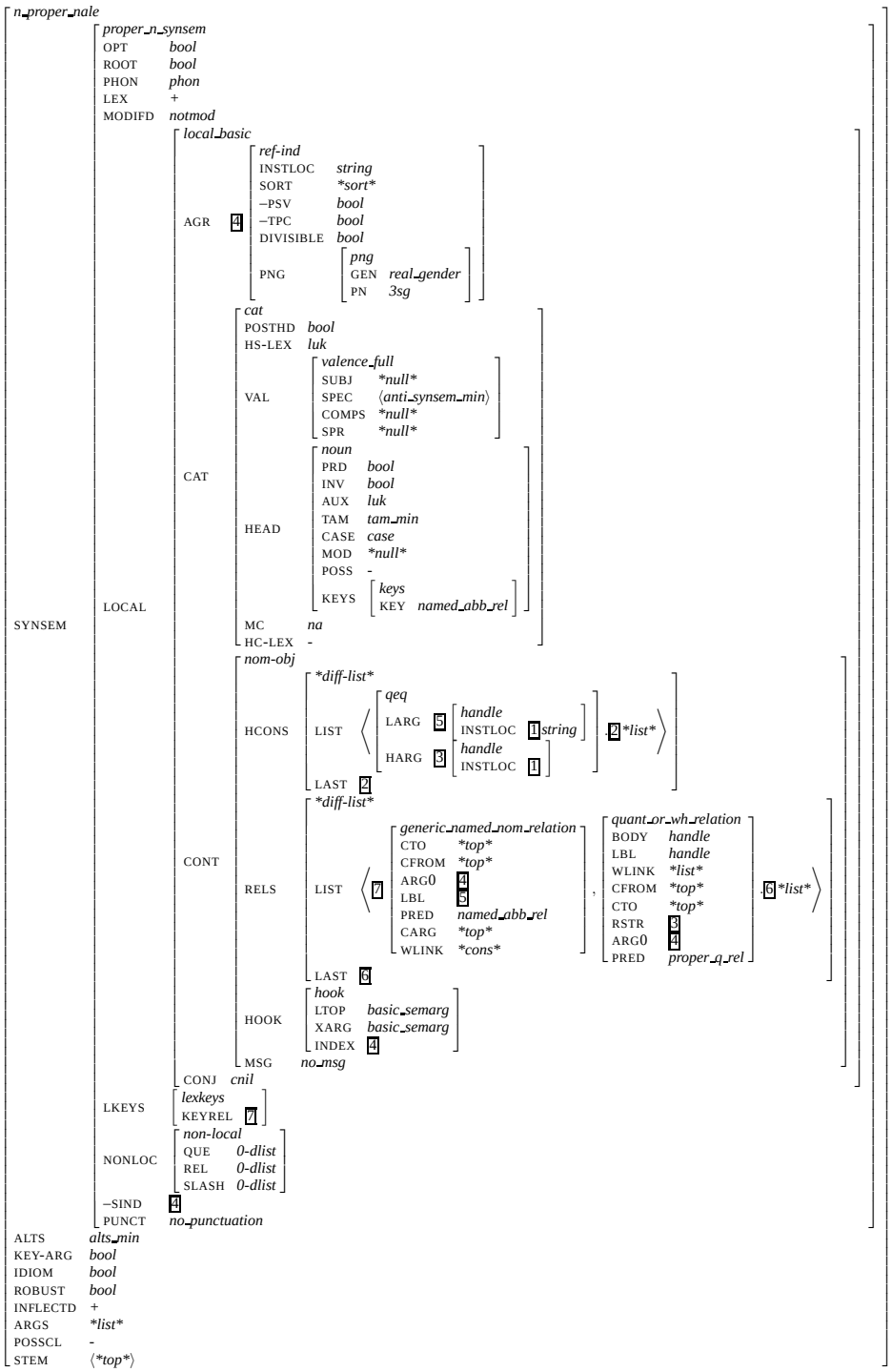


Figure 3.6: An HPSG lexicon entry for generic named entities

3.1.2.4 Head-Driven Approach

Finally, the 'H' in HPSG stems from the central role of the *heads* as information-bearing *nuclei* of phrases that transport features and its values over phraseal projections (X' theory; Jackendoff 1977). The *percolation* of feature information in the heads is forced by the *Head Feature Principle* that informally can be expressed as 'if a feature structure has a head (i.e., is of type *headed-structure* or inherits from it), then its head features are shared with the head features of the head daughter' (a feature structure may bear other features than the head features that are not percolated by this mechanism).

$$\begin{bmatrix} \textit{headed-structure} & \\ \textsc{head} & \boxed{1} \\ \textsc{head-dtr} & \begin{bmatrix} \textit{headed-structure} \\ \textsc{head}\ \boxed{1} \end{bmatrix} \end{bmatrix}$$

Examples are the case and number features of the head noun that become also head features of the NP containing the noun by virtue of the Head Feature Principle. By inheritance, the Subcategorization Principle, a *head-argument-structure*, a subtype of *headed-structure*, is constrained by the Head Feature Principle (of type *headed-structure*), as shown in Figure 3.7.

$$\begin{bmatrix} \textit{head-argument-structure} & \\ \textsc{head} & \boxed{1} \\ \textsc{subcat} & \boxed{2} \\ \textsc{head-dtr} & \begin{bmatrix} \textit{head-argument-structure} \\ \textsc{head}\ \boxed{1} \\ \textsc{subcat}\ \textit{append}\langle \boxed{2}, \langle \boxed{3} \rangle \rangle \end{bmatrix} \\ \textsc{non-head-dtrs} & \langle \boxed{3} \rangle \end{bmatrix}$$

Figure 3.7: Subcategorization Principle

During parsing, the percolation of the head features throughout phrase structures and saturation of the subcategorization list of the main verb are two of the key operations that are performed only by means of typed unification.

They not only determine the syntactic structure of the sentence, but at the same time also build the key semantic representation part of the analysis (details will be presented in the next section), and both are driven by the notion of heads.

There are many more (and more complex) principles regulating e.g. word order, coordination, semantics construction, quantification *etc* which we will not discuss here, cf. Pollard and Sag (1994); Müller (2007) for details.

One should keep in mind that the feature structures that are built during parsing typically get sizes of thousands of nodes because unification is a monotonic

operation, and already the initial lexicon entries have considerable size. A naïve implementation without optimization would render any more-than-toy-size HPSG grammar intractable (as were the first approaches). However, through sophisticated techniques, it can be avoided that efficiency poses a principal barrier (except for pathological cases), cf. Section 3.2.2.

3.1.2.5 Output: Semantic Representation

By way of the principles, an HPSG parser checks validity (satisfiability) of the composed structures (with the lexical entities as leaves). At the same time, construction of the semantic representation as 'output' of a parsed sentence is performed uniformly by inheritance/unification of rules, special principles for semantics construction and lexicon entries by the commutative, monotonic and associative unification operation of the declaratively specified linguistic knowledge.

The resulting semantic representation is stored in dedicated features of the parse result and can be extracted from there (Pollard and Sag, 1994). Depending on the required information, also morpho-syntactic features (e.g. gender, number, tense) could be retrieved from the parse result by an application.

In principle, HPSG is open to produce different kinds of semantics representations. Situation Semantics (Barwise and Perry, 1983) has been originally proposed in Pollard and Sag (1987). MRS (Minimal Recursion Semantics; Copestake *et al.* 2005a) is a more recent and probably better suited and elaborated approach. The key idea is to provide a language similar to predicate calculus, but with a syntactically flat structure. The *elementary predicates* (relations with arguments) are never embedded within one another.

However, there is treatment of scope which most other flat semantics representations do not provide. Scope is modeled by handle variables, and constraints over the handles can be used to select specific scopes or relaxed to leave the scope underspecified, thus providing a compact representation of multiple semantic interpretations.

Here is an example of a MRS representation of the sentence

Every dog chases some white cat.

$$h_1 : \text{every}(x, h_3, h_A), h_3 : \text{dog}(x), h_7 : \text{white}(y), h_7 : \text{cat}(y), h_5 : \text{some}(y, h_7, h_B),$$
$$h_4 : \text{chase}(x, y)$$

Two alternative scopes could e.g. selected by setting either $h_A = h_5$ and $h_B = h_4$ or $h_A = h_4$ and $h_B = h_1$.

Such analysis is computed within typed feature structures during parsing, i.e., via unification, and can be retrieved from the parse result.

The advantage of the flat MRS structure is that it (i) can be constructed compositionally during (HPSG) syntax analysis, (ii) can be encoded in typed feature structures, (iii) is powerful enough to represent linguistic meaning, (iv) allows for

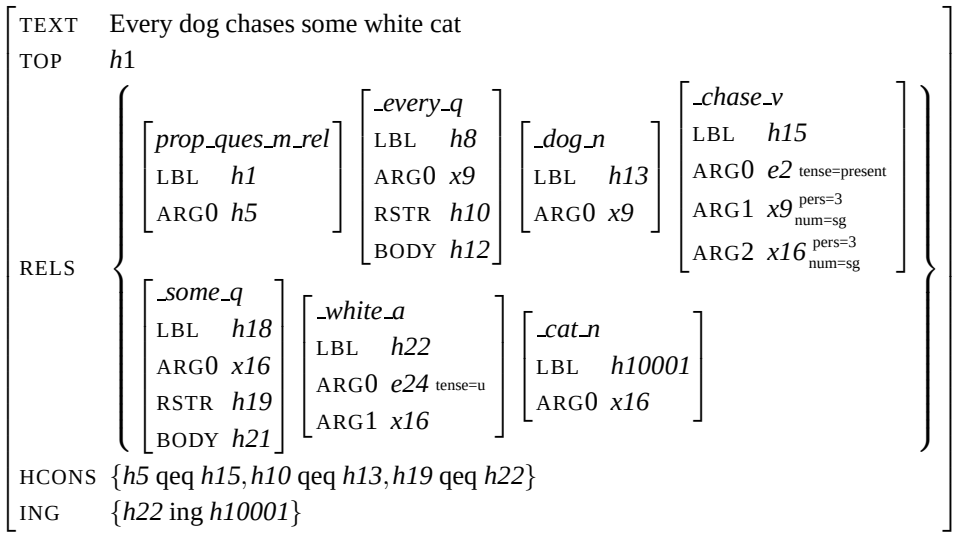


Figure 3.8: An RMRS representation generated by HPSG parsing

underspecification, (v) can be used for both parsing and generation (Copestake *et al.*, 2005a, 1995b).

A robust variant of MRS called RMRS (robust MRS; Copestake 2003) is better suited to support partial semantic analyses, e.g. produced by shallow parsers. An AVM-like, 'graphically' represented example with two slightly differing readings for the sentence is depicted in Figure 3.8. We will return to RMRS in Chapter 9.

3.2 Performance Properties of HPSG

3.2.1 Parsing Complexity

Although the HPSG formalism with typed feature structures and variables (coreferences) is in general equivalent to a Turing machine (cf. the discussion and references in Section 3.1.1), this is only of theoretical significance. A grammar that meets the so called offline parseability constraint (Kaplan and Bresnan, 1982; Pereira and Warren, 1983; Dymetman, 1994), avoiding empty and chain productions in the context-free backbone, e.g. can be guaranteed to terminate.

Moreover, there is much evidence that existing grammar-parser combinations might be in the class of mildly context-sensitive grammars (Joshi *et al.*, 1991), and thus be parsable in polynomial time as is the case for other contemporary grammar formalisms.

3.2.2 Implementations and Efficiency

During the last decade, several implementations of HPSG formalisms, parsers and grammars have been published, ranging from very lean (but rather slow) engines directly relying on Prolog SLD resolution up to very comprehensive grammar development environments such as LKB (Copestake, 2002) or PAGE (Uszkoreit *et al.*, 1994) (both developed in LISP), with compactly encoded type hierarchies, quite fast parsers, and powerful analysis and inspection tools.

A performance boost has been achieved through a collaborative approach of measuring performance and incrementally adding sophisticated efficiency-improving methods such as quick check, packing and unfilling (Oepen and Callmeier, 2000).

The framework that resulted from this international effort consists of the LKB system for grammar development and debugging, the efficient PET HPSG parser (Callmeier, 2000, 2001) implemented in C++ (and pure ANSI C for memory management), TSDB (Oepen, 2001) and a couple of reference grammars (including TSDB test suites and data), the English Resource Grammar (ERG; Flickinger 2002) being the most prominent, elaborate and best-tested one.

This ensemble currently presents itself as the most comprehensive and at the same time most efficient HPSG implementation framework. Many of the aforementioned efficiency techniques have been implemented first in the LISP parser an then ported to the more efficient C implementation within PET.

Parsing sentences is now possible within fractions of seconds for typical average sentence lengths, and within seconds for longer sentences, depending on the language, grammar and various parser option settings. These runtimes also include parse ranking, i.e., ordering multiple parses for a sentence according to a statistical model trained on a corpus.

3.2.3 Robustness

While the efficiency problem of HPSG parsing seems to be solved, what remains is a *robustness problem*, and this is mainly what this thesis tries to tackle. In this section, we motivate the robustness problem and related problems and briefly present an outlook to the envisaged solution.

As explained on page 43, lexicon entries play a crucial role in HPSG grammars. Although most of the information necessary to form the lexicon entry for a specific word comes through inheritance, for every word that occurs in the sentence to parse, an appropriate lexicon entry has to be found. Otherwise, the sentence cannot be analyzed, i.e. no spanning chart item for larger structures such as phrases or the whole sentence can be computed, nor can a compositional semantics representation be built of the entire sentence.

In other words, it is not possible to omit words from the input for which no lexicon item can be found, because no analysis would be possible (or if possible, it would probably be wrong). This fact, being rooted in the principle of how deep

processing works in general, has an important impact on robustness of HPSG parsing. Every attempt to circumvent the problem by adding 'robustness' rules to the grammar that allow parsing incomplete input is penalized by (1) increased ambiguity (2) the unwanted effect of admitting ungrammatical input as well.

To give an example with only a hand-crafted lexicon in the German HPSG with 35000 entries (approx. 350000 full forms) of which 25000 where semi-automatically generated, approx. 71 percent of the sentence in a newspaper text fail just because of missing lexicon information (Crysmann *et al.*, 2002).

Hence the solution is to have generic lexicon entries and interfaces to other NLP components that can contribute information that is necessary to fill the gaps in generic entries.

Solutions have been presented that address specific instances of HPSG grammars and parsers and specific shallow systems, e.g. for ALEP and a German HPSG-like Grammar (Declerck and Maas, 1997), for a Spanish HPSG Grammar (Marimon, 2002a), Grover and Lascarides (2001) for an English HPSG and a PoS tagger, and Kaplan *et al.* (2002) for the Xerox PARC LFG parser.

The focus of this thesis is to lift this to a more general level and provide architectures that support integration in a more principled way.

Such architectures will not only address the robustness problem, but also could help to (1) reduce ambiguity by selecting readings on lexical, or even phrase or topological sentence structure level with the help of external NLP components (divide and conquer) (2) provide a means to correct or repair ungrammatical or malformed input (e.g. email text, or speech analyzed by a speech recognition system), (3) extend the initial sequential approach (shallow processing first as input to deep processing) to more flexible ones, permitting also post-parsing integration and combination of parsed sentence fragments in case no full parse could be computed.

Furthermore, not just lexical information could be incorporated, but also results from other NLP components beyond the lexical level such as chunk parsers, topological parsers *etc*.

Chapter 4

Shallow Processing and Linguistic Markup

In this chapter, we will introduce the most commonly used tasks of shallow natural language processing and exemplify shallow representation formalisms. We will introduce XML and the relation to shallow markup and linguistically annotated corpora and give a motivation for the adoption of commonly available XML standards and implementations.

Then, we will show how linguistic representations can be encoded within XML and what the advantages are for processing, communication, portability, internationalization, and storage in databases or tree banks.

We will address general representation problems arising when linguistic data is encoded in XML, e.g. (a) encoding of ambiguities/multiple readings, (b) encoding of structure sharing and references, (c) overlapping and discontinuity, *etc* and show how these can be solved using *standoff annotation*.

4.1 Shallow Natural Language Processing

Shallow natural language processing has emerged as an alternative paradigm to traditional deep linguistic processing in the eighties and nineties[1]. The main motivation was lack of robustness and efficiency of deep NLP implementations at that time that made it impossible to use them for real world texts.

In contrast to deep processing, shallow processing is driven and dominated rather by specialized, application-oriented tasks than by linguistic theories. Meanwhile, there is broad common sense on what the different tasks or applications of shallow processing are. Moreover, standardized evaluations exist for many of the different shallow tasks such as CoNLL (Tjong Kim Sang and Buchholz, 2000) for chunking or MUC (Grishman and Sundheim, 1996) for named entity recognition.

[1] However, according to Joshi and Hopely (1996), a first finite-state parser has been developed as early as 1958.

The term shallow mainly relates to the resulting analyses, which could be characterized as simpler, less complex structured, partial, non-exhaustive compared to what a deep parser would (ideally) provide as result.

However, the methods employed for computation can be quite different. They comprise machine learning (or statistical) methods, rule-based (e.g. finite-state, context-free), and combinations of these.

4.1.1 Tokenization

The simplest step in shallow processing is tokenization, the separation of words and other symbols, e.g. punctuation. Tokenization is basically a preprocessing step to ease subsequent processing such as morphologic analysis and lexicon lookup. Example:

”What should I do?”, asked Fred.

could be tokenized into

”$_{t_1}$ What$_{t_2}$ should$_{t_3}$ I$_{t_4}$ do$_{t_5}$?$_{t_6}$ ”$_{t_7}$,$_{t_8}$ asked$_{t_9}$ Fred$_{t_{10}}$.$_{t_{11}}$

Additionally, a class of token such as punctuation symbol, capitalized, lowercase, uppercase word is often assigned to each token. Depending on the tokenizer, this can be done very fine grained (even with alternative readings which already introduce ambiguity in this early analysis step) or rather coarse grained with only a few or no distinct token classes.

Tokenization may also include some simple kind of normalization, e.g.

we’ll → we will
4.5% → 4.5 percent
8pm → 20:00
7” → 7 inches

Typically, tokenization is defined by finite-state rules (e.g. in Grefenstette and Tapanainen 1994). For Japanese or other Asian languages, tokenization also includes the task of word boundary recognition which is only possible with lexicon lookup – in contrast to languages based on Latin characters where spaces and punctuation separate words.

In many NLP systems, tokenizers are closely integrated with the other linguistic processing components which in turn make assumptions on what the input tokenization is and how token classes are defined. Hence, different interpretations of how a token is defined may exist if these NLP systems are combined.

4.1.2 Finite-State Morphology and Compound Recognition

Morphology describes the relation between the surface forms of words and the lexical form which consists of a lemma and the grammatical description. This can easily be modeled, at least for Indo-European languages, with finite-state automata (Koskenniemi, 1983). The idiosyncratic information is stored in a lexicon (database of lemmata and hints for morphologic regularities).

The following list enumerates the morphological readings of the German word 'Ranke' which could be a noun (N), adjective (A) or verb (V), as analyzed by the SPPC system (Piskorski and Neumann, 2000).

```
pos="N" gender="M" case="DAT" number="SG"
pos="A" gender="F" case="NOM" number="SG" comp="P" det="NONE"
pos="A" gender="F" case="AKK" number="SG" comp="P" det="NONE"
pos="A" gender="M" case="NOM" number="PL" comp="P" det="NONE"
pos="A" gender="F" case="NOM" number="PL" comp="P" det="NONE"
pos="A" gender="N" case="NOM" number="PL" comp="P" det="NONE"
pos="A" gender="M" case="AKK" number="PL" comp="P" det="NONE"
pos="A" gender="F" case="AKK" number="PL" comp="P" det="NONE"
pos="A" gender="N" case="AKK" number="PL" comp="P" det="NONE"
pos="A" gender="F" case="NOM" number="SG" comp="P" det="INDEF"
pos="A" gender="F" case="AKK" number="SG" comp="P" det="INDEF"
pos="A" gender="M" case="NOM" number="SG" comp="P" det="DEF"
pos="A" gender="F" case="NOM" number="SG" comp="P" det="DEF"
pos="A" gender="N" case="NOM" number="SG" comp="P" det="DEF"
pos="A" gender="F" case="AKK" number="SG" comp="P" det="DEF"
pos="A" gender="N" case="AKK" number="SG" comp="P" det="DEF"
pos="V" tense="PRES" person="1" number="SG"
pos="V" tense="SUBJUNCT-1" person="1" number="SG"
pos="V" tense="SUBJUNCT-1" person="3" number="SG"
pos="V" form="IMP" number="SG"
```

For many languages such as German, Dutch and Finnish, which allow rather free word formation of nouns by compounding, an additional step of compound segmentation is necessary. This is in many cases a regular process and hence can also be handled by finite-state rules. Exceptions such as the selection of the appropriate *Fugenmorphem* for a compound noun can be encoded in the lexicon, e.g. *Kind**s**taufe* vs. *Kind**er**stube*.

4.1.3 Part-of-Speech Tagging

A further step towards recognizing sentence structure is part of speech (PoS) tagging, where the class of a word (noun, verb, adjective, preposition *etc*) is computed for each token in the input text, and attached as a 'tag'. In the example above, PoS tagging would compute which of adjective (A), noun (N) or verb (V) is the most likely in the context of a sentence); this can help to abstract from specific lexicon entries e.g. for subsequent parsing.

There are rule-based (e.g. Brill 1992) and statistical approaches (e.g. Brill and Marcus 1992) to this task, where the best statistical taggers outperform rule-based taggers generally on unseen text, with per-token accuracy of around 96 percent. The statistical approaches assign probability values per word in its context, e.g. through Hidden Markov Models (HMM) obtained through training trigrams on manually annotated text data.

The following sentence e.g. is correctly analyzed (with probability 1.0 for each tag) by the trigram-based TnT system (Brants, 2000) that has been trained on the NEGRA corpus for German (Skut *et al.*, 1998) and e.g. the Penn Tree Bank for English.

> Die_{ART} $Kriminalpolizei_{NN}$ $verfolgte_{VFIN}$ die_{ART} $Bankräuber_{NN}$ bis_{APPR} $zur_{APPRART}$ $niederländischen_{ADJA}$ $Grenze_{NN}$.

However, it is generally possible to have several possible readings for the class of a word, and statistics-based systems can rank the possible readings by means of their trained model.

Another big advantage mainly of statistical part-of-speech taggers is that they can be used to guess word classes of unknown words by 'interpolating' on the basis of surrounding tags, provided the word classes of surrounding words could be determined reliably.

4.1.4 Chunking

Chunking is another useful preprocessing and abstraction step for parsing. Chunks are non-overlapping groups of words forming small syntactic units (phrases) such as noun phrases consisting of an optional determiner, followed by an optional adjective, followed by a noun. Text chunking divides the input text into such phrases and assigns a type such as NP for noun phrase, VP for verb phrase, PP for prepositional phrase in the following example, where the chunk borders are indicated by square brackets:

> $[Die\ Kriminalpolizei]_{NP}$ $[verfolgte]_{VP}$ $[die\ Bankräuber]_{NP}$ $[bis\ zur\ niederländischen\ Grenze]_{PP}$.

Chunking is sometimes also called chunk parsing, partial parsing, or light parsing (Abney, 1991). There are different definitions in literature of what a chunk exactly is. Sometimes, chunks are required to be non-recursive, i.e., no other chunk may be embedded within a chunk, other definitions admit this (hence importing e.g. inherent attachment ambiguity).

The CoNLL-2000 shared task for chunking (Tjong Kim Sang and Buchholz, 2000), e.g. has found out precision and recall values at around 93 % for the best systems on the commonly defined task based on training and test data on the Penn Tree Bank (English Wall Street Journal articles). Techniques for chunking are e.g. cascaded HMMs (Skut and Brants, 1998), Support Vector Machines (Kudo

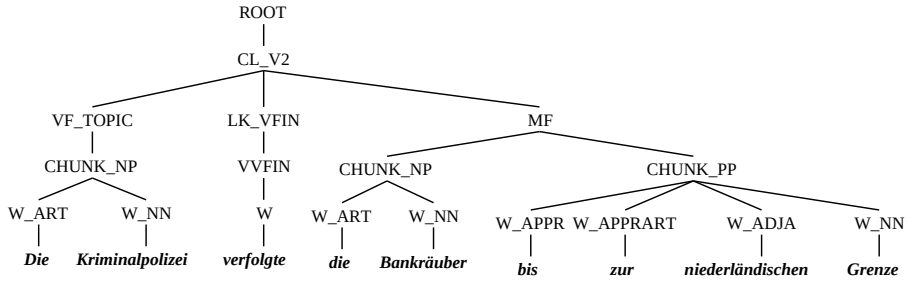


Figure 4.1: Topological tree as result of shallow parsing

and Matsumoto, 2000), Cascaded Finite State Parsing (Abney, 1996; Grefenstette, 1996) or combinations thereof.

4.1.5 Shallow Parsing

Shallow parsing is an analysis of a sentence which identifies the constituents (verbs, noun phrases, *etc*), but does not specify their internal structure, nor their role in the whole sentence. However, in contrast to chunking, the formation of a sentence-spanning structure, or substructure for the topological fields (in German Vorfeld, Mittelfeld, Nachfeld) of a sentence is tried ('topological parsing').

A popular way of implementing such parsers is by probabilistic context-free grammars (PCFG) which can be trained on annotated corpora (Schmid, 2000; Dubey and Keller, 2003). Shallow parsing can thus be considered an extension of chunking. An example is shown in Figure 4.1.

4.1.6 Named Entity Recognition

Named entity recognition (NER) is the identification of proper names, product or organization names, location names such as cities and countries in text and the specification of the type of named entity. More or less fine-grained variations of the definition of what a named entity (NE) type is exist in literature. A very simple classification is the one of the Message Understanding Conferences (MUC; Grishman and Sundheim 1996). Only basic types such as DATE, LOCATION, PERSON, ORGANIZATION exist:

> *Five unsuccessful attempts were made by a single team led by [Larry Newman]$_{PERSON}$ of [Scottsdale]$_{LOCATION}$, [Ariz.]$_{LOCATION}$, in his [Earthwinds]$_{ORGANIZATION}$ balloon before [Newman]$_{PERSON}$ abandoned his efforts [one year ago]$_{DATE}$.*

A modern, finer-grained classification is that of ACE (Automatic Content Extraction[2]). Depending on the application, even finer-grained typologies of NEs may be desirable. An example is normalization of time expressions, where slots for year, month, day, day of week, hour and minute could be filled or inferred from context or left unspecified. The *SProUT* system that will be described in Chapter 7 goes beyond typical NE recognition systems in that it allows for finer-grained and more domain-specific structured output.

4.1.7 Summary

We have briefly introduced different types and stages of shallow processing. Many implementations and systems exist following the described paradigms. Moreover, there are also implementations that combine different processing stages within one system. An example is the SPPC system (Piskorski and Neumann, 2000) for German (cf. Figure 4.4 for a sample output).

The advantage is increased efficiency and optimally combined and interleaved shallow processing stages. However, it is hard to exchange or separate stages with other components, so the price for efficiency here is inflexibility.

RASP (Briscoe and Carroll, 2002) shares some similarities, and focuses on English. Other systems such as GATE (Cunningham *et al.*, 2002) provide a general architecture for combining various shallow systems. We will discuss GATE and related systems and architectures in Chapter 6.

Simple language technology-based applications can be built on top of such shallow processing systems, the outstanding being probably morphological analysis and named entity recognition. In contrast to deep natural language parsing, shallow processing is typically non-exhaustive and partial. Words or groups of words for which no information exists, will remain unanalyzed. Depending on the underlying methods, probabilities may be assigned to analyses. A threshold can be used to filter out improbable readings if more than one have been computed.

However, there is no semantic analysis available directly from such shallow systems. In the best case, some kind of underspecified, partial semantic description can be derived from shallow analyses. Additional knowledge and computation is necessary in order to obtain semantic analyses such as predicate-argument structure as deep processing can deliver.

The idea of combining several shallow systems in such a way that a deep analysis can be computed seems appealing. However, attempts made so far only reach very limited single aspects of deep analyses, and require a lot of additional work. An example of such a single (syntactic) deep analysis aspect is detection of long distance dependencies, described in Dienes and Dubey (2003).

[2] `http://projects.ldc.upenn.edu/ace/`

4.2 The Relation between Shallow Processing and XML Representations

The partial character of shallow analysis immediately and naturally leads to the idea of storing shallow linguistic analysis as *markup* of the analyzed text. In this section, we discuss the relation between shallow processing and XML representations. We start with a short introduction to and history of XML and SGML. We then present some standards for linguistic markup and discuss problems of the representation languages, and how they can be solved within the XML framework.

4.2.1 SGML

The idea of augmenting electronically stored text by markup dates back to William Tunnicliffe (1967, 'generic coding')[3] and ideas of the book designer Stanley Rice ('editorial structure tags'; also in the late 60ies). The word *markup* itself goes back to the earlier, pre-electronic technique of adding manually tags to book or newspaper manuscripts as hints for typesetters (e.g. formatting instructions).

The 'generic markup' evolved into the Generalized Markup Language (the acronym GML also formed by the initial letters of the main author's names Goldfarb, Moshier and Lorie) at IBM in 1969. Finally, Goldfarb took GML with some extensions to the ISO standard 8879 in 1986 under the name SGML (Standard Generalized Markup Language).

GML and SGML have been designed for the formal description of structural constraints on text documents, mainly for interoperability of document-processing software (e.g. typesetting) and for document quality assurance in large companies and institutions (legal, government, military) that is described in grammars called DTDs (document type descriptions).

A DTD may e.g. enforce that a document of type book has at least one author, a title and several chapters, a chapter may or may not be divided into numbered sections. Complex conditions can be added that e.g. put technical manuals to obey very detailed structural rules.

Because of the complexity of the standard, implementation of SGML processing and checking software was a hard task, and SGML became a success story at best in large companies and institutions, but not as a broadly accepted standard in the originally envisaged wide range of daily use. Moreover, the vision in the foreword of Yuri Rubinsky in the SGML Handbook (Goldfarb, 1990) is still what it was in 1990 – a vision.

> *'The next five years will see a revolution in computing. Users will no longer have to work at every computer task as if they had no need or*

[3]No publication available; 'Many credit the start of the generic coding movement to a presentation made by William Tunnicliffe, chairman of the Graphic Communications Association (GCA) Composition Committee, during a meeting at the Canadian Government Printing Office in September 1967: his topic – the separation of information content of documents from their format.' (Goldfarb, 1990).

> *ability to share data with all their other computer tasks, they will not need to act as if the computer is simply a replacement for paper, nor will they have to appease computers or software programs that seem to be at war with one another.*'

(Not only) SGML failed in fulfilling this prediction, although some advances have been made into the visionary direction, and the main contribution is (still) that SGML defines a text document format for exchange and persistent storage different from legacy data and vendor- and platform-specific word processor formats that could and can be used over decades.

4.2.2 XML

The success of the World Wide Web, based on HTTP, the HyperText Transfer Protocol, and HTML, the HyperText Markup Language (which can be described by a small SGML DTD) led to the development of XML, the eXtensible Markup Language. HTML had been initially developed in 1992 as a mixture for describing simple hypertext document structure (in the spirit of SGML, e.g. by tags for title, headings, enumerations) and formatting (bold face, line breaks, font name and size). HTML became a standard ('recommendation') of the newly founded World Wide Web Consortium (W3C) in 1994.

Because SGML was too complicated and HTML was not clear and powerful enough for the upcoming need to describe the structure and semantics of documents independently from its layout, XML, which is basically a restricted, less complicated subset of SGML, has been made a first W3C working draft in 1996, and a recommendation version 1.0 in 1998.

The original goal of XML, namely separation of structured content markup (XML) and layout markup (HTML, PDF, *etc*) is still not achieved in the majority of currently published web pages. However, powerful and mature techniques, software and tools exist today, that will help to make XML a true replacement for HTML for content storage and turn the content and layout separation paradigm into reality in the (near) future.

Some introductory publications on XML claimed that XML would assign semantics to documents by means of speaking element and attribute names. This notion of XML semantics is, however, not the formal semantics that is addressed in computer science or computational linguistics. Hence, the role of XML is formally in the best case that of an abstract syntax or carrier syntax, and the true semantics in the mathematical sense has to be defined outside the XML framework and depending on the envisaged application domain. We will see examples in the NLP and Semantic Web domain later.

4.2.3 Well-Formed and Valid Documents

The basic concepts of XML (shared with SGML) are

- *text*
- *elements*, also called tags, forming the 'markup'. Elements enclose text or embed other elements and hence can form a hierarchical structure on documents. An element name should semantically describe its content:
  `<heading>` *text* `</heading>`
- *attributes*, adding non-structured information (modifiers) to elements are specified together with the opening element:
  `<heading level="2">` *text* `</heading>`

One of the design goals of XML (and SGML) was to make the syntax both human-readable and machine-readable, and this is probably why the ending element designator repeats the element name redundantly (in contrast to parenthesis syntax in LISP s-expressions which the XML/SGML designers obviously did not consider human-readable).

To distinguish elements from text, the starting and ending element names in XML must be enclosed in angle brackets, the closing element is indicated additionally by a single slash after the opening angle bracket. Empty elements (elements that do not comprise text or other elements) may be abbreviated as `<element/>`. Angle brackets and three other characters used for markup have to be quoted when occurring in normal text within an SGML or XML document. The elements in an XML document form a tree and hence must be balanced (element borders must not cross). An XML document must have a single root element.

An XML document is *well-formed* if it meets these conditions (plus some other mentioned in the standard such as Unicode-conformity *etc*), i.e., if it is syntactically correct.

An XML document is *valid* if it is conforming with a DTD (document type description) that describes the structure of a class of documents in a grammar with a BNF-like description of element containment, order and repetition, as well as constraints on attributes.

Such DTDs are optional, i.e., the XML recommendation requires XML documents to be well-formed, but they do not necessarily have to be valid. A DTD e.g. states which element and attribute names are admitted in the document class, which element is the root element, which elements may be enclosed by which other elements (and possibly the order), which elements are mandatory or optional, and where text is allowed within elements. Examples for NLP-related DTDs can be found in the DTD Appendix (page 285ff).

Instead of a DTD, a schema can be used to validate an XML document. Schemata allow for finer-grained validity checking than DTDs, e.g. by user-definable data types which do not exist in DTDs. XML Schema (by the World Wide Web Consortium; Thompson *et al.* 2004) and Relax NG Schema[4] (by the OASIS consortium) are the most popular schema definition languages.

[4] `http://relaxng.org`

For the purposes of this thesis, DTDs are preferable, because we are mainly interested in the coarse structure of valid documents which can be defined concisely in DTDs, while XML Schema and Relax NG syntax which themselves are defined in XML syntax, are verbose, harder to read and less intuitive.

Both SGML and XML provide a means for describing document structure in form of an *abstract syntax* via a DTD or schema. However, they do not provide a *semantics* of the document schemata or instances unlike the ISO/ITU standard ASN.1 (Abstract Syntax Notation; Dubuisson 2000) that includes a semantics description in form of world-wide unique *object identifiers* (OIDs).

In XML, semantics is specified only implicitly and informally by giving elements and attributes speaking names. The XML-generating and the XML-parsing ends must be guaranteed to interpret the content in the appropriate way. However, optionality of elements and attributes is a quite elegant way to cope with the fact that it may make sense to have XML-consuming software that only looks at those pieces of XML input that it knows about, and ignores the rest (that in turn may be of interest for another consumer).

One main difference between SGML and XML is that XML makes more restrictions with respect to the wellformedness conditions than SGML, while SGML provides a more powerful language for describing validity of documents. Both properties together make XML easier to implement than SGML. Moreover, DTDs are mandatory in SGML, while they are optional in XML.

Further concepts of XML are

- *Uniform Resource Identifiers (URIs)*. URIs are used to reference external resources (similar to HTML hyperlink references). However, an explicit linking mechanism is not part of the core XML standard, but is defined in separate standards such as XPointer (DeRose *et al.*, 2002), XLink (DeRose *et al.*, 2001) or XInclude (Marsh and Orchard, 2001).

- *Namespaces*. Namespaces are, similar to packages in programming languages, dictionaries of identifiers that make e.g. elements with the same name, but in different DTDs, distinguishable. The namespaces used in an XML document are declared at the beginning using a URI uniquely defining the namespace and a local name as reference that can then be used as a prefix for element names, separated by colon, e.g.
  ```
  <invoice xmlns:edi='http://ecommerce.org/schema'>
   <edi:price units='Euro'>32.18</edi:price>
  </invoice>
  ```

- *ID/IDREF*. ID and IDREF are special attributes for indexing and searching elements within an XML document. To this end, ID attributes must be unique within a document. The XPath language we will describe below provides an `id()` function that can be used to access XML nodes via its unique ID specified as argument.

- *Entities*. Entities are abbreviations, e.g. for often repeated character sequences. Entities can be defined in a separate DTD or at the beginning of an XML file.

- *Unicode*. The XML recommendation obliges implementations of XML to support Unicode (other character sets may be supported optionally). This 'greatest common denominator' of character encoding ensures e.g. that multilingual documents can be processed uniformly. A further very important property is character length. Unicode introduces (in contrast to previous encoding initiatives) the concept of a code vs. encoding. Each Unicode character has a single, unique code, although there may be different encodings or representation formats with fixed-length (UCS-2, UCS-4) or variable length (UTF-8, UTF-16) binary representations. The existence of an equal-length character code is very important for standoff annotation references that are based on unique text positions and string operations in multilingual applications.

The above mentioned essentials of XML syntax of course constitute only a partial description. The complete XML syntax is described in the W3C recommendation (Bray *et al.*, 1998). The W3C XML recommendation (the official standardization document) itself makes references to other, lower-level standards such as Unicode for character encoding of text and elements, and IETF RFC 1738 for the Uniform Resource Identifier (URI) syntax.

4.2.4 Strictly Structured vs. Semi-Structured Documents

The XML paradigm for document markup is similar to that of SGML. The markup is a means of structuring documents semantically (where 'semantics' is not formally defined, but informally described by the name of elements and attributes). An XML document even need not be based on a text. It may instead consist of regularly structured data, such as address book entries where e.g. all information is encoded in elements and attributes, and the marked up text is 'empty'.

One of the outstanding advantages of XML is that it is suitable for both *semi-structured documents* (sometimes also called markup-structured, loosely-structured, or document-centric) containing *heterogeneous data* and *structured documents* (also called *strictly structured* or *data-centric* documents) containing *homogeneous data*.

Semi-structured documents are those that follow the above mentioned generalized markup idea. Natural language text (unstructured data from the viewpoint of a computer scientist) is enriched with some additional structuring markup, indicating properties of text portions and maybe hierarchically structuring the document's content. Typical for the semi-structured paradigm are mixed content (text or elements are admitted as children of an element), optional and recursively embedded elements, and the significance of element order. An example is the MUC markup for named entities as shown on page 70.

The structured data paradigm corresponds to strongly typed data, such as tables in relational databases or sorted graphs in object hierarchies or databases (Abiteboul, 1997). Recursion, mixed content or optionality is not present or only limited (in case of optional attributes, default values can be declared in the DTD). The order of elements may be relevant to some extent. Application examples are storage of address book entries, stock quotes, flight schedules, sales orders or application data exchange through XML documents.

While strictly structured documents can be mapped to a simple relational database schema, this is generally not true for semi-structured markup. Here, mappings to relational database schemata are possible, but often result in huge, complicated database schemata[5].

Many XML-based standards or quasi-standards, described by DTDs or XML schemata have been defined by various institutions, companies or consortia, and the number is still growing. While SGML was primarily used for text document markup for publishing and as persistent vendor-independent text storage format, XML usage has quickly been extended to various applications that use markup as a kind of abstract syntax and container format for inter-application data exchange.

4.2.5 XML as Carrier Syntax for Computer Languages

Formally, XML can be characterized as a parenthesis grammar (Knuth, 1967; McNaughton, 1967), a special form of context-free grammars with rules of the form $N \rightarrow cMc'$ where N and M are non-terminals, and c, c' the parenthetic beginning and ending elements. For this reason, XML is often used as a *carrier syntax* for programming and other formal description languages that could also be expressed in a Backus-Naur form. Examples are

- XSLT, a programming language for transforming XML documents we will describe later
- XML schema, an XML-encoded description of XML document formats similar to DTDs
- Apache ant, a scripting language similar to that of the Unix `make` tool
- SVG (Scalable Vector Graphics), a language to describe vector graphics
- InkML, a data format for representing ink entered with an electronic pen or stylus
- MathML, a specification for describing mathematics as a basis for machine to machine communication

[5] under the assumption that full text storage of XML documents is not a serious, practical alternative because queries on the structure and content cannot be supported then without fully parsing the data for each query.

- SMIL (Synchronized Multimedia Integration Language)
- OASIS Open Document Format for Office Applications (word processing etc.)

4.2.6 XML as Open Data Structure

One of the major differences between SGML and XML is that XML validation is optional, i.e., an XML document is processable even if no DTD or schema is given (provided that it is well-formed). Moreover, even with a DTD or schema, much freedom is left by admitting optional elements and attributes.

This turns XML into a kind of *open data structure* or *lingua franca* for ad-hoc markup and storage or exchange of structured information of any kind, such as software configuration data or document metadata and linguistic markup, e.g. for rapid prototyping. However, not defining a DTD or schema always bears the risk of violating assumptions that are made by XML-processing software (e.g. unexpected element names *etc*).

4.2.7 Linguistic Markup

Markup (enrichment) of text with *linguistic information* comes close to the original GML markup idea of adding tags to text and corresponds to the semi-structured document paradigm of XML and SGML. The linguistic term for (manually) added markup is *annotation*. The term *linguistic annotation* covers any descriptive or analytic notation applied to raw language data.

Corpora are text collections, e.g. from newspapers, or speech transcriptions *etc*, serving as observation data.

Annotated corpora – text enriched with linguistic markup – play an important role e.g. for machine learning and evaluation of automated linguistic processing.

For machine learning, manually annotated corpora are used to train a statistical model (maybe on a initially small corpus). In a bootstrapping way, markup is then generated on new text through NLP based on the trained model.

This automatically generated markup is then corrected manually or semi-automatically, e.g. by repairing wrong analyses, selecting the readings a human would understand, or adding information for unknown words or constructions. From the corrected markup, an improved model can then be trained. The whole process can be iterated, e.g. to obtain better models for specific text classes or sources (cf. Figure 4.2).

Automatic and manual (or semi-automatic) markup can thus be conceived as an interactive process and it is therefore only natural to use XML as output format for NLP components. Moreover, it is possible to exchange and combine linguistic markup of NLP components via XML and XML-processing tools – one of the central ideas underlying this thesis.

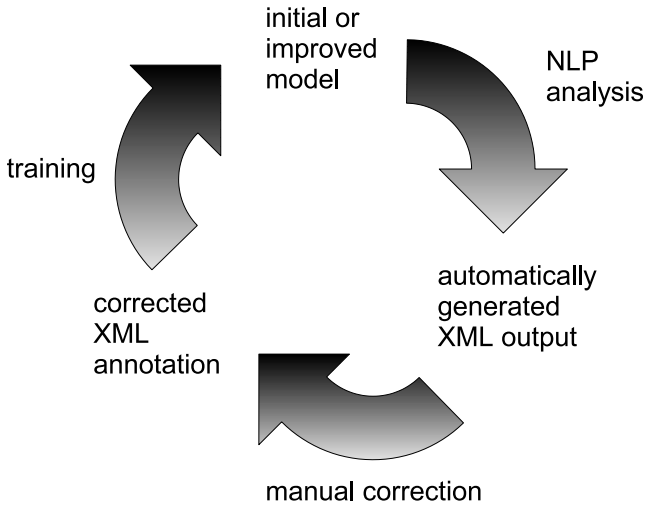


Figure 4.2: Training-annotation/correction-NLP analysis cycle

4.2.8 Standards for Linguistic Markup

There are a number of standards and proposals for linguistic annotation, some going even back to the time when XML has not yet been invented. We will briefly discuss some of them . Although processing of annotated corpora is not in the focus of our thesis, standards and especially XML plays an important role because NLP may have corpora at both ends – NLP components may both use them as input (e.g. for training statistical models) and produce markup as output, e.g. automatically annotated corpora. Thus, corpus annotation frameworks and NLP-generated markup are bound up with each other.

However, because of the limitations of NLP components, the markup of a specific NLP component typically is only a subset of a corpus annotation scheme which often is designed to cover a broader variety of linguistic phenomena. Generally speaking, the same criteria that are crucial for corpus annotation also are important for NLP component output. Ide and Romary (2002) e.g. name the consistency of tag set and encoding schema, recoverability of source text, validatability, processability, extensibility, compactness and readability.

4.2.8.1 Text Encoding Initiative (TEI)

In the early days of linguistic corpus annotation, SGML was proposed and used for corpus markup. Already in 1987, the separation of text and layout for content markup, and the independence of systems, hardware, software-specific data formats has been the motivation for the Text Encoding Initiative (TEI), a consortium

of institutions and projects related to history, literature, linguistics, philology *etc*, to setup a standard for content-oriented document annotation.

> *'The Text Encoding Initiative (TEI) Guidelines are an international and interdisciplinary standard that facilitates libraries, museums, publishers, and individual scholars represent a variety of literary and linguistic texts for online research, teaching, and preservation.'*
> (Sperberg-McQueen and Burnard, 1994)

Text structures (DTDs) have been defined for e.g. prose, verse, drama, speech transcription, dictionary, terminology, but also for linguistic information such as part-of-speech tags or inflection, and even feature structures. The proposed tag sets are very comprehensive, and organized in modular DTDs. The first series of guidelines was published in 1990 as TEI P1. In 1998, TEI has adopted XML as additional markup syntax.

Elements for linguistic markup are e.g.

- `<s>` for sentence-like division of a text
- `<cl>` for grammatical clause
- `<phr>` for grammatical phrase
- `<w>` for grammatical (not necessarily orthographic) words
- `<m>` for grammatical morpheme
- `<c>` for character

An XML example taken from the TEI P5 Guidelines (Sperberg-McQueen and Burnard, 1994)

```
<p>
 <s>
  <cl type="finite declarative" function="independent">
   <phr type="NP" function="subject">Nineteen fifty-four,
     <cl type="finite relative declarative" function="appositive">
       when <phr type="NP" function="subject">I</phr>
       <phr type="VP" function="predicate">was eighteen years old</phr>
     </cl>
   </phr>,
   <phr type="VP" function="predicate">
     <phr type="V" function="main verb">is held</phr>
     <phr type="NP" function="complement">
       <cl type="nonfinite" function="predicate nom.">
         <phr type="V" function="copula">to be</phr>
         <phr type="NP" function="predicate nom.">a crucial turning point
            <phr type="PP" function="postmodifier">in
              <phr type="NP" function="prep.obj.">the history
                <phr type="PP" function="postmodifier">
```

```
                of the Afro-American</phr>
            </phr>
          </phr>
          -
          <phr type="PP" function="appositive postmodifier">for
            <phr type="NP" function="prep.obj.">the U.S.A.
              <phr type="PP" function="postmodifier">as a whole</phr>
            </phr>
          </phr>
        </phr>
        -
        <phr type="NP" function="appositive pred.nom.">the year
          <cl type="finite relative" function="adjectival">
            <phr type="NP" function="subject">segregation</phr>
            <phr type="VP" function="predicate">
              <phr type="V" function="main verb">was outlawed</phr>
              <phr type="PP" function="postmodifier">
                   by the U.S. Supreme Court</phr>
            </phr>
          </cl>
        </phr>
      </cl>
    </phr>
  </phr>.
 </cl>
</s>
</p>
```

Although TEI is frequently referenced by other approaches and annotation schemata and is one of the oldest annotation standardization efforts, many corpora are not using the TEI schema, but other, simpler ad-hoc annotation schemata designed for the actual, specific needs. The main reason is that TEI suffers from the SGML sickness that in aiming at describing any phenomenon and foreseeing every case and feature, the schema becomes complex and confusing (cf. Witt 1998).

At the same time, TEI leaves room for more specific extensions (therefore the term ‘guidelines’), and is organized in a modular way. However, there are also aspects that TEI didn’t cover at all, such as semantic annotation, and that are not easy to make conforming to the guidelines. Although it is possible to add extensions to a TEI schema, people often end up in defining their own, TEI-independent schema, taking TEI as a start point. In Chapter 5, we will argue why this does not do much harm from a technical perspective. However, the question remains about the value of a standard that is too general on the one side, and too inflexible on the other side.

4.2.8.2 CES and XCES

CES (Corpus Encoding Standard; Ide 1998) has been developed as an application of TEI (firstly in SGML) and as part of the EAGLES (Expert Advisory Group on Language Engineering Standards) guidelines. As such, CES lays a much stronger focus on (linguistic) corpus annotation than TEI did. CES extends TEI specifi-

cations and makes them more specific where appropriate, and on the other hand limits the TEI scheme to include only that subset of tags that is relevant for corpus annotation.

Like TEI, CES has migrated to XML under the name XCES (Ide *et al.*, 2000a; Ide and Romary, 2001, 2002). The approach is in a clear way top-down-oriented, and the basic concepts have also influenced the ISO standardization efforts for linguistic annotation we will describe in the next section. XCES defines an abstract *Structural Skeleton* for syntactic structures that are common to all (in the corpus world) possible annotation schemes, and a *Data Category Registry* that defines general categories such as phrase types in a hierarchy using RDF (cf. Section 4.2.9).

Both the Structural Skeleton and the Data Categories are instantiated for a specific annotation scheme (called AML, the Annotation Markup Language), where e.g. the noun phrase category is defined to be an attribute value or the name of an element as well as the rest of syntactic structure. This AML corresponds then to and can be written as an XML DTD.

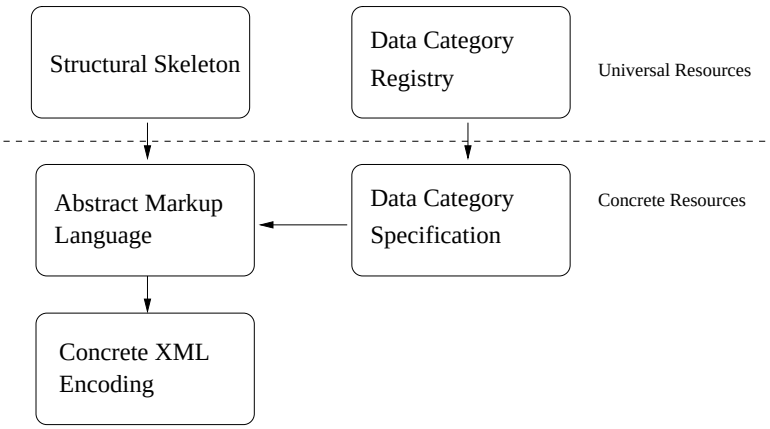


Figure 4.3: XCES annotation framework (simplified)

What makes XCES interesting for our needs in deep-shallow integration is the abstract top-down view to annotation, the concrete realization, and the clear adoption of XML transformation to realize the top-down approach of XCES in implementations, including concepts such as stand-off annotation and linking of annotation. These points will be discussed later.

4.2.8.3 ISO

A recent development is the ISO standardization of linguistic markup for computational linguistics, computerized lexicography, and language engineering, defining '*standards by specifying principles and methods for creating, coding, processing and managing language resources, such as written corpora, lexical corpora, speech corpora, dictionary compiling and classification schemes. These standards*

will also cover the information produced by natural language processing components in these various domains.'[6].

This claim makes ISO fit into our goal of using XML[7] for NLP component integration and it turns out that the ISO working group is mainly complementing, and partly overlapping existing TEI approaches. TEI, e.g. is not specific enough on morphology and although feature structure markup is defined by TEI, ISO tries to cover it in more principled and elaborated way (Lee *et al.*, 2004).

Moreover, ISO also aims at putting more focus on multilingual, multimedia and multimodal aspects than TEI did so far. However, the standardization process by the joint ISO/TEI working group (TC 37 SC 4) is still ongoing, and only one (not so near) day could become the ISO DIS 24610 standard. Another focus of ISO will also consist in standardization of non-textual linguistic resources such as lexica which are also less covered by TEI.

4.2.9 Further XML Standards Related to Linguistic Processing

Besides morphosyntactic markup of written text, there exist already several established XML standards for lexicon and terminology exchange, speech and the Semantic Web. All of them are closely related to linguistic markup and are worth a short discussion here, because they play important roles in linguistic processing – and XML being their common basis, makes it easy to integrate them in the framework we envisage. However, we will discuss them only briefly because they are not directly used and necessary for the deep-shallow integration scenarios we will focus on in this thesis.

The relation of speech, lexicon and terminology to linguistic processing should be obvious, and the Semantic Web will without doubt play an important role as application and aim of natural language processing in the near future, as will also be discussed in Section 9.7 and 9.10.2.

Semantic Web Being promoted by the World Wide Web Consortium, the following knowledge representation languages for the Semantic Web are all based on XML syntax (examples will be shown in Section 9.7 and 9.10.2).

- RDF (Resource Description Framework; Klyne and Carroll 2004), a simple language to describe objects (e.g. web resources) in form of subject-predicate-object triples ('statements'). As already briefly discussed in Section 4.2.2, the fact that RDF uses XML syntax does not imply that a formal semantics is defined. This is only defined in the following frameworks that themselves build on RDF.

[6] from `http://www.tc37sc4.org`

[7] The ISO working group has made a clear commitment to build 'on and around W3C standards' such as XML, RDF, OWL, SOAP.

- RDF Schema (or RDF Vocabulary Description Language; Brickley and Guha 2004) is a language to describe vocabularies, similar to classes in object-oriented programming languages, and uses RDF itself as base syntax,
- OWL (Web Ontology Language; Bechhofer *et al.* 2004) with its sub-languages OWL-light, OWL-DL and OWL-full is designed to define ontologies in a description logic-like manner (Baader *et al.*, 2003). OWL uses RDF as base syntax in the same way RDF Schema does.

Speech

- SSML (Speech Synthesis Markup Language[8]) has been designed to model and assist the generation of synthetic speech in applications. Aspects of speech generation such as pronunciation, volume, pitch, rate, *etc* can be controlled across different synthesis-capable platforms. This can be used as the speech front-end in NLP-based applications.
- VoiceXML (Voice Extensible Markup Language[9]) defines a standard for creating audio dialogs that feature synthesized speech, digitized audio, recognition of spoken and DTMF key input, recording of spoken input, telephony, and mixed-initiative conversations. Dialogs can be modeled e.g. by creating templates to be filled in by speech dialogs.

Lexicon and terminology interchange

- OLIF (Open Lexicon Interchange Format[10]) is an XML-based interchange format for lexical and terminological information (databases), originally developed in the OTELO project (Open Translation Environment for LOcalization; Lieske *et al.* 2001). It is e.g. used for translation memories in industrial machine-assisted translation of written documentations.

4.3 Common Properties and Challenges of XML-Based Linguistic Annotation

Although some popular annotated corpora such as the Penn Treebank (Marcus *et al.*, 1994) or the German NEGRA treebank (Skut *et al.*, 1998) are in non-XML format, the trend goes to XML annotation of corpora and even existing non-XML treebanks are converted to XML (cf. Teich *et al.* 2001). As already motivated in Section 4.2, manual corpus annotation and automatic annotation produced by NLP components are closely related, and therefore both benefit from the advantages of XML encoding such as

[8] `http://w3c.org/TR/speech-synthesis/`
[9] `http://w3c.org/TR/voicexml20/`
[10] `http://www.olif.net`

- having a common base syntax that can be seen as a kind of 'abstract syntax' generalizing over low-level syntax for grouping, e.g. parentheses *etc*, separation of entities
- Unicode as comprehensive character encoding framework supporting true multilinguality
- powerful, but – in contrast to SGML – not too complicated document structuring grammar syntax
- optional validation of document structure
- wide range of existing powerful software tools for parsing, editing, visualization, transformation

In analogy to XML corpus annotation, XML output of NLP components is becoming increasingly important and popular. Some recent NLP components directly produce an XML corpus format as output. An example is the LingPipe implemented by Bob Carpenter (Carpenter, 2005), a statistical named entity recognizer that produces XML output compatible with the MUC annotation format (Message Understanding Conference; Grishman and Sundheim 1996):

```
<?xml version="1.0" encoding="UTF-8"?>
<DOCUMENT>
  <P>
    <sent>
      <ENAMEX id="6" type="PERSON">George W. Bush</ENAMEX> is the
      president.
    </sent>
    <sent>
      <ENAMEX id="6" type="MALE_PRONOUN">He</ENAMEX>
      is the commander in chief of the
      <ENAMEX id="7" type="LOCATION">United States of
        America</ENAMEX>.
    </sent>
  </P>
</DOCUMENT>
```

Similar for taggers, e.g. the part-of-speech-tagged example sentence

Die$_{ART}$ *Kriminalpolizei*$_{NN}$ *verfolgte*$_{VFIN}$ *die*$_{ART}$ *Bankräuber*$_{NN}$ *bis*$_{APPR}$ *zur*$_{APPRART}$ *niederländischen*$_{ADJA}$ *Grenze*$_{NN}$.

from Section 4.1.3 could be annotated in XML format as

```
<?xml version='1.0'?>
<text>
  <w id="T0" pos="ART">Die</w>
```

```
    <w id="T1" pos="NN">Kriminalpolizei</w>
    <w id="T2" pos="VVFIN">verfolgte</w>
    <w id="T3" pos="ART">die</w>
    <w id="T4" pos="NN">Bankräuber</w>
    <w id="T5" pos="APPR">bis</w>
    <w id="T6" pos="APPRART">zur</w>
    <w id="T7" pos="ADJA">niederländischen</w>
    <w id="T8" pos="NN">Grenze</w>
    <w id="T9" pos=".">.</w>
  </text>
```

and furthermore enriched with chunk information

```
<?xml version='1.0'?>
<text>
  <chunk cat="NP">
    <w id="T0" pos="ART">Die</w>
    <w id="T1" pos="NN">Kriminalpolizei</w>
  </chunk>
  <w id="T2" pos="VVFIN">verfolgte</w>
  <chunk cat="NP">
    <w id="T3" pos="ART">die</w>
    <w id="T4" pos="NN">Bankräuber</w>
  </chunk>
  <chunk cat="PP">
    <w id="T5" pos="APPR">bis</w>
    <w id="T6" pos="APPRART">zur</w>
    <w id="T7" pos="ADJA">niederländischen</w>
    <w id="T8" pos="NN">Grenze</w>
  </chunk>
  <w id="T9" pos=".">.</w>
</text>
```

and finally with PCFG parser output for topological sentence fields (cf. Figure 4.1 on page 55).

```
<?xml version='1.0'?>
<text>
  <root>
    <cl_v2>
      <vf_topic>
        <chunk cat="NP">
          <w id="T0" pos="ART">Die</w>
          <w id="T1" pos="NN">Kriminalpolizei</w>
        </chunk>
      </vf_topic>
      <lk_vfin>
        <vvfin>
          <w id="T2" pos="VVFIN">verfolgte</w>
        </vvfin>
```

```
      <lk_vfin>
      <mf>
        <chunk cat="NP">
          <w id="T3" pos="ART">die</w>
          <w id="T4" pos="NN">Bankräuber</w>
        </chunk>
        <chunk cat="PP">
          <w id="T5" pos="APPR">bis</w>
          <w id="T6" pos="APPRART">zur</w>
          <w id="T7" pos="ADJA">niederländischen</w>
          <w id="T8" pos="NN">Grenze</w>
        </chunk>
      </mf>
    </cl_v2>
    <w id="T9" pos=".">.</w>
  </root>
</text>
```

The tagger and chunker markup above has been generated automatically by TnT and Chunkie (Skut and Brants, 1998), details in Section 9.5.2.1 and 9.5.3.1.

There are two things that can be seen from the examples: (1) there is a natural way of formulating hierarchical (tree) structure by embedding elements (chunks embed words), i.e., the linguistic tree structure is reflected in the XML tree, (2) markup can be optional. The chunker e.g. does not provide chunk information for the VP (token T2 in the above sentence). This optional markup reflects the agnostic way in which shallow processors typically work: information may be partial, and this can easily be modeled in XML by omitting tags or attributes.

We present another example in Figure 4.4 for even deeper nested hierarchical annotation produced by the shallow processor SPPC (Piskorski and Neumann, 2000) that also includes text structuring elements such as S for sentence and PARAGRAPH for paragraphs, and additional markup of named entities (NE) which in turn can be part of chunks.

Although this example looks elegant in that it combines different strata of linguistic analysis within a nested structure, this kind of representation is either only possible after full disambiguation of multiple readings of the different analysis levels (such as morphology, chunks, named entities), or – as in this case – if only a single processing component produces such output that ensures the tree structure of the output annotation.

In the general case, if dedicated, independent NLP components (e.g. a morphology/lexicon component, named entity recognizer, and chunker) are involved in generating XML output for the same text, the results may contain different spans of recognized entities and multiple readings for analyses. The resulting structure would contain crossing elements, which is not admitted in well-formed XML.

```
<SPPC_XML>
  <PARAGRAPH>
    <S>
      <CHUNK type="NP">
        <NE type="PERSON" subtype="UNTITLED">
          <W tc="first_capital_word" pos="N" stem="george">George
          </W>
          <W tc="initial_capital_period">W.</W>
          <W tc="first_capital_word" pos="N" stem="bush">Bush</W>
        </NE>
      </CHUNK>
      <CHUNK type="VF_AUX_FIN">
        <W tc="lowercase_word" pos="AUX" stem="sei">ist</W>
      </CHUNK>
      <CHUNK type="NP">
        <NE type="POSITION" subtype="POSITION">
          <W tc="first_capital_word" pos="N" stem="praesident">
            Praesident</W>
        </NE>
      </CHUNK>
      <CHUNK type="NP">
        <W tc="lowercase_word" pref="0" pos="PART DEF"
           stem="der d-det">der</W>
        <NE type="LOCATION" subtype="LOCATION_NP">
           <W tc="all_capitals_word">USA</W>
        </NE>
      </CHUNK>
      <W tc="separator_symbol">.</W>
    </S>
  </PARAGRAPH>
</SPPC_XML>
```

Figure 4.4: SPPC analysis for 'George W. Bush ist Präsident der USA'

4.3.1 Standoff Annotation

The standard solution for such cases is *standoff annotation* (Thompson and McKelvie, 1997). Instead of preserving the nested structure, the different analysis strata are separated into multiple output documents, and linked via ID attributes or other linking mechanisms such as XLink or XPointer, or by linking indirectly using character spans (start and end character in original input text) or a common tokenization. The standoff documents together form a graph. As a variation, the annotation

can go to a single output document containing different sections[11].

The advantage of standoff annotation is that the resulting document structures remain simple, and that any kind of linking (and also multiple linking mechanisms) can be applied. In most cases, the simple ID/IDREF mechanism supported by the XML standard can be used that allows ID attributes to be declared as unique identifiers in DTD (partial declaration of the ID attribute types is sufficient, cf. the example below) that can then be referred to as targets with IDREF attributes or a special XPath id function mechanism we will discuss later. The only support a (validating) XML parser provides for these directed links is validation for uniqueness of ID target labels and availability of all IDREF targets within the document in an efficient way.

Example in DTD (XML Schema provides an analogous facility):

```
<?xml version="1.0"?>
<!DOCTYPE standoff [
  <!ATTLIST w id ID #REQUIRED >
  <!ATTLIST ne parts IDREFS #REQUIRED >
]>
<standoff>
  <tokens>
    <w id="W0">Gerhard</w>
    <w id="W1">Schröder</w>
  <tokens>
  ...
  <namedentities>
    <ne parts="W0 W1"/>
  </namedentities>
</standoff>
```

The ID/IDREF mechanism shown here for intra-document linking also works for inter-document linking as long as the document management is handled by the XML processor or in XPath (described later) using the `document()` function.

W3C has defined additional and more elaborated standards for linking, namely XPointer (DeRose *et al.*, 2002) and XLink (DeRose *et al.*, 2001). While XLink introduces a fixed set of linking types such as bidirectional links, link groups, or link titles and roles and uses URI to identify link targets, XPointer with its sub-language XPath (Clark and DeRose, 1999) supports pointing to e.g. complex descriptions of sub-document ranges. However, as most XML processing software does not support these extensions[12], the simple ID mechanism (or alternatively an ordinary, dedicated attribute) is used in most cases and systems.

[11] A later, related term for standoff annotation used in a more general context than NLP is *concurrent markup* or *extreme markup* (Durusau and O'Donnell, 2002). In the computational linguistics context, the terms *multi-dimensional markup*, *multi-level* or *multi-layered annotation* have also been used in recent years, both for standoff markup in multiple documents and in a single document.

[12] There are multiple reasons for this, e.g. the problem of different uses and representations of the pointer targets, and more profanely, patent issues.

4.3.2 Related Annotation Standards

Standoff annotation is also useful for annotating translated corpora (each language in a separate layer), or time-aligned corpora, typically transcription of speech and video, as in the Universal Transcription Format (UTF; of Standards and Technology 1998) by the US National Institute of Standards and Technology (NIST), cf. Figure 4.5 for an example.

Here, a timeline is defined in the document which the annotation makes reference to. Various existing time-aligned speech annotation formats are discussed in Bird and Liberman (2001) and a generalization called *annotation graphs* is presented. Formally, annotation graphs are labeled directed acyclic graphs with a time function that assigns to each graph node an element of the timeline (an ordered set).

```
<turn speaker="Roger_Hedgecock" spkrtype="male" dialect="native"
      startTime="2348.811875" endTime="2391.606000"
      mode="spontaneous" fidelity="high">  ...
  <time sec="2378.629937"/> now all of those things are in doubt
  after forty years of democratic rule in
  <enamex type="ORGANIZATION">congress</enamex>
  <time sec="2382.539437"/>
  <breath/>
  because
  <contraction e_form="[you=>you]['ve=>have]">you've
  </contraction> got quotas
  <breath/>
  and set-asides and rigidities in this system that keep you
  <time sec="2387.353875"/>
    on welfare and away from real ownership
  <breath/>
  and <contraction e_form="[that=>that]['s=>is]">that's
  </contraction> a real problem in this
  <overlap startTime="2391.115375" endTime="2391.606000">country
  </overlap>
</turn>
<turn speaker="Gloria_Allred" spkrtype="female" dialect="native"
      startTime="2391.299625" endTime="2439.820312"
      mode="spontaneous" fidelity="high">
  <overlap startTime="2391.299625" endTime="2391.606000">well i
  </overlap> think the real problem is that %uh these kinds of
  republican attacks
  <time sec="2395.462500"/>
  i see as code words for discrimination  ...
</turn>
```

Figure 4.5: XMLified example of the Universal Transcription Format (UTF)

Furthermore, there is an immediate analogy between the TIPSTER annotation

format (Grishman, 1997) and standoff annotation. In TIPSTER, all annotations are expressed in terms of byte offsets into the original text. The byte offsets to text could be interpreted as *timeline* because they form an ordered set[13].

The ACE (Automatic Content Extraction)[14] program by the American National Institute of Standards and Technology (NIST) generalizes this even further to text (newswire), speech (ASR) and image (OCR) annotation in one DTD. The ACE DTD defines a reference key annotation of EDT entities and RDC relations (EDT = entity detection and tracking task, RDC = relation detection task). The ACE annotation format foresees markup for persons, organizations, geographical-political entities, locations and human-made artifacts of architecture and civil engineering.

An entity can be identified via a character span or sequence (if it is text), a time span (if it is speech or audio-video signal), or a list of bounding box coordinates (if it is scanned text; the position description of a word may be a list of boxes because a word may be wrapped in lines). Thus, this annotation format allows for flexible markup not only of text and OCR input, but also of speech and multimedia streams. The relevant fragment of the DTD can be found in the DTD Appendix on page 285.

Discontinuous constituents (e.g. topicalization, scrambling, clause union, pied piping, extraposition, split NPs and PPs) are linguistic phenomena that exceed the standard XML element tree, but can be easily modeled by standoff annotation. An example is the sentence 'Turn the switch off' where turn off is the discontinuous constituent. By linking the constituent parts via ID attributes, e.g. as shown here[15], even distant phrase elements are tied together.

```
<s>
  <w id="W0">turn</w>
  <w id="W1">the</w>
  <w id="W2">switch</w>
  <w id="W3">off</w>
  <vp id="V0" constituents="W0 W3"/>
  <np id="N0" constituents="W1 W2"/>
</s>
```

An XML-independent discussion of an annotation scheme for free word order languages (including discontinuous constituents) is e.g. addressed in Skut *et al.* (1997).

Mengel and Lezius (2000) describe a largely linguistic theory-neutral XML format (TIGER XML) based on the non-XML NEGRA (Skut *et al.*, 1998) annotation schema giving up the pure tree model in favor of an instrument to model crossing edges in the analysis. Both annotation formats allow crossing edges (edge_1_5 and edge_1_6 in the example below).

[13] A disadvantage of the byte offset representation is that some character encodings may have variable byte-length representation of characters (e.g. for Japanese). Here, XML technology based on Unicode and character offset representations has a clear advantage over byte offsets.

[14] `http://projects.ldc.upenn.edu/ace/`

[15] Typically, standoff annotation is distributed over multiple XML documents. However, it is also possible and sometimes convenient to put multiple standoff layers into one XML document.

```
<s id="s2" href="#id(n1_502)"/>
<n id="n1_500" cat="S">
  <edge id="edge1_1" label="SB" href="#id(w1_4)"/>
  <edge id="edge1_2" label="HD" href="#id(w1_5)"/>
</n>
<n id="n1_501" cat="NP">
  <edge id="edge1_3" label="NK" href="#id(w1_0)"/>
  <edge id="edge1_4" label="NK" href="#id(w1_1)"/>
  <edge id="edge1_5" label="RC" href="#id(n1_500)"/>
</n>
<n id="n1_502" cat="S">
  <edge id="edge1_6" label="HD" href="#id(w1_2)"/>
  <edge id="edge1_7" label="SB" href="#id(n1_501)"/>
</n>
<w id="w1_0" word="Ein" pos="ART"/>
<w id="w1_1" word="Mann" pos="NN"/>
<w id="w1_2" word="kommt" pos="VVFIN"/>
<w id="w1_3" word="," pos=","/>
<w id="w1_4" word="der" pos="PRELS"/>
<w id="w1_5" word="lacht" pos="VVFIN"/>
<w id="w1_6" word="." pos="."/>
```

Additional crossing edges are introduced by the SALSA extension to the TIGER XML format (Erk and Padó, 2004) that adds semantic role annotation which can cross syntactic boundaries.

4.3.3 Summary

In this chapter, we have examined shallow linguistic markup for the most common shallow natural language processing tasks. We have introduced XML and its role as *lingua franca*, open data structure and abstract syntax for linguistic annotation.

One of the main advantages is flexibility and extensibility (role of elements and attributes). We have shown the close relation between corpora and NLP component output, and noticed the increasing availability of tools and standards for both corpora and online natural language processors.

We have learned that the work on standardization of corpus annotation is also related to standardization of NLP component output formats, e.g. namings and schemata for morphologic, syntactic and semantic markup, and that the same criteria that are crucial for corpus annotation also are important for NLP component output, e.g. consistency of tag set and encoding schema, recoverability of source text, validatability, processability, extensibility, compactness and readability.

After a discussion of the deep-shallow mapping problem in the next chapter, we will propose how deep and shallow components can be integrated using XML transformations.

Chapter 5

Deep-Shallow Integration by Transformation

In this chapter, we first discuss general problems that arise when representations generated by different NLP components with different granularities, spans and with different namings for linguistic entities *etc* are combined. We focus here on deep-shallow integration. However, some of the problem classes also arise when multiple shallow systems are combined. In the second part, we will propose a technical solution by transformation of XML annotation and discuss some variants and alternatives.

5.1 The Deep-Shallow Mapping Problem

The main goal when integrating deep and shallow natural language processing components is increased robustness of deep parsing by exploiting information for words (or more general, character sequences) that are not contained in the deep lexicon. The type of unknown words or word sequences, e.g., can be guessed by statistical models, domain-specific expressions or time expressions can be parsed efficiently and reliably with finite-state devices and resources.

Named entities such as proper names or location names can be recognized by statistical and/or rule-based components and with the help of gazetteers etc. Moreover, the search space of a deep parser can also be shaped and reduced by preprocessing of sentence structure, partial parsing and ranking of possible readings for a word class with statistical methods.

Although the list of possible advantages that could be gained by combining deep and shallow methods looks promising, many problems occur if it comes to a real integration of (pre-)existing components. For economical reasons, deep-shallow integration is based on the assumption that (re)designing shallow or deep components from scratch for a smoother integration is both expensive and not really wanted because the components also function as standalone modules in other application contexts that are not to be neglected. Hence, a translation or mapping

step in between is necessary in most cases.

In the following, we will briefly discuss the main problem classes of deep and shallow component integration and describe possible solutions on a more abstract level. The architecture implementations described in Chapters 8 and 9 will provide concrete solutions for the integration of existing NLP components.

Namings, types, welltypedness Although there are ongoing standardization efforts for linguistic markup *etc*, there is no unique and common naming standard for linguistic entities and analyses. Adjectives e.g. are named JJ, adj, ADJ, A or adjective in different NLP components. In many cases, there is no or no one-to-one correspondence[1]. In the former case (no correspondence), information from shallow analysis that has no correspondence in the deep grammar may be lost. However, it remains accessible in standoff annotation of the shallow component, e.g. for an application. The latter case (no one-to-one correspondence) is discussed below as granularity of classification problem.

However, it is not only the names of analyzed entities that have to be translated for deep-shallow integration. Efficient HPSG parsers e.g. require feature structures to be well-typed, i.e., only attributes with appropriate values are admitted for a specific type. A non-welltyped feature structure would result in parse failure (cf. Chapter 3).

An example is morphology information that can consist of a bunch of feature-value pairs. These have to be defined and must be well-typed in the deep grammar.

The solution is to provide (total) mappings that always return the appropriate type that is admitted and expected in the target component, e.g. by using translation tables or rules.

Granularity of classification Granularity designates the number of classes an NLP component recognizes for an entity.

A simple example for different granularities is the token type in tokenization. While some tokenizers only distinguish words from punctuation characters and numbers, others provide a finer-grained classification, e.g. including ordinal and cardinal numbers, hypotheses on sentence boundaries, email addresses, abbreviations *etc*. The shallow processing system SPPC (Piskorski and Neumann, 2000) e.g. distinguishes 52 token classes.

Besides separation of punctuation from words, in HPSG grammars tokenization is mainly used for recognizing open word classes such as numbers, dates *etc* that are mapped to lexical types. Thus, an external tokenization must match the classes foreseen by the deep grammar, or a mapping must be provided.

Another example are part of speech tags. While some PoS taggers such as TnT (Brants, 2000) recognize more than 20 different part-of-speech types for English and German, only a few basic categories are typically defined in HPSG such as

[1] Paris (2002) discusses a closely related problem of different tag sets in annotated corpora.

noun, verb or adjective. Additional information related to word classes may be defined in additional features and types.

Part of speech types that could be assigned to different classes such as adjectival verbs as in *der bebende Berg* could e.g. be classified by a shallow component as verb form and as adjective by a deep grammar (or *vice versa*).

If the granularities differ between shallow and deep NLP components, a mapping between the classes has to be defined manually. The boundaries may not always be obvious. It may be the case that different instances belonging to one class in the source component have to be mapped to different classes in the target component and *vice versa*.

Moreover, the concept of a type hierarchy with inheritance that forms the basis for deep (HPSG-based) grammars, has no correspondence in shallow systems[2]. Hence, implications that can be expressed by using general (super)types in a type system of a deep grammar have to be enumerated as subtypes in non-hierarchical classification e.g. of shallow NLP systems.

Even if granularities of classification are similar or isomorphic, the problem remains to find the correct mapping because names and underlying concepts may differ.

Because the granularity is determined by underlying linguistic resources (grammars, trained models based on annotated corpora), it is normally too expensive to adapt their definitions. The solution is hence to map the different classes by translation tables or rules created manually on the basis of the underlying resources (models, grammars, theories).

Partial analysis results Shallow natural language processing components typically deliver partial analysis results. The absence of an analysis e.g. of a word may denote either that the word does not belong to a class, or that no information was available about the word. This could be misinterpreted by a deep system relying on the shallow input.

Because a deep grammar relies on the fact that each input word must be known, and typically will fail to deliver a full parse of the whole sentence in case of partial information, this could form a problem for a deep parser that tries to fill lexical gaps by shallow input.

A fall-back could be to (a) rely on shallow input only in case of no full deep analysis is available (b) use fragments remaining on the deep parse chart in that case. Both solutions will be discussed in Chapter 9.

An example for partial analysis is named entity recognition. If a word is not marked as named entity (NE), this does not necessarily mean that it cannot be a named entity, because no NE recognition component has complete knowledge about all proper and location names.

To cope with partial information, a solution is to use the underspecified, minimal information that is available as fall-back. For an example, we consider the

[2] except in *SProUT*, cf. Chapter 7.

case that a word is not in the deep lexicon. Normally, a deep analysis would fail because of the gap in the parser's chart. The analysis of a shallow named entity recognizer would be used instead (which would cause a generic lexicon entry for the named entity to be put on the chart of the deep parser).

If the named entity recognizer also gives no hint on the word, the output of a statistical part-of-speech tagger could be used as final fall-back and a generic lexicon entry for the guessed part-of-speech would be added to the chart. If information is available neither from the shallow nor from the deep analysis, then the deep parser could try a fragmentary deep parse.

Structural Richness Richness is related to the information complexity or structural complexity of NLP analyses (in contrast to granularity which only comprises the number of recognized classes).

A deep lexicon entry for a verb e.g. contains information on the subcategorization frame of the verb, i.e., which kinds of object the verb requires and which semantic roles are assigned to them, such as

$$\begin{bmatrix} \textit{give} & \\ \text{POS} & \textit{v} \\ \text{AGENT} & \textit{giver} \\ \text{OBJECT} & \textit{given} \\ \text{RECIPIENT} & \textit{givee} \end{bmatrix}$$

for the verb to give. A shallow lexicon typically only contains the information that to give is a verb.

Similarly, deep grammars deliver a highly structured syntactic analysis of a sentence while shallow parsers in most cases return more or less flat sentence structure, if any.

To give an example for the opposite case (structured shallow analysis), a named entity recognition component may deliver detailed information on the structure, type and nature of a named entity,

$$\begin{bmatrix} \textit{NE\_person} & \\ \text{TITLE} & \texttt{"Dr."} \\ \text{SURNAME} & \texttt{"Reinhart"} \\ \text{GIVENNAME} & \texttt{"Rühmenkorff-Bohlander"} \end{bmatrix}$$

whereas a deep grammar may only provide or foresee information on the span and part-of-speech type: $[\text{Dr. Reinhart Rühmenkorff-Bohlander}]_{\text{NE\_person}}$.

Structural differences are probably the most likely case for mismatches between shallow and deep analyses, e.g. chunks are typically non-recursive in shallow chunkers, while recursively branching in deep analyses, and even if they were

recursive also in shallow analyses, then the branching structure might be different (left vs. right-branching)[3].

The difficulty is to match or map structures of differing richness without loss of information, in order to get a maximally rich structure. Usually, unification is the operation that returns such structures, but because of different granularities and namings in deep and shallow analyses, the unification operation may not always be feasible and well-defined.

The general solution would be to map different structures to each other depending on their type and on the basis of the underlying models, theories and grammars. This is easier when either source or target of the mapping is not structured but consists only of a single class.

Boundaries Deep and shallow analyses may differ in what is conceived as part of a recognized item such as a token, a phrase, a named entity or a larger structure. This is in most cases rooted in the underlying models, training corpora, grammars or linguistic theories.

A proper name in a prepositional phrase may be recognized as proper name as a whole in a shallow analyzer, or without the preposition in a deep grammar, e.g. in $[\text{Paris}]_{\text{NE\_location}}$ vs. $[\text{in Paris}]_{\text{NE\_location}}$

Another example is tokenization. While a deep grammar may contain the word 'sister-in-law' as a single token in the lexicon, it may consist of 5 tokens when analyzed by a shallow tokenizer, e.g.

`<w>sister</w><t>-</t><w>in</w><t>-</t><w>law</w>.`

The general problem here is mapping of different spans of recognized linguistic items. Because the span information or verbatim input text is often the only common information that two independent natural language processing components share, the correct span is crucial for deep-shallow integration.

The boundary problem is closely related to structural mapping (richness) and hence the solution is similar to that: Mappings must be based on underlying linguistic resources such as grammars, models, theories. In case of not exactly matching spans, structural information of the recognized entities can be used to extract and map parts of analyses.

The common measuring unit for span information exchange between NLP components should be character positions (or counts) in the original input text as the least common denominator. Other units such as tokenization may not be compatible between components (cf. example above) and can therefore not form a reliable basis for syncing boundaries of higher-level linguistic items.

A further solution is to add the shallow tokenization as alternative input to deep parsing. This may increase ambiguity slightly, but also moves resolution to a single and probably best informed place, namely to the deep parser.

[3] An exception to the former case is the trigram-based chunker Chunkie (Skut and Brants, 1998) that computes recursively embedded chunks, provided that the underlying corpus used for training provides such information. However, a deep grammar may still support chunks of any depth.

Ambiguity Ambiguity may be introduced at any level of linguistic processing, e.g. tokenization, part-of-speech tagging, morphological analysis, lexicon access, chunking, named entity recognition, shallow parsing may all introduce ambiguity as well as deep grammars (because of lexical, syntactic and semantic ambiguity that is inherent in deep grammars).

If several shallow components are combined, then the number of ambiguous analyses may multiply at each integration step. However, they will collapse in many cases by coincidence of boundaries *etc*. On the other hand, it is also possible to use shallow preprocessing to filter unlikely readings by exploiting information a specialized component such as a statistical chunker can deliver. Some examples for concrete ambiguities

- *tokenization*: a dot may form the delimiter of a sentence or indicate an abbreviation or both at the same time
- *pos-tagging*: The same word can be a verb as well as a noun
- *morphology*: number, gender, case ambiguities
- *lexicon*: polysemy or homographs
- *named entities*: Paris may be the given name of a person or the name of a town

Ambiguity in deep-shallow integration is both a blessing and a curse. On the one side, shallow components can recognize readings for a word or syntactic structure that a deep grammar and lexicon does not provide because of resource limitations, and hence increase theoretically the chance that a sentence can be analyzed deeply.

On the other side, the fact that each shallow component may introduce additional readings, may increase the number of items on the deep parser's chart and hence the search space and run time of the deep analysis. The solution for this dilemma is to make decisions as early as possible in the integration process whether to include an additional reading induced by shallow preprocessing or to drop it. Heuristics on the reliability of outputs may help in the decision process.

A further aspect is reduction of ambiguities in deep parsing through the help of shallow analyses. Those readings that come closest to shallow analyses could be preferred over the others (cf. Chapter 8).

Finally, a backtracking strategy could be pursued that tries a full deep parse with one of the readings induces by shallow components, and only if this fails tries the next shallow reading and so on.

Probability Both shallow and deep processing systems may come with underlying probabilistic models that can be used to filter out readings, e.g. below a probability threshold. However, the problem remains how to set probabilities against

each other and which thresholds to choose. This is closely related to the ambiguity problem and probabilities may help to solve it.

We consider an example in part-of-speech tagging. Three words in the sentence analyzed by the TnT tagger get multiple readings (do, snails, live). The probability gained from the trained model is indicated in the `prio` attribute value.

```
<w id="TNT0" cstart="0" cend="4">
  <surface>Where</surface>
  <pos tag="WRB" prio="1.000000e+00"/>
</w>
<w id="TNT1" cstart="6" cend="7">
  <surface>do</surface>
  <pos tag="VBP" prio="8.754433e-01"/>
  <pos tag="VB" prio="1.245567e-01"/>
</w>
<w id="TNT2" cstart="9" cend="13">
  <surface>apple</surface>
  <pos tag="NN" prio="1.000000e+00"/>
</w>
<w id="TNT3" cstart="15" cend="20">
  <surface>snails</surface>
  <pos tag="NNS" prio="9.446836e-01"/>
  <pos tag="VBZ" prio="5.531639e-02"/>
</w>
<w id="TNT4" cstart="22" cend="25">
  <surface>live</surface>
  <pos tag="VBP" prio="7.822439e-01"/>
  <pos tag="JJ" prio="1.550179e-01"/>
  <pos tag="VB" prio="6.273819e-02"/>
</w>
<w id="TNT5" cstart="26" cend="26">
  <surface>?</surface>
  <pos tag="?" prio="1.0"/>
</w>
```

The problem with probabilities is that there is no general way of comparing or even propagating the values from one component to another.

A simple strategy that avoids explosion of search space caused by multiple readings is to take only the n most probable readings at each analysis step above a threshold y and to omit the others, n and y being configurable values. In the example above, $n = 1, y = 0.5$ would safely return the correct readings for all three ambiguously tagged words.

Conflicts, contradictory information When different NLP components based on different, independent linguistic resources (grammars, models, theories) are combined, conflicts are likely to occur.

Example: *'Essen ist gesund.'*

A named entity recognizer may recognize Essen as the name of a German city, while the deep grammar may recognize it as a verb. In this example, both readings could be valid. Similarly, conflicts between deep and shallow components may arise in tokenization, part-of-speech tagging, lexicon, chunking *etc*.

In general, conflicts may e.g. occur if a shallow component recognizes a word or constituent different from a deep parser. A conflict resolution might be necessary that could be based on reliability assessment. If a shallow component cautiously recognizes named entities, e.g., its voting would be preferred over a deep parser's lexicon-based alternative reading. The reliability of deep grammars could be judged higher e.g. for chunk or macro-sentential structures than for shallow parsers.

Alternatively, if the reliability of the NLP components is unclear, the contradicting reading from the shallow grammar could be introduced as additional item on the deep parser's chart, and the parser would either solve the conflict by knowledge encoded in the grammar, or by propagating the additional reading to the overall parse result.

Errors, correctness Both rule-based and statistics-based linguistic processing components make errors. The errors may be caused by errors in the resources, or by statistical models that do not fit the actual input text. Strategies are necessary that try to detect and possibly correct errors in order to avoid propagation of errors throughout a shallow-deep integration that could make the overall results worse.

In the following example, the word 'orange' is wrongly tagged by the TnT tagger as noun because of deficits in its trigram model. By applying correction rules that go beyond the underlying trigram model of the tagger (taking into account macro sentence structure), the reading with lower probability (JJ for adjective, 14%) could be made the correct analysis.

```
<w id="TNT0" cstart="0" cend="2">
  <surface>Why</surface>
  <pos tag="WRB" prio="1.000000e+00"/>
</w>
<w id="TNT1" cstart="4" cend="7">
  <surface>does</surface>
  <pos tag="VBZ" prio="1.000000e+00"/>
</w>
<w id="TNT2" cstart="9" cend="11">
  <surface>the</surface>
  <pos tag="DT" prio="1.000000e+00"/>
</w>
<w id="TNT3" cstart="13" cend="16">
  <surface>moon</surface>
  <pos tag="NN" prio="1.000000e+00"/>
</w>
<w id="TNT4" cstart="18" cend="21">
  <surface>turn</surface>
```

```
    <pos tag="NN" prio="7.511812e-01"/>
    <pos tag="VB" prio="1.887845e-01"/>
    <pos tag="VBP" prio="6.003432e-02"/>
  </w>
  <w id="TNT5" cstart="23" cend="28">
    <surface>orange</surface>
    <pos tag="NN" prio="8.592189e-01"/>
    <pos tag="JJ" prio="1.407811e-01"/>
  </w>
  <w id="TNT6" cstart="29" cend="29">
    <surface>?</surface>
    <pos tag="?" prio="1.0"/>
  </w>
```

The general solution is similar to conflict resolution (cf. above), but can be harder because conflicts can be detected immediately, while errors without conflicts may remain undiscovered until deep parsing is performed. However, a conflict case is of course always a good candidate for an error. In that case, the output of the more reliable component should be taken. If no conflict is signaled, the input e.g. of a shallow component may still be wrong. In order to increase correctness, it would be possible to run e.g. two different part-of-speech taggers concurrently, and compare the results word by word. A conflict could signal a potential source of error, and a voting mechanism could be applied to decide for one or the other result.

5.1.1 Summary

We have discussed, on a rather abstract level, problems that may occur when shallow and deep NLP components are combined. As a general strategy, it would be advantageous to only map reliable, useful information and try to reduce ambiguity. Furthermore, facilities for translating structure and namings as well as a configurable and flexible should be provided to ease integration. We will discuss concrete solutions in the following chapters. In Chapter 9, we will show how integration on a semantics representation level (in addition to a sequential shallow-deep architecture) can partially circumvent some of the described problems in application-oriented contexts.

5.2 Integration of Linguistic Representations by Transformation

From the similarity of markup produced by NLP components and annotated corpora that we observed in Chapter 4, a close relationship follows between access to NLP component output (online) and querying annotated corpora (offline).

In this chapter, we examine and compare XML query languages and show how NLP annotation markup can be accessed using standard XML query languages.

We propose XSLT as a simple, but powerful, efficient and standardized transformation language for online integration of NLP markup and motivate the choice.

While we will postpone discussion and examples of shallow NLP markup XSL transformation to later chapters where we will present implemented architectures, we will concentrate in the end of this chapter on the less obvious, but also important task of transforming XML-encoded typed feature structures using XSLT.

5.2.1 Querying Multi-level (Standoff) Annotation

The term *multi-level annotation* has been established as a generic term for nested and standoff annotation of various kinds (cf. also footnote remark in Section 4.3.1) comprising those mentioned as examples in the previous chapter.

For NLP component integration on the basis of multi-level annotation and XML annotation in general, it is important to see how the annotation can be queried, i.e., how pieces of annotation such as part-of-speech tags of words can be accessed, how structural information such as what are the constituents of a phrase can be obtained, and how the information can be combined.

Thus, a query language providing (a) data access, (b) compositionality of queries, (c) transformation capabilities and (d) integration facilities is being sought. Similar requirements have been identified by Taylor (2003) for querying linguistic corpora. It is obvious that properties and requirements for corpus query languages also hold for markup integration and access produced online by NLP components.

In the following, we will briefly discuss query languages and tools for multi-level (or general) linguistic XML annotation.

5.2.1.1 XPath

XPath 1.0 (Clark and DeRose, 1999) is a simple, but powerful language that describes patterns of the XML tree structure in form of (element and attribute) path expressions. XPath is not specifically designed for linguistic queries and therefore lacks e.g. the ability to query items that exceed the XML tree schema such as overlapping hierarchies. Moreover, advanced constructions such as query variables, quantification or negation of arbitrary path components are not supported. XPath patterns may denote node sets in the XML document tree, or strings or numbers. Examples for XPath expressions are shown in Table 5.1.

XPath provides a number of predefined basic functions for simple arithmetics, Boolean, string operations and document tree navigation (siblings, ancestors, parents, children, descendants, cf. Figure 5.1), counting nodes, converting data types *etc*. XPath lacks a full document integration and transformation facility, it can only map XML documents to node sets or to elementary data types such as strings, Booleans or numbers. However, XPath is part of XSLT and XQuery (discussed later) which provide integration and transformation facilities.

XPath	description
`chunk`	matches a chunk element
`*`	matches any element
`chunk|w`	matches chunk or w elements
`chunk/w`	matches w element with chunk parent
`last()`	matches last daughter element of current node
`w[3]`	matches the 3rd w child of the current node
`/`	matches the root element of the document
`//w`	matches all w elements in the document
`w[position() mod 2 = 1]`	matches odd-numbered w daughters
`w[last() = 1]`	matches a w daughter that is the only daughter
`id('W34')`	matches element with unique id W34
`w[@cstart='23']`	matches w element with attribute cstart='23'
`@*`	matches any attribute daughter
`w[3]/@cstart`	returns cstart attribute value of 3rd w daughter
`count(w)`	returns number of w daughters of current node
`count(//w)`	returns number of all w elements in document
`substring('hi',2)`	returns 'i'
`contains('hello','hell')`	returns true
`name()`	returns name of current element

Table 5.1: XPath expressions (examples)

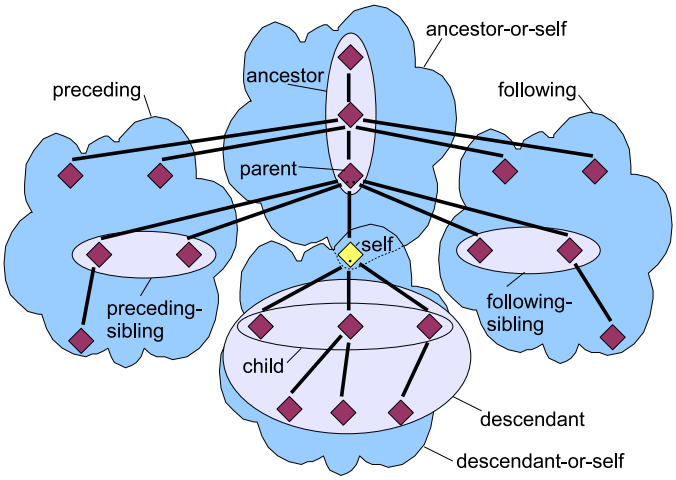


Figure 5.1: XPath axes in the XML document tree

5.2.1.2 XPath 2.0 and XQuery

XPath 2.0 extends XPath 1.0 in many ways making it a more powerful language, e.g. by adding regular expressions, extended string functionality and XML schema

support.

XQuery (Boag *et al.*, 2006), building on XPath 2.0, is an XML query language unifying the two different strands of XML usage mentioned in Section 4.2.4, namely XML-as-document (reflecting XML's original roots in SGML) and XML-as-data (corresponding to relation database data models).

XQuery adopts from SQL the FLWOR query concept (FLWOR for the for-let-where-order-return query structure, loosely analogous to SQL's SELECT-FROM-WHERE) and combines it with XPath 2.0 to a powerful query language. We show an example of a query that generates a list consisting of book title, price and number of authors per book.

```
<books>
  { for $x in doc("bookstore.xml")/bookstore/book
    let $a := $x/author
    where $x/price>30
    order by $x/title
    return <book> {$x/title} {$x/price}
              <authors>{count($a)}</authors>
           </book> }
</books>
```

The overall query structure of XQuery looks similar to SQL, but the underlying data model is quite different. While the basic data structure in relational databases are flat tables, data in XML documents is hierarchically structured. Similarly, the query and result data structures are hierarchical. Moreover, the sequential order in XML documents must be preserved from the input to the output structures while the order is generally undefined in relational databases.

Slightly simplifying, the XQuery expression arguments of the for, let, where, order-by and return statements consist of XPath 2.0 expressions which makes the language modular and powerful. Data type definitions can by imported from XML Schema definitions via references to external resources. A wealth of XQuery use cases is documented in a W3C working draft (Chamberlin *et al.*, 2006).

5.2.1.3 Related Work

We briefly discuss related work in XML query languages for linguistic markup. Cassidy (2002) discusses XQuery (an earlier version of it) in a use case analysis as linguistic annotation query language and concludes that XQuery has weaknesses in expressing sequential constraints while it is quite powerful for querying with hierarchical constraints.

Both XPath and XQuery are general languages for XML tree access and not specifically designed for operating on linguistic markup with annotation graphs and time alignment of linguistic annotation. For that reason, various XPath extensions have been proposed during the last years that attack that specific feature.

While Bouma and Kloosterman (2002) argue that XPath is powerful enough for querying dependency treebanks encoded in XML (provided that they were converted into an appropriate format), Carletta *et al.* (2003) and Heid *et al.* (2004) propose the NXT Search query language that extends XPath by adding query variables, regular expressions, quantification and special support for querying temporal and structural relations.

Their main argument against standard XPath is that it is impossible to constrain both structural and temporal relations within a single XPath query and that standard XML query mechanisms would not be intuitive to linguists. The following is a 'cross-level' NXT Search example from Heid *et al.* (2004), in this case for querying frame and syntactic annotations, using variables ($), existential quantification (exists), structural dominance (^) for querying.

```
($f1 frame)($f2 frame)
(exists $phrase syntax)
(exists $target word):
$f1 >"target" $target and
$f2 >"target" $target and
$f1 ^ $phrase and
$f2 ^ $phrase and
$f1 != $f2
```

Another recent corpus query language is LPath (Bird *et al.*, 2005). LPath lays a special focus on subtree scoping, immediate precedence and edge alignment with intuitive XPath syntax extensions. Moreover, translation to SQL queries is performed for efficient access to annotation stored in a relational database.

Beyond XPath extensions, many other query languages for (not only) XML-based multi-level annotations have been developed. Bird *et al.* (2000) discuss an annotation graph query language. (Simplified) MMAXQL (Müller, 2005) is a query language that similarly to NXT Search supports query variables, regular expressions and relation operators for structural queries, but lacks quantification.

Teich *et al.* (2001), Taylor (2003) and Lai and Bird (2004) present overviews and a comparison of different query languages for annotated corpora.

5.2.2 Using Corpus Query Languages for NLP Component Integration?

As shown in the previous section, besides XPath that is not powerful enough and XQuery which is not yet a stable standard, there are many proposals for query languages for linguistically annotated corpora. However, the presented approaches specifically designed for linguistic corpus access show some disadvantages that make them inappropriate for NLP component integration.

Lack of portability The proposed annotation languages, e.g. those forming an extension to XPath, have been implemented for a specific XPath engine and can-

not be used without adaptation to e.g. XPath implementations in other programming languages, even porting to another implementation in the same programming language requires additional implementation work. The LPath implementation in addition requires and relies on annotation stored in a relational database.

Lack of efficiency XPath extensions such as NXT Search are reported to have severe performance problems (which is clearly related to the expressive power of the query language), especially when a query combines several input documents. Existing optimizations e.g. of the standard XPath document tree accessing methods are highly probable to be incompatible with these XPath extensions and therefore cannot be applied. In a recent experiment, Mayo *et al.* (2006) report on a re-implementation of the NXT query language in XQuery (using the Saxon[4] implementation) that showed drastic speedups for some query types, but required a preprocessing technique called 'knitting'.

Limited re-usability Each of the discussed query languages comes with its own syntax that is incompatible with the others. High learning costs are inevitable, and the inflation of new query language reduces the usability and usefulness of corpora in general. Partly, the query syntax has been designed for specific corpora or corpora types, and may not be applicable for others. Because of this, and as a standard seems not to show up in the near future, queries written for one query language cannot be reused.

Restricted extensibility Although some of the most important shortcomings of standard XPath and similar query languages with respect to linguistic annotation processing are addressed, the proposed extensions solve only a fixed set of problems. Further extensions such as combining input from multiple annotations or computing values or annotation must be solved outside the query language.

Given these common restrictions of corpus query languages (and their implementations), there seems to remain serious doubt whether a corpus query language can be appropriate for online NLP component integration.

5.3 Markup Transformation and Query with XSLT

As an alternative to the highly specific but hardly extensible corpus query languages with their above described disadvantages, we propose XSLT (eXtensible stylesheet transformation language, Clark 1999b) for NLP component integration. Already McKelvie *et al.* (1998); Ide (2000); Ide and Romary (2001); Carletta *et al.* (2002) have discussed and proposed XSLT as corpus query language and general purpose transformation language for linguistic annotation. For our application in

[4] `http://saxon.sourceforge.net`

deep-shallow NLP component integration, *online* aspect and *integratability* of the query language play a central role, and as we will see, XSLT fulfills these requirements well.

XSLT, as part of the XSL language family, builds on XPath and is a specialized programming language for transformation of XML documents into another XML format, HTML, text, PDF *etc*.

Probably because of the name ('stylesheet'), XSLT has often been misunderstood or underestimated as a web (HTML) style definition language such as CSS[5], but this is only one aspect. XML tree transformation is a further application of XSLT that has also been proposed and used in other contexts, including electronic business and even general (XML-based) software architecture frameworks such as the one proposed by Löwe and Noga (2002) which will be discussed further in Section 9.13.

The XML to XML transformation we focus on can be seen as a query that extracts information from XML input (annotated corpus or NLP component output) and represents it in a different way. As XSLT extends (or uses) XPath, and is at the same time a Turing-machine equivalent programming language[6], we believe that it is an appropriate and powerful alternative to the specialized corpus query languages and well-suited for NLP component integration.

Taylor (2003) discusses several query languages for linguistic (corpus) annotation and shows, by giving translation schemata and examples, that queries of EMU (Cassidy and Harrington, 1996) and Tgrep2 (Rohde, 2005), both query languages that support specific data models for linguistic annotation, can be translated into XSLT queries.

The advantages we see in XSLT are:

Portability In contrast to corpus query languages, where often only a single implementation exists, XSLT 1.0 processors have been implemented for almost all current programming and scripting languages. This means that the queries can easily be exchanged across different platforms or implementations, and XSLT can form an independent basis for annotation interchange. The broad and quite mature implementation status of XSLT processors in various programming languages and platforms makes it possible to easily implement also online integration of NLP components through XSLT.

Efficiency Most XSLT implementations provide optimizations for XML tree and XPath processing and the XML ID/IDREF mechanism. In XSLT, an additional key mechanism exists for constructing unique keys as in relational database systems.

[5] Cascading Style Sheets; `http://w3c.org/Style/CSS/`

[6] Cf. `http://www.unidex.com/turing/utm.htm`, where a XSLT 1.0 stylesheet is defined that encodes a universal Turing machine and hence proves by construction that XSLT 1.0 is Turing-complete.

This makes XSLT access to XML tree annotation typically faster than straightforward DOM tree search and navigation as it is the case in most XPath extensions.

Moreover, in recent implementations such as Apache Xalan-J, an XSLT processor implemented in Java, a stylesheet compilation facility called XSLTC can be enabled which compiles XSLT stylesheets into *Translets* that make transformation significantly faster (3-6 times on average, up to 1000 times in specific queries, further performance boosts can be expected in later implementations), and have a much smaller memory footprint than interpreting XSLT processors.

Recent overviews over XSLT benchmarks and comparisons of XSLT processors are presented in Bittner (2004) and Früh *et al.* (2004). Although the benchmark cases are quite simple and limited in the stylesheet complexity compared to realistic natural language corpus queries, it can be seen that there is still room left for optimizations, especially in the Java XSLT implementations.

Re-usability Because a well-established, standardized language is used, pre-existing queries (stylesheets) can be re-used for new corpora and NLP component formats. This can help to reduce the overall amount of coding work for annotation access. Re-usability is also possible across different corpus formats because of the similarity of their document structures.

In some cases, only the names of attributes or elements in XPath expressions need to be replaced. Libraries for typical query types can be built using named templates[7] where the structure and names of elements *etc* could be made customizable parameters.

Extensibility In contrast to the discussed XPath extensions for linguistic queries, XSLT queries (stylesheets) can be extended by user-defined subroutines or functions (Turing machine equivalent; see above). The XSLT template mechanism (named and matching templates) can be used to define operations on node sets, strings or numbers.

The templates can take parameters with default values and can return element nodes or other elementary XSLT/XPath data types. Queries can be formulated through templates that may become necessary for new kinds of annotation, and that are not expressible in the existing linguistic XPath extension frameworks.

Openness XSLT is open both on the input and on the output side.

Openness with respect to input: Although XML is the obligatory input format for XSLT, XSLT is not bound to a specific DTD or schema, a stylesheet can be applied to different document formats that can only be described by different DTDs. An example is a stylesheet that only has relative matching paths such that any DTD containing the matching elements would be appropriate, and even *vice versa*, a stylesheet may contain matches to elements that never occur in one input

[7] XSLT 2.0 additionally has user-definable functions that can also be included by reference.

document type, but which may be contained in other DTDs. This independence of a DTD may ease XML processing and transformation.

Openness with respect to output: XSLT can by design be used to output other formats than XML. However, for transformation and query of linguistic annotation, XML should be the preferred output format. Ide (2000) propose XSLT also for the visualization, formatting and presentation of linguistic annotation.

Formatting XML has been (and probably still is in the majority of applications) clearly one of the initial main motivations for inventing XSLT. Throughout the thesis, we will (in the implementations to be presented) also make use of XSLT stylesheets for visualization of XML output generated from linguistic components, e.g. for trees, typed feature structures, robust semantics representations, cf. Sections 7.5, 8.7.8 and 9.8.

5.3.1 Brief Introduction to XSLT

XSLT has been introduced as part of the W3C eXtensible Stylesheet (XSL) family[8] that forms a standard (called recommendations in the W3C jargon) for defining XML document transformation and presentation. The XSL family comprises three parts

- XSL Transformations (XSLT), a language for transforming XML
- the XML Path Language (XPath) an expression language used by XSLT to access or refer to parts of an XML document. XPath is also used by the XML Linking specification XPointer and as part of XQuery.
- XSL Formatting Objects (XSL-FO), an XML vocabulary for specifying formatting semantics

While we already have briefly introduced XPath, we will not further discuss XSL-FO. The main application of XSL-FO is layout generation from XML sources beyond HTML, and currently mostly the PDF language is used to this end in (web-based) publishing applications.

Historically, XSLT is based on DSSSL (Document Style Semantics and Specification Language), a transformation language for SGML, defined as ISO 10179 standard (ISO/IEC, 1996). While DSSSL is a LISP dialect and as such based on S-expression syntax, XSLT uses XML syntax, but a similar concepts, constructs and processing model as DSSSL.

Similarly to DSSSL, XSLT uses a declarative processing model following the tree structure of the XML input document and very much relies on formulation of algorithms through the use of recursion as in other functional programming languages. The focus of DSSSL has been SGML to SGML transformation. As already mentioned, XSLT is more open.

[8] `http://w3c.org/Style/XSL/`

While the input of an XSL transformation must be a valid XML document, the output may be XML, HTML, or any other format (including non-text). The reason for this is that XSLT has been from the beginning designed as a multi-purpose transformation language, with a strong focus on formatted output with HTML (or XSL-FO alternatively).

The XSLT processing model (cf. Figure 5.2) consists of an XSLT processor (or transformer) that takes a so called XSLT stylesheet containing the transformation rules and an XML input document plus optional named parameters (e.g. strings or numbers) for the stylesheet as input. The transformation process produces an output document according to what has been defined in the stylesheet. In XSLT 2.0, a stylesheet can produce multiple output documents within a single transformation.

There is an alternative way of calling a stylesheet-based transformation on an XML document by associating the stylesheet in the XML source document (Clark, 1999a). This mechanism depends on the capabilities of the XML parser, browser or transformer that opens the XML document, but many current browsers and XSLT processors support this now. The effect would be the same as running the transformer on the stylesheet and XML input document.

According to the stylesheet and parameters, different output formats can be generated. However, XSLT itself does not (cannot) verify the correctness of the output syntax, except for some basic XML and HTML structure if these are the specified output formats.

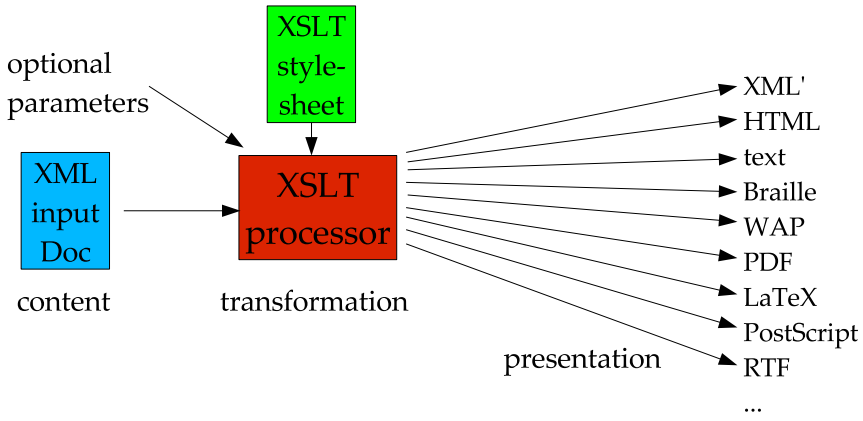


Figure 5.2: XSLT processing model

Stylesheet concept An XSLT stylesheet is an XML document with root element `xsl:stylesheet` (or the synonym `xsl:transform`). 'xsl:' indicates the namespace which must be declared as `http://www.w3.org/1999/XSL/Transform` for XSLT 1.0. The stylesheet contains the transformation instructions in elements and attributes defined in the XSLT recommendation, i.e., the transformation programming language is encoded in XML syntax.

Optional sub-elements of `xsl:stylesheet` e.g. declare global parameters (that can be passed to the stylesheets), the output format *etc*. Other XML elements not defined in the XSL namespace are returned as part of the resulting output document. Numerous stylesheet examples can be found in the XSLT Appendix starting from page 295.

Processing model and matching templates An XSLT processor first parses the XML input document into an internal XML tree representation which is then recursively traversed. During traversal, *matching templates* which are subroutines are applied when they match via a specified XPath pattern the XML tree fragments. The XSLT syntax element for matching templates is `xsl:template match="`*match*`"`.

The recursive traversal can be canceled, repeated, selected (via specific subelements) in the `xsl:apply-templates` instruction or sorted from within matching templates and loops. This mechanism provides a powerful and (at least in the default case of recursive tree traversal) a quasi-declarative specification of XML tree transformation through XPath patterns.

An optional mode attribute can be used to provide alternative template code depending on the context, e.g. to define rules for multiple output formats within the same stylesheet. When a node in the XML input tree matches more than one template rule, a simple conflict resolution strategy applies based on a static analysis of the match expressions. Alternatively, a priority value can be assigned to a template in the stylesheet.

Named templates Named templates are subroutines or functions that can be called via their name (independently of a match with the XML input tree; syntax: `xsl:template name="`*name*`"`.), such as subroutines or functions in other programming languages. Because of the static behavior of variables in XSLT (cf. below), recursion through named templates with locally scoped variables plays an important role.

Data model and basic data types XSLT shares with XPath the data model, i.e., basic data types are node set, string, Boolean, number with the same automatic conversion rules between the data types.

XPath expressions as first-class citizens XPath is not only used to match XML tree fragments in matching templates, but also for computation of expressions e.g. for string output in the result document or for testing Boolean expressions in conditional instructions.

Built-in functions Built-in functions can be used in XPath expressions for computations on numbers, strings, Booleans and node sets. These functions comprise

the typical, simple operations on the basic data types known from other programming languages, such as addition, string concatenation, *etc*.

The document() function deserves a special mention. It can be used to include additional external XML input documents (and via XPath portions of them) in addition to the standard input document that is always passed to the XSL processor. While there is only a fixed set of predefined functions available in XSLT 1.0, functions can be defined similar to named templates in XSLT 2.0, and employed in XPath expressions.

Variables and parameters Variables in XSLT aren't variable. They behave like variables in mathematics that are assigned at most once rather than as names for memory cells as in most programming languages where they can change their values. However, variables have local scopes in templates, and recursion can be used for counting *etc*. Both matching and named templates may take named parameters as in other programming languages. Moreover, the stylesheet itself can take named parameters from outside (e.g. the transformer call).

Conditional control structures and loops A single conditional instruction can be expressed with `xsl:if` and a test based on a Boolean XPath expression, more complex case selections with `xsl:choose`, `xsl:when`, `xsl:otherwise`.

`xsl:for-each` in conjunction with an XPath node selection expression can be used to define loops over tree elements.

Modularity through inclusion of external stylesheets Both `xsl:include` and `xsl:import` support inclusion of external stylesheets where the definitions in the importing stylesheets take precedence over code imported with `xsl:import`.

Output: text, elements, attributes, *etc* Elements, attributes and text can either be output by writing them directly in the template code, or by the explicit XSLT instructions `xsl:element`, `xsl:attribute`, `xsl:text` and `xsl:comment`, with XPath expressions that can be used to compute or compose values e.g. through string functions. Nodes from the input can be copied with `xsl:copy` and `xsl:copy-of` with an XPath expression selecting the node(s), where `xsl:copy-of` performs deep copies.

To summarize briefly, XSLT is a small but powerful, specialized programming language for XML tree transformation and query. It provides powerful, quasi-declarative tree navigation and selection through XPath expression matches. Compared to DOM navigation in classical programming languages such as Java, C, Python *etc*, XSLT is a much more concise and better maintainable alternative when flexible adaptation to new or modified XML formats is required.

5.3.2 XQuery vs. XSLT

The only serious, powerful and viable alternative to XSLT we see is the upcoming XQuery standard (basing on XPath 2.0). It shares most of the advantages we discussed for XSLT. Until the XQuery standard is established (as a W3C standard) and efficient implementations are available, XSLT is a good alternative for which many different implementations, partly with efficiency optimizations, already exist.

What remains to be discussed is the relation between XQuery and XSLT. While both share many properties and are at large parts overlapping in their application areas, the focus and strength of XQuery seems to be the data-centric queries (regularly structured markup), while XSLT has its advantages in document-centric queries (semi-structured markup).

Thus, both languages have advantages and disadvantages depending on the type of task to perform. The same holds for efficiency which varies to a great extent also depending on the task. As in all programming languages, solutions for the same problem can be formulated in many ways. Here, the early implementation status of XQuery processors and also ongoing research in optimization and compilation techniques for XSLT would render any comparison unbalanced.

The current focus in research and development of both XSLT and XQuery processing is in fact compilation and optimization. Major improvements are to be expected in both languages.

Lenz (2003) discusses the differences and commonalities of XQuery and XSLT (the former at an earlier stage, not in the current state with XPath 2.0), and illustrate how XQuery queries can be encoded in XSLT, supporting his thesis that most things that can be done in XQuery are also feasible in XSLT. He shows, e.g., the correspondence to FLWOR expressions in XSLT, namely

XQuery	XSLT
`FOR`	`xsl:for-each` (or `xsl:template` calls)
`LET`	`xsl:variable`; often not necessary because of context node switch in the `xsl:for-each` body
`WHERE`	`xsl:if`
`ORDER-BY`	`xsl:sort`
`RETURN`	`xsl:copy(-of)`, `xsl:value-of`, `xsl:element`, *etc.*

This is of course simplifying because extensions and variations exist in both XQuery and XSLT, but the table roughly illustrates how XQuery queries could be translated to XSLT.

Graaumans (2005) thoroughly compares the usability of XQuery, XSLT and SQL/XML, an extension of the SQL ISO/ANSI standard to XML data stored in a relational database. Although the author does not focus on NLP markup, the study presents interesting insights into the performance and usability of XSLT vs. XQuery in different query tasks, both for data-oriented and document-oriented markup.

To conclude, both XSLT and XQuery have a large common application area, and strengths exist in both frameworks. Both are being continued and extended by the W3C (e.g. by a common XPath 2.0 subset and XML Schema support) and both have their own justification to exist. However, because of the more general transformation approach (e.g. to fundamentally re-organize XML document structure) through templates and the more stable and settled standardization and implementation status, we opt for XSLT in the rest of the thesis, keeping in mind that in most cases, one could replace an XSLT processor by an XQuery processor if necessary.

5.3.3 NLP Integration and Computation with XSLT

XSLT has been proposed in several natural-language processing contexts such as natural language generation or text-to-speech.

Wilcock (2001) discusses XSLT for natural language generation in pipeline architectures, template-based generation with XSLT templates, and tree-to-tree transformations.

Ide *et al.* (2000b) discuss how a general model of lexical information with inheritance encoded in XML can be realized in different formats using XSLT. Ide (2000) also proposes XSLT for formatting and visualizing linguistic information.

Foster and White (2004) present an XSLT-based approach for logical form generation for text generation. An XSLT processor is treated as a top-down rule expander structuring and aggregating the content and performing lexical choice.

Schröder and Breuer (2004) use XSLT to plug together different text-to-speech systems.

Coming back to the integration task of multiple NLP components on XML basis, XSLT can serve as 'glue' between the components with the advantages mentioned above with respect to portability, re-usability, extensibility and efficiency. XSLT also solves the problem of interlinking XML documents, e.g. different analyses of an input text produced by different NLP components with standoff annotation. This is important because no commonly adopted and implemented linking *standard* exists[9].

We propose XSLT for the integration of deep and shallow natural language components because (1) most shallow processors produce XML output natively or can be easily adapted to do so (2) XSLT can be used to combine standoff annotation produced by multiple (shallow) natural language components and to combine, translate and compute the information a deep parser requires from shallow preprocessing (3) XSLT can be used for post-processing e.g. of deep analysis results (e.g. semantics, syntactic tree structures, typed feature structures) or to repair processing results if deep processing fails.

Furthermore, because XSLT plays an important role in general XML-based software architectures (e.g. for web publishing), participation in the improvements

[9] XPointer and XLink are W3C recommendations, but only a few, and rarely used implementations exist, partly because of patent issues.

and optimizations of implementations and porting to new platforms and programming or scripting languages and the further development of the standards are considerable advantages over niche solutions that are specific for language (corpus) technology.

We will come back to XSLT by demonstrating applications of it in the later architecture chapters (Chapters 7–9).

5.4 Transforming XML-encoded Typed Feature Structures

5.4.1 Accessing and Transforming Feature Structure XML

In the sections so far, we have implicitly discussed query and access to shallow XML annotation because this is what most corpus query languages aim at. But annotation access could also include deep analysis results encoded in typed feature structures. Typed feature structures provide a powerful, universal representation framework for linguistic knowledge (cf. Chapter 3), not only in deep analysis, but also for shallow analysis results (we will describe a shallow system utilizing typed feature structures in Chapter 7).

While it is in general inefficient to use XML to represent typed feature structures during processing (e.g. for unification, subsumption operations in HPSG parsing), there are several applications that may benefit from a standardized system-independent typed feature structure XML, e.g. as exchange format for

- deep NLP component results (e.g. parser chart or parts thereof, but also shallow analysis results),
- grammar sources, XML format as ‘abstract syntax’ for increased portability between different formalisms or implementations,
- feature structure renderers or editors such as in *SProUT* (cf. Chapter 7) or Thistle (Calder, 2000),
- feature structure ’tree banks’ of analyzed corpora

We adopt an embedding XML representation for typed feature structures originally developed by the Text Encoding Initiative (TEI). It is compact and widely accepted (Sperberg-McQueen and Burnard, 1994). The markup is also part of a proposal for an ISO standard on feature structure representation in XML ISO TC37 SC-4 (Lee *et al.*, 2004) and also of MAF, the proposed morph-osyntactic annotation format (Clément and Villemonte de la Clergerie, 2005).

An in-depth justification for the naming and structure of the TEI feature structure DTD is presented in Langendoen and Simons (1995). We focus here on the feature structure DTD subset that is able to encode the basic data structures of deep systems such as LKB (Copestake, 2002), PET (Callmeier, 2000), PAGE (Uszkoreit

et al., 1994), or the shallow system *SProUT* (Drożdżyński *et al.* 2004; cf. Chapter 7) which use a subset of TDL (Krieger and Schäfer, 1994) as their common basic formalism[10].

The TFS DTD in the DTD Appendix (page 286) is structured as follows. The `FS` tag encodes typed feature structure nodes, `F` encodes features. Atoms are encoded as typed feature structure nodes with empty feature list. The `coref` attribute encodes coreferences (reentrancies; structure sharing) between feature structure nodes. An illustrative example is shown in Figure 5.3.

```
<FS type="synsem">
  <F name="FEAT1">
     <FS type="t1" coref="1">
        <F name="FEAT3">
           <FS type="*top*"/>
        </F>
     </FS>
  </F>
  <F name="FEAT2">
     <FS coref="1"/>
  </F>
</FS>
```

$$\begin{bmatrix} \textit{synsem} & \\ \text{FEAT1}\ \boxed{1} & \begin{bmatrix} \textit{t1} & \\ \text{FEAT3} & \textit{*top*} \end{bmatrix} \\ \text{FEAT2}\ \boxed{1} & \end{bmatrix}$$

Figure 5.3: XML-encoded typed feature structure (left) with structure sharing between attributes `FEAT1` and `FEAT2` (through `coref="1"`) and the corresponding AVM notation (right)

5.4.2 The Role of Feature Structure XML Transformation for the Integration of NLP Components

One of the main motivations for XML feature structure markup is the interchange of linguistic data. This can be done *offline*, e.g. for the exchange of lexica, grammatical resources, or annotated documents.

A further application is *online* integration of NLP components, where several, specialized modules contribute to improved (e.g. disambiguated or more precise) linguistic analyses.

In both cases, online or offline integration, different representations of linguistic data can be involved, where feature structures can either form the source or the target representation or even both.

To illustrate the use of XML transformation of feature structure markup, we present concrete, simple examples.

[10] This is only the common, minimal basis of the different formalisms, each formalism has its own extensions such as sets, disjunctions, distributed disjunctions, and differing interpretations of type semantics, e.g. open world vs. closed world.

5.4.2.1 Feature Structure XML as Target Representation

Construction of typed feature structures from other XML representations that are e.g. produced by a shallow NLP system. Specific elements with attributes are translated to possibly nested feature-value pairs, e.g. for input to an HPSG parser *etc*. In the following example, `<infl num="singular"/>` is translated to the corresponding feature structure, with value `singular` inserted for the XPath expression `{@num}`. Of course, also symbolic names, e.g. `sg` to `singular` *etc* could be translated.

```
<xsl:template match="infl">
   <FS type="infl">
       <F name="NUMBER">
          <FS type="{@num}"/>
       </F>
   </FS>
</xsl:template>
```

Feature structure XML as grammar exchange format or meta syntax. An example is XTDL in *SProUT* (cf. Chapter 7), where a TDL-based grammar syntax (Krieger and Schäfer, 1994) is translated to an internal representation based on feature structure XML. The internal XML representation (*XTDL* DTD on page 286) is used as input for type checking and finite-state compilation; details in Section 7.4.

Feature structure XML for data exchange between NLP components. As an example, a morphology component could encode generated linguistic information in typed feature structure XML for further use in other components, e.g. a parser.

5.4.2.2 Feature Structure XML as Source Representation

Extraction or projection of information encoded in typed feature structures such as morphology to other formats or as API accessors, e.g. an XPath expression such as

```
<xsl:template match="FS[@type='infl']">
  <infl num="F[@name='NUMBER']/FS[@type='num']/@type"/>
</xsl:template>
```

is the inverse of the template example above.

AVM visualization tools or editors such as the feature structure renderer in *SProUT* or Thistle (Calder, 2000) both take (different) descriptions of typed feature structures and render a graphical representation of the feature structure as attribute-value matrix (AVM). Examples are depicted in Figure 7.8, 8.13, 8.12 and 9.24.

Extraction of tree structures encoded in a complex HPSG feature structure (parse result), e.g. for further linguistic processing, treebanking *etc*.

Extraction and transformation of semantics representation. An example is a transformation of typed feature structures to RMRS XML (Copestake, 2003) which e.g. forms the basic representation for the exchange of deep and shallow NLP results in the Heart of Gold architecture (Callmeier *et al.*, 2004), cf. Figure 5.4. Details will be discussed in Chapter 9.

```
<MATCHINFO rule="en_city" cstart="3" cend="7">
  <FS type="sprout_rule">
    <F name="OUT">
      <FS type="ne-location">
        <F name="LOCNAME">
          <FS type="&quot;Paris&quot;"/>
        </F>
        <F name="LOCTYPE">
          <FS type="city"/>
        </F>
      </FS>
    </F>
  </FS>
</MATCHINFO>
```

```
-->
```

```
<rmrs cfrom="3" cto="7">
  <label vid="1"/>
  <ep cfrom="3" cto="7">
    <gpred>ne-location</gpred>
    <label vid="2"/>
    <var sort="x" vid="2"/>
  </ep>
  <rarg>
    <label vid="2"/>
    <rargname>CARG</rargname>
    <constant>"Paris"
    </constant>
  </rarg>
</rmrs>
```

Figure 5.4: Transformation of *SProUT* feature structure XML to RMRS

5.4.2.3 Feature Structure XML as Both Source and Target Representation

Translation between different feature structure syntaxes or systems. We exemplify lists that can be encoded differently in typed feature structure markup. The XSLT template below takes a list encoded as nested `FIRST-REST` list typed `*cons*` and translates it to the 'flat' XML `<list>` element with embedded elements from the `FIRST` attribute values in the input. The template works recursively on `FIRST-REST` lists of any length.

```
<!-- ====================================
Initial template. Enclose list elements from
FIRST-REST list in <list> element
==================================== -->
<xsl:template match='FS[@type="*cons*"]'>
 <xsl:element name="list">
  <xsl:call-template name="listlist">
   <xsl:with-param name="node" select="."/>
  </xsl:call-template>
 </xsl:element>
</xsl:template>
```

```
<!-- ====================================
recursive template: list all list elements
==================================== -->
<xsl:template name="listlist">
 <xsl:param name="node"/>
 <xsl:copy-of select='$node/F[@name="FIRST"]/FS'/>
 <xsl:if test='$node/F[@name="REST"]/FS/@type="*cons*"'>
  <xsl:call-template name="listlist">
   <xsl:with-param name="node"
        select='$node/F[@name="REST"]/FS'/>
  </xsl:call-template>
 </xsl:if>
</xsl:template>
```

Similarly, transformation can reorganize the structure of information encoded in typed feature structures, e.g. move values to a different feature path, or rename features and types *etc*. For a list of further applications of XML-based feature structure transformation cf. Section 5 in Lee *et al.* (2004).

5.4.2.4 Reentrancies and Transformation

A general issue that arises in the case where feature structures are source representations is reentrancies. Here, 'dereferencing' is necessary on the basis of lookup in the XML source in order to have access to every node in the DAG (e.g. for feature path access); XML ID/IDREF declarations support faster access as discussed already before. If cyclic reentrancies are disallowed, copying of shared values when generating the features structure representation is an easy and probably faster way in order to get the full access to shared values. Identity information is preserved through the reentrancy attribute (`coref` in the above examples) anyway.

5.5 Summary

Starting from a discussion of general problems that arise when deep and shallow natural language processing components are combined, such as different granularity, namings, structure, boundaries, ambiguity and conflicts, we have motivated the role of transformation and query for XML-based integration of linguistic annotation.

We have examined existing linguistic corpus query languages, but also general W3C-supported XML query and transformation languages such as XPath, XSLT and XQuery, and compared them. We have motivated our choice of using XSLT for online integration of NLP component output.

XSLT has advantages over similar frameworks in that various efficient implementations exist. XSLT code is portable, re-useable and partly even declarative. Moreover, the extensibility and openness of XSLT as well as the embedding in

lively standards that are further developed also in other contexts such as World Wide Web and Semantic Web, makes it a promising framework also for future use.

Finally, we have addressed the problem of transforming XML-encoded typed feature structures that may play a role for deep-shallow integration architectures, using XSLT.

In the remainder of the thesis, we will concentrate on describing various applications of XSLT-based NLP component integrations.

Chapter 6

Hybrid Architectures

This chapter serves the purpose of a lead-in into the core part of the thesis describing hybrid architectures. We first motivate why architectures are needed to perform the deep-shallow integration task. We then present an overview of related work and state of the art in general architectures for linguistic processing and XML-based architectures. We conclude with an outlook to the following three chapters in each of which the author's contributions to a deep-shallow integration architecture will be described.

6.1 Motivation and Requirements

In the preceding chapters, we have shown what the problems of deep processing are, how shallow processing can help to improve (mainly robustness of) deep processing, that shallow processing results can be naturally encoded in XML, and what the difficulties are when combining shallow and deep processing results. What is missing so far is an *infrastructure* or *architecture* that supports the combination of various (pre-existing) shallow and deep natural language processors in such a way that the benefits from the combination can be exploited in applications. We collect some properties that such architectures should possess.

Flexibility and configurability. First of all, deep-shallow combination in language processing and language technology is a new field, and any current application is still experimental. Thus, a necessary property of such architectures is *flexibility* with respect to how and which NLP components are combined, e.g. processing order (sequential, on demand, in parallel) and information (annotation) flow. Related to this topic is the requirement that *integration of new NLP components* should be easy, as should be *configuration of NLP components*.

Openness, XML standoff support. *Openness* to linguistic theories and representation formalisms is an important concern, because there is neither a commonly

accepted, universal theory of language nor formalism for NLP. However, integration of components obeying different paradigms should be possible and may be useful.

As a consequence, and as motivated in the previous chapters, *XML support* is a necessary prerequisite, both for *XML access* to and for *XML transformation* of NLP component output and input. This also includes *standoff annotation* access facilities.

Online integration support. In contrast to e.g. XML-based corpus annotation which is primarily an offline task, the *online combination* of NLP components has to be supported, i.e., several NLP components, possibly implemented in different programming languages, should be supported to run in parallel or in a sequence without interruption or manual intervention.

Multilinguality, Unicode support. Finally, while Unicode support is implied by the XML condition just mentioned, a further issue is *multilinguality* of linguistic resources and processors that may have an impact on the architecture as well.

From this short enumeration, it is obvious that for a flexible application integration, these conditions cannot be met by *ad hoc* combinations of natural language processors, but only by well-designed *architectures*. In the following chapters, we will describe three architectures, with different focuses, for deep-shallow integration.

6.2 Related Work

The distinction between *ad hoc* NLP component combination vs. architecture already divides the field of existing approaches into two areas, namely general NLP architectures (for shallow processing, without a claim to specifically support deep processors or deep-shallow integration), and more or less *ad hoc* integrations of a fixed set of deep and shallow NLP component instances (in most cases for a fixed language, too). While examples exist for both areas, what was missing so far, and hence is in the focus of this thesis, is a general architecture for deep-shallow integration.

In this section, we will focus on related work in general NLP architectures. Specific (mainly *ad hoc*) deep-shallow integrations and architectures (e.g. Grover and Lascarides 2001; Prins and van Noord 2001; Marimon 2002a,b; Daum *et al.* 2003) will be discussed in related work in Chapter 8.

Cunningham *et al.* (1997) present a classification of software infrastructures for NLP by distinguishing three models they call

- *referential* (analyses are stored as separate representations with pointer references into the original text),

- *additive* (e.g. cumulative SGML/XML annotation markup), and
- *abstraction-based* (as in typed feature structures of deep analysis where the analysis result consists of a closed, integrated information structure for larger text entities, typically a whole sentence).

From the requirements we formulated above for deep-shallow processing (cf. page 107) follows that all three of them are needed for architectures integrating deep and shallow NLP, where additive markup can be easily simulated by referential storage.

TIPSTER TIPSTER (Grishman, 1997) falls into the class of *referential* models. It mainly provides a document architecture aiming at facilitating the integration of shallow NLP components on very large document collections, e.g. for information extraction. The (relational) database view on the analysis results imposes restrictions on the data models that are supported, but is sufficient for shallow component analyses and has the advantage of supporting very fast access to large analyzed document collections.

Corelli Corelli (Zajac *et al.*, 1997; Zajac, 1998) extends TIPSTER by generalizing feature-value pairs to typed feature structures with type declarations and type checks, hence opening TIPSTER towards the *abstraction-based* approach. However, the status and availability of the implementation is unclear and the architecture seems not to have been used outside the initial project.

GATE GATE (Cunningham *et al.*, 2002) augments TIPSTER by *additive* markup extensions. GATE also introduces multilinguality, comes with impressive amount of resources and ready-to-use components, and has a vivid user community. The lack of a declarative formalism (beyond pattern matching) for the development of e.g. domain-specific resources with the need to fall-back to C or Java program code makes the system somewhat awkward and look more ad-hoc than desirable.

In that sense, GATE's focus is more an architecture shell with flexible and graphically definable NLP analysis workflows. GATE can well be used for combining multiple shallow preprocessors, but as GATE completely ignores deep linguistic processing, e.g. on the basis of typed feature structures, it is not an ideal candidate for deep-shallow integration. GATE will also be discussed in related work in Chapter 7.

ALEP ALEP (Simpkins, 1994) is a parser for HPSG-like grammars implemented in Prolog that has been in a later development phase extended by an SGML interface for external part-of-speech taggers and open-class words such as number, time and named entity expressions (Bredenkamp *et al.*, 1996; Declerck and Maas, 1997), the, as the authors call them, *messy details* in large-scale grammar development.

ALEP foresees an explicit interface between preprocessing ('text handling' in their terminology) and the deep parser ('linguistic structure'). The interface consists of Prolog terms that insert information gathered by a (rule-based) tagger and `awk` or `perl`-encoded named entity identification and normalization rules into the typed feature structures of the core ALEP parser.

Because of pre-parsing disambiguation of part-of-speech information through the tagger (and the generally slow Prolog parser), the parsing time for a sentence could be drastically reduced (approx. factor 5 according to the paper) simply by filtering out alternatives that would otherwise have to be processed by time-costly unification.

The system has been developed in the LS-GRAM and MELISSA projects applied to newspaper texts and command and control for English, German and Spanish. The ALEP system itself has two severe restrictions that make its use for real, state-of-the-art HPSG grammars hard. The first one is the lack of multiple inheritance in the formalism (it must be simulated through macros). Secondly, the overall performance of the parser is extremely poor compared to more recent implementations such as PET.

LT XML LT XML (Brew *et al.*, 2000), originally developed as LT NSL for SGML, consists of a set of C programs for combining and querying XML annotations. It is file-based and, besides command line tools, comes with a C API. Most of the features such as pointers, querying, transforming, counting, searching, sorting and text stripping XML that are available as separate tools, can nowadays be formulated more conveniently in XPath or XSLT which were not available at the time LT NSL and LT XML were developed. In that sense, LT-XML is an interesting early approach to *additive* markup, but somewhat technologically overcome by later W3C developments. There is no special handling or support for deep processing.

VERBMOBIL ICE VERBMOBIL ICE (INTARC Communication Environment; Amtrup 1995) is an infrastructure for communication in distributed AI systems that has been implemented on top of a virtualization machine PVM (parallel virtual machine). ICE provides a channel-based model for communication in heterogeneous networks and interfaces for programming languages such as C, C++, LISP, Prolog and TCL. ICE formed the communication infrastructure in the speech-to-speech translation system demonstrator and prototype of the VERBMOBIL project (Wahlster, 2000).

Component communication, especially for speech analysis, is clearly in the foreground of ICE, whereas the content combination or mediation is left to the underlying modules or applications. An explicit concurrency strategy (time constraints; interruptible components; the fastest component 'wins') is interesting, but contrary to our approach that tries to benefit from synergy gained through the combination of results of different components, assuming that they can complete their

computation, and that one can wait for the full output of the other (sentence-wise), thus treating NLP components as black boxes that are not interruptible[1]. As the recently developed NLP components are reasonably fast (compared to VERBMOBIL times), this is a realistic and not too limiting assumption.

Because of the time-critical task in VERBMOBIL speech-to-speech translation, deep processing had to be interrupted often before a full parse could be computed. In this case, chart fragments could be used, after topological sorting, to construct a partial, robust analysis (Kasper *et al.*, 1999). While this *anytime* approach may not lead to satisfactory semantic analyses (which is the key benefit one expects when employing deep analysis) in the general case, it was sufficient for short utterances in dialog situations as addressed by VERBMOBIL.

There was also an accuracy-oriented mode for situations without time constraints in VERBMOBIL, where deep analysis was not interrupted and could deliver full results. However, there was no content combination or mediation of NLP component output either.

MULTIPLATFORM testbed Similar to VERBMOBIL ICE, MULTIPLATFORM testbed (Herzog *et al.*, 2003, 2004) focuses on speech input, and, in addition, multimodal dialog, e.g. through gestures etc. The architecture has been developed for the large-scale SmartKom project (Wahlster, 2006) dealing with multimodal dialogs on various different devices from PDA, home information system, to public stationary communication kiosks.

The PVM-based architecture from VERBMOBIL has been extended by a publish/subscribe message system with named message queues, mainly to overcome the bottleneck of the point-to-point communication as imposed by the VERBMOBIL architecture. An XML language M3L has been developed to encapsulate the outputs of the various components. Just as in VERBMOBIL, component communication is in the foreground, not the close integration of different linguistic representations.

The architectures that will be described in the following chapters focus on written text and document analysis with an emphasis on high precision and detailed, complete analyses and hence do not require or admit the special time constraint and communication mechanisms implemented for VERBMOBIL and SmartKom. However, a combination of the described approaches could be useful at a later stage for e.g. speech processing.

6.3 General XML Processing Frameworks

In this context, we can only briefly discuss some XML-based or XML-supporting software architecture frameworks. They have been developed completely independently of natural language processing, and hence lack many of the concepts and

[1] As a matter of fact, none of the many pre-exsiting NLP components we use features anytime capabilities.

requirements that are needed for NLP. However, as they deal with XML representations, control flow and architecture, they could at least (and in fact are partly) used in the frameworks that will be described in the sequel.

The **XML Pipeline Definition Language** note[2] describes a framework for processing XML documents with minimal conversion overhead and definable workflow. XML pipelines describe the processing relationships between XML resources. A pipeline *document* specifies the inputs and outputs to XML processes and a pipeline *controller* uses this description to figure out the chain of processing that must be executed in order to get a particular result.

Apache Ant[3] is a build tool similar to the Unix `make` tool, but is working platform-independently as it is implemented in Java. It uses XML as *carrier syntax* for project and target descriptions. Besides dependency resolution of definable targets (inherited from the `make` concept), Ant supports parallelism, sequence, and XSL transformations. The definition of the workflows is done either implicitly using target dependencies (similar to XML pipelines, but without implicit flow of information) or explicitly as in other scripting languages. Ant will be discussed in a testing and evaluation application in Section 7.8.

Apache Cocoon[4] is a web publishing framework with explicit XSLT support with a focus on dynamic multi-channel web publishing and strict separation of application control, logic, content, and style (layout). It provides 'component pipelines', each component on the pipeline specializing on a particular operation. However, a strong focus is on HTML and PDF output of the XML-encoded content.

XBeans (Martin, 2000) is a generic framework that uses the Java beans technology to provide a component-oriented approach to define workflows on the basis of XML DOM documents. XBeans provide a nice and elegant way to implement distributed XML-based architectures on the basis of pattern-like components. Although already published as open source tool in the year 2000, the technology seems not to have reached the acceptation in the Java/XML community it may deserve.

Apache Tomcat[5] is a Web application server implemented in Java providing a reference implementation of the Java servlet and Java server page technology. Tomcat can be used to distribute application services (e.g. NLP) over a network architecture.

As stated above, these frameworks are too general for NLP component integration (e.g., there is no support for standoff annotation), but at least constitute interesting related frameworks, partly with explicit XSLT features, to XML-based software architecture.

[2] `http://w3c.org/TR/xml-pipeline/`
[3] `http://ant.apache.org`
[4] `http://cocoon.apache.org`
[5] `http://tomcat.apache.org`

6.4 The Deep-Shallow Architectures Trilogy

In the three following chapters, we will present our contributions to three architectures related to deep-shallow integration.

- **Chapter 7:** *SProUT* (Drożdżyński *et al.*, 2004; Becker *et al.*, 2002; Krieger *et al.*, 2004) is a recent, rule-based formalism and system with an XML architecture. It differs from the other two frameworks below in that full parsing in the HPSG spirit is not directly supported (in that sense, *SProUT* is shallow). However, the underlying formalism shares with HPSG the typed feature structures and the powerful unification operation, i.e., *SProUT* is hybrid on the formalism level.

  This permits highly structured information both encoded in rules and as output result, e.g. for information extraction. Moreover, *SProUT* interpreters can be cascaded and hence support more powerful ('deeper') analyses. *SProUT* itself is, as a structured shallow component, part of many multilingual deep-shallow integration scenarios we will describe for the third framework, Heart of Gold.

- **Chapter 8:** WHITEBOARD (Neumann and Schäfer, 2002; Crysmann *et al.*, 2002; Frank *et al.*, 2003) is a sequential architecture for standoff markup, based on XSLT. The focus lies on filtering for search space reduction as input for the deep processing located at the end of the sequence. Advantages are improved lexical and syntactic coverage and parsing speedup.

  WHITEBOARD is probably the most comprehensive broad-coverage deep-shallow integration that has been implemented so far, with support for parsing open-domain newspaper text. However, it has been fully instantiated for German only, and is now in many respects superseded by the third framework, Heart of Gold.

- **Chapter 9:** Heart of Gold, the DEEPTHOUGHT core architecture framework (Callmeier *et al.*, 2004; Schäfer, 2006a) is a generalization of WHITEBOARD with more flexible configuration facilities, the depth of analysis can be chosen by application clients. In addition to syntactic standoff markup, a uniform robust semantics representation formalism (RMRS; Copestake 2003) is employed, serving e.g. as additional fall-back and *post-parsing* integration layer.

  Heart of Gold extends WHITEBOARD with respect to distributability over a network, multilinguality support, less restrictive processing model (not just sequential, but also concurrent NLP), *etc.* Thus, Heart of Gold can be characterized as NLP middleware in between NLP-based applications and existing NLP components. Heart of Gold includes *SProUT* as important, however one of many integrated, multilingual components.

Chapter 7

SProUT

7.1 Introduction

The first of the three architectures to be described is *SProUT*. *SProUT* is different from the other two architectures we will present in the next two chapters in that is does not constitute a proper, dedicated architecture for the integration of (pre-existing) deep and shallow processing components, nor is it possible or intended to parse HPSG or related deep grammars with *SProUT* alone.

Instead, *SProUT*[1] is an amalgamation of two basic concepts from shallow processing, namely finite-state methods, and from deep (constraint-based) processing, namely typed unification based on an underlying type inheritance hierarchy, combined in a single, new, declarative grammar formalism.

In other words, not the architecture is hybrid, but the formalism. However, as *SProUT* also provides a flexible architecture combining subcomponents for tokenization, morphological analysis and gazetteers (in addition to its grammar formalism), it forms a powerful general-purpose multilingual natural language processor that fits well into the architecture trilogy.

Although the implemented processing of *SProUT* strategy is rather shallow than deep, *SProUT* grammars could be combined to fulfill tasks that could well be characterized as deep processing. Moreover, the structured data model of *SProUT* resembles (and in fact is inherited from) unification-based grammar formalisms of deep parsers.

Thus, *SProUT* is 'deeper' and more flexible than most other shallow processors (that are specialized in typically a single task), and can also be used as part of hybrid deep-shallow integration architectures. Although shallow in the processing paradigm, *SProUT* stands out from the crowd of typical shallow systems through its rich declarative formalism and flexibility, and may help to fill the gap between classical shallow and deep natural language processors.

The development of the *SProUT* formalism and implementation is joint work

[1] *SProUT* is an abbreviation for 'Shallow Processing with Unification and Typed Feature Structures'.

(Becker *et al.*, 2002; Drożdżyński *et al.*, 2004). After a brief introduction, we will therefore focus on our own contributions to formalism, interfaces and XML facilities related to deep-shallow processing and XML annotation transformation. Applications of *SProUT* as part of hybrid deep-shallow architectures will be discussed in Chapter 9.

7.2 A Brief Introduction to *SProUT*

7.2.1 Motivation

The *SProUT* formalism combines unification of typed feature structures and regular expressions in a rule-based framework. Regular expressions, not directly available in typed feature structures, are a simple, efficient and intuitive means to represent patterns over symbols with potential repetition. Typed feature structures generalize symbols (strings) and add further information-bearing constructs, namely (1) arbitrarily nested feature-value pairs (2) types ordered in a closed world inheritance hierarchy, (3) structure sharing between feature values (cf. Section 3.1.1).

The motivation for developing a hybrid grammar formalism combining both paradigms for shallow natural language processing is driven by the observation that (i) simple regular expression matching over input symbols (text strings or abstractions thereof) is insufficient, error-prone and inappropriate for advanced language technology tasks (e.g. in languages with rich morphology). (ii) full parsing with typed feature structures is computational overkill and not robust enough for simple language technology tasks such as named entity extraction or template-based information extraction.

7.2.2 Targeted Applications

Applications of *SProUT* are various domain-specific basic NLP techniques such as multilingual named entity recognition with structured output, morphological analysis and shallow parsing, but also advanced information extraction tasks such as template-based information extraction, shallow relation extraction, opinion mining, *etc*. As both formalism and implementation are generic, other applications are feasible, even outside the field of language technology.

The advantage of using unification and typed feature structures in unification-based grammar formalisms (Kay, 1979; Pereira and Warren, 1980; Shieber *et al.*, 1983; Shieber, 1986)[2] has already been motivated in Chapter 3. They provide a *monotonic* and declarative representation language for linguistic knowledge on which a parser/generator or a uniform type deduction mechanism acts as the inference engine.

In contrast to simple feature-value pairs with atomic values as they are used e.g. in GATE's JAPE formalism discussed in the next section, typed feature struc-

[2] A slightly more general, but often synonymously used term is constraint-based grammar formalism.

tures provide additional expressivity through (1) types ordered in a type inheritance hierarchy, (2) nested feature-value structures (3) coreferences as an explicit structure-sharing facility between feature values (cf. variables in logic programming), altogether with a well-researched set-theoretical semantics.

The well-defined unification operation, set-theoretically an intersection of the denotation of two typed feature structures that is both constructive and determines satisfiability (compatibility), and subsumption (subset/superset relation between the denotations of two typed feature structures) provide two powerful operations that can be used to compare and merge typed feature structures in a monotonic way.

7.2.3 Related Work

Both finite-state techniques and unification-based typed feature structures have a long tradition in natural language processing.

The pure finite-state-based shallow processing approaches have proved to be very efficient in terms of processing speed. The first applications of finite-state processing in NLP were morphology and phonology (Koskenniemi, 1983; Karttunen, 1983; Kaplan and Kay, 1994) but have been extended later to many NLP tasks including tokenization and 'light' parsing, *etc* (Karttunen *et al.*, 1996). Piskorski and Neumann (2000) present SPPC, a highly efficient system, which uses cascades of simple finite-state grammars, based on a small number of basic predicates. Complex constraints cannot be encoded in the finite-state device. The idea of using more complex annotations on the transitions of FS automata has been considered in SMES (Neumann *et al.*, 1997) which uses regular grammars with predicates over morphologically analyzed tokens.

These (LISP) predicates inspect arbitrary properties of the input tokens such as part of speech or inflectional information. Van Noord and Gerdemann (2001) introduce arbitrary predicates over symbols and discuss various operations on finite-state acceptors and transducers. They observe that automata with predicates generally have fewer states and transitions. However, the discussed automata only operate on symbols of a finite input alphabet. As a drawback of using too many or too complex predicates, standard optimization techniques are hardly applicable.

Cascaded finite-state systems have been developed for information extraction. The most successful systems provide high-level specification languages for grammar writing. The FASTUS system (Hobbs *et al.*, 1997) uses CPSL (Common Pattern Specification Language). The more recent GATE system (Cunningham, 2000) provides JAPE (Java Annotation Patterns Engine), which is similar in spirit to CPSL and borrows features from CPSL. A CPSL/JAPE grammar contains pattern-action rules.

The LHS (left hand side) of a rule is a regular expression over *atomic* feature-value constraints called annotations (the recognition part), but without types, unification or typed feature structure concepts, while the RHS (right hand side) is an *annotation manipulation statement* for output production, which calls native code

(e.g. C or Java), making rule writing difficult for non-programmers. Furthermore, even though there is a mechanism for variable binding which is responsible for copying values to the RHS, this mechanism is not capable of declaratively describing structure sharing among the rule elements. The annotation model of GATE is based on TIPSTER (Grishman, 1997), already discussed in Section 6.2.

Like *SProUT*, Ellogon is a 'multilingual, cross-platform, general-purpose text engineering environment' (Petasis *et al.*, 2002). It shares with GATE the TIPSTER annotation concept. However, while GATE provides at least a simple pattern-based grammar formalism, Ellogon is a pure architecture and visualization shell to combine existing NLP components programmatically through APIs. Both systems do not provide a powerful and generic declarative formalism comparable to *SProUT*'s *TDL* and *XTDL*.

7.2.4 The *SProUT* Formalism

The *SProUT* formalism consists of two parts, *TDL* (Krieger and Schäfer, 1994; Uszkoreit *et al.*, 1994) for building the type hierarchy, and *XTDL* for the rule syntax that incorporates *TDL* and extends its typed feature structure part by regular expressions, sets *etc*. Both are grounded in a closed type world semantics version that is shared with grammar formalisms of PET (Callmeier, 2000) and LKB (Copestake, 2002). The closed-world semantics means that types are pairwise incompatible unless they are in a subtype relationship or share an explicitly defined common subtype (greatest lower bound, GLB).

7.2.4.1 *TDL*

In the BNF syntax of *TDL* (Figure 7.1), *identifier* and *string* are pre-terminals defined as in usual programming language syntax for identifiers such as class names or variable names and strings (character sequences enclosed in double quotes).

$$
\begin{array}{lll}
\textit{typedef} & \rightarrow & \textit{type}\ \texttt{":="}\ \textit{avm}\ \texttt{"."} \mid \\
 & & \textit{type}\ \texttt{":<"}\ \textit{type}\ \texttt{"."} \mid \\
 & & \textit{string}\ \texttt{":<"}\ \textit{type}\ \texttt{"."} \\
\textit{type} & \rightarrow & \textit{identifier} \\
\textit{avm} & \rightarrow & \textit{term}\ \{\ \texttt{"\&"}\ \textit{term}\}^{*} \\
\textit{term} & \rightarrow & \textit{type} \mid \textit{fterm} \mid \textit{string} \mid \textit{coref} \\
\textit{fterm} & \rightarrow & \texttt{"["}\ [\textit{attr-val}\ \{\texttt{","}\ \textit{attr-val}\}^{*}]\ \texttt{"]"} \\
\textit{attr-val} & \rightarrow & \textit{identifier}\ \textit{avm} \\
\textit{coref} & \rightarrow & \texttt{"\#"}\textit{identifier}
\end{array}
$$

Figure 7.1: *TDL* syntax for type definitions

The *typedef* production rule is used to introduce new types. The left type name is the new type to be defined, the right type name is the supertype. If additional features or feature value refinements are introduced, the `":="` must be used to

indicate the type definition, and the ampersand is used for combination (indicating set-denotational intersection), e.g.

```
ne-location := enamex & [LOCTYPE loc-type,
                         LOCNAME string,
                         AREA string].
```

introduces the new type *ne-location* as subtype of enamex and introduces three new features LOCTYPE, LOCNAME and AREA. As can also be seen from this example, feature-value pairs are enclosed in square brackets and separated by comma.

Multiple inheritance is defined by specifying a complex supertype expression on the right side, where the supertypes are again combined using the ampersand symbol:

```
inf := fin_inf & inf_infzu & inf_prp & inf_psp.
```

Otherwise, `":<"` indicates type introduction without feature refinement, e.g.

```
noun :< part_of_speech.
```

Coreferences indicate structure sharing using variables with a leading hash sign, e.g.

```
[ATTR1 #shared_value, ATTR2 #shared_value]
```

In addition to what is stated in the (simplified) BNF, there are abbreviation constructs ('syntactic sugar') for list-valued feature structures. < and > enclose list elements (*avm*) separated by comma, an abbreviation for the **list** and **cons** types for first-rest lists encoded as typed feature structures.

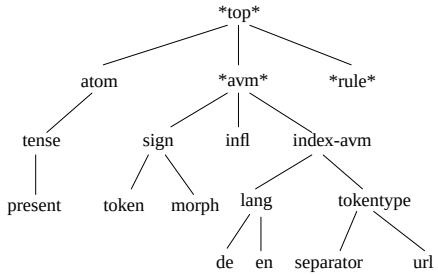
To this aim, besides **top**, the most general type in the type hierarchy, and **avm**, the supertype of all attributed types, the following types are predefined:

```
*avm* :< *top*.
*list* :< *avm*.
*null* :< *list*.
*cons* := *list* & [FIRST *top*,
                    REST *top*].
```

In the sample definition below, the *morph* type inherits from *sign* and introduces three more morphological attributes with the corresponding value type restrictions.

```
morph := sign & [POS  pos,
                 STEM string,
                 INFL infl,
                 SEGMENTATION list].
```

Similarly, basic types such as *tokenizer* and *gazetteer* exist for each of the predefined *SProUT* processing components that will be described later. The next figure illustrates a small (upper) fragment of a type hierarchy.



Except for the predefined types **top**, **avm**, **null**, **list** and **cons**, all type definitions are optional and can be assigned freely for task-specific purposes. The type hierarchy is compiled into an efficient bit-vector encoding using the grammar preprocessor `flop` that is part of the PET system (Callmeier, 2000).

7.2.4.2 *XTDL*

The *SProUT* rule syntax, called *XTDL*, is an extension of the *TDL* syntax for defining type hierarchies, but without the *typedef* production (Figure 7.1). Instead, it is possible to define a grammar consisting of rules with regular expression patterns on the LHS that do not match simple atomic symbols, but typed feature structures (production rule *avm* in the BNF in Figure 7.1).

The typed feature structure input stream is e.g. generated by a tokenizer or other preprocessing components such as morphology or a gazetteer lookup module from an input text. Alternatively, arbitrary typed feature structures can be given as input via an API or as XML input document. The *XTDL* grammar is compiled and interpreted by the *XTDL* interpreter at runtime.

The LHS of an *XTDL* rule is a regular expression over typed feature structures (in *TDL* syntax), representing the recognition pattern. The RHS consists of a single typed feature structure specifying the output structure. Consequently, equality of atomic symbols is replaced by *unifiability* of typed feature structures and the output is constructed using typed feature structure *unification* w.r.t. the type hierarchy defined in *TDL*.

The rule concept (already without the regular expression over typed feature structures extension) is comparable to *lexical rules* in unification-based grammars such as the following from the first HPSG book (Pollard and Sag, 1987).

$$\begin{bmatrix} \textit{base} \\ \text{PHON } \boxed{1} \\ \text{3RDSNG } \boxed{2} \\ \text{SYN|LOC|SUBCAT } \boxed{3} \\ \text{SEM|CONT } \boxed{4} \end{bmatrix} \rightarrow \begin{bmatrix} \textit{3rdsng} \\ \text{PHON } f_{\textit{3RDSNG}}(\boxed{1},\boxed{2}) \\ \text{SYN|LOC|SUBCAT } \boxed{3} \\ \text{SEM|CONT } \boxed{4} \end{bmatrix}$$

Such a rule, including the transport mechanism of feature values from the LHS to the RHS through coreferences, and the functional operator f_{3RDSNG} computing a phonological variant, could be directly expressed in *XTDL* syntax, the vertical bar | being syntactic sugar for embedded feature paths.

The implemented processing strategy is *longest match*: The rules are applied to the input sequence. If multiple rules apply to the same sequence of input items, then the rule(s) with the longest matching input sequence win, are evaluated such that the RHS is instantiated, and the resulting output feature structures is appended to the output of the rule interpreter.

An *XTDL* rule (cf. BNF in Figure 7.2) starts with a named label for the rule name. Rules can be 'called' from other rules using the `seek` operator and indicating the rule via its label. After the `:>` separator, the LHS recognition part follows (a regular expression), then the LHS/RHS separator `->`, finally the RHS output feature structure. Each rule is terminated by a dot.

For the LHS regular expression over typed feature structures, the standard operators `*`, `?`, `+`, `{n}`, `{m,n}` can be employed for Kleene star, optionality, Kleene plus, n-fold repetition and range. A simple space between the terms signifies concatenation (sequence of input items).

rule → *rulename* {`":>"` | `":/"`} *regexp* `"->"` [*avm*] [*fun-op*] `"."`
rulename → *identifier*
regexp → *avm* | `"@seek("` *rulename* `")"` | `"("` *regexp* `")"` |
regexp {*regexp*}$^+$ | *regexp* {`"|"` *regexp*}$^+$ |
regexp {`"*"` | `"+"` | `"?"`} | *regexp* `"{"` *int* [`","` *int*] `"}"`
fun-op → `", where"` *coref* `"="` *fun-app* {`","` *coref* `"="` *fun-app*}*
fun-app → *identifier* `"("` *term* {`","` *term*}* `")"`

Figure 7.2: *XTDL* rule syntax as extension of *TDL*

We briefly exemplify the conciseness of the formalism. The first *XTDL* grammar rule describes a sequence of morphologically analyzed tokens (of type *morph* with attributes POS and INFL). The first TFS matches one or zero items (`?`) with part-of-speech `Determiner`. Then, zero or more `Adjective` items are matched (`*`). Finally, one or two `Noun` items ({`1,2`}) are consumed.

The use of a variable (e.g. `#case`) in different places establishes a coreference (i.e., structure sharing) between features. This example enforces e.g. agreement in case, number, and gender for the matched items. I.e., all adjectives must have compatible values for these features.

If the recognition pattern on the LHS successfully matches the input, the description on the RHS creates a feature structure of type *phrase*. The category is shared with the category `Noun` of the right-most token(s) and the agreement features result from the unification of the agreement features of the *morph* tokens. An extended example of morphology input items for a complete sentence is depicted on page 214.

```
np :> morph & [POS  Determiner,
               INFL [CASE #case, NUM #number, GEN #gender ]] ?
      morph & [POS  Adjective,
               INFL [CASE #case, NUM #number, GEN #gender ]] *
      morph & [POS  Noun & #cat,
               INFL [CASE #case, NUM #number, GEN #gender ]] {1,2}
   -> phrase & [CAT #cat,
               AGR agr & [CASE #case, NUM #number, GEN #gender ]].
```

in AVM notation, we use the bullet (•) sign to indicate sequence on the LHS. This corresponds to a whitespace in the *XTDL* syntax. The usage of the other symbols for regular expressions *etc* should be obvious.

np :> [*morph*; POS *Determiner*; INFL [CASE case; NUM number; GEN gender]]$^{?}$ • [*morph*; POS *Adjective*; INFL [CASE case; NUM number; GEN gender]]* • [*morph*; POS cat *Noun*; INFL [CASE case; NUM number; GEN gender]]$^{\{1,2\}}$ → [*phrase*; CAT cat; AGR [*agr*; CASE case; NUM number; GEN gender]] .

The second example addresses the recognition of river names. The rule matches either expressions consisting of an (unknown) capitalized word (via match with token type *1stcapwd*), followed by a noun with stem *river* or *brook* (via the English morphology component; disjunction has a higher precedence than concatenation), or Gazetteer entries of type *gaz_river* containing English river names represented by the Gazetteer type *gaz_river*.

The generated output structure of type *ne-location* contains a location type *river* and the location name transported via the coreference symbol `#lname`. To sum up, this rule recognizes both unknown river names (via a pattern involving morphology lookup) and known river names (via a gazetteer match).

```
river :> (token & [TYPE 1stcapwd, SURFACE #lname]
            (morph & [STEM "river", POS noun, SURFACE #key]
           | morph & [STEM "brook", POS noun, SURFACE #key]))
| (gazetteer & [GTYPE gaz_river, CONCEPT #lname, DESIGNATOR #key])
-> ne-location & [LOCTYPE river, LOCNAME #lname, DESCRIPTOR #key].
```

in AVM notation

river :> ([*token*; TYPE *1stcapwd*; SURFACE lname] • ([*morph*; STEM "river"; POS *noun*; SURFACE key] | [*morph*; STEM "brook"; POS *noun*; SURFACE key]))

$$\left[\begin{array}{ll} \textit{gazetteer} & \\ \text{GTYPE} & \textit{gaz\_river} \\ \text{CONCEPT} & \boxed{\text{lname}} \\ \text{DESIGNATOR} & \boxed{\text{key}} \end{array}\right] \rightarrow \left[\begin{array}{ll} \textit{ne-location} & \\ \text{LOCTYPE} & \textit{river} \\ \text{LOCNAME} & \boxed{\text{lname}} \\ \text{DESCRIPTOR} & \boxed{\text{key}} \end{array}\right].$$

XTDL provides a functional operator facility that can be used to place function calls with values and arguments linked through coreference constraints into the rules. The functions can be defined in Java code that is associated with the grammars. Typical applications of this operator are user-defined string operations.

Example (taken from the English named entity grammar):

```
;; Dummy rule for "en_temp_unit_amount"
n_and_a_half :/
   gazetteer & [GTYPE gaz_cardinal, CONCEPT #num, CSTART #cs]
   token & [SURFACE "and"]
   morph & [STEM "a"]
   token & [SURFACE "half"]
-> interval & [TIMEX_AMOUNT #num_half, CSTART #cs],
   where #num_half = Append(#num, ".5").
```

in AVM notation

$$\text{n\_and\_a\_half} :> \left[\begin{array}{ll} \textit{gazetteer} & \\ \text{GTYPE} & \textit{gaz\_cardinal} \\ \text{CONCEPT} & \boxed{\text{num}} \\ \text{CSTART} & \boxed{\text{cs}} \end{array}\right] \bullet \left[\begin{array}{ll} \textit{token} & \\ \text{SURFACE} & \texttt{"and"} \end{array}\right] \bullet \left[\begin{array}{ll} \textit{morph} & \\ \text{STEM} & \texttt{"a"} \end{array}\right] \bullet$$

$$\left[\begin{array}{ll} \textit{token} & \\ \text{SURFACE} & \texttt{"half"} \end{array}\right] \rightarrow \left[\begin{array}{ll} \textit{interval} & \\ \text{TIMEX\_AMOUNT} & \boxed{\text{num\_half}} \\ \text{CSTART} & \boxed{\text{cs}} \end{array}\right]$$

where

$$\boxed{\text{num\_half}} = \text{Append}\left(\boxed{\text{num}}, \texttt{".5"}\right).$$

where `Append` is the usual string-appending function[3].

The rule recognizes number expressions written as e.g. ‘two and a half’ and copies the normalized numeric amount into the output structure (in this case for time expressions/duration) under the feature TIMEX_AMOUNT’, e.g. as `"2.5"`. In this rule, the number as text recognition is implemented via a gazetteer lookup (first pattern).

Nonterminals not defined in the *XTDL* BNF (Figure 7.2) are shared with the *TDL* syntax. The *TDL* BNF rule for *term* is augmented in *XTDL* by set-valued

[3]There is only a handful of elementary functional operators needed for the later described multilingual named entity grammars. Normally, grammar writers do not need to define new functional operators.

attribute values (in curly brackets) and the collect operator that collects values repeated under Kleene star or plus in a set or list on the RHS.

$$\begin{aligned} \textit{term} &\rightarrow \textit{type} \mid \textit{fterm} \mid \textit{set} \mid \textit{coref} \mid \textit{collect} \\ \textit{set} &\rightarrow \texttt{"\{"}\ [\textit{term}\ \{\texttt{","}\ \textit{term}\}^*]\ \texttt{"\}"} \\ \textit{collect} &\rightarrow \texttt{"\%"}\textit{identifier} \end{aligned}$$

A weak form of negation is also supported, but at the top-level of LHS pattern expressions only, i.e., not as feature values [4].

The following sample rule matches (via the German morphology component) noun phrases such as ‘der grüne Baum’ or ‘den großen Bäumen’, but excludes ‘der seltene Baum’ or ‘des seltenen Baumes’.

```
baum_rule :> morph & [STEM "der"]
             morph & ~ [STEM "selten"] & [STEM #stem]
             morph & [STEM "baum"]#
          -> out & [DESCR #stem ].
```

in AVM notation

$$\text{baum\_rule} :> \begin{bmatrix} \textit{morph} \\ \text{STEM} \ \texttt{"der"} \end{bmatrix} \bullet \neg \begin{bmatrix} \textit{morph} \\ \text{STEM} \ \texttt{"selten"} \\ \text{STEM} \ \boxed{\text{stem}} \end{bmatrix} \bullet \begin{bmatrix} \textit{morph} \\ \text{STEM} \ \texttt{"baum"} \end{bmatrix} \rightarrow \begin{bmatrix} \textit{out} \\ \text{DESCR} \ \boxed{\text{stem}} \end{bmatrix}.$$

In contrast to negation, sets are only admitted as feature values. We present an example in combination with the collect operator that collects the values under Kleene star on the LHS of a rule in a set on the RHS. The following rule matches (via the German morphology component) phrasal expressions such as ‘die kleinen grünen Männchen’ and outputs the stems of the adjectives, e.g. {”klein”, ”grün”}, in a set under the DESCR attribute in the output structure.

```
collect_adjs :>  morph & [POS det]
                (morph & [POS adjective, STEM %1])*
                 morph & [POS noun]
              -> out & [DESCR %{1}].
```

in AVM notation

$$\text{collect\_adjs} :> \begin{bmatrix} \textit{morph} \\ \text{POS} \ \textit{det} \end{bmatrix} \bullet \begin{bmatrix} \textit{morph} \\ \text{POS} \ \textit{adjective} \\ \text{STEM} \ \boxed{1} \end{bmatrix}^* \bullet \begin{bmatrix} \textit{morph} \\ \text{POS} \ \textit{noun} \end{bmatrix} \rightarrow \begin{bmatrix} \textit{out} \\ \text{DESCR} \ \boxed{\{1\}} \end{bmatrix}.$$

[4] This would require extending the *TDL* formalism with disjunction, a great source of inefficiency. Instead, disjunction e.g. of morphological feature values, can easily and very efficiently be expressed using the type hierarchy.

The above rules also exemplified how morphological analysis can be used to write general rules that apply to a whole bunch of collocated words, thus making grammar writing quite comfortable.

A final remark on the examples. For the sake of simplicity and also for showing the application of built-in morphology and gazetteer components, the rule input was chosen to be generated by one of these ready-to-use *SProUT* components. However, it should be pointed out here that also general feature structures could be used as input, e.g. produced by a previous component (e.g. another *SProUT* grammar) in a cascade, or by any other NLP component, transformed into a format ingestible by the *SProUT* interpreter. In the next section, we will show how the architecture is set up and what general (XML) input and output formats exist in order to connect *SProUT* with the outside world, including applications.

7.2.5 Architecture and Components

Central to the *SProUT* architecture is the *interpreter*, the core algorithm for executing the *XTDL* grammar that was transformed into a minimized automaton at compile time. The interpreter tries to match input sequences of typed feature structures with the *XTDL* grammar LHS of the grammar rules using typed feature structure unifiability (and unification in case of a match).

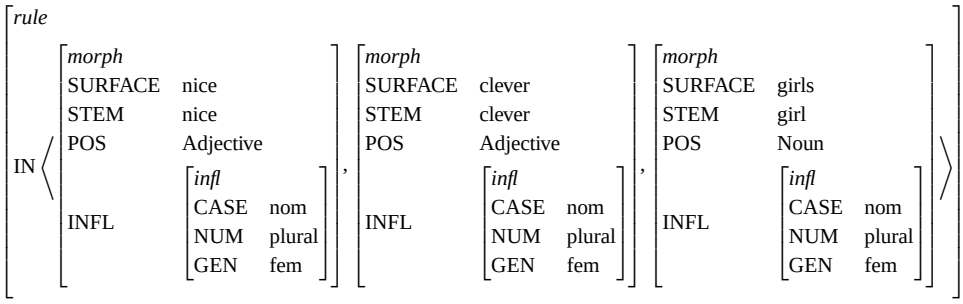


Figure 7.3: The matched input sequence for the phrase ‘nice clever girls’

The output is the sequence of the RHSes of the successfully applied (matching) rules that may include values from the LHSes when transported to the RHS via unification. The implementation of the ‘transport’ mechanism itself is straightforwardly performed by putting LHS and RHS into a single, wrapping feature structures under attributes IN in a list (LHS) and OUT (RHS), structure sharing and unification will do the real transport of values.

We give a short example for the noun phrase rule defined on page 122 matching an input sequence ‘nice clever girls’. The morphological analysis of that phrase as input to the interpreter is presented in Figure 7.3, the angle brackets being syntactic sugar for a first-rest list representation.

The morphology components of *SProUT* for European languages such as English, German, French, Italian or Spanish are based on MMorph resources (Petitpierre and Russell, 1995; Krieger and Xu, 2003). Japanese segmentation and PoS tagging is based on ChaSen (Asahara and Matsumoto, 2000), Chinese segmentation on ShanXi (Liu, 2001).

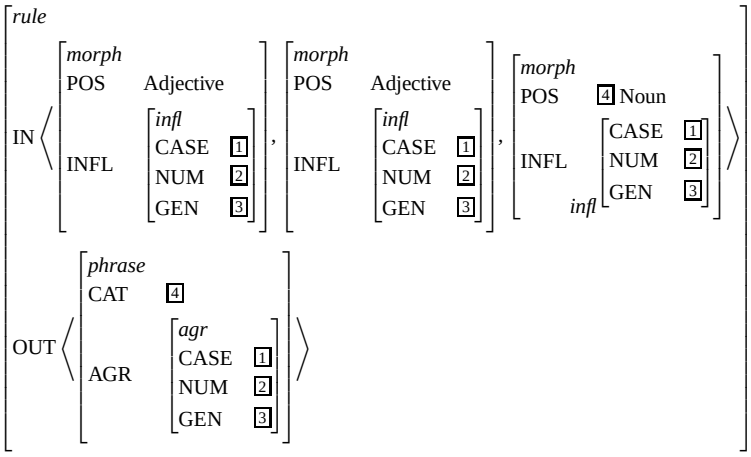


Figure 7.4: The successfully expanded NP rule from page 122 with no determiner (pattern: `?`), two adjectives (`*`), and a noun (`{1,2}`)

From the rule definition and the input token sequence, the interpreter will construct a TFS with an instantiated LHS pattern as a valid expansion of the regular expression in the rule definition (Figure 7.4).

Unification of the morphology input sequence will result in the structure shown in Figure 7.5, where the output of the rule application can be found under feature OUT.

Figure 7.6 displays an overview over the major, standard components of the system and their connection to the interpreter. Most of the resources of the standard components (morphology, type hierarchy, gazetteer, grammar) are compiled through specialized (mostly finite-state) compilers for efficient processing at runtime.

For details on the various compilation and minimization techniques developed and implemented for the *SProUT* system, we refer the reader to Krieger *et al.* (2004), as we concentrate here on the I/O embedding of *SProUT*.

The common data structure for the basic components such as tokenizer, morphology, gazetteer and *XTDL* grammar are typed feature structures[5]. Each type and attribute output by the components must have an appropriate definition in the *TDL* type hierarchy. Otherwise, a runtime error will be signaled. I.e., types are not only the 'glue' between components and the interpreter, but also serve as a means

[5]This also includes the added interfaces to external morphology/segmentizer tools for Asian languages mentioned above.

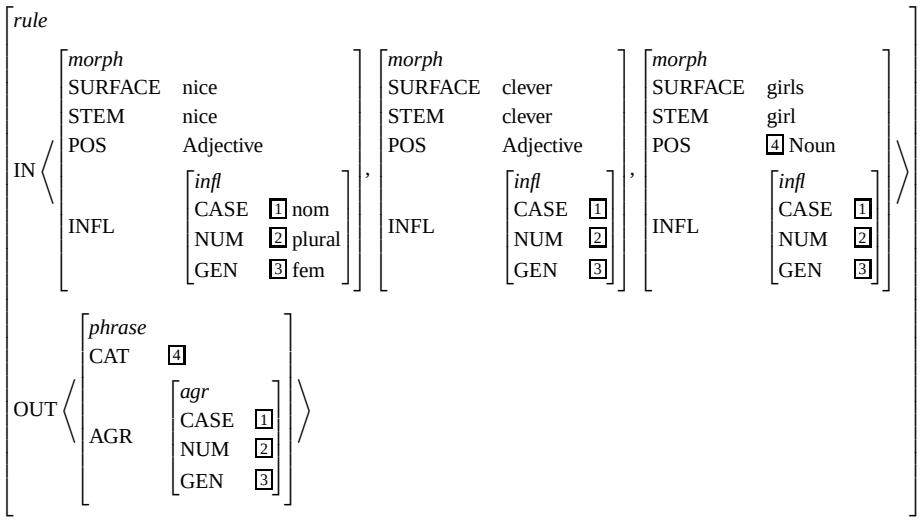


Figure 7.5: The final result from the unification of an expanded instance of the noun phrase rule from page 122 with the TFS for the input ‘nice clever girls’ from Figures 7.3 and 7.4

for consistency checking (cf. Section 7.4).

The type hierarchy is compiled from the *TDL* sources to a compact binary representation using the `flop` preprocessor of the PET system (Callmeier, 2001). The binary representation is read by the JTFS (Java Typed Feature Structures) subsystem of *SProUT* implemented by Hans-Ulrich Krieger that uses the compact encoding of the type hierarchy and provides Java classes for representing typed feature structures including methods for unification and subsumption testing.

7.3 *SProUT*put DTD: XML and XSLT Transformation of Results

In this section, we discuss transformation of the typed feature structures serialized to XML. As mentioned and motivated in the architecture description above (Section 5.4.1), XML-encoded typed feature structures can play an important role for interfacing and integration of components.

XML serialization and transformation is e.g. useful for (1) interfacing applications and external components that do not use typed feature structures or use other representations of typed feature structures than the Java implementation (JTFS) of *SProUT*; (2) persistent storage of analysis results, e.g. as automatically annotated corpora for training statistical models; (3) for visualization of results in e.g. AVM or tree representation.

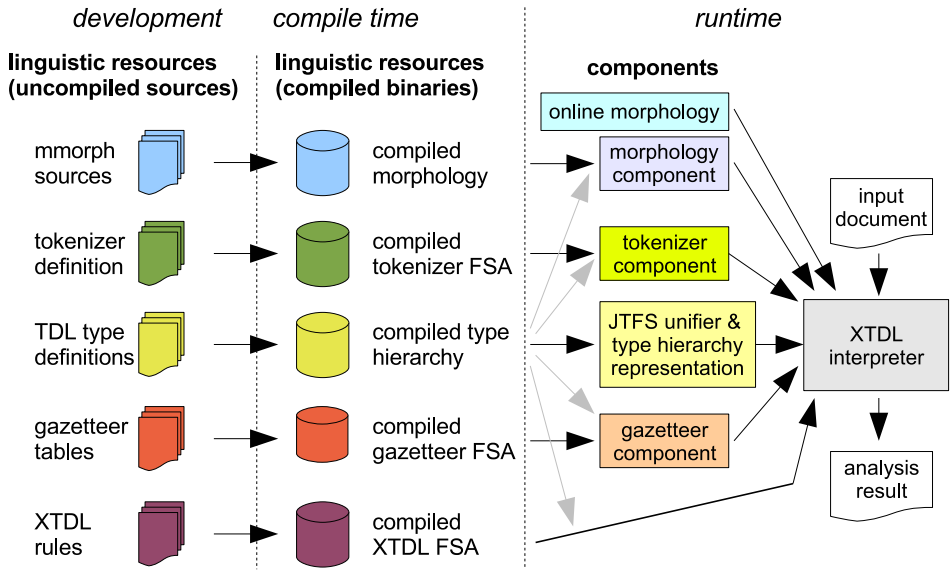


Figure 7.6: *SProUT* Architecture

The typed feature structures of *SProUT* can be serialized (without the type hierarchy information which is supposed to be stored externally in a compressed bit-vector encoding format for efficiency reasons[6]) to XML using an extended version of the *minimal typed feature structure DTD* described in Section 5.4.1. We call this format the *SProUT*put DTD.

The full DTD is contained in the DTD Appendix on page 288. A *SProUT*put document consists of a disjunction of MATCHINFO items, each MATCHINFO containing all readings recognized for a character span (or token span) in the input. A reading is simply a typed feature structure as generated by the *SProUT* interpreter, in XML transcription.

This is basically a one-to-one mapping of the data structures[7] the *SProUT* implementation uses. The same XML format and corresponding JTFS data structure can also be given as input to the *SProUT* interpreter, i.e., as a stream of MATCHINFO objects. This also enables straightforward *cascading* of *SProUT* grammars: One grammar can take the output of the previous grammar as input.

The XML format can be transformed into any other XML or text format using the built-in *SProUT*put XSLT transformer[8] that provides convenient wrappers around Java's XSL transformation framework that is part of the standard JAXP

[6] Although this is not a major concern in *SProUT* where type hierarchies are (up to now) by far smaller than in HPSG grammars.

[7] Class `de.dfki.lt.sprout.runtime.MatchInfo`.

[8] Class `de.dfki.lt.sprout.runtime.XmlTransformer`.

API.

An extended example for an XSL stylesheet working on *SProUT*put will be presented in Section 9.5.4.1. There, the output of the standard named entity grammar of *SProUT* (for English, German, Greek and Japanese) is transformed into the robust semantics representation format RMRS. The transforming stylesheet is generated automatically and solely on the basis of the *TDL* type definitions for the named entity output types.

Similarly, any XML format different from the *SProUT*put DTD could be read by the *SProUT* interpreter as input after an appropriate XSLT translation. It is now also easy to see how new, external components could be integrated with the interpreter via the XML interface, e.g. for programming languages other than Java. Together with the generic feature structure format, *SProUT*put XML thus forms a highly generic and flexible, structured XML format for input to and output from the *SProUT* interpreter.

7.4 Compile Time Type Check

As stated above, welltypedness of the typed feature structures is important to ensure correct and efficient unification operations in the interpreter at runtime. It has to be ensured, e.g., that each feature occurring in the grammars is appropriate for the associated type, and that all types occurring a defined in the type hierarchy.

While the preprocessor for the *TDL* type hierarchy is *SProUT* has its own checking algorithm for the simple typed feature structures supported by the core *TDL* syntax, this algorithm is not applicable to the *XTDL* syntax which is a superset of *TDL*. Therefore, a new, appropriate type check had to be developed that checks *XTDL* definitions for compliance with the type hierarchy defined in *TDL*.

The type check constitutes the second stage in a three-stage preprocessing phase for grammar compilation. The first step is *XTDL* syntax parsing, then the type check and generation of an intermediate XML representation follows on which finally the finite state grammar compilation operates.

The first step has been implemented using JavaCC (Viswanadha and Sankar, 2002), a parser generator for LL(k) grammars. The *XTDL* syntax as defined in the BNF in Figure 7.2, but with refinements and elaborations not presented there for the sake of simplicity, is defined in a LL(1) grammar with additional contextual constraints formulated in attached action rules.

The action rules also generate an intermediate XML representation; its DTD (DTD Appendix on page 286) is called the *XTDL* DTD. We do not go into details here as the DTD is roughly isomorphic to the BNF structure, with additional attributes e.g. for access to character positions in the original *XTDL* source (received through the JavaCC-generated parser) in order to provide type check errors with a precise location in the grammar source, as exemplified in the *SProUT* IDE screenshot in Figure 7.7.

We roughly sketch the type checking algorithm that is executed while the inter-

mediate representation for finite state compilation is generated recursively during parsing the *XTDL* grammar sources. The type hierarchy compiled by the `flop` preprocessor is represented in JTFS and can be queried quickly via hash table access (type-code mappings) and bit vector operations (GLB computation, subsumption checking; cf. Aït-Kaci *et al.* 1989).

Type definition check at feature structure (CFS) nodes. The first check tests if a type is defined in the type hierarchy. String attribute values do not have to be present in the type hierarchy, but the value type must be compatible ('appropriate', cf. page 40) with the *TDL* type *string* in the type hierarchy. If a type is not in the type hierarchy, an error is signaled.

In case multiple types are specified at a feature structure (CFS) node (this is possible in the XTDL syntax), their corresponding GLB (greatest lower bound) type is looked up in the type hierarchy (including subtype relations). If it does not exist, an error is signaled (because of the closed type world assumption imposed by the type system). If it exists, the complex type expression is replaced by the single GLB type and a warning is generated for the grammar developer that a common subtype has been found and should be used instead in the rule.

Appropriateness and welltypedness checks. The concepts of appropriateness and welltypedness were already defined on page 40. The appropriateness check tests whether all features occurring in a typed feature structure are licensed by a *TDL* type definition (directly or through inheritance). A feature occurring without a licensing type will cause an error. Welltypedness implies appropriateness. The welltypedness check additionally tests whether the type of each attribute value is subsumed by either the feature-introducing type definition or any possible refinement following through inheritance.

The list of detected error or warnings is collected during parsing, and presented (at least up to the first syntax error) in such a way that the grammar writer can immediately correct them in the editor (either built-in IDE or through a generic editor interface that e.g. supports the text editor `emacs`) by simply clicking on the error message (Figure 7.7). The character position stored in the intermediate XML representation of the grammar allows to automatically point the cursor to the problematic location.

As a result, the implementation of the compile time type checking algorithm with exact error positioning has helped to drastically reduce errors during and after grammar development.

7.5 Visualization

We briefly discuss tools we have implemented for graphical rendering of XML-encoded typed feature structures. They have been implemented both as Swing

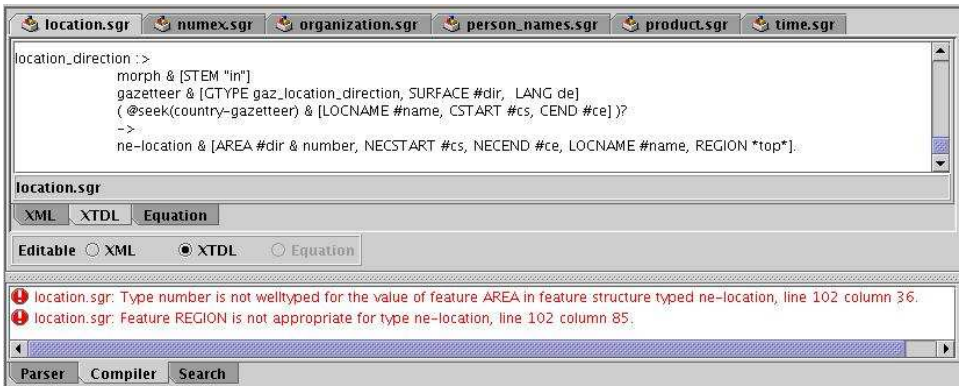


Figure 7.7: Type check results in the *SProUT* IDE

components for use in the integrated *SProUT* development environment and as Java applets for viewing in a Java-enabled browser.

The main motivation is to support convenient graphical representations in AVM notation (cf. page 36) of the complex, structured analysis results of the *SProUT* interpreter and the *XTDL* grammar rules that would otherwise (in text or XML representation) make reading of the nested and interlinked structures difficult.

The implementation takes advantage of the common typed feature structure subset that is part of both *XTDL* and *SProUT*put and the fact that both representation formats can be encoded in XML for output. A SAX parser generates the same (intermediate) object representation for the graphical elements from both XML input formats (*XTDL* DTD and *SProUT*put DTD), which is then used to draw (or update) Swing primitives. Through the XML format, the feature structure representation is decoupled from the *SProUT* runtime engine, and the visualization code is lightweight, which is e.g. important for the Java applet implementation.

An example of *SProUT*put and *XTDL* visualizations as part of the *SProUT* IDE is depicted in Figure 7.8.

A second way of visualization are the LaTeX AVMs and graphical *XTDL* rules we have seen in this chapter. These have been generated via an XSLT stylesheet that reads the *XTDL* DTD and *SProUT*put DTD and generates LaTeX code of it. Thanks to the modular DTD design, a single stylesheet can handle both DTDs, and generate the appropriate LaTeX source code.

7.6 Applications

IE systems are becoming commercially viable in supporting diverse information discovery and management tasks. The *SProUT* platform has been adopted as the core IE component in several EU-funded and industrial projects, supporting tasks

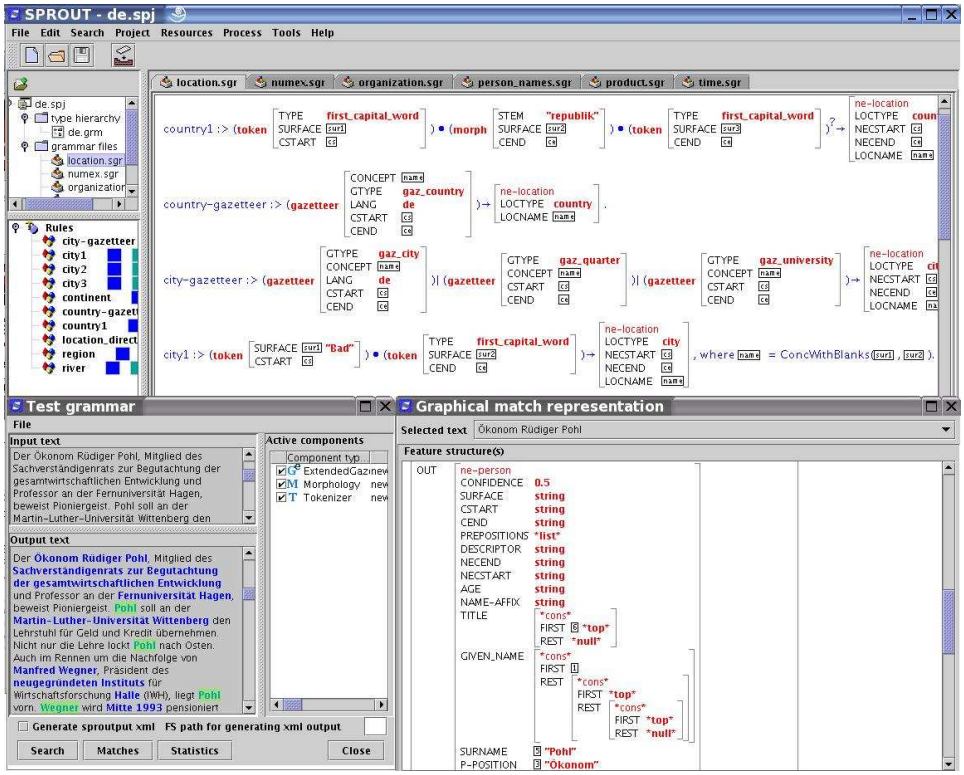


Figure 7.8: AVM rendering in the *SProUT* IDE: XTDL visualization in the top right panel, *SProUT*put in the bottom right window

such as content extraction and acquisition for text/data mining, dynamic hyperlinking, machine translation, and text summarization.

These applications yielded valuable feedback for further improvements and extensions of *SProUT*. Grammars are usually developed utilizing the integrated development environment with which *SProUT* is shipped out. Runtime systems, on the other side, make use of *SProUT*'s rich programming API.

The common core set of grammars used in the projects for named entity recognition comprises rules and gazetteer entries *etc* for person, organization and location names, currency and temporal expressions. While these named entity classes roughly are equivalent to the MUC named entity classes already mentioned in previous chapters, the output of recognized items is *structured* whereas in MUC, only the class of the named entity is represented in the annotation.

Below, we show the four named entities recognized by *SProUT* in the following two sentences, including its structured output.

Geschäftsführer Prof. Dr. hab. Peter S. van den Berg war seit Anfang

Juli 2004 bei der Söhnlein GmbH & Co. KG tätig. Er arbeitete vom 2. Juli 2004 bis 31. Mai 2005 dort.

ne-person	
SURFACE	"Geschäftsführer Prof. Dr. hab. Peter S. van den Berg"
CSTART	"0"
CEND	"51"
TITLE	⟨"Prof. Dr. hab."⟩
GIVEN_NAME	⟨"Peter","S."⟩
SURNAME	"van den Berg"
P-POSITION	"Geschäftsführer"
VARIANT	"van den Berg | Prof. Dr. hab. van den Berg | Geschäftsführer van den Berg"

span		
SURFACE	"seit Anfang Juli 2004"	
CSTART	"57"	
CEND	"77"	
SPAN_TYPE	*time_span*	
FROM	*point*	
	TEMPORAL-UNIT	*temporal-unit*
	SPEC	*temp-point*
	MUC-TYPE	*date*
	PREPOSITIONS	⟨*seit*⟩
	YEAR	*2004*
	MONTH	*07*
	PART-OF-MONTH	*begin_of*
TO	*point*	
	TEMPORAL-UNIT	*temporal-unit*
	SPEC	*temp-point*
	MUC-TYPE	*date*
	YEAR	*string*
	MONTH	*string*
	PART-OF-MONTH	*part-of-point-type*

ne-organization	
SURFACE	"Söhnlein GmbH & Co. KG"
CSTART	"87"
CEND	"108"
PREPOSITIONS	**list**
ORGTYPE	*org-type*
ORGNAME	1 "Söhnlein"
DESIGNATOR	*gmbh_&_co_kg*
VARIANT	1

$$
\begin{bmatrix}
\textit{span} & \\
\text{SURFACE} & \texttt{"vom 2. Juli 2004 bis 31. Mai 2005"} \\
\text{CSTART} & \texttt{"130"} \\
\text{CEND} & \texttt{"162"} \\
\text{SPAN\_TYPE} & \textit{time\_span} \\
\text{FROM} & \begin{bmatrix}
\textit{point} & \\
\text{TEMPORAL-UNIT} & \textit{temporal-unit} \\
\text{SPEC} & \textit{temp-point} \\
\text{MUC-TYPE} & \textit{date} \\
\text{PREPOSITIONS} & \langle \textit{von} \rangle \\
\text{YEAR} & \textit{2004} \\
\text{MONTH} & \textit{07} \\
\text{DOFM} & \textit{02}
\end{bmatrix} \\
\text{TO} & \begin{bmatrix}
\textit{point} & \\
\text{TEMPORAL-UNIT} & \textit{temporal-unit} \\
\text{SPEC} & \textit{temp-point} \\
\text{MUC-TYPE} & \textit{date} \\
\text{PREPOSITIONS} & \langle \textit{bis} \rangle \\
\text{YEAR} & \textit{2005} \\
\text{MONTH} & \textit{05} \\
\text{DOFM} & \textit{31}
\end{bmatrix}
\end{bmatrix}
$$

Some projects using *SProUT* extended or partly replaced the core named entity grammars by domain-specific rules and other resources. The modular design of the grammars that is facilitated through the powerful type system eased fulfilling this requirement.

Following is an incomplete list of applications developed in research project context at DFKI. There are several further applications not mentioned here that have been developed by other, external research groups and companies using the *SProUT* system.

Integrating information extraction and automatic hyperlinking. EXTRALINK (Busemann *et al.*, 2003) is a novel information system combing IE technology and automatic hyperlinking. Semantic concepts identified by the *SProUT* named entity grammars are mapped onto a domain ontology that relates concepts to a selection of hyperlinks, which are directly visualized using a standard web browser. EXTRALINK showcases the extraction of relevant concepts from German texts in the tourism domain, offering the direct connection to associated web documents on demand.

Multilingual information extraction for AIR FOReCast in Europe. The EU-funded project AIRFORCE targets at developing ideas and components which

support building a database of European events and trends, helping to forecast air traffic (Busemann and Krieger, 2004). *SProUT* has been adopted for building up domain-specific named entity and relation extraction grammars with language-neutral output for automatically extracting relations from official travel warnings, published regularly in the Internet by the ministries for foreign affairs of France, Germany and the UK.

Multilingual IE for machine translation and text summarization. The EU-funded MEMPHIS project (Kasper *et al.*, 2004) has developed a platform for cross-lingual premium content services, targeting mainly at portable thin clients such as mobile phones, PDAs, *etc*. The core of the system consists of a transformation layer, integrating cross-lingual information extraction and summarization of source documents, translation to the customers' target languages, a domain ontology-based knowledge management for extracted information as well as multilingual generation of documents according to the requirements of the target devices. *SProUT* is used primarily as a document indexing engine for tokenization, morphological analysis, and named entity recognition, and secondly for boosting the performance of text summarization and machine translation components.

Information extraction for Polish in the financial and medical domain. An attempt in applying *SProUT* in the process of constructing an Information Extraction engine for Polish and adopting it to the processing of Slavic languages is reported in Piskorski *et al.* (2004). The IE tasks focus on the identification of typical named entities from financial texts (Piskorski, 2005) and on extraction of data about pathological changes from a medical corpus containing descriptions of mammographic examinations (Kupść *et al.*, 2004).

Opinion mining. The ARGOSERVER system, developed by the Italian company Celi, analyzes on a daily basis forums and newsgroups on different car manufacturers in order to retrieve interesting messages and trends. *SProUT* is applied here to handle the information extraction task. The extracted opinions are input to statistical post-processing, yielding, e.g. the total number of comments (or attitudes) expressed by the forum/newsgroup users in the monitored period.

Hybrid deep and shallow methods for knowledge-intensive information extraction. In the DEEPTHOUGHT project, English, German and Japanese named entity recognition of *SProUT* is employed in the hybrid architecture integrating deep and shallow natural language processing components we will describe in Chapter 9. Prototype application domains are precise information extraction for business intelligence, e-mail response management for customer relationship management, and creativity support for document production and collective brainstorming (Uszkoreit *et al.*, 2004).

Here, the *SProUT* XML output is converted to the semantic formalism RMRS (robust minimal recursion semantics) and to the deep parser's input chart through XSLT stylesheets (Schäfer, 2004b). The application will be described in more detail in Section 9.10.1.

Question answering with hybrid deep and shallow processing. QUETAL HoG-QA is a system for domain-restricted question answering from structured knowledge sources (in contrast to unstructured text sources), based on robust semantic analysis in a hybrid NLP system building on the Heart of Gold architecture we will describe in Chapter 9. Question interpretation and answer extraction is performed in an architecture that builds on a lexical-semantic conceptual structure for question interpretation, and is interfaced with domain-specific concepts and properties in a structured knowledge base.

SProUT is employed here for German and English both as a pre-processing component for robust deep parsing (named entity recognition) and a richer semantics representation construction in an information extraction-like fashion of domain-specific terms and named entities (Frank *et al.*, 2006). Details will be presented in Section 9.10.2.

Customer care question answering. The CCA (Customer Care Automation) project centers around a question answering system with dialog facilities for the telecommunication domain developed for the leading German mobile phone provider (Burkhardt *et al.*, 2005).

SProUT is deployed in the component for shallow semantic analysis of the questions (named entity and NP/PP grammars for German) and for indexing the question-answer database.

Information extraction in SmartWeb. In the large German Semantic Web research project SmartWeb, *SProUT* is used for information extraction of soccer match descriptions that are written in German (Buitelaar *et al.*, 2006). Match situations are output by the *SProUT* grammar in frame-like feature structures capturing information gathered over several sentences. The grammar is a modular extension of the existing general named entity recognition rules for person names, location, time expressions *etc*, and heavily makes use of the extended gazetteer facility and the morphology component. An event recognition rule from the developed grammar is shown in Figure 7.9, cf. also the SmartWeb description in Section 9.11

7.7 Evaluation

As can be seen from the examples above, *SProUT* is a very versatile NLP processor that has been successfully deployed for many different tasks in novel challenging research and industrial applications. To demonstrate the competitiveness of the system, we report on an evaluation of (a 2005 snapshot of) the multilingual named

```
scoregoal :> syn_args & [HEAD syn_verb & [SYN_STEM goalscore & #ds,
                                          SYN_CSTART #cs,
                                          SYN_CEND #ce],
                         ARGS #args]
-> s_playeraction & [SPORTACTIONDESCR #ds,
                     SPORTACTIONTYPE scoregoal,
                     COMMITTEDBY #player_fs,
                     NECSTART #cs,
                     NECEND #ce],
where #act_subj = InList(act_subj, #args),
      #nelist = FeatVal("NE_LIST", #act_subj),
      #player_fs = FeatVal("NE_FS", #nelist).
```

Figure 7.9: An event recognition rule from the SmartWeb soccer grammar

entity recognition *SProUT* grammars, because evaluation corpora exist for this task.

7.7.1 Evaluation Snapshot of the Multilingual NE Grammars

Many of the above presented applications build on named entity recognition grammars that have been developed since the first version of *SProUT* was available. A large portion of the different project or domain-specific grammars is shared, also cross-lingually. In a modular way, the core grammar is extended with domain-specific rules depending on project-specific requirements. The soccer grammar of SmartWeb, e.g., builds on the common person name recognition rules, but adds further rules for player roles, match situations, *etc*. Specific gazetteer tables are added for names, roles and domain-specific extensions.

To support true multilinguality, an effort was started to standardize the output format of the named entity grammars for the different languages. Although the grammars partly have different internal structure and organization[9], they shall provide the same output structures for comparable phenomena in the different languages. As the *SProUT* formalism does not impose an a-priori structure (types, names, attributes, granularity *etc* can in general be chosen freely), this had to be designed in an iterative process that included feedback from the projects and subsequent amendments.

On the other hand, the existing *de facto* annotation standard MUC-6 for named entity recognition (Grishman and Sundheim, 1996) has already early been considered insufficient as it foresees much less fine-grained and less structured analyses than are possible with *SProUT*. While MUC-6 e.g. only distinguishes span and named entity type information, *SProUT* grammars output more detailed analyses

[9] Either motivated by the structure of the natural language or by the grammar development history.

of recognized names such as the internal structure of a person name (title, surname, given name *etc*), the granularity of locations (city vs. region, country names *etc*), or normalization of metric data such as time and date.

However, in order to provide a means for comparison with existing, competing tools, a somewhat impoverishing mapping had to be defined that (in the case of MUC-6 annotation for English) even included the elimination of *SProUT* rules for phenomena that MUC-6 does not cover.

The grammars were evaluated with the JTaCo evaluation tool (cf. Section 7.8.4; Bering *et al.* 2003; Bering 2004) which supports user-defined mappings between different NE classes, for controlled partial overlap between recognized and annotated NEs, and supports user-defined mappings between text-based and semantics-based annotations and output structures. Table 7.1 and 7.2 contain the evaluation results in terms of precision-recall figures of the core NE grammar for English and German, respectively. For the evaluation of the English grammar, an excerpt of circa 1MByte from the MUC-6 annotated corpus (newspaper texts) was used, whereas for German, a manually annotated corpus of German newspaper texts with a slightly different annotation scheme has been created.

Type	#entities	precision	recall	f-measure
NUMEX-PERCENT	34	1.0	1.0	1.0
NUMEX-MONEY	103	0.971	1.0	0.986
ENAMEX-LOCATION	1398	0.959	0.987	0.973
ENAMEX-ORGANIZATION	1747	0.949	0.911	0.930
ENAMEX-PERSON	1123	0.917	0.950	0.933
TIMEX-DATE	854	0.963	0.943	0.953
TIMEX-TIME	162	1.0	0.926	0.961

Table 7.1: English named entity grammar evaluation

Type	#entities	precision	recall	f-measure
NUMEX-PERCENT	280	1.0	0.989	0.995
NUMEX-MONEY	679	1.0	0.969	0.984
NUMEX-QUANTITY	16	0.941	1.0	0.969
ENAMEX-LOCATION	784	0.982	0.989	0.985
ENAMEX-ORGANIZATION	1017	0.974	0.982	0.978
ENAMEX-PERSON	58	0.950	0.983	0.966
TIMEX-TIME	530	1.0	0.926	0.961
NUMEX-NUMBER	227	0.904	0.991	0.945

Table 7.2: German named entity grammar evaluation

As can be seen from the results, the *SProUT* grammars perform quite well compared to typical statistical systems. Recall will presumably lower on domains for which no appropriate gazetteers are contained, e.g. in the chemistry domain.

However, the strengths of the general *SProUT* recognition approach, namely (1) flexible adaptability to new domains, (2) concise, declarative formalism (3) clear-cut specialized and compilable resources, (4) modularity of grammars and architecture, (5) structured and fine-grained analysis output, are largely neglected by the bare figures.

Thus, *SProUT* is a suitable tool for high-precision and structured domain-specific named entity recognition tasks – not to forget the fact that the system could be used for other purposes than just named entity recognition. An example will be presented in Chapter 9 where several cascaded *SProUT* grammars are, in addition to named entity recognition, also used to compute a shallow semantic representation from both *SProUT* morphological analysis and a statistical chunker.

7.8 Building, Testing and Evaluation with *SProUT*omat

In this section, we present *SProUT*omat, an automatic build, testing and evaluation tool we have developed (Schäfer and Beck, 2006; Bering and Schäfer, 2006) for *SProUT* as a means for periodical automatic testing and evaluation, mainly for immediate feedback after grammar or (re)source changes, but also to test integrity and performance of the overall system including program code.

7.8.1 Motivation

The development of multilingual resources for language technology (LT) components is a tedious and error-prone task. Resources for a complex, multi-purpose, multilingual system like *SProUT*, such as tokenizers, morphologies, lexica, grammars, gazetteers *etc* for multiple languages can only be developed in a distributed manner, i.e., many people work on different resources.

However, the resulting systems are supposed to deliver the same good recognition quality for each language. Dependencies of resources and subsystems may lead to suboptimal functioning, e.g. reduced recognition rates, of the overall systems in case of errors creeping in during the development process. Hence, in analogy to software engineering, testing and evaluation of the developed resources has to be performed on a regular basis, both for quality assurance (QA) and comparability of results in different languages.

7.8.2 *SProUT*omat

*SProUT*omat is a tool for daily automatic building, testing the *SProUT* development and runtime system from the Java source code and for compiling, testing and evaluating linguistic resources for English, German, French, Spanish, Greek, Japanese, Italian and Chinese named entity and information extraction grammars from a version control system. Although parts of the build mechanism are specific for *SProUT*, most of the testing and evaluation parts could be re-used for other systems.

7.8.3 Building and Testing

*SProUT*omat is an extension of the build mechanism for language technology components and resources we have developed for the *SProUT* system using Apache Ant[10]. Ant is a standard open-source tool for automatic building and packaging complex software systems. On the basis of target descriptions in an XML configuration file, Ant automatically resolves a target dependency graph and executes only the necessary targets.

Before testing and evaluating, a system has to be built, i.e., compiled from the sources checked out from a source control system. The Java program code compilation of *SProUT* is a straightforward task best supported by Ant. The case is, however, different for lingware sources all of which are compiled in *SProUT* as well for efficiency reasons.

While the appropriate Java code compilation tasks know what a compiled class file is and when it has to be recompiled (source code changes, dependencies), this has to be defined explicitly for lingware resources which Ant natively is not aware of. The `uptodate` task of Ant (a predicate) can be used to compare source files (.tdl in the following example) against their compiled version (.grm).

```
<uptodate property="tdl_input_is_uptodate"
          srcfile="${typehierarchy}.tdl"
       targetfile="${typehierarchy}.grm"/>
```

For each of the different lingware types, these source file dependencies are defined as are the calls to the dedicated *SProUT* compilers and parameters for their compilation.

Lingware-specific targets have common parameters and properties such as `lang`, `project` and the lingware type that are used to locate e.g. the source and compiled files in the hierarchically defined directory trees, or `charset` to specify encodings for source files to read.

Dependencies between different lingware types are handled by calls to defined sub-targets. Figure 7.10 shows the definition of the `compile_ne` target that calls four other compilation sub-targets. Each sub-target compiles only when necessary, and the `compile_ne` target itself depends on the `jar` target that provides working and up-to-date *SProUT* lingware compilers.

Besides the program and lingware compilation, many other targets exist e.g. to generate documentation, package runtime systems, start the integrated development environment (IDE) *etc*.

Thus, using a single command, it is possible to compile the whole system including code and all dependent available linguistic resources, or to update it after changes in the sources.

The daily automatic testing and evaluation mechanism is an extension of the build procedure. *SProUT*omat first updates all program sources and linguistic resources from the version control system, and compiles them. For each language

[10] `http://ant.apache.org`

```
<!--usage : ant compile_ne -Dlang=en -->
<target name="compile_ne" depends="jar"
        description="Compile NER grammar.">
  <property name="lang" value="en"/>
  <property name="project" value=""/>
  <property name="charset" value="utf-8"/>
    <!-- compile type hierarchy: -->
  <antcall target="compile_tdl"/>
    <!-- compile tokenizer: -->
  <antcall target="compile_tokenclass"/>
    <!-- compile gazetteer: -->
  <antcall target="compile_gazetteer"/>
    <!-- compile XTDL grammar for NER: -->
  <antcall target="compile_grammar"/>
</target>
```

Figure 7.10: A sample target definition: named entity grammar compilation

resource to test, a reference text is then analyzed by the *SProUT* runtime system. This checks for consistent (re)sources. The next step is comparison of the generated named entity and information extraction annotation against a gold standard.

7.8.4 Evaluation with JTaCo

*SProUT*omat uses a batch version of JTaCo (Bering *et al.*, 2003; Bering, 2004) for the automatic evaluation and computation of precision, recall and f-measure. For English, the annotated corpus is e.g. taken from the MUC evaluation data (Grishman and Sundheim, 1996). For other languages for which no MUC annotations exist (e.g. German), a manually developed corpus is employed. JTaCo can be easily customized for comparison with other XML annotation formats.

JTaCo provides unified use of variably annotated source material for testing. The component developer provides suitably, i.e., usually semi-manually or manually marked-up reference sources on the one hand, and a parser or similar NLP component (here *SProUT*) on the other hand. JTaCo extracts the original annotation from the corpus, compares this annotation with the markup the component in question generates for the same input, and generates statistics and reports from the comparison results[11].

Since a focus of JTaCo lies on the integration of diverse manual annotation schemes on the one hand and differing NLP components on the other, JTaCo employs a modular architecture in which its different processing stages allow independent adaptations to varying input and different environments. JTaCo is realized as

[11] JTaCo stands for Java Tagging Comparator.

a pluggable, lightweight, mostly architecture-independent framework. Currently, there are two JTaCo plug-in realizations for usage with grammars developed in *SProUT*: A GUI plug-in integrated in the *SProUT* IDE, and a batch version integrated in *SProUT*omat.

7.8.4.1 JTaCo's Processing Stages

JTaCo works in four separate transformational processing stages. Figure 7.11 illustrates these stages, their input and the results they generate. The process starts from an annotated written corpus against which the NLP component or resource is to be tested. In the first step, JTaCo uses an *AnnotationParser* to separate the corpus into

- the *raw* text contained in the corpus (i.e., the text without any annotation) and
- its *true* annotation (interchangeably also called the *reference* or *manual* annotation).

The extracted text is fed into the *Parser* or a similar component to test, in this case the *SProUT* interpreter, yielding the annotation to compare with the manual annotation. The comparison is executed by a *TaggingComparator*. The comparator's result in turn is used by an *OutputGenerator* to select, format and output the needed information.

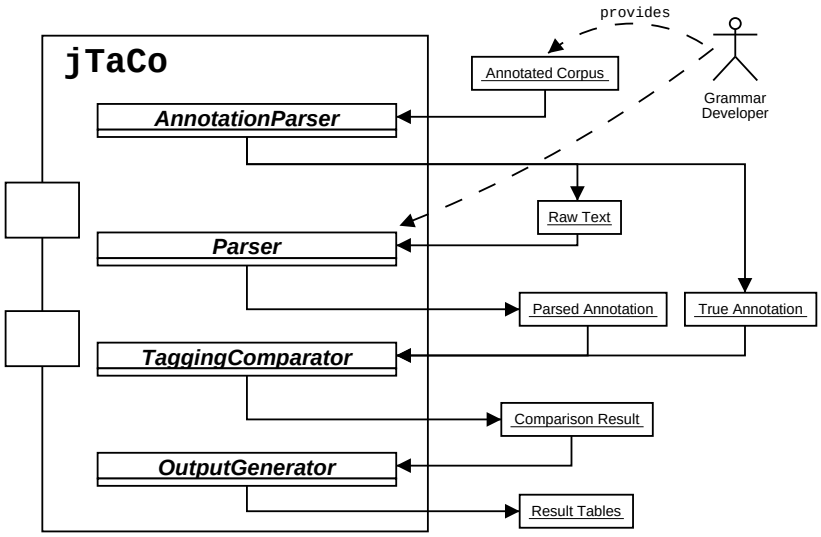


Figure 7.11: An overview of JTaCo's processing stages

7.8.4.2 Reading the Annotated Corpus

For use at the following processing stages, JTaCo extracts from the annotated corpus the 'raw' content, i.e., the written text without any markup, on the one hand, and the reference annotation on the other. Both the extraction of the text and of the annotation can be configured according to the specific annotation scheme. A corpus usually not only contains the annotated textual material, but also meta-information intended for, e.g. administrative purposes. Such information has to be excluded from the text extracted to be used for testing. JTaCo includes support for annotations which satisfy certain regular constraints and for XML annotations such as found in MUC corpora (Grishman and Sundheim, 1996). For use with *SProUT*, JTaCo transforms the XML-encoded entities into typed feature structures.

As an illustration, consider the following MUC time expression:

```
<TIMEX TYPE="DATE">07-21-96</TIMEX>
```

The textual content consists just of the date expression *07-21-96*. JTaCo transforms the tag information as well as the surface and character offsets into feature-value pairs in a feature structure:

$$\begin{bmatrix} \textit{timex} & \\ \text{TYPE} & \texttt{"DATE"} \\ \text{CSTART} & \texttt{"27"} \\ \text{CEND} & \texttt{"34"} \\ \text{SURFACE} & \texttt{"07-21-96"} \end{bmatrix}$$

Here, CSTART and CEND indicate the inclusive start and end character positions of the annotated element in the 'raw' text, i.e., without counting the markup. The resulting reference annotation is the collection of all feature structures generated from the corpus. More complex, embedded annotations would be translated in a similar manner.

7.8.4.3 Parsing the Extracted Text

At this second processing stage, JTaCo feeds *SProUT* with the text retrieved from the previous stage, and *SProUT* in turn produces some specific markup of the text. As at the previous stage, JTaCo transforms this annotation into a format which it can compare with the reference annotation.

For the previously employed example expression, *07-21-96*, *SProUT*'s named entity recognition markup delivers structured output in an XML-encoded typed feature structure ('*SProUT*put' DTD), where CSTART and CEND indicate start and end character positions of the matched named entity in the input text:

$$\begin{bmatrix} \textit{point} & \\ \text{SPEC} & \textit{temp-point} \\ \text{MUC-TYPE} & \textit{date} \\ \text{CSTART} & \texttt{"27"} \\ \text{CEND} & \texttt{"34"} \\ \text{SURFACE} & \texttt{"07-21-96"} \\ \text{YEAR} & \texttt{"1996"} \\ \text{MONTH} & \texttt{"07"} \\ \text{DOFM} & \texttt{"21"} \end{bmatrix}$$

7.8.4.4 Comparing the Annotations

At this stage, the annotations obtained from the two previous transformation processes are compared, i.e., the 'manual' annotation read directly from the corpus, and the 'parsed' annotation obtained through the NLP component. For JTaCo, an annotation is a collection of tags, where a tag consists of some linguistic information about a piece of text. Minimally, a tag contains

- some name, e.g. a linguistic label,
- the surface string to which the label applies,
- position information about where this string is found in the corpus.

Usually, the setup uses tags which incorporate more information, and the relation used to determine entity equality between the two annotations typically depends on this information. For use with *SProUT*, JTaCo generates an annotation consisting of tags which are augmented with feature structure information. The equality notion of these tags is defined though unification.

An important feature of JTaCo is that the comparison can be configured to accommodate for a variety of systematic differences in annotations:

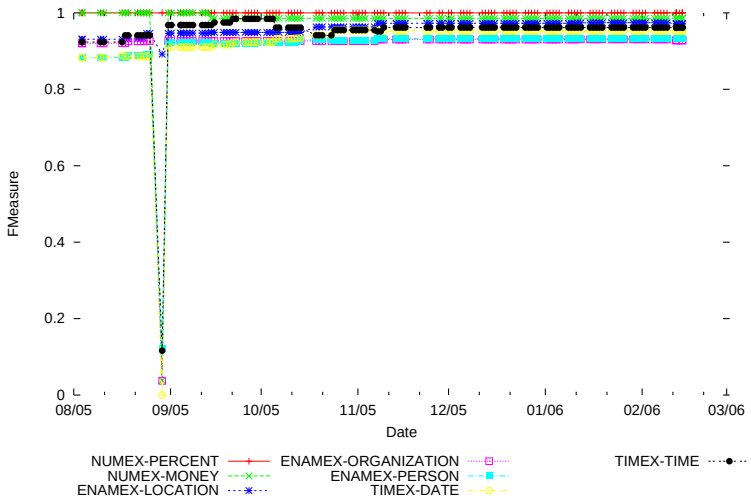
- The annotations may use different labels, differing perhaps even in granularity. One annotation, e.g., might globally use the label *organization*, while the other uses subclasses such as *university*, *government*, *etc*.
- The annotated entities may differ in their surface spans. One annotation, e.g., might consider the expression *President Hugo Chavez* to be a named entity, while the other might exclude the title.
- One annotation may contain sequences of entities which in the other annotation correspond to one single entity. For instance, MUC will usually separate a date followed by a time into two named entities (`TIMEX-DATE` and `TIMEX-TIME`), while *SProUT* considers this to be one entity.

From: SProUTomat
Date: 20/02/2006 06:35 AM
Subject: SProUTomat -- Status: OK

SProU 6

CVS update...
U src/grammar/extendedgazetteer/common/location.gaz
U src/grammar/xtdl/ne/en/location.sgr

Building runtime system and grammars... log
Testing English grammar... log result
Running JTaCo... log
JTaCo result English



Generating runtime system Javadoc
Generating Antdoc for build.xml
Generating Ant call graph diagram

Start: 20.02.2006 06:22:01
End: 20.02.2006 06:35:38

Figure 7.12: A report generated by *SProUT*omat (excerpt for a single language)

7.8.5 Report

After counting and comparing matches through JTaCo, a report is generated and emailed to the developers with an overall status (OK or ERROR) for quick infor-

mation. The report (example in Figure 7.12) also contains diagrams consisting of precision/recall/f-measure curves since beginning of regular measurements per language generated with `gnuplot`[12] that graphically present an overview of the resource development progress over time. To this end, the evaluation figures are also added to a global evaluation database.

Further information sources such as Ant target and Javadoc documentation as well as a visual dependency graph representation of the Ant targets are also generated automatically.

7.8.6 Summary and Outlook

We have presented a comprehensive tool for automatically testing and evaluating linguistic resources and language technology components. The system is in daily use since March 2005 and successfully helps to maintain the quality and reliability of the multilingual language processor with its various resources that are developed by many authors and used in several projects.

The tool greatly helps to improve and accelerate the development - evaluation/comparison - refinement - cycle (cf. Figure 4.2) and gives motivating feedback (such as raising recall and precision curves over time). Although daily testing has been described above, the testing and report generation could be started at any time. A complete build from scratch, testing of four languages including Javadoc and generation of the runtime system plug-in into the Heart of Gold platform for deep-shallow integration (cf. Chapter 9) *etc* takes less than 14 minutes, while only a few seconds are required after modification of a single resource.

7.9 *SProUT* Summary and Relation to Deep Processing

In this chapter, we have presented our contributions to *SProUT*, a new declarative formalism and system that combines finite-state with constraint-based processing. The advantages we see in the new approach are the (1) openness of the formalism (in contrast to specialized shallow systems), (2) potentially rich, structured output, (3) a highly declarative formalism with good trade-off between expressivity and efficiency, (4) based on two simple, well-known basic concepts, namely typed feature structures and regular expressions.

The usefulness of the formalism and the overall system as well as re-usability of developed resources has been thoroughly and successfully demonstrated in various large projects, both scientific and industrial.

The structure paradigm of *SProUT* grammars (so far best visible in the named entity grammars) shows an important and useful relation between the recognition (or matching) part of a rule and its output. The structure of the matched LHS part

[12] `http://www.gnuplot.info`

can not only be used (1) to recognize and disambiguate items[13] using the powerful language of typed feature structures with unifiability tests, but also to (2) transport the information that is encoded in rules to the RHS output structure in such a way that knowledge about structure implicitly encoded in the rules is preserved in the output. This is a big advantage over classical shallow systems that often can only determine the type and character span of a NE, but not its internal structure.

SProUT has advantageous properties (not only) for integration in hybrid deep-shallow architectures. The primary use of and original intention for developing *SProUT* has been shallow processing, e.g. for named entity recognition and information extraction. As such, *SProUT* can be used for preprocessing words unknown to the lexicon of a deep parser in a sequential deep-shallow architecture. However, *SProUT* grammars can also be cascaded to produce themselves 'deeper' analyses.

In addition, *SProUT* shares with HPSG grammars the general concept of type hierarchies and typed feature structures (and uses the same type hierarchy preprocessor as the deep HPSG parser PET). This makes sharing of (linguistic) knowledge among deep and shallow grammars natural and easy. Moreover, *SProUT*'s information extraction-like structured output can form an important supplement to an HPSG parser's analysis.

A tight integration of *SProUT* with HPSG, an extension to incorporate ontology concept and structured instance information in *SProUT* lingware resources as well as a 'deeper' application of cascaded *SProUT* grammars interleaved with XSLT stylesheets will be demonstrated in Chapter 9.

[13] A preposition such as *in*, e.g., can be used to distinguish the city name Paris from a person name Paris. It is possible to use such context information for the match, but omit it in the output.

Chapter 8

WHITEBOARD

8.1 Introduction and Motivation

The aim of the WHITEBOARD project was to develop an architecture for the integration of deep and shallow natural language processing components and to investigate the benefits that could result from interleaved deep-shallow processing, with a strong focus on integration of German HPSG parsing with shallow preprocessing. The targeted application scenarios comprised (controlled) language checking and domain-specific information extraction.

To this aim, the coverage of the HPSG grammar for German developed at DFKI (main development in the VERBMOBIL project for speech dialogs) on a German corpus of newspaper texts had to be increased through deep-shallow integration.

A further research goal was to answer the question whether shallow preprocessing would be able to reduce the search space of the deep parser by reducing lexical and structural ambiguity.

Finally, the benefits of integrating the full range of shallow processing component types (in combination or separately), e.g. part-of-speech taggers, chunkers, named entity recognition, shallow sentence parsers, were investigated.

The closely interleaved processing model could be seen as an extension of the VERBMOBIL[1] architecture (Wahlster, 2000), where deep and shallow components ran concurrently instead of exploiting synergy.

One of the new key architecture ideas, already formulated by Hans Uszkoreit in the project proposal (Bredenkamp *et al.*, 1999), was to use multi-layered XML annotation to store and retrieve multiple NLP component results during processing.

As in the other two chapters of the architecture trilogy, the overall integration work was a collaborative effort involving many people, and we will focus here again on the software architecture and XML-based component integration, leaving out e.g. many linguistic and other implementation details.

[1] WHITEBOARD, conducted 2000–2002 at DFKI, started immediately after VERBMOBIL had finished. One main difference, however, is that VERBMOBIL had speech in focus, while WHITEBOARD concentrates on written documents.

8.2 The WHITEBOARD Architecture

The WHITEBOARD architecture aims at integrating different NLP components by enriching an input document through *annotations*. XML is used as a uniform means of representing online and keeping persistently (offline) the results of the various processing components.

Because not all interesting linguistic information can be directly represented within the basic XML tree structure, e.g. linguistic phenomena such as coreferences, ambiguous readings, and discontinuous constituents, the WHITEBOARD architecture employs a distributed multi-level representation of different annotations. Instead of translating all markup into one format in a single XML document, they are stored in different standoff annotation layers.

The challenge for the deep-shallow integration architecture was to combine the different NLP data representation models that characterize deep and shallow NLP analysis results. As already described in Section 6.2, they are e.g. defined in Cunningham *et al.* (1997) as a trichotomy. While a single shallow component markup contains *additive* information, the combination of multiple (shallow) markup strata is typically established by references or links (standoff markup) which can be characterized as *positional* representation.

The advantages and benefits of the standoff annotation model have already been motivated in Chapter 5 in detail. Linking via XML ID attributes and 'span' information together supports efficient access between layers. In contrast, deep analysis results encoded in typed feature structures representing NLP entities sentence-wise in a uniform, linguistically motivated form, constitute *abstraction-based* representations.

The WHITEBOARD architecture solves the multiple representation problem by providing a multi-level chart that serves as a container for all three kinds of representation. The multi-level chart is managed by and can be accessed via the WHITEBOARD Annotation Machine (WHAM) we will describe in the next section.

The initial integration scenario (WHITEBOARD I, Section 8.4) comprised the HPSG parser PET (Callmeier, 2000) with a VERBMOBIL-derived broad-coverage grammar for German (Müller and Kasper, 2000) as the deep component, and the shallow (mostly finite-state/rule-based) component SPPC (Piskorski and Neumann, 2000) for part-of-speech tagging, sentence boundary recognition, morphological analysis of words unknown to the HPSG lexicon and named entity recognition for German.

In the second phase (WHITEBOARD II, Section 8.7), larger sub-sentential constructions have been integrated through shallow chunking and topological parsing.

8.3 The WHITEBOARD Annotation Machine (WHAM)

The WHITEBOARD Annotation Machine (WHAM) is the core engine that provides the necessary integration facilities as depicted in Figure 8.1. For NLP-based appli-

cations (top), WHAM provides (1) an interface that accepts input text documents to be analyzed and (2) methods to access the computed analyses. On the NLP component side (left), WHAM supports interfaces to integrate software components producing XML-encoded NLP markup.

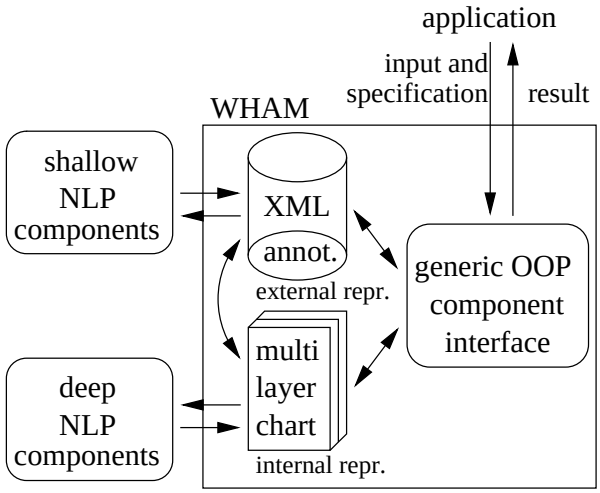


Figure 8.1: The WHITEBOARD Annotation Machine (WHAM)

WHAM also manages the multi-level chart. For efficiency reasons, the internal online storage (during processing) of shallow analyses uses a compact binary encoding with index-sequential access methods. Deep analyses are kept in their typed feature structure format, while persistent, external storage optionally uses an XML file format (TFS DTD motivated and described in Section 5.4.1).

An example of a multi-level representation of shallow results is presented in Figure 8.2. Each annotation level contains type (label) information for a text span.

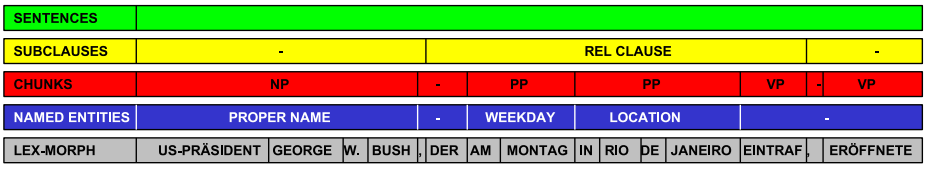


Figure 8.2: Index-sequential annotation structures

It may be useful to externally store and re-use the shallow XML representation as a kind of annotated corpus, e.g. for further processing, manual correction for machine learning *etc*. In contrast, storage of the full resulting feature structures of deep processing components may not be wanted for online processing because of their huge size (TFS DTD as discussed above; the resulting feature structures also contain a 'record' of the structure-building unification operations during parsing as explained in Chapter 3).

Instead, only interesting extracted information from deep processing such as phrase structure, subcategorization frames, semantic representation *etc* can be transformed into and stored as XML. Vice versa, it is possible to generate the index-sequentially stored shallow chart representation from the (offline) XML format in order to provide fast access to large annotated corpora.

The WHAM interface operations (access to shallow results *etc*) are not only used to interface NLP component-based applications, but also for the integration of deep and shallow processing components itself, i.e., the shallow part of the online multi-level chart is also actively queried by the deep parser via the access methods of WHAM.

Both applications and the integrated components access the WHAM results through a generic object-oriented programming (OOP) interface which is designed as general as possible in order to abstract from component-specific details, while preserving shallow and deep paradigms. The interfaces of the actually integrated components form subclasses of the generic interface. New components can be integrated by implementing this interface and by specifying transformation rules for the chart.

The OOP interface provides iterators that support walking through the different annotation levels (e.g. token spans, sentences, cf. Figure 8.3), reference and seek operators that allow to switch to corresponding annotations on a different level (e.g. return all tokens of the current sentence, or move to next named entity starting from a given token position), and accessor methods that return the linguistic information contained in the chart, e.g. the type of a phrase or of a named entity.

Similarly, general methods support navigating the type system and feature structures of the DNLP components, e.g. by returning the type value under a feature path or the subtype or supertype of a given type. The resulting output of the WHAM can be accessed via the OOP interface or as XML representation.

```
S = new SentenceIterator(SPPCAnalyzer, Document);
while (S.valid()) {
  NE = new NamedEntityIterator(S);
  while (NE.valid()) {
    print NE.getText() + NE.getType();
    NE.next();
  }
  S.next();
}
```

Figure 8.3: Iterator-based programming interface to annotation layers

8.4 WHITEBOARD I: Integrated components, results and applications

In this section, we describe the components that have been integrated in the first phase of WHITEBOARD (Crysmann *et al.*, 2002), a first evaluation and two initial information extraction applications that have been implemented on the basis of the architecture.

8.4.1 Components

8.4.1.1 Shallow Component: SPPC

Shallow preprocessing is performed by SPPC, a rule-based system which consists of a cascade of weighted finite-state components responsible for performing an analysis pipeline consisting of tokenization, lexico-morphological analysis, part-of-speech filtering, named entity recognition, sentence boundary detection, chunk and sub-clause recognition. SPPC is described in Piskorski and Neumann (2000); Neumann and Piskorski (2002).

We will briefly describe those components of SPPC which we integrated with the deep parser.

The SPPC *tokenizer* first segments words from punctuation symbols and returns a (compared to other tokenizers) relatively fine-grained token classification (52 different token classes), e.g.

```
<ITEM id="3" type="two_digit_number"/>
<ITEM id="4" type="four_digit_number"/>
<ITEM id="6" type="number_percent_compositum"/>
<ITEM id="7" type="decimal_number_with_period"/>
<ITEM id="8" type="number_dot_compositum"/>
<ITEM id="16" type="email_address"/>
<ITEM id="17" type="url_address"/>
<ITEM id="20" type="initial_capital_period"/>
<ITEM id="21" type="lowercase_word"/>
<ITEM id="22" type="first_capital_word"/>
<ITEM id="33" type="simple_word_dash_first_capital"/>
<ITEM id="48" type="abbreviation"/>
<ITEM id="50" type="word_followed_by_dots"/>
<ITEM id="51" type="end_of_paragraph"/>
```

Tokens identified as potential word forms are then *morphologically analyzed*. 420 different morphological types are distinguished, representable as feature-value pairs, e.g.

```
<ITEM id="14" gender="M" case="GEN" number="PL"/>
<ITEM id="32" person="2" case="NOM" number="SG"/>
<ITEM id="33" person="3" gender="M" case="NOM" number="SG"/>
<ITEM id="38" tense="PRES" person="3" number="SG"/>
```

```
<ITEM id="70" tense="SUBJUNCT-1" person="3" number="PL"/>
<ITEM id="72" form="INFIN"/>
<ITEM id="91" gender="M" case="GEN" number="SG" comp="P"
              det="INDEF"/>
```

Lexical information (list of valid readings including stem, part-of-speech and inflection information) is computed using a full-form lexicon of about 700000 entries that has been compiled out from a stem lexicon of about 120000 lemmata. After morphological processing, *PoS disambiguation* rules are applied which compute a preferred reading for each token (the deep parser, however, can also back off to all readings). The following 24 different PoS types are recognized by SPPC

```
<ITEM id="1" type="N"/>
<ITEM id="2" type="V"/>
<ITEM id="3" type="AUX"/>
<ITEM id="4" type="MODV"/>
<ITEM id="5" type="A"/>
<ITEM id="6" type="ATTR-A"/>
<ITEM id="7" type="DEF"/>
<ITEM id="8" type="INDEF"/>
<ITEM id="9" type="RELPRON"/>
<ITEM id="10" type="PERSPRON"/>
<ITEM id="11" type="REFPRON"/>
<ITEM id="12" type="POSSPRON"/>
<ITEM id="13" type="WHPRON"/>
<ITEM id="14" type="ORD"/>
<ITEM id="15" type="CARD"/>
<ITEM id="16" type="VPREF"/>
<ITEM id="17" type="ADV"/>
<ITEM id="18" type="WHADV"/>
<ITEM id="19" type="COORD"/>
<ITEM id="20" type="SUBORD"/>
<ITEM id="21" type="INTP"/>
<ITEM id="22" type="PART"/>
<ITEM id="23" type="PREP"/>
<ITEM id="24" type="STOP-WORD"/>
```

Named entity recognition is based on simple, string-based pattern matching techniques to recognize e.g. organizations, persons, locations, temporal expressions and quantities (13 NE types, 24 subtypes)

```
<ITEM id="1" type="date"/>
<ITEM id="2" type="organization"/>
<ITEM id="3" type="location"/>
<ITEM id="4" type="monetary"/>
<ITEM id="5" type="person"/>
<ITEM id="6" type="percentage"/>
<ITEM id="7" type="time"/>
<ITEM id="8" type="number"/>
```

```
<ITEM id="9" type="address"/>
<ITEM id="10" type="person_candidate"/>
<ITEM id="11" type="organization_candidate"/>
<ITEM id="12" type="location_candidate"/>
<ITEM id="13" type="position"/>
```

Next, NE-specific *reference resolution* is performed through the use of a dynamic lexicon which stores abbreviated variants of previously recognized named entities. Finally, the system *splits the text into sentences* by applying only few, but highly accurate contextual rules for filtering implausible punctuation signs. These rules benefit directly from NE recognition which already performs a restricted punctuation disambiguation.

The output of SPPC comes in XML format that is transformed by WHAM into the above described index-sequential format for fast random access through the WHAM shallow API.

8.4.1.2 Deep Component: PET

The HPSG parser integrated in the WHITEBOARD system is PET (Callmeier, 2000). Initially, PET was built to experiment with different techniques and strategies for processing unification-based grammars. The resulting system provides efficient implementations of the best known techniques for unification and parsing and is still the fastest parser for HPSG grammars.

While PET is basically a runtime parser for fast processing of HPSG grammars, the grammar source can be developed, tested and debugged with the LKB system (Copestake, 2002), that shares with PET a common TDL formalism (Krieger and Schäfer, 1994) subset and a compatible type hierarchy and typed feature structure model.

Being designed as an experimental system, the original PET parser lacked open interfaces for flexible integration with external components. For instance, in the beginning of the WHITEBOARD project, the system only accepted full-form lexica and plain text input.

Bernd Kiefer extended the system in collaboration with Ulrich Callmeier. Instead of single word input, input items where then allowed to be complex, overlapping and ambiguous, i.e., essentially word graphs. Dynamic creation of atomic type symbols, e.g. to be able to add arbitrary symbols as feature values, has been implemented as well.

Finally, a flexible interface has been implemented that uses API calls to WHAM for the integration of morphology, tokenization and named entity recognition analysis results. As WHAM is implemented in Java, and PET in C++, we defined this interface in JNI (Java Native Interface). Through the object-oriented WHAM API layer, PET could in principle also be integrated with other shallow systems than SPPC. We will discuss some shortcomings of the JNI-based API interface in Section 8.7 (WHITEBOARD II).

The German HPSG grammar in WHITEBOARD is based on a large-scale grammar by Müller (1999), which was further developed in the VERBMOBIL project for translation of spoken language (Müller and Kasper, 2000). It therefore covers many constructions that occur frequently in spontaneous speech. After VERBMOBIL, the grammar was adapted mainly by Berthold Crysmann to the requirements of the LKB/PET system (Copestake, 2002; Callmeier, 2000), and to written text, i.e., extended with constructions such as free relative clauses that were irrelevant in the VERBMOBIL scenario.

The grammar consists of a rich hierarchy of 5069 lexical and phrasal types. The core grammar contains 23 rule schemata, 7 special verb movement rules, and 17 domain specific rules. All rule schemata are unary or binary branching. The lexicon contains 38549 stem entries, from which more than 70% were semi-automatically acquired from the annotated NEGRA corpus (Skut *et al.*, 1998).

A further semi-automatic technique has been applied to acquire semantic types for nouns unknown to the deep lexicon using information available from GermaNet (Hamp and Feldweg, 1997). The approach is elaborated in Siegel *et al.* (2001). The semantic types are needed for (syntactic) disambiguation based on semantic information and thus help to reduce ambiguity and restrict search space for the parser.

8.4.2 Integration

Morphology and part-of-speech tagging The morphological analyses delivered by SPPC are mapped to the German HPSG morphology types. The mapping table has been generated by identifying the corresponding classes feature-wise. The actual mapping is then performed automatically in order to be able to easily track changes in both shallow and HPSG morphology geometry.

Shallow PoS tagging is used in two ways by the deep parser. First, HPSG lexicon entries that are marked as preferred by the shallow component (via their PoS value) are assigned a higher priority than the rest. Thus, the correct reading is more likely to be found early without excluding any reading. When no entry for a word is found in the HPSG lexicon, a default entry based on a generic HPSG lexicon type is automatically created based on the part-of-speech tag of only the reading marked as preferred by the shallow preprocessor. This strategy increases robustness, while preventing an increase in ambiguity.

Named entity recognition Similar to the unknown word strategy, named entities are mapped to generic lexical types that expand to feature structures during parsing. In other words, a simple mapping from shallow named entities to HPSG generic types is sufficient, filling the FORM feature value with the surface string from the recognized named entity.

In the following example the HPSG type *pn_type_person* is mapped from the shallow NE type `person untitled`.

SPPC output for the named entity:

```
<NE id="N0" type="person" subtype="untitled">
  <W id="W13" tc="first_capital_word">Martina</W>
  <W id="W14" tc="first_capital_word">Regel</W>
</NE>
```

The HPSG feature structure generated by the extended PET interface then looks as follows.

$$\begin{bmatrix} \textit{morph-type} \\ \text{FORM} \ \texttt{"Martina Regel"} \\ \text{STEM} \ \textit{pn\_type\_person} \\ \text{HEAD} \begin{bmatrix} \textit{nmorph-head} \\ \text{MAJ} \ \textit{noun} \\ \text{INFL} \begin{bmatrix} \textit{agr-type} \\ \text{AGR} \begin{bmatrix} \textit{nagr-feat} \\ \text{NUM} \ \textit{sg} \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

This simple strategy helps to drastically increase the coverage of the HPSG grammar on the large open class of named entities, as we will see in the first evaluation.

8.5 First Evaluation

An evaluation has been started using the NEGRA corpus, which contains about 20,000 newspaper sentences (Skut *et al.*, 1998). The main objectives were to evaluate the syntactic coverage of the German HPSG on German newspaper text and the benefits of integrating deep and shallow analysis. The sentences of the corpus were used in their original form without stripping, parenthesized insertions, *etc*.

The HPSG lexicon was extended semi-automatically from about 10000 to 35000 stems, which roughly corresponds to 350000 full forms. Then, the lexical coverage of the deep system on the whole corpus was checked, which resulted in 28.6% of the sentences being fully lexically analyzed. The corresponding experiment with the integrated system yielded an improved lexical coverage of 71.4%, due to the techniques described in Section 8.4.2. This increase is not achieved by manual lexicon extension, but only through synergy between the deep and shallow components.

To test the syntactic coverage, the subset of the corpus that was fully covered lexically (5878 sentences) was processed with deep analysis only. The results are shown in Table 8.1 in the second column. In order to evaluate the integrated system, 20568 sentences from the corpus were processed without further extension of the HPSG lexicon (see table 8.1, third column).

About 10% of the sentences that were successfully parsed by deep analysis only could not be parsed by the integrated system, and the number of analyses per

	Deep	Integrated
# sentences	20568	
avg. sentence length	16.83	
avg. lexical ambiguity	2.38	1.98
avg. # analyses	16.19	18.53
analyzed sentences	2569	4546
lexical coverage	28.6%	71.4%
overall coverage	12.5%	22.1%

Table 8.1: Evaluation of German HPSG in WHITEBOARD I

sentence dropped from 16.2% to 8.6%, which indicates a problem in the morphology interface of the integrated system at the time of this first evaluation.

The overall coverage increased from 12.5 to 22.1%. It has to be noted that, although the growth is impressing, the low absolute coverage is mainly due to the fact that the German HPSG at that time still was close to the VERBMOBIL grammar specialized in speech dialogs in the appointment negotiation domain, and not yet extended to general newspaper texts with long, complex sentences etc.

8.6 Applications on the Basis of WHAM

The WHAM has been used for two initial information extraction applications, the first one being shallow only, but testifying the speed of the index-sequential storage mechanism in an online template-based information extraction system using Google search for document retrieval. The second one was a feasibility study for using both deep and shallow analysis for improved precision and recall in restricted domains.

8.6.1 WAG – Mining Answers in German Web Pages

The fast index-sequential interface to XML annotation provided by WHAM can also be used for shallow-only analysis of large document collections. WHAM is e.g. used in the WAG system for online information extraction from websites (Neumann and Xu, 2003).

WAG is a study for a question answering system for German that takes factoid queries formulated as structured templates by letting the user fill in a form, and tries to find relevant answers in Web documents received from online Google searches based on keywords in the query template.

In the system, WHAM is used to extract named entities from the Web documents that are recognized by the shallow SPPC system. Web redundancy is exploited to compute weights on the named entities found. The ranked named entities

are then used for paragraph selection and answer identification.

The system has been evaluated for person and location questions taken from a German quiz book, e.g.

- Welches Pseudonym nahm Norma Jean Baker an? *(person)*
  Which pseudonym did Norma Jean Baker use?
- Wer wurde 1949 erster Ministerpräsident Israels? *(person)* Who became Israel's first prime minister in 1949?
- In welcher ehemaligen Sowjetrepublik befand sich das Kernkraftwerk Tschernobyl? *(location)*
  In which former soviet state was the nuclear power plant Chernobyl?
- In welcher europäischen Stadt nennt man die Altstadt Alfama? *(location)*
  In which European city is there an old city part called Alfama?

In cases where Google returned at least one answer (17 out of 20 person questions; maximally 50 documents per question), the system reached a recall score of 0.64 for the top 3 and 0.53 for the top 1 exact answers, for locations, and 0.43 (top 3) viz. 0.31 (top 1) for locations.

The system only needed a few seconds for the overall online retrieval and extraction process per template. Further details about the system and more on the evaluation are presented in Neumann and Xu (2003).

8.6.2 WHIES – Integrating Shallow and Deep NLP for Information Extraction

WHAM has been used for both deep and shallow NLP in a template-based information extraction system called WHIES (Xu and Krieger, 2003).

The idea of WHIES is to go further than mainstream information extraction (IE) systems that do not attempt an exhaustive deep analysis of all aspects of a text, but rather try to identify and analyze only those text portions that contain relevant information. The shallow-only strategy warrants speed and robustness, but performs moderately on unrestricted natural language text (cf. WAG in the section above).

Appelt and Israel (1999) argue that the current IE technology seems to have an upper performance level of approx. 60%. However, complex scenario-based information extraction with shallow methods and highly specialized, domain-specific fine-tuning seems to be able to break that barrier.

Moreover, from deep analyses of text the observation is that precision and recall could be potentially be higher on restricted domains. In contrast to shallow methods, structured linguistic relationships can be provided such as grammatical functions and referential relationships, including e.g. passive, control/raising, long-distance dependencies and free word order.

Xu and Krieger (2003) present the following example containing both a passive and a control construction (wurde gebeten, zu übernehmen).

- Hans Becker wurde aufgrund des Rücktritts von Peter Müller gebeten, die Presseabteilung zu übernehmen.
  Hans Becker was due to the resignation of Peter Müller asked, to take over the press division.

- Aufgrund des Rücktritts von Peter Müller wurde Hans Becker gebeten, die Presseabteilung zu übernehmen.
  Due to the resignation of Peter Müller Hans Becker was asked, to take over the press division.

Relationships such as the one between *Hans Becker* and the division name *Presseabteilung* cannot be formulated by regular expressions (in a general case). The relatively free word order of German allows reversing the order of the two names, by keeping the same meaning. In this sense, German is 'harder' than English for information extraction.

In a study conducted by Xu and Krieger (2003), the WHITEBOARD architecture has been used to showcase the possible way an information extraction application could employ deep and shallow analyses, and to elaborate cases where deep analyses could provide information that cannot be found by shallow processing only.

A demonstrator application for WHIES has been developed for the domain of management succession (Figure 8.4) using the deep and shallow programming interface of WHAM as fully functional back-end for hybrid processing.

The information extraction system WHIES consists of a template filling component and a template merging component.

The template filling component is hybrid. *Pattern-based template filling rules* are applied to shallow results (tokens, simple lexical items, named entities, phrases). *Lexicalized unification-based template filling rules* operate on the MRS structures output by the deep parser that contain predicate-argument structures.

The filled templates are represented as typed feature structures, and the *template merging* component combines the filled shallow and deep templates with (scenario-specific) template merging rules by means of feature structure unification and subsumption tests to remove non-maximally specific templates.

Figure 8.4 shows an example of template merging and the result for the sentence

> Der Aufsichtsrat hat den Rücktritt von Hermann Kronseder, Vorstandssprecher der Krones AG, angekündigt. Lorenz Raith wurde Hermann Kronseder zufolge gebeten, die Stelle zu übernehmen.

Xu and Krieger (2003) report on an initial evaluation of the 50 most-relevant sentences out of a corpus of 299 documents (management succession reports taken from a dpa/German press agency collection) showed promising recall results (0.92

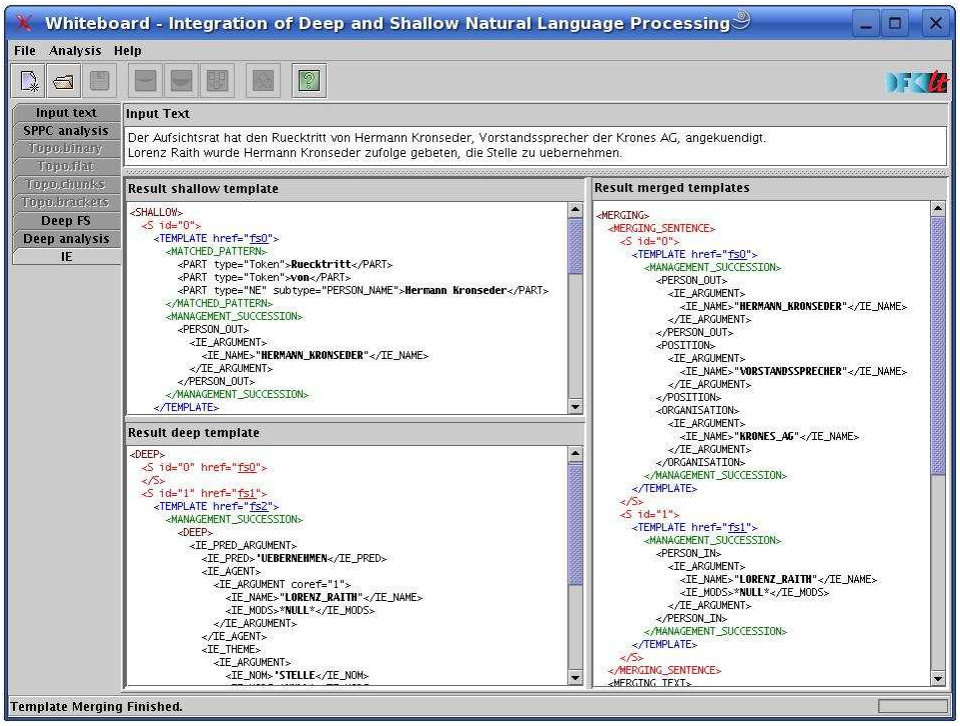


Figure 8.4: The WHIES demonstrator GUI for hybrid information extraction

for deep-shallow compared to 0.31 for shallow only) for the template merging task based on deep and shallow sentence analyses.

Because of the early and immature status of the German HPSG grammar at that time with respect to semantics representation output and disambiguation of multiple readings, the template filling task has been performed manually, thus was merely a simulation of what template filling of HPSG analyses could achieve in principle.

8.7 WHITEBOARD II: Annotation Access and Transformation with WHAT

Although the first architecture prototype for deep-shallow integration was stable, useful, and showed impressive improvements on the NEGRA corpus with respect to lexical and overall (parsing) coverage (cf. Section 8.4), a shortcoming became obvious when new components, mainly to integrate shallow components for phrase and topological sentence structure, were added.

The problem was more a practical, not a principal one: The close integration

of the (shallow) API within the deep parser and the postulated isomorphism of the XML annotation and the API structure (modulo naming of entities) made the integration of additional or alternative components a tedious task that required recompilation of the API bridge almost every time the annotation format changed or a new component had to be added. On the other side, given the wide range of different annotation formats and paradigms, a fully automatic mapping from arbitrary annotation formats to API routines in a usable and useful way was not feasible.

An additional layer of abstraction between shallow annotation and basic API routines turned out to be necessary, filling the deep parser's chart, without the need to adapt the parser's API for each change in shallow annotation formats.

As motivated extensively in Chapter 5, XSLT has been chosen as transformation and query language for annotation access as it is declarative for simple mappings, but also provides programming language power for complex annotation computation and combination. The additional component that is added for XSLT transformation to the WHAM is called WHAT.

8.7.1 WHAT, the WHITEBOARD Annotation Transformer

WHAT is built on top of a standard XSL transformation engine. It provides uniform access to standoff annotation through queries that can either be used from non-XML aware components to get access to information stored in the annotation (as an extension of XPath), or to transform (modify, enrich, merge) complete XML annotation documents.

WHAT XSLT queries are specific for a standoff document structure (DTD or schema) of a component's XML output format, i.e., they must be written once for a new component and are collected in a template library (cf. Figure 8.5). However, as output of e.g. different taggers is similar, the query code could at least partially be reused. WHAT queries are embedded in WHAM API calls, mainly to abstract from component-specific details such as namings of types *etc*.

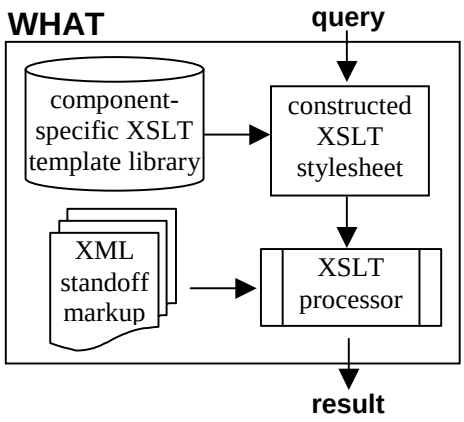


Figure 8.5: WHAT and XSLT template library

A WHAT query consists of component name, query name, and query-specific parameters such as an index or identifier. To return the result of a WHAT query on a given XML input document, the query code is looked up in the XSLT template library for the specified component by name. The associated XSLT stylesheet is constructed, returned and applied to the XML document by the XSLT processor. The result of stylesheet application is then returned as the answer to the WHAT query.

There are basically three kinds of results: (1) strings (including non-XML output), (2) references to nodes in the XML input document via identifiers, (3) XML documents.

Formulating queries as functions, we distinguish the following three query signatures, with C being the component, D denoting an XML document, P^* a (possibly empty) sequence of parameters, S^* a sequence of strings, and N^* a sequence of nodes.

- **V-queries**. getValue: $C \times D \times P^* \longmapsto S^*$
  V-queries return string values from XML attribute values or text. The simplest case is a single XPath lookup, e.g. of the gender of a word encoded in a shallow XML annotation.

- **N-queries**. getNodes: $C \times D \times P^* \longmapsto N^*$
  N-queries compute and return lists of node identifiers (e.g. to answer structural queries) that can again be used as parameters for subsequent queries, e.g. all named entity nodes within a token or character range specified as query parameters.

- **D-queries**. getDocument: $C \times D \times P^* \longmapsto D$
  D-queries return transformed XML documents. This constitutes the classical, general use of XSLT. Complex transformations that modify, enrich or produce (standoff) annotation can be used for many purposes such as converting formats, merging, modifying or computing annotations.

8.7.2 WHAT Query Types

8.7.2.1 V-Queries (getValue)

V-queries return string values from XML attribute values or text. The simplest case is a single XPath lookup. As an example, we determine the type of named entity 23 in a shallow XML annotation produced by the SPPC system (Piskorski and Neumann, 2000). The WHAT query

```
getValue("NE.type", "de.dfki.lt.sppc", 23)
```

would lead to the lookup of the following query in the XSLT template library for SPPC

```
<query name="getValue.NE.type" component="de.dfki.lt.sppc">
  <!-- returns the type of named entity as number -->
  <xsl:param name="index"/>
  <xsl:template match="/WHITEBOARD/SPPC//NE[@id=$index]">
    <xsl:value-of select="@type"/>
  </xsl:template>
</query>
```

The query basically consists of the XPath match expression that matches the NE element with the desired id attribute, and a select expression that returns the value of the attribute `type`. On appropriate SPPC XML annotation, containing the named entity tag e.g. `<NE id="23" type="location"?>` somewhere below the root tag, this query would return the String ”location”.

By adding a lookup to a translation table (through XML entity definitions, as part of the input document or an external XML-encoded mapping table or as a call to the component-specific template library), it would also be possible to translate namings, e.g. in order to map NLP-component-specific type names to HPSG type names.

We see from this example how WHAT helps to abstract from component-specific DTD structure and namings by providing an annotation-independent interface. However, queries need not be that simple (in fact, the query presented could be formulated as a single XPath expression as well). Complex computations can be performed, e.g. through recursive named templates, there can be multiple input annotations included via the XPath `document()` function, and the return value can also be numbers, e.g. for queries that count elements, words, *etc*.

8.7.2.2 N-Queries (getNodes)

An important feature of WHAT is navigation within the annotation. N-queries compute and return lists of node identifiers that can in turn be used as parameters for subsequent (e.g. V-)queries.

The sample query returns a list of node identifiers of all named entities (NE elements) that are in the given range of tokens (W elements). The template calls a recursive auxiliary template that seeks the next named entity until the end of the W range is reached. The WHAT query

```
getNodes("W.NEinRange", "de.dfki.lt.sppc",3,19)
```

would lead to the lookup of the following query in the XSLT template library for SPPC.

```
<query name="getNodes.W.NEinRange" compon.="de.dfki.lt.sppc">
<!-- returns NE nodes starting exactly at token $index to
    (at most) token $index2 -->
  <xsl:param name="index"/>
  <xsl:param name="index2"/>
```

```
  <xsl:template match="/">
   <xsl:variable name="X" select="//W[@id=$index]/ancestor::NE"/>
   <xsl:if test="$X//W[1]/@id = $index">
    <xsl:call-template name="checknextX">
      <xsl:with-param name="nextX" select="$X"/>
      <xsl:with-param name="lastW" select="$index2"/>
    </xsl:call-template>
   </xsl:if>
  </xsl:template>

  <xsl:template name="checknextX">
   <!-- auxiliary template (recursive) -->
   <xsl:param name="nextX"/>
   <xsl:param name="lastW"/>
   <xsl:variable name="Xtokens" select="$nextX//W"/>
   <xsl:if test="number(substring($Xtokens[last()]/@id, 2)) <=
                 number(substring($lastW, 2))">
    <xsl:value-of select="$nextX/@id"/>
    <xsl:text> </xsl:text>
    <xsl:call-template name="checknextX">
     <xsl:with-param name="nextX" select="//NE[@id=concat('N',
                 string(1 + number(substring($nextX/@id,2))))]"/>
     <xsl:with-param name="lastW" select="$lastW"/>
    </xsl:call-template>
   </xsl:if>
  </xsl:template>
</query>
```

Again, the query forms an abstraction from DTD structure, e.g. in SPPC XML output, named entity elements enclose token elements. This need not be the case for another shallow component; its template would be defined differently, but the query call syntax would be the same.

8.7.2.3 D-Queries (getDocument)

D-queries return transformed XML input documents. This is the classical, general use of XSLT. Complex transformations that modify, enrich or produce (standoff) annotation can be used for many purposes. Examples are

- conversion from a different XML format
- merging of several XML documents into one
- auxiliary document modifications, e.g. to add unique identifiers to elements, sort elements *etc.*
- interfacing NLP applications (up to code generation for a programming language compiler)

- visualization and formatting (trees, feature structures, HTML, PDF, *etc*)
- complex computations on XML input

The last application is perhaps the most important. (Linguistic) computation and transformation can turn a WHAT query into a kind of NLP component itself. This is e.g. intensely used in the shallow topological field parser integration we will describe in Section 8.7.5. There, multiple queries are applied sequentially in order to transform a topological field tree into a list of constraints over syntactic spans that are used for initialization of a deep parser's chart.

We show only a short example here, an auxiliary query that inserts unique identifier attributes into an arbitrary XML document without id attributes by recursively walking through the document, copying all element nodes, adding an `id` attribute with a unique ID value to each element node, and copying all other attributes of the original node (in the `xsl:for-each` loop).

```
<query name="getDocument.generateIDs">
 <!-- generate unique id for each element -->
 <xsl:template match="*">
  <xsl:copy>
   <xsl:attribute name="id">
    <xsl:value-of select="generate-id()"/>
   </xsl:attribute>
   <xsl:for-each select="@*">
    <xsl:copy-of select="."/>
   </xsl:for-each>
   <xsl:apply-templates/>
  </xsl:copy>
 /xsl:template>
</query>
```

Moreover, this an example for a stylesheet which is completely independent of a DTD, it just works on any XML annotation and thus shows how generic XSL transformation rules can be.

Another example of d-queries is transformation of syntactic XML tree representations into Thistle trees for visualization (arbora DTD; see Calder 2000). While the output DTD is fixed (imposed by the arbora DTD), this is again not true for the input document which can contain arbitrary element names and branches that must not be known to the stylesheet in advance. Thistle visualizations of syntactic XML trees generated through WHAT d-queries are reproduced in Figure 8.7, 8.8, and 8.9 in the next sections.

8.7.3 Topoparser Integration

As mentioned in the beginning of this section, the aim of the second phase of WHITEBOARD was to integrate shallow components beyond the lexical level (part-of-speech tagging of words unknown to the deep lexicon, and named entities) with

the deep parser, to see how one can take further advantage of partial knowledge provided by shallow pre-processing to pre-structure the search space of the deep parser.

The scenario we will describe has been first presented in Frank *et al.* (2003), on which parts of the following description are based. Again, as we concentrate on the architecture and technical infrastructure that makes the integration possible, we only briefly touch the linguistic details here.

The key idea of the topoparser integration is structuring (*shaping*) the deep parser's search space by bracketing information computed by a statistical topological field parser running ahead, i.e., those items on the deep parser's chart that are 'licensed' by the shallow parser are ranked higher, thus guiding deep parsing towards a best-first analysis suggested by shallow analysis constraints.

On the other side, constituents which are incompatible with the precomputed shape are penalized by assigning a lower rank. Additional information about proposed constituents, such as categorial or feature constraints, provide further criteria for prioritizing compatible, and penalizing incompatible constituents in the deep parser's chart.

This strategy requires that the concepts of the brackets are compatible, i.e., they must have a common subset of properties, minimally a character span and a type, that can be mapped. In most cases, shallow parsers do not deliver more than that (which makes it a kind of least common denominator), while deep HPSG parsing yields more structured and fine-grained analyses.

The bracketing information would not only include chunks comprising only a few words, but also much larger portions of a sentence such as the German Vorfeld, Mittelfeld, Nachfeld positional fields, in which a deep parser alone would probably become lost in longer sentences because of the wealth of hypothetical boundaries.

8.7.4 Finding Appropriate Linguistic Structures

Finding compatible constructions (including the definition of transformations that make non-isomorphic structures comparable) is the linguistic challenge for integrated deep-shallow processing.

Although the next logical step after using lexical information would be extension to chunks, this strategy has not been followed seriously. The reason is that chunks delivered by state-of-the-art shallow parsers are not isomorphic to deep syntactic analyses that explicitly encode phrasal embedding structures.

As a consequence, the boundaries of deep grammar constituents in (1) a. cannot be pre-determined on the basis of a shallow chunk analysis (1) b. Moreover, the prevailing greedy bottom-up processing strategies applied in chunk parsing do not take into account the macro-structure of sentences. They are thus easily trapped in cases such as (2).

(1) a. [$_{\text{CL}}$ There was [$_{\text{NP}}$ a rumor [$_{\text{CL}}$ it was going to be bought by [$_{\text{NP}}$ a French company [$_{\text{CL}}$ that competes in supercomputers]]]]].

b. [$_{CL}$ There was [$_{NP}$ a rumor]] [$_{CL}$ it was going to be bought by [$_{NP}$ a French company]] [$_{CL}$ that competes in supercomputers].

(2) Fred eats [$_{NP}$ pizza and Mary] drinks wine.

Therefore, the insight in WHITEBOARD was (at least for German and English) that state-of-the-art shallow chunk parsing does neither provide sufficient detail, nor the required accuracy to act as a 'guide' for deep syntactic analysis.

Therefore, shallow analyses that determine the clausal macro-structure of sentences seem to be more promising for integration with HPSG. The *topological field model* of German syntax (Höhle, 1983) divides basic clauses into distinct fields – *pre-*, *middle-*, and *post-fields* – delimited by verbal or sentential markers, which constitute the left/right sentence brackets. This model of clause structure is underspecified, or *partial* as to non-sentential constituent structure, but provides a theory-neutral model of sentence *macro-structure*.

The topological field model provides a pre-partitioning of complex sentences that is (i) highly compatible with deep syntactic analysis, and thus (ii) maximally effective to increase parsing efficiency if interleaved with deep syntactic analysis; (iii) partial results regarding the constituency of non-sentential material ensure robustness, coverage, and processing efficiency.

Wauschkuhn (1996) described a topological parser of German on the basis of hand-crafted CFG rules. Braun (1999) implemented a cascaded finite-state grammar for identifying topological fields. Becker and Frank (2002) explored a corpus-based stochastic approach to topological field parsing, by training a non-lexicalized PCFG on a topological corpus derived from the NEGRA treebank (Skut *et al.*, 1998) of German.

The topological parser employed in WHITEBOARD was provided a tagger front-end for free text processing, using the TnT tagger (Brants, 2000). The grammar was ported to the efficient LoPar parser of Schmid (2000).

Due to the combination of scrambling and discontinuous verb clusters in German syntax, a deep parser is confronted with a high degree of local ambiguity that can only be resolved at the clausal level. Highly lexicalized frameworks such as HPSG, however, do not lend themselves naturally to a top-down parsing strategy. Using topological analyses to guide the HPSG will thus provide external top-down information for bottom-up parsing.

More details on the linguistic aspects of the topoparser integration are discussed in Frank *et al.* (2003).

8.7.5 Architecture of the Hybrid Deep-Shallow System

In this section, we describe the WHAT-based architecture underlying the integrated system, and then provide some evaluation figures in Section 8.7.6.

The fully online-integrated hybrid WHITEBOARD topoparser architecture consists of the efficient deep HPSG parser PET (Callmeier, 2000) utilizing tokenization, part-of-speech, morphology, lexical, compound, named entity, phrase chunk

and topological sentence field analyses from shallow components in a sequential architecture.

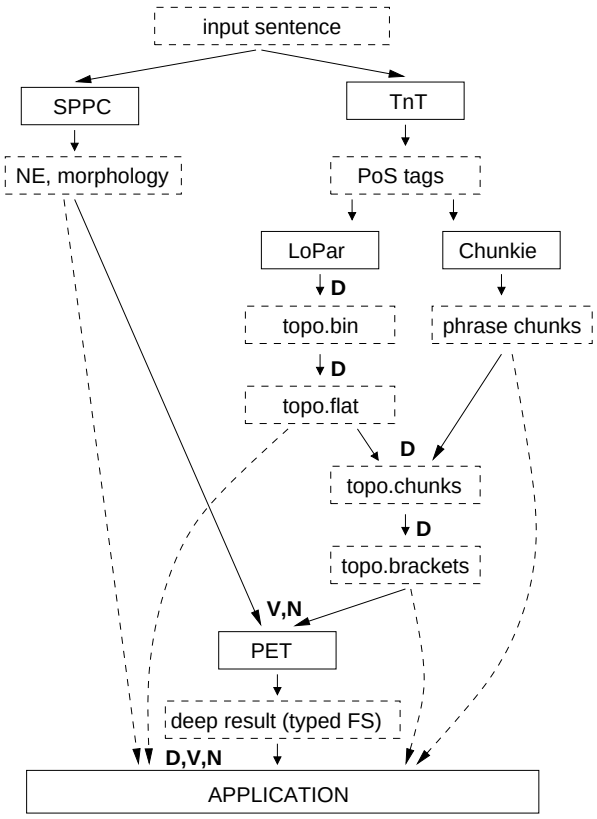


Figure 8.6: XSLT-based architecture of the hybrid parser

The simplified diagram in Figure 8.6 depicts the components and places where WHAT comes into play in the hybrid integration of deep and shallow processing components (V, N, D denote the WHAT query types, i.e., XSLT transformations). Solid boxes indicate components that produce annotation, dashed boxes indicate the produced XML annotation.

Solid-line arrows represent the transformations used in the online integration. Dashed-line arrows indicate possible access to the intermediate annotation that could be accessed from an application (bottom box), e.g. the Thistle (Calder, 2000) tree visualizations that show the XML annotation tree structures (Figure 8.7 through 8.9) have been created through WHAT D-queries out of the intermediate topo.* XML trees.

The system takes an input sentence, and runs four shallow systems on it:

- the rule-based shallow SPPC (Piskorski and Neumann, 2000) for named entity recognition, compound analysis for German, and morphology and stemming of words unknown to the HPSG lexicon,

- TnT, a statistical PoS tagger (Brants, 2000),
- Chunkie, a statistical chunker based on TnT (Skut and Brants, 1998),
- LoPar, a probabilistic context-free parser (Schmid, 2000), which takes PoS-tagged tokens as input, and produces binary tree representations of sentence fields, e.g. topo.bin in Figure 8.7. For a motivation for binary vs. flat trees cf. Becker and Frank (2002).

The results of these components are multiple XML standoff annotations for the input sentence. Named entity, compound, morphological and stem information from SPPC is used by the deep parser to initialize the chart with prototypical feature structures that are filled with shallow information through V-queries for words unknown to the HPSG lexicon and for named entities.

In addition, preference information on part-of-speech is used for prioritization of the deep parser. Details have been described above in the WHITEBOARD-I description (Section 8.4). PoS tagging from TnT is used as input for Chunkie to produce chunking and as input for the shallow topological PCFG parser (Frank *et al.*, 2003).

The examples in Figure 8.7 through 8.11 show the analyses of the German sentence

> *Untergebracht war die Garnison in den beiden Wachlokalen Hauptwache und Konstablerwache. (Located was the garrison at the two guard houses, the main guard house and the Constabler guard house.)*

with a fronted verb in topic position, which the topological parser identifies correctly. This macro-sentential information can be used to direct the deep parser's search space towards the correct (and rather infrequent) construction, avoiding alternative exploration of the search space.

The following XML document is the output of the topological parser (topo.bin), graphically represented in Figure 8.7.

```
<?xml version="1.0"?>
<ROOT>
  <CL fn="V2">
    <VF fn="TOPIC">
      <RK fn="VPART">
        <VVPP>
          <W id="W0">Untergebracht</W>
        </VVPP>
      </RK>
    </VF>
    <LK fn="VFIN">
      <VAFIN>
        <W id="W1">war</W>
      </VAFIN>
    </LK>
```

```
      <MF>
        <ART>
          <W id="W2">die</W>
        </ART>
        <MF>
          <NN>
            <W id="W3">Garnison</W>
          </NN>
          <MF>
            <APPR>
              <W id="W4">in</W>
            </APPR>
            <MF>
              <ART>
                <W id="W5">den</W>
              </ART>
              <MF>
                <PIDAT>
                  <W id="W6">beiden</W>
                </PIDAT>
                <MF>
                  <NN>
                    <W id="W7">Wachlokalen</W>
                  </NN>
                  <MF>
                    <NN>
                      <W id="W8">Hauptwache</W>
                    </NN>
                    <MF>
                      <KON>
                        <W id="W9">und</W>
                      </KON>
                      <MF>
                        <NN>
                          <W id="W10">Konstablerwache</W>
                        </NN>
                      </MF>
                    </MF>
                  </MF>
                </MF>
              </MF>
            </MF>
          </MF>
        </MF>
      </MF>
    </CL>
  </ROOT>
```

In order to extract this type of global constituent-based information, a sequence

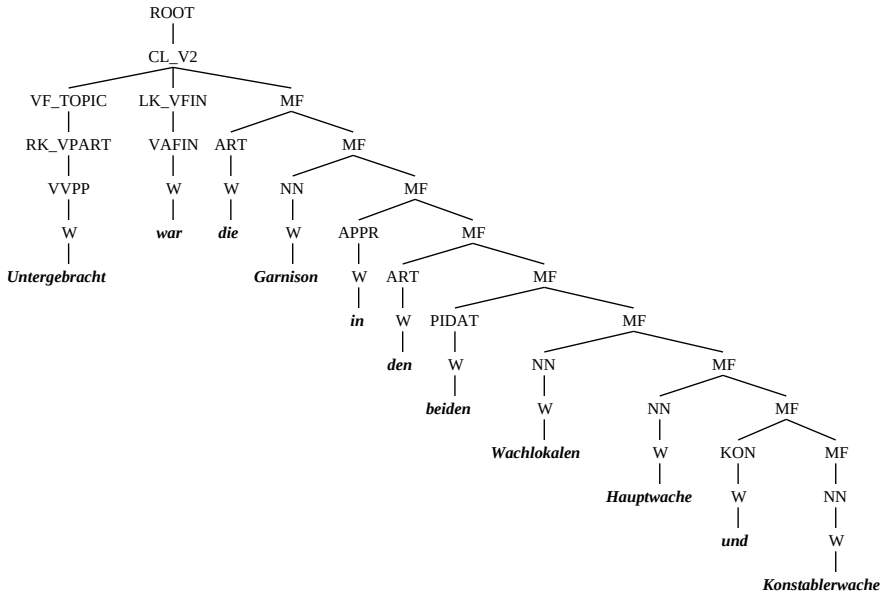


Figure 8.7: Result of the topological parser (topo.bin) as binary tree

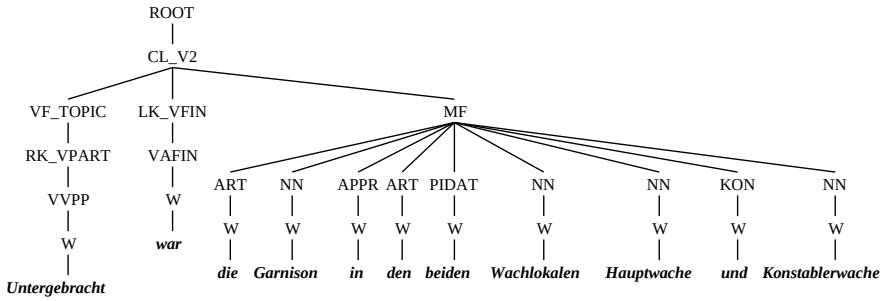


Figure 8.8: The topological tree after flattening (topo.flat)

of D-queries is applied to flatten the binary topological trees that are output by LoPar (result is topo.flat, Figure 8.8) and merge the tree with shallow chunk information from Chunkie (topo.chunks, Figure 8.9). In a next step, we apply the main D-query which computes bracket information for the deep parser from the merged topological tree and chunks (topo.brackets, Figure 8.11).

In order to communicate the structural constraints from the topological parser to the deep parser, the topological tree is scanned for relevant configurations by the stylesheet code of the D-query, and the span information is extracted for the target HPSG constituents. The resulting 'map constraints' (Figure 8.11) encode a

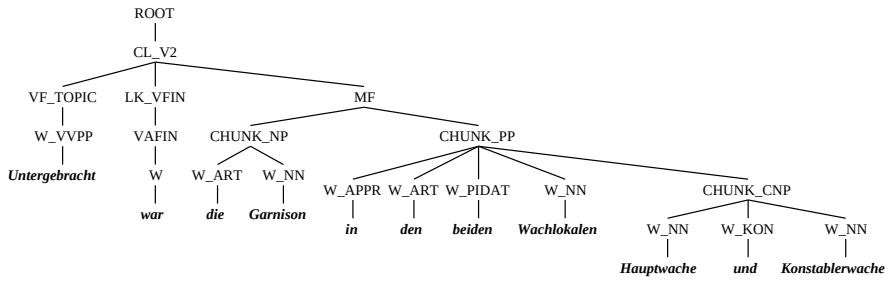


Figure 8.9: The topological tree merged with chunks (topo.chunks)

bracket type name (34 different bracket types are mapped) that identifies the target constituent and its left and right boundary, i.e., the concrete span in the sentence.

The bracket information is used to prioritize chart elements of the deep parser that match the constituent boundaries computed by the shallow parser and chunker. The stylesheet directly generates the names of the appropriate HPSG types (value of the rule attributes in Figure 8.11).

Bernd Kiefer extended the PET parser in such a way that it can exploit computed priorities for the chart elements assigned to the phrasal constraints (brackets) from the topological analysis, in addition to the word-based PoS ranking described in WHITEBOARD-I above.

A related approach can be found in Kiefer *et al.* (2000) for parsing in a speech translation application, where prosodic boundaries help to structure the search space of an HPSG parser.

Depending on the bracket type and chart edge configuration (*left-*, *right-*, and *fully matching* brackets), application of corresponding HPSG rules is either penalized or ranked higher. A parser task in the following is a configuration of passive and an active chart edge.

A right-matching bracket may affect the priority of parser tasks whose resulting edge will end at the right bracket of a pair such as, for example, a task that would combine edges *C* and *F* or *C* and *D* in Figure 8.10. Left-matching brackets work analogously. For fully matching brackets, only tasks that produce an edge that matches the span of the bracket pair can be affected, such as a task that combines edges *B* and *C* in Figure 8.10.

If, in addition, specified rule as well as feature structure constraints hold, the task is rewarded if they are positive constraints, and penalized if they are negative ones. All tasks that produce *crossing* edges, i.e., where one endpoint lies strictly inside the bracket pair and the other lies strictly outside, are penalized, e.g. a task that combines edges *A* and *B* in Figure 8.10.

Details of the algorithm that computes these priorities are explained in Frank

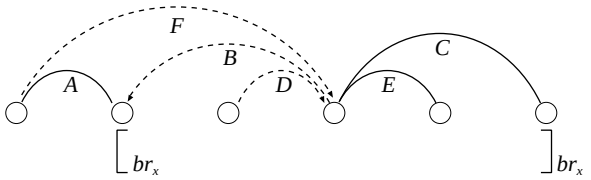


Figure 8.10: An example chart with a bracket pair of type *x*. The dashed edges are active

et al. (2003).

The priority of a parser task is modified relative to a default priority using two confidence values, one, $conf_{ent}(br_x)$, based on tree entropy of a topological parse (per sentence), and one, $conf_{pr}(br_x)$, based on a measure of expected accuracy for each bracket type. These confidence parameters take into account the fact the stochastic topological parser may deliver (partially) wrong analyses and try to correct them. If confidence is high, the topological brackets are fully considered for prioritization. If it is low, their impact is decreased or completely ignored.

Both confidence values are weighted using a heuristically determined weight factor γ, and all these parameters together are used to either add to or subtract from the default priority, depending on whether the bracket and chart configuration triggers reward or penalty. Thus, the priority $p(t)$ of a task t involving a bracket br_x is computed from the default priority $\tilde{p}(t)$ by:

$$p(t) = \tilde{p}(t) * (1 \pm conf_{ent}(br_x) * conf_{pr}(x) * \gamma)$$

```
<TOPO2HPSG>
  <MAPC type="chunk_np+det" rule="chunk" left="W2" right="W3"/>
  <MAPC type="chunk_pp" rule="chunk" left="W4" right="W10"/>
  <MAPC type="v2_cp" rule="vfronted" left="W0" right="W10"/>
  <MAPC type="vfronted_vfin-rk" rule="vfronted" left="W1" right="W1"/>
  <MAPC type="vfronted_vfin+vp-rk" rule="vfronted" left="W1" right="W10"/>
  <MAPC type="v2_vf" rule="vfronted" left="W0" right="W0"/>
  <MAPC type="v2_vfin_pvp-rk" rule="vfronted" left="W1" right="W1"/>
</TOPO2HPSG>
```

Figure 8.11: The extracted brackets (topo.brackets)

Finally, the modified deep parser PET is started with a chart initialized using V-queries to access lexical (morphology, stemming, compounds, PoS preferences) and named entity information gathered from SPPC. The computed bracket information (Figure 8.11) is accessed through WHAT V and N-queries during parsing in order to prioritize constituent analyses motivated by the topological parser and additional syntactic information.

The abstraction provided by WHAT facilitates exchange of the shallow input components of PET, e.g. it would be possible to exchange some of the used components without rewriting the deep parser's code.

The complete input for the deep parser is encoded in a single XML document (per input sentence), e.g.

```
<?xml version="1.0"?>
<WHITEBOARD>
  <SPPC_XML version="2002-06-24" type="transformable">
    <ENCODING_TABLES>
     ...
    </ENCODING_TABLES>
    <DOCUMENT_STATISTICS tokens="12" lexical_items="12" unknown_words="1"
      words_found_in_lexicon="11" words_with_prefered_reading="9"
      named_entities="0" phrases="4" sentences="1" subclauses="0"/>
    <PARAGRAPH id="P0">
      <S id="S0">
        <CHUNK id="H0" type="6">
          <W id="W0" tc="22">Untergebracht
            <READINGS id="D0" pref="R0">
              <R id="R0" pos="2" stem="unterbring" infl="152" code="94"/>
            </READINGS>
          </W>
          <W id="W1" tc="21">war
            <READINGS id="D1" pref="R1">
              <R id="R1" pos="3" stem="sei" infl="36 37" code="54"/>
            </READINGS>
          </W>
        </CHUNK>
        <CHUNK id="H1" type="1">
          <W id="W2" tc="21">die
            <READINGS id="D2" pref="NONE">
              <R id="R2" pos="22" stem="die" infl="" code="5"/>
              <R id="R3" pos="7" stem="d-det" infl="18 21 13 23 29 16 26 30"
                 code="23"/>
             </READINGS>
          </W>
          <W id="W3" tc="22">Garnison
            <READINGS id="D3" pref="R4">
               <R id="R4" pos="1" stem="garnison" infl="18 19 20 21"
                  code="13"/>
            </READINGS>
          </W>
        </CHUNK>
        <CHUNK id="H2" type="2">
          <W id="W4" tc="21">in
            <READINGS id="D4" pref="R5">
              <R id="R5" pos="23" stem="in" infl="3 4" code="4"/>
            </READINGS>
          </W>
          <W id="W5" tc="21">den
            <READINGS id="D5" pref="NONE">
              <R id="R6" pos="22" stem="den" infl="" code="5"/>
```

```
      <R id="R7" pos="7" stem="d-det" infl="7 15 25 27" code="20"/>
    </READINGS>
  </W>
  <W id="W6" tc="21">beiden
    <READINGS id="D6" pref="R8">
      <R id="R8" pos="14" stem="beid" infl="334 335 336 337 338 339
         340 341 342 343 344 345 346 347 348 349 350 351 352 353 354
         355 356 357 358 359 360 361 362 363 364 365 366 367 368 369
         370 371 372 373 374 375 376 377" code="159"/>
    </READINGS>
  </W>
  <W id="W7" tc="22">Wachlokalen
    <READINGS id="D7" pref="R9">
      <R id="R9" pos="1" stem="wachlokal" infl="27" code="73"/>
    </READINGS>
  </W>
</CHUNK>
<CHUNK id="H3" type="1">
  <W id="W8" tc="22">Hauptwache
    <READINGS id="D8" pref="R12">
      <R id="R10" pos="5" stem="wach" infl="129 130 131 132 133 134
         135 136 137 138 139 140 141 142 143" code="82"/>
      <R id="R11" pos="2" stem="wach" infl="43 62 63 17" code="71"/>
      <R id="R12" pos="1" stem="wache" infl="18 19 20 21" code="13"/>
    </READINGS>
    <COMPOUND id="C0">
      <SEGMENT id="M3420215" surface="haupt">
        <READINGS id="D9">
          <R id="R13" pos="1" stem="haupt" infl="9 10 11" code="9"/>
        </READINGS>
      </SEGMENT>
      <SEGMENT id="M3420216" surface="wache">
        <READINGS id="D10">
          <R id="R14" pos="5" stem="wach" infl="129 130 131 132 133
             134 135 136 137 138 139 140 141 142 143" code="82"/>
          <R id="R15" pos="2" stem="wach" infl="43 62 63 17"
             code="71"/>
          <R id="R16" pos="1" stem="wache" infl="18 19 20 21"
             code="13"/>
        </READINGS>
      </SEGMENT>
    </COMPOUND>
  </W>
  <W id="W9" tc="21">und
    <READINGS id="D11" pref="R17">
      <R id="R17" pos="19" stem="und" infl="" code="40"/>
    </READINGS>
  </W>
  <W id="W10" tc="22">Konstablerwache
    <READINGS id="D12" pref="R18">
      <R id="R18" pos="1" stem="konstablerwache" infl="18 19 20 21"
         code="13"/>
    </READINGS>
  </W>
```

```
        </CHUNK>
        <W id="W11" tc="1">.</W>
      </S>
    </PARAGRAPH>
  </SPPC_XML>
  <TOPO sno="0">
    <TOPO2HPSG>
      <MAPC id="T1" mapc="chunk_np+det" rule="chunk" left="W2" right="W3"/>
      <MAPC id="T2" mapc="chunk_pp" rule="chunk" left="W4" right="W10"/>
      <MAPC id="T3" mapc="v2_cp" rule="vfronted" left="W0" right="W10"/>
      <MAPC id="T4" mapc="vfronted_vfin-rk" rule="vfronted" left="W1"
                                                          right="W1"/>
      <MAPC id="T5" mapc="vfronted_vfin+vp-rk" rule="vfronted" left="W1"
                                                             right="W10"/>
      <MAPC id="T6" mapc="v2_vf" rule="vfronted" left="W0" right="W0"/>
      <MAPC id="T7" mapc="v2_vfin_pvp-rk" rule="vfronted" left="W1"
                                                         right="W1"/>
    </TOPO2HPSG>
  </TOPO>
</WHITEBOARD>
```

The result of deep parsing including the constructed semantics representation of the analyzed sentence can be accessed through the chart interface of PET as typed feature structures (Figure 8.13), or via subsequent XSLT transformation in other formats, extracting only partial information.

8.7.6 Evaluation Results

For the evaluation, a subset of the NEGRA corpus consisting of 5060 sentences (24.57%) that were parsable by the HPSG grammar with the PoS tagging integration from WHITEBOARD I, was used. This test set is different from the corpus that has been used for determining the confidence values. Average sentence length was 8.94, ignoring punctuation; average lexical ambiguity was 3.05 entries per word.

As baseline, a run without topological information was performed, yet including PoS prioritization from tagging. A series of tests explored the effects of alternative parameter settings. Furthermore, the impact of chunk information was tested. To this end, phrasal fields determined by topological parsing were fed to the chunk parser of Skut and Brants (1998).

Extracted NP and PP bracket constraints were defined as left-matching bracket types, to compensate for the non-embedding structure of chunks. Chunk brackets were tested in conjunction with topological brackets and in isolation, using the labeled precision value of 71.1% in Skut and Brants (1998) as a uniform confidence weight. For all runs, the maximum number of passive edges was set to the comparatively high value of 70000.

For all runs, the absolute time needed to compute the first reading was measured. Various variants have been tried, with $\gamma = 1$ and $\gamma = 0.5$ as confidence weight, combined with and without both confidence values. Details are presented in a table (Frank *et al.*, 2003). We content ourselves here with the overall result

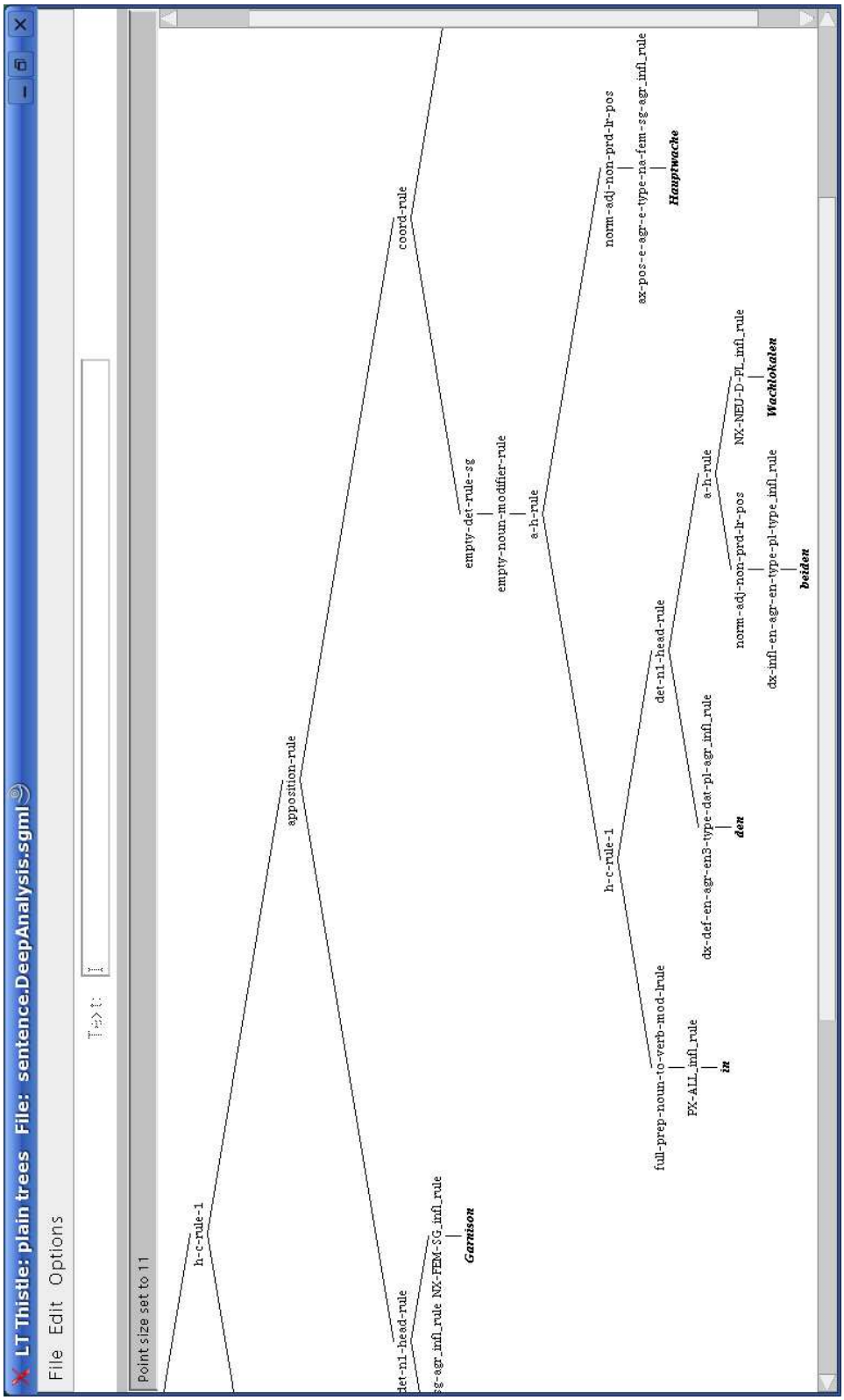


Figure 8.12: Part of the derivation tree of the deep parser

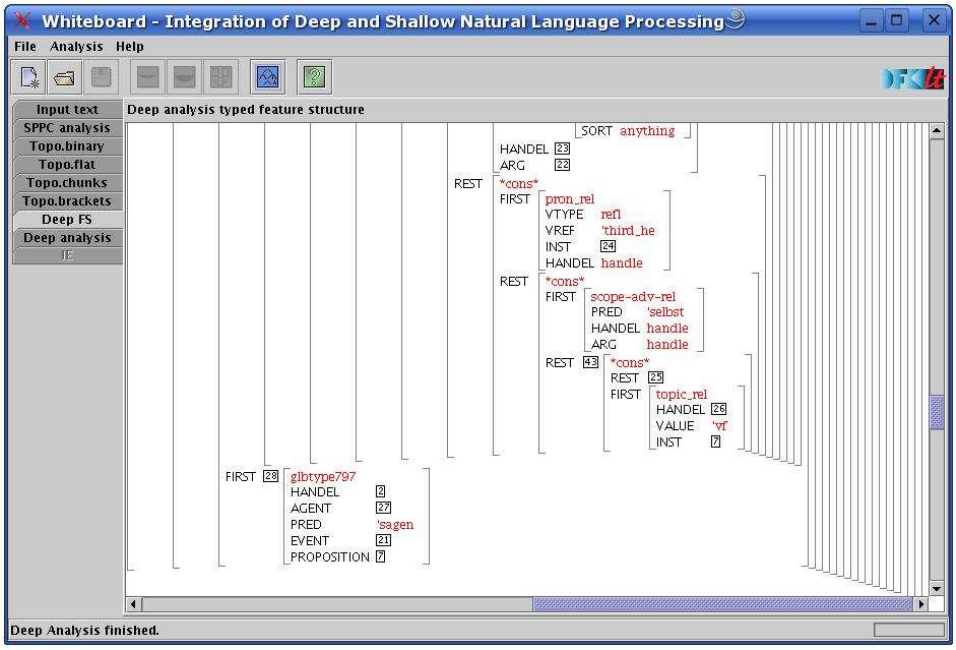


Figure 8.13: Part of a deep parsing result in the WHITEBOARD GUI

which is a speedup of factor 2.25 on average with the topoparser integration. Even without the confidence factor, the speedup is still 2.19 ($\gamma = 0.5$).

Additional use of chunk information decreases the speedup slightly to approx. 2.15, with chunk brackets only to 1.0, both probably due to lower precision of chunk brackets. A similar observation has been reported by Daum *et al.* (2003) for integration of chunk and dependency parsing, they measured a gain of factor 2.76 relative to a non-PoS-guided baseline, which reduces to factor 1.21 relative to a PoS-prioritized baseline, as in our scenario.

8.7.7 Conclusion

In the WHITEBOARD II phase, it could be shown that the integration of shallow topological and deep HPSG processing results in significant performance gains, of factor 2.25 – at a high level of deep parser efficiency. It was shown that macro-structural constraints derived from topological parsing improve significantly over chunk-based constraints. Fine-grained prioritization in terms of confidence weights could further improve the results.

The XML and XSLT-based architecture on the basis of WHAM and WHAT is flexible and was easily extended to address robustness issues beyond lexical matters. By extracting spans for clausal fragments from topological parses, in case

of deep parsing failure, the chart can be inspected for spanning analyses for sub-sentential fragments. Fragment output of deep parsing as fall-back result will be further addressed in Chapter 9. Moreover, the input sentence could be simplified by pruning adjunct sub-clauses, and trigger re-parsing on the pruned input, thus performing a divide and conquer strategy that could help to cope with long sentences where a deep parser often fails because of the huge search space.

8.7.8 Transformation for Visualization

As described in Chapter 5, XSLT can be used to transform linguistic markup for visualization purposes. The tree visualizations in Figure 8.8 through 8.12 have been generated through a generic WHAT D-query transforming the XML-encoded topological parse tree into a Thistle visualization tree (Calder, 2000).

The target SGML format (Thistle arbora DTD) can be generated thanks to the openness of XSLT with respect to output formats. The Thistle editor mode could in principle be used to edit the generated tree representations (Figure 8.14). The resulting, corrected SGML file could be used to improve the underlying stochastic topological parser that originally generated the topological tree.

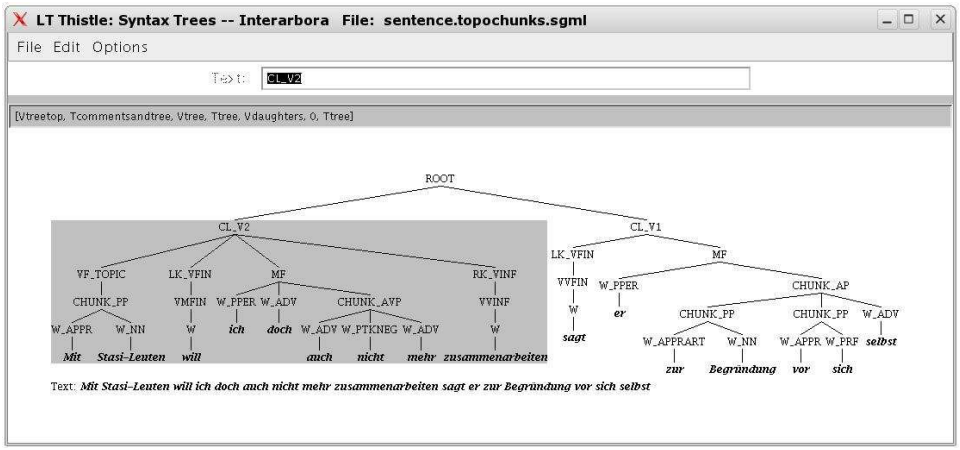


Figure 8.14: Topological parse tree in Thistle editor mode

8.8 Related Work

WHITEBOARD was the first implemented system that integrated multiple shallow processing components (not only PoS tagging) with an advanced, high-performance deep HPSG-based parser. In addition, WHITEBOARD provides an architecture framework that supports easy integration of other shallow components by means of XML annotation and through XSL transformation. These facts make the architecture superior to other, in most cases *ad hoc* integrations of specific systems.

Another NLP architecture also called WHITEBOARD has been developed at ATR Kyoto (Boitet and Seligman, 1994). The focus of that prototypical system designed for speech translation was to overcome restrictions of both pipeline and blackboard architectures by postulating a *coordinator* that would schedule NLP components and mediate between them. However, the ATR WHITEBOARD idea is different from our WHITEBOARD in that access to NLP component results is only possible via the coordinator. Moreover, the assumed and supported data structures are specific for speech processing (time-aligned lattice) and not directly usable for concepts such as abstraction-based deep and annotation-based shallow processing results.

There exists only very little other work that considers integration of shallow and deep NLP utilizing an XML-based architecture, most notably Grover and Lascarides (2001) for the HPSG precursor GPSG. However, their integration efforts are largely limited to the level of PoS tag information.

Ad hoc integrations of PoS tagging and specific HPSG grammars have been conducted for Dutch (Prins and van Noord, 2003) and Spanish (Marimon, 2002a).

There was also integration work in deep parsing other than HPSG, e.g. Daum *et al.* (2003) combined PoS tagging and chunking with a dependency parser. Kaplan and King (2003) and Kaplan *et al.* (2004) combine PoS tagging and finite-state preprocessing with the LFG parser. The common observation from their results is that mainly PoS tagging as preprocessing increases coverage and robustness that the deep frameworks alone would not accomplish in an economically way. The results we have obtained in WHITEBOARD support this observation.

8.9 Summary

In this chapter, we have presented the key architecture concepts of WHITEBOARD, the WHITEBOARD Annotation Machine (WHAM) and the WHITEBOARD Annotation Transformer (WHAT). We have demonstrated an application scenario with highly integrated multiple shallow preprocessors and a deep parser, and shown the advantages of integrating both for increased robustness (recognition of words unknown to the deep lexicon) and search space reduction (shallow pre-shaping of the deep parser's search space).

An evaluation of 5000 sentences of a German newspaper corpus showed that the already high efficiency of deep parsing could be further improved by a factor of 2.25 on average, lexical coverage increased from 28 to 71% and overall parsing coverage (full parses) from 12.5 to 22%. It has to be noted that these results were obtained at a very early stage of German HPSG grammar development, where the grammar was more elaborated on speech dialog (VERBMOBIL) than on general newspaper text.

WHITEBOARD, extended with WHAT, is an open, flexible and powerful infrastructure based on standard XSLT technology for the online and offline combination of natural language processing components, with a focus on, but not limited to, hy-

brid deep and shallow architectures.

The infrastructure is portable. As the programming language-specific wrapper code is relatively small, the framework can be quickly ported to any programming language that has XSLT support (which holds for most modern programming and scripting languages). XSLT makes the transformation code portable and declarative which it could not be when being based on DOM manipulation in an ordinary programming language.

The WHAT framework can easily be extended to new NLP components and document DTDs. This has to be done only once for a component or DTD through XSLT query library definitions, and access will be available immediately in all programming languages for which a WHAT implementation exists.

WHAT can be used to perform computations and complex transformations on XML annotation, provide uniform XML annotation access in order to abstract from component-specific namings and DTD structure. WHAT makes it easier to exchange results between components (e.g. to give non-XML-aware components access to information encoded in XML annotation), and to define application-specific architectures for online and offline processing of NLP XML annotation.

Due to its flexibility, the infrastructure is well suited for rapid prototyping of hybrid NLP architectures as well as for developing NLP applications, and can be used to both access NLP markup from programming languages and to compute or transform it.

Besides the integration within NLP architectures described in this section, the XSLT-based infrastructure (WHAT) could also be used for interfacing applications, e.g. to translate to Thistle (Calder, 2000) for visualization of linguistic analyses and back from Thistle in editor mode, e.g. for manual, graphical correction of automatically annotated texts for training *etc*.

Because of the unstable standardization and implementation status, we did not yet make use of XQuery, an XML query language discussed in Chapter 5. However, the WHAT framework is open, and it might be worth considering XQuery as a future extension. Which engine to ask, an XSLT or an XQuery processor, could be encoded in each `<query>` element of the template library using an additional attribute. Similarly, extension of the current framework to XSLT 2.0 which among other things supports user-definable functions that can be part of XPath expressions, should be straightforward.

Compared to *ad hoc* integrations of specific deep parsers with specific PoS taggers, the XML and XSLT-based WHITEBOARD architecture approach offers much more flexibility. This allowed to easily also integrate further levels of natural language processing other integrated systems do not provide, such as named entity recognition or topological parsing.

However, it has to be noted that although obviously there is huge potential in combining many more existing shallow and deep NLP components and in different ways through a general architecture such as WHITEBOARD, only some concepts could be tried within the project, and even less could also be poured into applications utilizing the new approach.

An interesting application of the architecture that has been formulated already in the project proposal but not tried in an implementation so far, is to use deep processing to support shallow processing on demand. Using this strategy, e.g. in information extraction or opinion mining, it could be possible to both preserve the high robustness of shallow processing and achieve high precision on crucial parts of a text that could have been identified by shallow methods.

Also mainly because of time and resource limitations, the architecture was not fully instantiated for languages other than German.

A further generalization of WHITEBOARD towards more robust, application-oriented integration of deep and shallow NLP components that is even better suited for high coverage and high precision in restricted domains and Semantic Web-related applications will be presented in the next chapter.

Chapter 9

Heart of Gold

9.1 Introduction and Motivation

In the previous chapter, we have described an integration architecture for deep and shallow natural language processing components called WHITEBOARD. Although WHITEBOARD has been designed for flexible integration of components, more or less a single scenario for German (with and without topo-parsing) has been fully implemented. The architecture was successful in the sense that the benefits of integrating deep and shallow approaches to NLP could well and clearly be shown in a mature and stable implementation that was robust enough to parse German newspaper corpora and other unseen text online.

However, the focus of WHITEBOARD was to demonstrate the feasibility and evaluate the benefits of the hybrid approach from the linguistic, mainly syntactic, perspective. Many aspects that would become important when deep-shallow integration would be explored in real NLP-based applications, could not be addressed during the WHITEBOARD project, one main reason being the fact that the German HPSG grammar at that time did not provide a sufficiently functional semantics construction.

The aspects missing in WHITEBOARD with respect to architecture that had to be addressed further comprise (1) true multilinguality (also in parallel), (2) integration support for components implemented in different programming languages other than Java and C/C++, (3) more flexible, configurable processing order of components, (4) fully networking-enabled architecture, (5) post-parsing and fallback integration on a semantics representation level.

9.2 Project Context: DEEPTHOUGHT and QUETAL

The aim of the EU-funded project DEEPTHOUGHT (October 2002-October 2004) was to investigate integrated shallow and deep processing in a multilingual, application-oriented context. Three application scenarios had been chosen to evaluate the hybrid processing approach, (1) email response management for customer

relationship management, (2) precise information extraction for business intelligence, (3) creativity support for document production and collective brainstorming (Uszkoreit *et al.*, 2004).

In addition to the more syntax-oriented WHITEBOARD approach, a new, common semantic representation format allowing for underspecification, RMRS (Copestake, 2003), was to be used as interface to applications and explored also for component integration. The idea is to conceive any NLP component output as a (possibly underspecified) semantic representation. In case a component (e.g. deep parsing) fails, a possibly less specific fall-back analysis of another component would still be available in a compatible format.

After DEEPTHOUGHT had been finished, the framework was further developed and extended in the QUETAL project where it has been mainly used for deep question analysis in restricted domains (Frank *et al.*, 2005, 2006; Schäfer, 2006a). A technical user documentation for Heart of Gold can be found in Schäfer (2005).

In this chapter, we describe the Core Architecture Framework called *Heart of Gold* we have developed for DEEPTHOUGHT in detail[1]. The key concept of Heart of Gold is to treat the core architecture as a flexibly configurable *middleware* in between NLP-based applications and (pre-existing) NLP components for which the middleware provides interfaces (wrapper classes), cf. Figure 9.1.

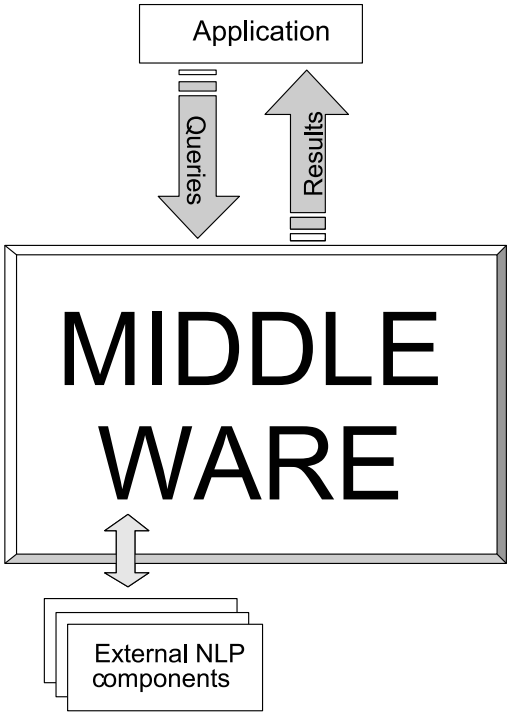


Figure 9.1: Heart of Gold middleware architecture

[1] For the names cf. Adams (1979).

We will first describe the middleware architecture, the most important NLP components for various languages we have integrated so far, grouped according to their functionality, then present some architecture instances and corresponding configurations, motivate some generic extensions, show evaluation results and implemented applications, and finally conclude with an outlook to further developments.

9.3 Middleware Architecture

9.3.1 Overview

Heart of Gold (Callmeier *et al.*, 2004) is an XML-based middleware for the integration of deep and shallow natural language processing components. It provides a uniform and flexible infrastructure for building applications that use RMRS-based and/or XML-based natural language processing components.

The main design goals where:

- flexible integration of NLP components
- simple application interface
- RMRS as (optional) uniform semantic representation language
- open to other XML standoff annotation formats
- integration through annotation transformation
- annotation database interface for storage and retrieval of computed linguistic analyses
- network-enabled architecture with distributed components and lightweight, platform- and programming language-independent communication through XML-RPC
- based on current standardized technology such as XML, XML-RPC, XSLT, XPath, XML:DB

Figure 9.2 depicts the general architecture. The Heart of Gold acts as mediator between applications (top) and NLP components (bottom), abstracting from component-specific interfaces and representations. Applications send queries (analysis requests) on text documents to the middleware which in turn passes the queries to one or more components according to a NLP component configuration initially specified by the applications. The resulting annotation(s), which can also be taken from the annotation database if already computed (caching), are then returned to the application.

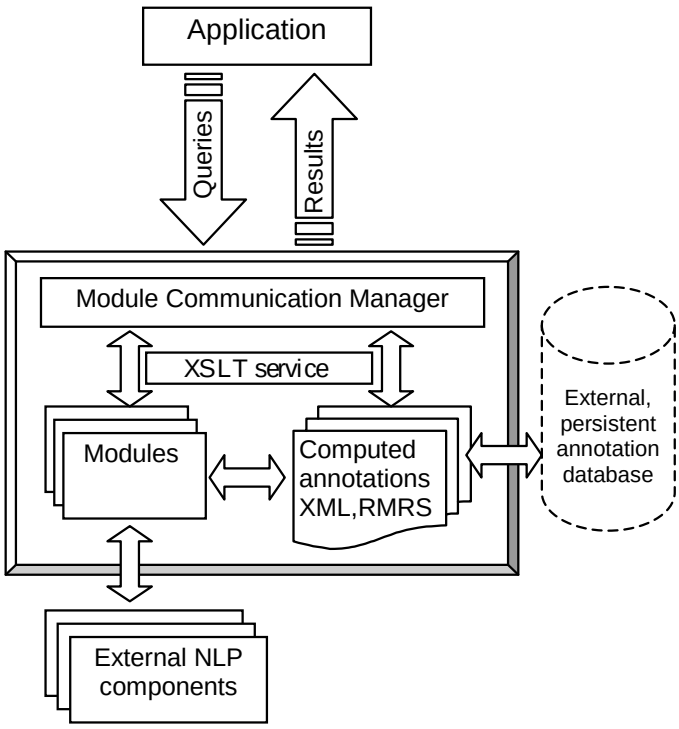


Figure 9.2: Heart of Gold (HoG) core architecture

9.3.2 The Module Communication Manager (MoCoMan)

The Module Communication Manager (MoCoMan) mediates between an application and the annotation-producing NLP components (Figure 9.2). MoCoMan receives a request (text documents, sentences) from an application, sends it to the configured NLP components, receives their analysis results, and returns the results back to the application. The interface to the Heart of Gold allows requests containing the following parameters

1. a language identifier for the language of the string to be analyzed (two-letter ISO 639 code such as `en` for English, `de` for German, `ja` for Japanese),

2. the text to be analyzed,

3. requested analysis depth (numerical).

An optional annotation database supports persistent storage of the computed analyses. MoCoMan is also responsible for the order in which the components are triggered. The implemented default strategy is to let the application specify the depth of desired analysis with the query, and trigger all modules starting from

the shallowest (e.g. tokenizer) up to the requested depth, with a fall-back to the previous component if no result was available from the component with the desired depth.

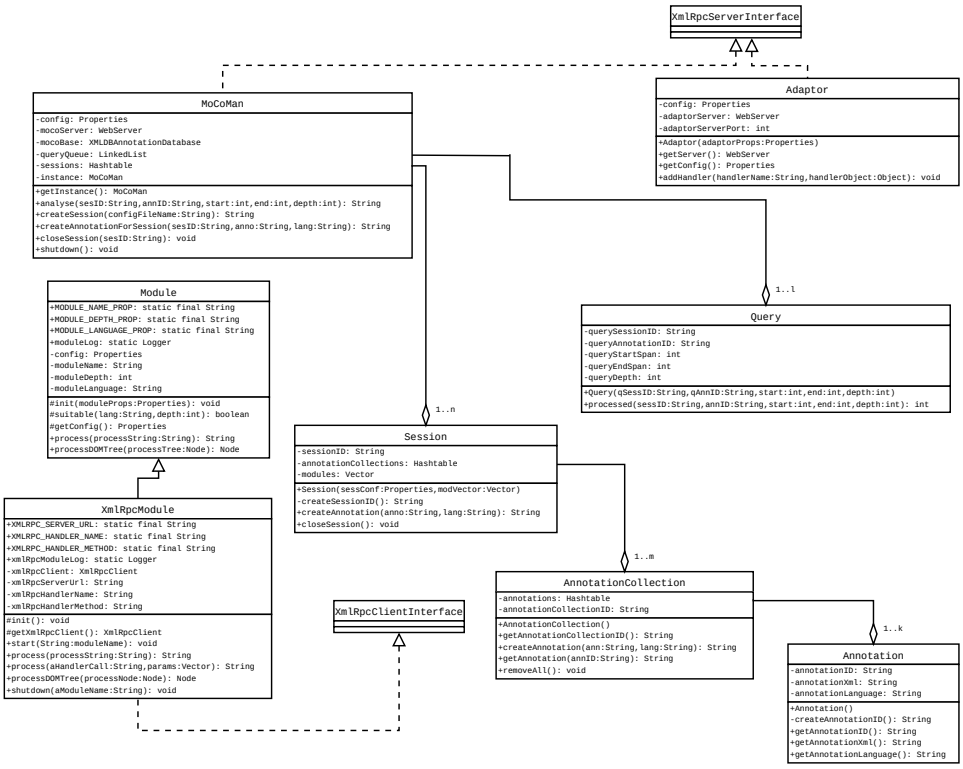


Figure 9.3: UML diagram of MoCoMan

Applications communicate with the Heart of Gold middleware through the Module Communication Manager via a Java API (Java applications) or XML-RPC (remote applications or applications written in programming languages other than Java).

XML-RPC is a lightweight protocol for web services that is supported, through additional libraries, by most current programming languages and scripting languages on various platforms. It is built on top of HTTP and hence can also be used for communication through firewalls which otherwise would have to be opened for specific ports other than the standard HTTP port. This is why XML-RPC has been chosen for the integration of natural language processing components into the architecture. In other words, XML-RPC provides an easy and portable means to network-enable both architecture components and the application interface in Heart of Gold.

While the core architecture is implemented in Java, components and applications can hence be written in other programming languages and connected through XML-RPC.

9.3.3 Modules and Components

Initially, an application starts an instance of the Heart of Gold architecture with a configuration setting for the required components (Figure 9.4 shows a sample configuration for English). MoCoMan then starts (or remotely connects via XML-RPC to) the appropriate components.

```
de.dfki.lt.hog.modules.JTokModule=conf/en/jtok.cfg
de.dfki.lt.hog.modules.TnTModule=conf/en/tnt.cfg
de.dfki.lt.hog.modules.SproutModule=conf/en/sprout.cfg
de.dfki.lt.hog.modules.RaspModule=conf/en/rasp.cfg
de.dfki.lt.hog.modules.PetModule=conf/en/pet.cfg
```

Figure 9.4: Sample configuration for English

Each component is initialized through MoCoMan according to a component-specific configuration file (examples will be shown in the later sections). More precisely, a launcher creates the specified module classes and registers them in a Registry. The module configuration, abstract `Module` and `Registry` classes are courtesy of the Memphis architecture (Kasper *et al.*, 2004) developed by Jörg Steffen, but with a different processing strategy that we will describe below. The `init()` methods of the `Module` classes are executed to start the real components. Each component has its own configuration setting.

From the viewpoint of MoCoMan, components are `Modules` (local Java-based components) or `XmlRpcModules` (remote, possibly non-Java, components). I.e., in order to integrate a new component in the architecture, it must inherit from either `Module` or `XmlRpcModule`. UML diagrams in Figure 9.3 and 9.5 show the core class structure.

The remote counterparts of an `XmlRpcModule` are called `Adapters`. A Java class can be used to implement a remote Java `Adapter`. Moreover, remote adapters can also be implemented in other programming languages and communicate with MoCoMan via XML-RPC.

Modules are required to return an XML (standoff) annotation as result. If a component does not provide XML output, then translation to and (possibly) from XML is to be implemented in the Module classes. RMRS can be used as optional common representation format, in this case, RMRS output can also be generated through XSL transformations.

An example for the integration of a component in this way is the *SProUT* module that uses XSLT transformations of the XML-encoded typed feature structure output of the named entity grammars along the ideas presented in Chapter 5 resp. Schäfer (2003) to generate an XML representation conforming to the RMRS DTD, cf. also Section 9.4 and 9.5.4.1.

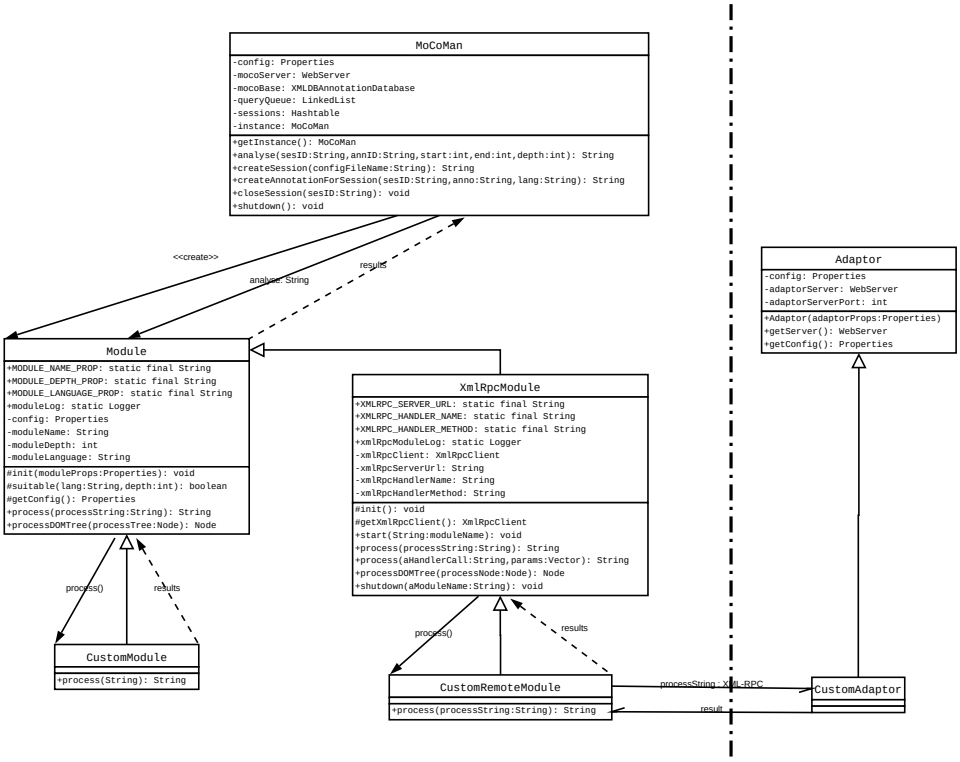


Figure 9.5: UML diagram of module and application communication

9.3.4 NLP Analysis

After the system configuration is finished, analysis requests with parameters such as language code and depth as described above can be passed to MoCoMan.

MoCoMan passes the request to the modules that are configured in the architecture instance and that are appropriate for the requested depth of analysis and language.

To this aim, MoCoMan selects modules using the `suitable()` predicate, taking into account language and depth of analysis. The text is then passed as input parameter to the `process()` method of the first selected component. The XML output of a module is called annotation. Then, in a cascade, the output annotation of a preceding module is taken as the input annotation of a subsequent module's `process()` method until all configured and (through depth and language) suitable components have returned their results.

The annotation computed by the last module is then returned to the application. All annotations can optionally be stored in an XML database; this is a global configuration option. If a query is passed to MoCoMan that has already been com-

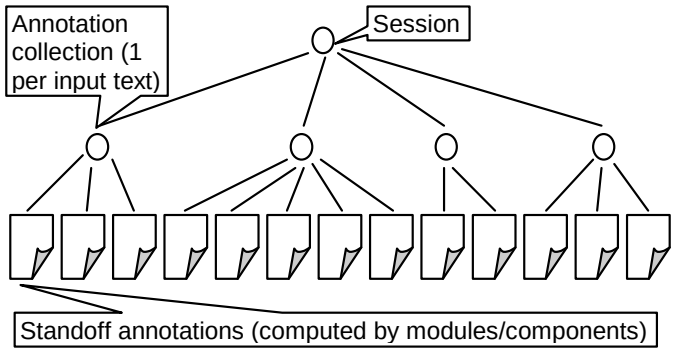


Figure 9.6: Session and annotation management

puted, i.e. with the same input text and query parameters, then the pre-computed result is returned by MoCoMan, either from the database or the Heart-of-Gold-internal storage of limited size, depending on configuration.

9.3.5 Default Processing Strategy

The shallowest component is started first (typically a tokenizer), then other components with increasing depth, up to the requested depth. A fall-back is performed to the result of the previous component if no result from the component with requested depth could be achieved, e.g. in case of an empty analysis of the deep parser.

Each component gets the output of the previous component as input plus the output from other components if configured (the default input may be ignored by the module). The result of the query is the result of the deepest component in the sequence. Analysis results from previous components are returned on request by MoCoMan (method `getAnnotation()`). A module may also produce multiple output annotations and store them explicitly in the active annotation collection as a side effect.

By varying the configured depth of the modules, additional input or output annotation *etc*, it is possible to flexibly adapt the processing order and information flow between modules. Further, even more flexible processing strategies (including parallelism and loops) can be implemented using the SDL extension we will describe in Section 9.5.7.

9.3.6 Session and Annotation Management

MoCoMan provides a session management, so that different input sessions with multiple input documents (texts) can be created and referenced (Figure 9.6). MoCoMan manages a collection of sessions where each session consists of a collection of annotation collections (one annotation collection corresponds to one input text or sentence) that contain RMRS/standoff annotations computed by the components of the configured architecture instance.

Sessions, annotation collections and annotations are referenced through context-unique IDs. Sessions, annotation collections and computed annotations can be stored in the optional XML annotation database.

There are MoCoMan server methods[2] to create sessions and annotation collections in the context specified through IDs as parameters. The annotations themselves are generated using the `analyse()` method that stores the annotations computed by the configured modules in the active annotation collection. The annotation ID is identical with the (configurable) module name that creates the annotation. Examples for the context-unique IDs will be presented in Section 9.3.9.

9.3.7 Metadata

Metadata on date, time, source, processing parameters, processing options and the component-specific configurations of the producing component are stored as part of each computed annotation in a `<metadata>` element. Its `<id>` sub-element contains processing parameters inserted by MoCoMan such as current session, annotation collection and annotation identifiers, component name, creation date and processing time. The `<conf>` sub-element contains a copy of the configuration settings of the module that produced the annotation. This allows to precisely reconstruct the environment under which an annotation was produced (Figure 9.7). This is an important feature when Heart of Gold is used to create linguistically annotated texts for permanent storage, because it allows to at least partially reconstruct *ex post* the circumstances under which an annotation has been produced.

9.3.8 XML Annotation Database

If a query that has already been computed (i.e., a known input text with the same query parameters) is passed to the MoCoMan, then the pre-computed result is returned. This can be done on the basis of the data gained during a session, but the Heart of Gold middleware optionally also provides a database interface for XML annotation storage. The main purpose is persistent storage of computed annotations for the automatic creation or enrichment of linguistic corpora *etc*.

The annotation database interface uses XML:DB[3] which is a vendor-independent interface to XML databases. The current implementation uses the open source

[2] `createSession()` and `createAnnotationCollection()`

[3] `http://xmldb-org.sourceforge.net`

```
<metadata>
  <id>
    <entry name="created" value="2004-03-04 15:23:15"/>
    <entry name="processingtime" value="00:00,90"/>
    <entry name="sessionid" value="session1"/>
    <entry name="acid" value="collection1"/>
    <entry name="component" value="Sprout"/>
  </id>
  <conf>
    <entry name="sprout.outputpath" value="OUT"/>
    <entry name="sprout.stylesheet" value="enamex2rmrs.xsl"/>
    <entry name="module.name" value="Sprout"/>
    <entry name="module.depth" value="1"/>
    <entry name="module.language" value="en"/>
    <entry name="module.rootelement" value="SPROUTPUT"/>
  </conf>
</metadata>
```

Figure 9.7: Annotation and module configuration metadata

(but disappointingly slow) Xindice 1.1b[4] XML database. However, other XML:DB supporting databases such as bdbxml [5] or Tamino[6] could be used instead. An interface class is provided that can be implemented in order to support other XML databases.

The Heart of Gold XML database interface[7] supports organization of XML annotations reflecting the session and annotation collection tree hierarchy of MoCoMan. Standard operations such as inserting, deleting collections and XML annotations, and a standardized query language based on XPath (Clark and DeRose, 1999) are supported. Similarly, existing annotations could also be modified using the XUpdate query language[8]. However, this is currently not actively supported by the Heart of Gold XML:DB interface, but it could become of interest for external, manual correction of computed annotations, e.g. for machine learning purposes.

An important feature of XML databases is indexing of XML document elements to guarantee efficient retrieval. Depending on the structure of the annotation, indexers can be defined through the database interface. These should be defined when integrating new Modules and could be stored as part of the Module configuration which in turn is part of the annotation metadata.

In the current implementation based on Xindice, the XML database can easily

[4] `http://xml.apache.org/xindice/`
[5] `http://dev.sleepycat.com`
[6] `http://www.tamino.com`
[7] Abstract class `AnnotationDatabase`
[8] `http://xmldb-org.sourceforge.net/xupdate/`

be separated physically from the rest of the architecture. The database can reside on a server different from the middleware server and provide its services through the Apache Tomcat web application server[9]. The XML database interface of MoCoMan then acts as a client to the XML database.

For most practical cases, a simple storage and retrieval mechanism based on annotation ID and annotation collection ID is sufficient. It could alternatively be implemented on file system basis becoming another subclass of the abstract class `AnnotationDatabase`.

9.3.9 Annotation Transformation Service

For the integration of NLP components, e.g. those not providing RMRS output formats natively, XSLT can be employed to transform component-specific XML output, e.g. of a chunker or a named entity recognition component, into the RMRS format or other XML formats required by other components.

The class `TransformationService` provides access to the XSL transformer of Heart of Gold. The idea is similar to WHAT in WHITEBOARD, but for the sake of conceptual clarity, we restrict transformation to what we called d-queries in WHAT, i.e. input and output of a transformation is required to be a well-formed XML *document*. This also makes differentiation in the programming interfaces obsolete for annotation access and thus simplifies component integration.

The transformation is performed in the module's `process()` method, the XSLT stylesheet name can be made part of the component-specific configuration. The transformation service is based on a standard XSLT engine, with a single, generic extension for access to other annotations (in the same annotation collection or even in another annotation collection, e.g. for searching anaphora antecedents in previous sentences *etc*).

Access to other annotations is provided through a 'HoG URI' of the form `hog://sid/acid/aid` via the XPath `document()` function, where `sid` is the session ID, `acid` is the annotation collection ID, and `aid` is the annotation ID. The IDs of the active session and annotation collection are automatically passed from the module as (default) parameters to the stylesheet.

An example for the integration of a component using annotation transformation is the *SProUT* module that uses XSLT transformations of the XML-encoded typed feature structure output of the named entity grammars to generate an XML representation conforming to the RMRS DTD, cf. Section 9.5.4.1 below.

The XSLT stylesheets are stored in subdirectories of the **xsl** directory of the Heart of Gold distribution, ordered according to the target transformation format, e.g.

- **html**: HTML visualization of RMRS structures (cf. Section 9.8)
- **latex**: LaTeX visualization of RMRS structures (cf. Section 9.8)

[9] `http://tomcat.apache.org`

- **pic**: transformation to PET input chart format (cf. Section 9.5.5.2), partly automatically generated, cf. description of *SProUT* module below (Section 9.5.4.1)
- **preproc**: preprocessing of input texts in XML format, sentence splitting (Section 9.5.5.2)
- **rmrs**: transformation of e.g. NE recognizer output formats (*SProUT*, LingPipe) to RMRS, partly automatically generated, cf. description of *SProUT* module below (Section 9.5.4.1)
- **sdl**: stylesheets for XSLT transformations as part of SDL sub-architecture processing cascades (Section 9.5.7)
- **xml**: auxiliary stylesheets e.g. for XML pretty-printing

9.4 RMRS as Common Semantic Annotation Format

As already mentioned above, one of the strategic ideas of the DEEPTHOUGHT project that aimed at exploring deep-shallow NLP for new, innovative applications, was to use a common, robust semantics format, RMRS, throughout the core architecture. Heart of Gold has been designed to support this idea and to provide facilities for RMRS generation, transformation and visualization. However, the whole middleware architecture does not necessarily rely on RMRS, it simply treats it as standoff annotation in the same way as it treats other XML formats produced by any integrated NLP component. In this section, we present a short overview over the RMRS idea and introduce the notation we will use later on.

RMRS stands for Robust Minimal Recursion Semantics. It has been introduced in a DEEPTHOUGHT deliverable (Copestake, 2003). The main difference (or generalization) to the MRS flat (non-recursive) semantics framework (Copestake *et al.*, 2005a) is the robustness aspect, i.e., in contrast to MRS which was meant as semantics representation framework for deep grammars based on typed feature structures, RMRSes can not only be *underspecified* for scope as MRSes, but also be *partially* specified, e.g. when the arguments of a transitive or ditransitive verb cannot be fully resolved by an NLP component. Therefore, RMRS is suited for representing output also from shallow NLP components.

The underlying idea is that in principle any shallow processing result, even from a part-of-speech tagger, can be conceived as a, typically underspecified, version of a compatible deep analysis. Several components could deliver partial analyses that could be combined into a single representation where information missing from one component is complemented by information from another component, or left underspecified where unresolved.

Moreover, in the deep-shallow integration scenario, a possibly underspecified semantics representation produced by a shallow component could be used as fallback where deep parsing of a sentence failed.

RMRS comes with an XML format which did not exist for MRS. However, as any MRS can be expressed as RMRS (but not necessarily vice versa), MRSes could via this inclusion be encoded in XML as well.

MRS is currently the semantics representation format of the most elaborated and popular HPSG grammar implementations such as ERG (Flickinger, 2002), the German HPSG grammar developed at DFKI (Crysmann, 2003), or the modern Greek grammar (Kordoni and Neu, 2004). Moreover, it forms the basis for the multilingual Matrix model of HPSG grammar development (Bender *et al.*, 2003), and is used for natural language parsing and generation as well as transfer for machine translation (Bond *et al.*, 2005).

An MRS or RMRS for e.g. a natural language sentence is a logical form consisting of conjunctively connected *elementary predications* (EPs) each being a single relation with associated arguments, e.g. beginning(x,y). Each EP typically corresponds to a lexeme, but there might also be additional EPs not representing a proper word, e.g. indicating question types, compounds *etc*.

Although (R)MRSes are flat, i.e., an EP is never embedded in another EP, there is a notion of quantification and scope that is encoded by augmenting EPs with variables called *handles*.

The handles can be used to constrain readings resulting from scope ambiguities. A deep NLP component may contain more knowledge about how to solve such ambiguities than a shallow one. Ambiguities may be left unresolved, thus providing compact ambiguity representation.

A small example taken from Copestake *et al.* (2005a) may illustrate this.

Every nephew of some famous politician runs.

Two scopes are valid (using a predicate calculus representation):
(a) every(x, some(y, famous(y) $\wedge$ politician(y), nephew(x,y)), run(x))
(b) some(y, famous(y) $\wedge$ politician(y), every(x, nephew(x,y), run(x)))
The MRS that represents the valid scopes is written as

$$\begin{aligned}&\langle h1, \{h2 : \text{every}(x,h3,h4), h5 : \text{nephew}(x,y), h6 : \text{some}(y,h7,h8),\\ &\quad h9 : \text{politician}(y), h9 : \text{famous}(y), h10 : \text{run}(x)\},\\ &\quad \{h1 =_q h10, h7 =_q h9, h3 =_q h5\}\rangle\end{aligned}$$

Hence, an MRS is a triple of top handle ($h1$ in the example), set ('bag') of elementary predications, and a set of handle constraints ('hcons'). The handle constraints are given as so-called qeq constraints (written as $=_q$) which stands for equality modulo quantifiers, and expresses scopal subordination.

A qeq constraint always relates a handle in an argument position (in an EP) to a label, where the argument is either directly filled by the label, or one or more quantifiers 'float' in between handle and label, i.e., the label of a quantifier fills the argument position and the body argument of that quantifier is filled either by the label, or by the label of another quantifier, which in turn must have the label directly

or indirectly in its body. More formal definitions are presented in Copestake *et al.* (2005a).

As they were designed for, MRSes can be straightforwardly encoded in typed feature structures in a type *mrs* with three attributes, HOOK, RELS and HCONS. HOOK contains the top handle (handles are encoded using structure sharing), RELS contains a list of EPs which themselves bear LBL (label) and different ARG1–ARGN attributes (quantifiers have a RESTR restrictor and a BODY attribute).

The ARG role features represent argument relations numbered according to their relative obliqueness. Basic predicate-argument structure is expressed by co-indexation of the argument's inherent variable (or, in the case of propositional or scopal arguments, its label) with the appropriate role feature of the predicate. Variables are either individual (x) or event (e) variables.

The HCONS attribute is again list-valued, with structured qeq constraints of type *qeq* with attributes HARG and LARG[10].

The MRS in Figure 9.8 is an example taken from Copestake *et al.* (2005a) of the sentence

Every dog probably sleeps.

$$\langle h1, \{h2 : \text{every}(x, h4, h5), h6 : \text{dog}(x), h7 : \text{probably}(h8), h9 : \text{sleep}(x)\},$$
$$\{h1 =_q h7, h4 =_q h6, h8 =_q h9\}\rangle$$

$$\begin{bmatrix} \textit{mrs} \\ \text{HOOK} \begin{bmatrix} \textit{hook} \\ \text{GTOP} \ \boxed{1}\textit{handle} \\ \text{LTOP} \ \boxed{7}\textit{handle} \end{bmatrix} \\ \text{RELS} \left\langle \begin{bmatrix} \textit{\_every\_rel} \\ \text{LBL} \ \boxed{2}\textit{handle} \\ \text{ARG0} \ \boxed{3}\textit{ref-ind} \\ \text{RSTR} \ \boxed{4}\textit{handle} \\ \text{BODY} \ \boxed{5}\textit{handle} \end{bmatrix}, \begin{bmatrix} \textit{\_dog\_rel} \\ \text{LBL} \ \boxed{6}\textit{handle} \\ \text{ARG0} \ \boxed{3} \end{bmatrix}, \begin{bmatrix} \textit{\_probably\_rel} \\ \text{LBL} \ \boxed{7} \\ \text{ARG1} \ \boxed{8} \end{bmatrix}, \begin{bmatrix} \textit{\_sleep\_rel} \\ \text{LBL} \ \boxed{9} \\ \text{ARG1} \ \boxed{3} \end{bmatrix} \right\rangle \\ \text{HCONS} \left\langle \begin{bmatrix} \textit{qeq} \\ \text{HARG} \ \boxed{1} \\ \text{LARG} \ \boxed{7} \end{bmatrix}, \begin{bmatrix} \textit{qeq} \\ \text{HARG} \ \boxed{4} \\ \text{LARG} \ \boxed{6} \end{bmatrix}, \begin{bmatrix} \textit{qeq} \\ \text{HARG} \ \boxed{8} \\ \text{LARG} \ \boxed{9} \end{bmatrix} \right\rangle \end{bmatrix}$$

Figure 9.8: MRS for 'Every dog probably sleeps'

RMRS is a modification of MRS that allows for partial analyses, taking into account the fact that several natural language processors could contribute to refined constraints on an initially underspecified semantic representation, the maximally

[10] Although RELS and HCONS are set-valued in the formalism, implementations typically represent them in (difference) lists, as most of the modern typed feature structure implementations do not support sets for efficiency reasons.

specific representation being equivalent to the MRS representation (as they are e.g. produced by deep parsing).

To this aim, the following modifications were made from MRS

- A concept of variable equality is added, i.e., it is possible to set different variables equal as soon as constraints that justify this have been computed during processing. In the MRS version, variables had to be either equal or distinct from the beginning.
- An explicit naming convention for relation (EP) names is added that allows to infer e.g. part-of-speech types (introduced as minimally specified RMRS by a PoS tagger) from the relation names.
- Predicate arguments that were fixed in MRS are made relations to allow for variable arity of predicate arguments ('Parsons style'), e.g. instead of $l2 : on(e', e, y)$ in MRS, one would write $l2 : on(e'), \mathrm{ARG1}(l2, e), \mathrm{ARG2}(l2, y)$.
- Implicit conjunction of elementary predications (expressed with identical labels in MRS) is made explicit through introduction of a special (non-lexical) relation CONJ (for conjunction). Additional in-g (for *in group*) relations express this membership in a conjunctively connected group of EPs.
- Finally, some morpho-syntactic features such as tense, gender, number are added as additional constraints to variables.

These additions are reflected in the RMRS DTD (DTD Appendix, page 290). The `surface` attributes in `rmrs` and `ep` elements contain related input text added for illustration in the following RMRS example for the sentence 'Every dog probably sleeps'.

```
<?xml version="1.0"?>
<rmrs-list>
  <rmrs cfrom="0" cto="24" surface="Every dog probably sleeps">
    <label vid="1"/>
    <ep cfrom="0" cto="24" surface="Every dog probably sleeps">
      <gpred>prop-or-ques_m_rel</gpred>
      <label vid="1"/>
      <var sort="e" vid="2" tense="present"/>
    </ep>
    <ep cfrom="0" cto="4" surface="Every">
      <realpred lemma="every" pos="q"/>
      <label vid="6"/>
      <var sort="x" vid="7" pers="3" num="sg"/>
    </ep>
    <ep cfrom="6" cto="8" surface="dog">
      <realpred lemma="dog" pos="n" sense="1"/>
      <label vid="10"/>
      <var sort="x" vid="7" pers="3" num="sg"/>
```

```
</ep>
<ep cfrom="10" cto="17" surface="probably">
  <realpred lemma="probable" pos="a" sense="1"/>
  <label vid="11"/>
  <var sort="u" vid="13"/>
</ep>
<ep cfrom="19" cto="24" surface="sleeps">
  <realpred lemma="sleep" pos="v" sense="1"/>
  <label vid="14"/>
  <var sort="e" vid="2" tense="present"/>
</ep>
<rarg>
  <rargname>MARG</rargname>
  <label vid="1"/>
  <var sort="h" vid="3"/>
</rarg>
<rarg>
  <rargname>RSTR</rargname>
  <label vid="6"/>
  <var sort="h" vid="8"/>
</rarg>
<rarg>
  <rargname>BODY</rargname>
  <label vid="6"/>
  <var sort="h" vid="9"/>
</rarg>
<rarg>
  <rargname>ARG1</rargname>
  <label vid="11"/>
  <var sort="h" vid="12"/>
</rarg>
<rarg>
  <rargname>ARG1</rargname>
  <label vid="14"/>
  <var sort="x" vid="7" pers="3" num="sg"/>
</rarg>
<hcons hreln="qeq">
  <hi>
    <var sort="h" vid="3"/>
  </hi>
  <lo>
    <label vid="11"/>
  </lo>
</hcons>
<hcons hreln="qeq">
  <hi>
    <var sort="h" vid="8"/>
  </hi>
  <lo>
```

```
        <label vid="10"/>
      </lo>
    </hcons>
    <hcons hreln="qeq">
      <hi>
        <var sort="h" vid="12"/>
      </hi>
      <lo>
        <label vid="14"/>
      </lo>
    </hcons>
  </rmrs>
</rmrs-list>
```

The AVM-like notation format in Figure 9.9 will be used in the following RMRS examples. It is more compact and similar to MRS in omitting variables *etc*, and gives a better readable overview of the RMRS structure. Similarly to typed feature structures, square brackets indicate a conjunction of constraints, in this case handle and arguments for elementary predications.

$$\left[\begin{array}{ll}
\textsc{text} & \text{Every dog probably sleeps} \\
\textsc{top} & h1 \\
\textsc{rels} & \left\{\begin{array}{l}
\left[\begin{array}{ll} \textit{prop-or-ques_m_rel} & \\ \textsc{lbl} & h1 \\ \textsc{arg0} & e2 \\ \textsc{marg} & h3 \end{array}\right] \left[\begin{array}{ll} \textit{_every_q} & \\ \textsc{lbl} & h6 \\ \textsc{arg0} & x7 \\ \textsc{rstr} & h8 \\ \textsc{body} & h9 \end{array}\right] \left[\begin{array}{ll} \textit{_dog_n} & \\ \textsc{lbl} & h10 \\ \textsc{arg0} & x7 \end{array}\right] \\
\left[\begin{array}{ll} \textit{_probable_a} & \\ \textsc{lbl} & h11 \\ \textsc{arg0} & u13 \\ \textsc{arg1} & h12 \end{array}\right] \left[\begin{array}{ll} \textit{_sleep_v} & \\ \textsc{lbl} & h14 \\ \textsc{arg0} & e2\ ^{\text{tense=present}} \\ \textsc{arg1} & x7\ ^{\text{pers=3}}_{\text{num=sg}} \end{array}\right]
\end{array}\right\} \\
\textsc{hcons} & \{h3 \text{ qeq } h11, h8 \text{ qeq } h10, h12 \text{ qeq } h14\} \\
\textsc{ing} & \{\}
\end{array}\right]$$

Figure 9.9: Human-readable RMRS notation that will used in this thesis

CONJ is always assumed as relation for the IN-G elements. In the Heart of Gold system, there is also an interactive visualization facility for RMRS using HTML and JavaScript that helps to quickly find and inspect variable occurrences through colors and highlighting (not visible here on paper).

The HTML and JavaScript code is generated directly from the RMRS XML document using an XSLT stylesheet. The LaTeX code of the AVM-/MRS-like visualizations in our thesis is generated analogously via a XSLT. Although in RMRS, the concept of character position of a word in the original input text is crucial for

identifying, combining[11] and sorting the EPs, this property is omitted (for space reason) in the AVM-like notation we will use from now on.

9.5 Integrated NLP Components

Table 9.1 lists the NLP components that have been integrated into the Heart of Gold so far, ordered by their suggested depth (middle table column), a parameter assigned to each component's default configuration setting that can be altered according to specific configuration needs.

Component	NLP Type	Depth	Languages	Implementation
JTok	tokenizer	10	de, en, it,...	Java
ChaSen	Japanese morph.	10	ja	C
TnT	statistical tagger	20	de, en,...	C
Treetagger	statistical tagger	20	en, de, es, it,...	C
Chunkie	stat. chunker	30	de, en,...	C
ChunkieRMRS	chunk RMRSes	35	de, en	XSLT, SDL
LingPipe	statistical NER	40	en, es,...	Java
SDL	sub-architectures		...	SDL/Java
Sleepy	shallow parser	40	de	OCaml
SProUT	shallow NLP	40	de, el, en, ja,...	Java
Corcy	coref resolver	45	en	Python
RASP	shallow NLP	50	en	C, Lisp
PET	HPSG parser	100	de, el, en, ja,...	C, C++, Lisp
RMRSmerge	RMRS merger	110	de, en,...	XSLT, SDL

Table 9.1: Integrated NLP components

In the following sections, we will describe only the most important most and interesting components to give a representative overview, ordered by their NLP task type (roughly corresponding also to their suggested default depth).

9.5.1 Tokenization, Word and Sentence Segmentation

The main purpose of tokenizer components is to provide a common tokenization for those modules that require it. Normally, Unicode character counts (or positions) are taken as the least common unit, as they are independent of different concepts of what constitutes a token. The character counts also form the basis for combining standoff and RMRS annotations. Because tokenization is more or less an auxiliary task and only used by a few components, there is no special conversion to the RMRS format as it exists for most other NLP tasks.

[11] In Section 9.5.7.5, we will show how multiple RMRSes generated by different components can be combined.

9.5.1.1 JTokModule

JTok, developed at DFKI by Jörg Steffen, is used for the purpose of tokenization and sentence boundary recognition for European languages. Currently, English, German and Italian are supported; it can be easily adapted to new languages by copying and adapting XML configuration files.

JTok is implemented in Java and integrated directly as Java module in Heart of Gold, inheriting from the `Module` class. The first part of the configuration file contains 4 configuration settings that are obligatory for all modules, `name` (which is used also as identifier for the produced XML output annotations), `depth` (integer), ISO 639 two-letter `language` code indicating the language the configured module supports (for each language, a separate configuration file is required), and the `rootelement` name of the XML output annotation.

The remaining configuration settings are specific for JTok, and in this case only consist of the path to the JToK configuration file (the configuration per language specifying resources *etc* coming with the JTok distribution).

A sample configuration file for English (conf/en/jtok.cfg) follows.

```
module.name=JTok
module.depth=10
module.language=en
module.rootelement=jtok
#
# config file for JTok API
jtok.configfile=components/jtok/conf/jtok.cfg
```

In Heart of Gold, the native XML format that JTok generates is employed. The output DTD (DTD Appendix, page 289) reflects the three structuring elements that are recognized by JTok, paragraphs (e.g. line breaks from original input text), text units (e.g. sentences), and tokens (words, numbers, punctuation).

9.5.1.2 ChasenModule

ChaSen (Asahara and Matsumoto, 2000) (default depth: 10) performs morphological analysis and word boundary recognition for Japanese. The module runs the open source ChaSen system and, through the flexible ChaSen output configuration mechanism, the output format of ChasenModule directly conforms to the PET input chart DTD (cf. DTD Appendix on page 292 and description of the PetModule in Section 9.5.5).

ChasenModule is a good example for compatibility problems of external components that are solved in the module code that wraps an external NLP component. In this case, the problem is caused by the inability of ChaSen to return Unicode character positions that are required for standoff annotation combination. By convention, the start and end character positions for each token are contained in the CSTART and CEND attributes that refer to the original input text. Unicode is

```
<pet-input-chart>
  <metadata>
    <id>
      <entry name="created" value="Tue, 25 Apr 2006 11:25:55 +0200"/>
      <entry name="processingtime" value="71 milliseconds"/>
      <entry name="sessionid" value="session1"/>
      <entry name="acid" value="collection1"/>
      <entry name="component" value="ChaSen"/>
      <entry name="diagnosis" value="OK"/>
      <entry name="empty" value="false"/>
    </id>
    <conf>
      <entry name="chasen.options" value="-r conf/.chasenrc-xml"/>
      <entry name="module.depth" value="10"/>
      <entry name="module.rootelement" value="pet-input-chart"/>
      <entry name="chasen.inputencoding" value="EUC-JP"/>
      <entry name="chasen.libs" value="components/chasen/lib"/>
      <entry name="module.language" value="ja"/>
      <entry name="chasen.inputannotation" value="rawtext"/>
      <entry name="chasen.outputencoding" value="EUC-JP"/>
      <entry name="chasen.binary" value="components/chasen/bin/runchasen"/>
      <entry name="module.name" value="ChaSen"/>
    </conf>
  </metadata>
  <w id="CHA0" cstart="0" cend="2">
    <surface>彼女</surface>
    <pos tag="名詞-代名詞-一般+n-n" prio="1269"/>
  </w>
  <w id="CHA1" cstart="2" cend="3">
    <surface>は</surface>
    <pos tag="助詞-係助詞+n-n" prio="0"/>
  </w>
  <w id="CHA2" cstart="3" cend="6">
    <surface>ボタン</surface>
    <pos tag="名詞-一般+n-n" prio="3378"/>
  </w>
  <w id="CHA3" cstart="6" cend="7">
    <surface>を</surface>
    <pos tag="助詞-格助詞-一般+n-n" prio="0"/>
  </w>
  <w id="CHA4" cstart="7" cend="10">
    <surface>付ける</surface>
    <pos tag="動詞-自立+一段-基本形" prio="1884"/>
  </w>
</pet-input-chart>
```

Figure 9.10: Output of ChasenModule in PET input chart format

the only sensible character set for multilingual frameworks such as Heart of Gold because each character has equal length (although different encodings exist).

While Heart of Gold expects these character positions to be Unicode character counts, ChaSen (at least in the version integrated in Heart of Gold) outputs only byte offsets related to the EUC-JP 8 bit multi-byte encoding it uses internally, even if Heart of Gold takes over the transcoding between EUC-JP and Unicode. Therefore, the ChasenModule contains code that corrects the character counts according to knowledge about the encodings available from standard Java Unicode classes.

9.5.2 Part-of-Speech Tagging

Statistical Part-of-Speech (PoS) taggers are used in deep-shallow integration (1) to give lexical items in the deep lexicon a higher priority on the deep parser's chart

that correspond to the most probable reading determined by the statistical tagger, (2) to guess the word class of unknown words (if they cannot be determined as named entity by a named entity recognition component), using a generic lexicon entry inserted into the deep parser's chart.

The strategy employed in the Heart of Gold implementation is the same as described in Chapter 8 for WHITEBOARD except that the new PET input chart XML format is used (cf. description in Section 9.5.5 below). In particular, similarly to WHITEBOARD, it is possible to insert multiple analyses (readings) for a word with priorities (e.g. taken from the trained model) into the deep parser's chart where the parser can resolve these ambiguities with the help of the deep grammar. We present a single example for an integrated tagger, TnT, here. Further taggers have been integrated as will be listed in Section 9.5.6.

9.5.2.1 TnTModule

TnT is a statistical, trigram-based part-of-speech tagger (Brants, 2000). Statistical models have been trained for German on NEGRA (Skut *et al.*, 1998), and for English on the Penn Treebank (Marcus *et al.*, 1994). Further languages could be and have already been trained, e.g. Portuguese by a group at the University of Lisbon, and also additional corpora for English and German.

TnTModule (default depth: 20) sends text tokenized using the JTokModule to TnT and converts the native (non-XML) output of TnT to both an isomorphic XML format and the PET input chart format (Section 9.5.5.2). The TnTModule DTD is shown in the DTD Appendix (page 289).

Configuration file conf/en/tnt.cfg for English with Penn Treebank model:

```
module.name=TnT
module.depth=20
module.language=en
module.rootelement=tnt
#
# path to tnt startscript
tnt.script=components/tnt/scripts/tnt.sh
# command line options for tnt
tnt.options=-z20 -v0 models/wsj
# input encoding
tnt.inputencoding=ISO-8859-1
# output encoding
tnt.outputencoding=ISO-8859-1
# name of generated PET input chart XML annotation
tnt.piXMLoutputannotation=TnTpiXML
#
# root element name of PET input chart XML annotation
tnt.piXMLrootelement=pet-input-chart
```

An example of the TnT output XMLified by TnTModule follows. An example of the additionally generated PET input chart format is illustrated in Sec-

tion 9.5.5.2.

```
<tnt>
  <metadata>...</metadata>
  <tokens>
    <w str="How" cstart="0" cend="2">
      <p pos="WRB" p="1.000000e+00"/>
    </w>
    <w str="cold" cstart="4" cend="7">
      <p pos="NN" p="6.513877e-01"/>
      <p pos="JJ" p="3.486123e-01"/>
    </w>
    <w str="should" cstart="9" cend="14">
      <p pos="MD" p="1.000000e+00"/>
    </w>
    <w str="a" cstart="16" cend="16">
      <p pos="DT" p="1.000000e+00"/>
    </w>
    <w str="refrigerator" cstart="18" cend="29">
      <p pos="NN" p="1.000000e+00"/>
    </w>
    <w str="be" cstart="31" cend="32">
      <p pos="VB" p="1.000000e+00"/>
    </w>
    <w str="?" cstart="33" cend="33">
      <p pos="." p="1.000000e+00"/>
    </w>
  </tokens>
</tnt>
```

9.5.3 Chunking and Shallow Parsing

9.5.3.1 ChunkieModule

Chunkie (Skut and Brants, 1998) is a statistical, trigram-based chunker built upon TnT. It potentially can deliver recursive chunks (in contrast to most other chunkers). ChunkieModule (default depth: 30) uses JTok tokenization as input and returns XML output of Chunkie chunk analyses including the selected PoS tags from TnT. As for TnT, models are available e.g. for English (Penn Treebank; Marcus *et al.* 1994) and German (NEGRA; Skut *et al.* 1998).

The output DTD of the module is specified in the DTD Appendix on page 290, an example can be found on page 71.

RMRS construction of Chunkie output is implemented in the separate ChunkieRMRS module we will describe later in Section 9.5.7.4. The ChunkieRMRS module also enriches Chunkie output with morphological information from *SProUT* analyses.

The ChunkieRMRS annotation can be used as fall-back annotation in case deep parsing fails, but Chunkie output is not fed into the deep parser because of partial

chunk incompatibilities with the HPSG grammars (cf. also Section 8.7).

9.5.3.2 RaspModule

RASP is a robust statistical parser for English developed in C and LISP on the basis of the ANLT system (Briscoe and Carroll, 2002). RASP delivers RMRS output of medium NLP depth, including (partial) predicate-argument structure. RASP uses its own tokenization, morphology and named entity recognition, thus is employed in the Heart of Gold integration as a black box.

RASP has an integrated, optional RMRS output facility. RaspModule (default depth: 50) uses this method to generate and forward the RMRS structures that are merely augmented with `<metadata>` information by the wrapper code.

As we illustrate for the sentence 'John gave Mary the book' in Figure 9.11, RASP correctly assigns predicate-argument structure, which is also why RASP is a good shallow fall-back component in case deep parsing fails.

$$\left[\begin{array}{ll}
\text{TEXT} & \text{John gave Mary the book} \\
\text{TOP} & h32 \\
\text{RELS} & \left\{\begin{array}{l}
\left[\begin{array}{ll} prpstn_m_rel & \\ \text{LBL} & h32 \\ \text{ARG0} & h35 \end{array}\right]
\left[\begin{array}{ll} proper_q_rel & \\ \text{LBL} & h1 \\ \text{ARG0} & x2 \\ \text{RSTR} & h4 \\ \text{BODY} & h5 \end{array}\right]
\left[\begin{array}{ll} named_rel & \\ \text{LBL} & h3 \\ \text{ARG0} & x2 \\ \text{CARG} & \text{john} \end{array}\right]
\left[\begin{array}{ll} give_v & \\ \text{LBL} & h6 \\ \text{ARG0} & e7\ \text{tense=past} \\ \text{ARG1} & x2 \\ \text{ARG2} & x9 \end{array}\right] \\
\left[\begin{array}{ll} appos_rel & \\ \text{LBL} & h25 \\ \text{ARG0} & e26 \\ \text{ARG1} & x9 \\ \text{ARG2} & x14 \end{array}\right]
\left[\begin{array}{ll} proper_q_rel & \\ \text{LBL} & h8 \\ \text{ARG0} & x9 \\ \text{RSTR} & h11 \\ \text{BODY} & h12 \end{array}\right]
\left[\begin{array}{ll} named_rel & \\ \text{LBL} & h10 \\ \text{ARG0} & x9 \\ \text{CARG} & \text{mary} \end{array}\right]
\left[\begin{array}{ll} the_q & \\ \text{LBL} & h13 \\ \text{ARG0} & x14 \\ \text{RSTR} & h20 \\ \text{BODY} & h21 \end{array}\right] \\
\left[\begin{array}{ll} book_n & \\ \text{LBL} & h15 \\ \text{ARG0} & x14 \end{array}\right]
\end{array}\right\} \\
\text{HCONS} & \{h35\ \text{qeq}\ h6, h4\ \text{qeq}\ h3, h11\ \text{qeq}\ h10, h20\ \text{qeq}\ h15\} \\
\text{ING} & \{h25\ \text{ing}\ h10\}
\end{array}\right]$$

Figure 9.11: RASP analysis of 'John gave Mary the book'

9.5.4 Named Entity Recognition and Information Extraction

9.5.4.1 SproutModule

The *SProUT* system (Drożdżyński *et al.*, 2004), a flexible multilingual, shallow processing component that combines finite state and typed feature structure technology and includes morphologic resources and named entity grammars for ten

languages, as described in Chapter 7, is integrated as a Heart of Gold module as well.

SProUT plays an important role in the multilingual deep-shallow integration scenarios, mainly for general and domain-specific named entity recognition for (1) preparation of generic named entity lexical entries for the deep parser through the PET input chart format (cf. Section 9.5.5.2) and (2) information extraction through generating finer-grained structured representations of recognized named entities than are provided by the general-purpose HPSG grammar, and passing them by as additional RMRS structures to the applications.

As *SProUT* is implemented in Java, SproutModule (default depth: 40) runs an instance of the *SProUT* interpreter via the *SProUT* runtime API with a configured grammar and other resources within the same virtual machine. Through the flexible configuration, *SProUT* can also be used in Heart of Gold for other purposes than named entity recognition, cf. also Section 9.5.7.4 for an example.

Further applications for which already appropriate grammars and resources exist are e.g. morphological analysis for various languages including compound segmentation for German and Dutch, sentence boundary recognition, anaphora resolution, shallow parsing and domain-specific template-based information extraction. Thanks to the flexible formalism and the modular framework, further NLP tasks could be implemented as well.

Automatic stylesheet generation at compile time A special feature and advantage of the *SProUT* integration in Heart of Gold is the automatic generation of transformation stylesheets from the declaratively defined output specifications of *SProUT* grammars we will describe in this section. This feature will also play an important role for the ontology integration we will describe in Section 9.7.

While many shallow named entity recognition (NER) systems come with a small, fixed and hard-wired set of recognized named entity types (e.g. those defined by MUC-6; Grishman and Sundheim 1996), *SProUT* allows to flexibly add new or more fine-grained output structures.

The output of a named entity grammar, a sequence of matched regular expressions over typed feature structures, is itself a typed feature structure (details and examples are explained in Chapter 7). As the output types have to be declared in the associated type hierarchy of the named entity grammars, it is possible to use this information in order to generate XSLT code at compile time. We have implemented an algorithm that generates XSLT mapping code from the TDL type definitions.

The generated stylesheets map at runtime the XML output of the *SProUT* runtime system to (1) generic HPSG lexicon entries via the PET input chart format (Section 9.5.5.2) for use in the deep parser, and to (2) RMRS structures with more structured and fine-grained, information extraction-like NER analyses that complement the deep sentence analysis with additional information that might be relevant for applications.

In the rest of this section, we will discuss an example of how this mapping works. The code generator is implemented in Java[12] and can be called via the Heart of Gold `ant` build target `generate_xsl`. It takes (per language) two files as input, the type hierarchy of the *SProUT* named entity grammar to map, efficiently encoded in a binary representation produced by `flop`, and a mapping table that maps *SProUT* named entity type names from the *SProUT* type hierarchy to generic HPSG lexicon types.

Generation of the RMRS transformation stylesheet RMRS relation names are constructed from the *SProUT* feature names plus `_rel` suffix. The entries of the mappings file are of the form

$$\text{type}_{SProUT} = \text{type}_{HPSG}$$

SProUT types not mentioned in the mappings are not mapped by the generated stylesheets. If a *SProUT* NE type is to be translated to RMRS, but not to the PET input chart format, then the RHS of the mapping must be left empty: Here is an example of such a mapping table for English; the German, Greek and Japanese versions are very similar.

```
ne-person=generic_name
ne-location=generic_name
ne-organization=generic_name
ne-product=generic_name
ne-address=generic_email
numex=generic_number
money=generic_money
duration=generic_time
span=generic_time
timex=generic_time
point=generic_time
percentage=generic_number
ne-sciencearea=generic_name
ne-prize=generic_name
ne-occupation=generic_name
ne-tech=generic_name
ne-technology=generic_name
ne-project=generic_name
ne-event=generic_name
ne-term=generic_mass_count_noun
```

An output type (part of the *SProUT* named entity grammars) is defined in TDL as follows, additional attributes are inserted via inheritance from the *enamex* supertype and further supertypes.

[12] Class `de.dfki.lt.hog.util.SproutRmrsTransformerGenerator`

```
ne-product := enamex & [PRODUCT-NAME string,
                        PRODUCT-MODEL string,
                        PRODUCT-VARIANT string].
```

where enamex has, besides the SURFACE, CSTART and CEND attributes for original text string, character start and end positions, also VARIANT and DESCRIPTOR string-valued attributes for alternatives and description text.

All these attribute values are, if they contain information more specific than their default values, mapped to corresponding RMRS relations connected with the identified named entity by the generated XSLT code.

The generated RMRS transformation stylesheets comprise e.g. 8700 lines of XSLT code (essentially a big case statement over the 20 different named entity output types of the *SProUT* grammars) for the English NER grammar enhanced with LT WORLD ontology information (Section 9.7), and approx. 4600 lines for the standard NER grammar.

A fragment of the generated RMRS transformation code is exemplified in the XSLT Appendix on page 295.

Generation of the PET input chart transformation stylesheet A separate stylesheet is generated at compile time for producing PET input chart entries for recognized named entities, and mapped to generic HPSG types for named entities (right hand side of the mapping table above), including span information. In the current mapping, features such as PRODUCT-NAME are omitted, as they have no correspondence in the HPSG grammar. In principle, also richer information could be transported by means of the flexible feature-path-value mechanism (cf. Section 9.5.5.2).

Transformation at runtime To illustrate the transformed analyses, we show a simple example here for the sentence 'The first user liked the Nokia 6810.' where *SProUT* recognizes a named entity 'Nokia 6810'. A more comprehensive example employing LT WORLD ontology instance information will be shown in Section 9.7. The *SProUT* output ('SProUTput') DTD is defined in Chapter 7.

The following output is produced by *SProUT* for the sample sentence.

```
<SPROUTPUT>
  <DISJ id="DI0">
    <MATCHINFO id="MI0" rule="en_cell_phone_gazetteer"
               cstart="25" cend="34">
      <FS type="ne-product">
        <F name="VARIANT">
          <FS type="*top*"/>
        </F>
        <F name="SURFACE">
          <FS type="string"/>
        </F>
```

```
        <F name="CSTART">
          <FS type="string"/>
        </F>
        <F name="CEND">
          <FS type="string"/>
        </F>
        <F name="PREPOSITIONS">
          <FS type="*list*"/>
        </F>
        <F name="DESCRIPTOR">
          <FS type="string"/>
        </F>
        <F name="NECEND">
          <FS type="string"/>
        </F>
        <F name="NECSTART">
          <FS type="string"/>
        </F>
        <F name="PRODUCT-VARIANT">
          <FS type="string"/>
        </F>
        <F name="PRODUCT-MODEL">
          <FS type="string"/>
        </F>
        <F name="PRODUCT-NAME">
          <FS type="&quot;Nokia 6810&quot;"/>
        </F>
      </FS>
    </MATCHINFO>
  </DISJ>
</SPROUTPUT>
```

The automatically generated stylesheet `en_types-sprout2rmrs.xsl` mentioned in the SproutModule configuration file below transforms the *SProUT* output into the following RMRS.

```
<rmrs-list>
  <rmrs cfrom="25" cto="34" reading="0" surface="Nokia 6810">
    <label vid="100"/>
    <ep cfrom="25" cto="34" surface="Nokia 6810">
      <gpred>ne-product_rel</gpred>
      <label vid="100"/>
      <var sort="x" vid="100"/>
    </ep>
    <rarg>
      <label vid="100"/>
      <rargname>CARG</rargname>
      <constant>Nokia 6810</constant>
    </rarg>
```

```
    <ep cfrom="25" cto="34" surface="Nokia 6810">
      <gpred>product-name_rel</gpred>
      <label vid="110"/>
      <var sort="x" vid="110"/>
    </ep>
    <rarg>
      <label vid="110"/>
      <rargname>CARG</rargname>
      <constant>Nokia 6810</constant>
    </rarg>
    <rarg>
      <label vid="110"/>
      <rargname>ARG1</rargname>
      <var sort="x" vid="100"/>
    </rarg>
  </rmrs>
</rmrs-list>
```

The automatically generated stylesheet `en_types-sprout2pic.xsl` mentioned in the SproutModule configuration file below transforms the *SProUT* output into the following PET input chart entry.

```
<pet-input-chart>
  <w id="SPR1.1" cstart="25" cend="34" constant="yes" prio="0.5">
    <surface>Nokia 6810</surface>
    <typeinfo id="TIN1.1" baseform="no">
      <stem>$generic_name</stem>
    </typeinfo>
  </w>
</pet-input-chart>
```

The whole *SProUT* runtime subsystem for use in the Heart of Gold SproutModule is generated from the *SProUT* sources using a single build target `sprout-2hog` that is part of the automatic build system for *SProUT* (Schäfer and Beck, 2006) we will describe in Section 7.8. The generated subsystem comprises the runtime jar, four named entity grammars for Greek, English, German and Japanese, and eight ChunkieRMRS cascade grammars for German and English (cf. Section 9.5.7.4). Additional *SProUT* resources can be easily included by extending the `sprout2hog` target.

The following configuration file conf/en/sprout.cfg contains the SproutModule configuration for the English named entity grammar including the automatically generated mapping stylesheets.

```
module.name=Sprout
module.depth=40
module.language=en
module.rootelement=SPROUTPUT
#
```

```
# config file for SProUT runtime API
sprout.configfile=components/sprout/Project/de.cfg
#
# stylesheet for transformation of FS-XML to RMRS
sprout.stylesheet=xsl/rmrs/de_types-sprout2rmrs.xsl
#
# feature path to output structure
# if undefined, the root FS (including IN and OUT) is returned
# feature separator in the path can be . or | (as in TFS API)
sprout.outputpath=
#
# name of raw text input annotation for Sprout
sprout.inputtextannotation=rawtext
#
# name of feature structure output annotation
sprout.outputfsannotation=SproutFS
#
# --- The following configurations are for PET input chart mode only
# The subsequent settings are ignored if sprout.output4pic is unset
#
# name of output annotation for PET input chart (pic) format
# no pic annotation is generated if this value is omitted
sprout.output4pic=SProUTpiXML
#
# stylesheet for transformation of Sproutput to PET XML input chart
# (ignored if sprout.output4pic is not set)
sprout.stylesheet4pic=xsl/pic/de_types-sprout2pixml.xsl
#
# --- End of configurations for PET input chart mode ---
```

Using *SProUT* Morphology analyses In this section, we describe how *SProUT* resources other than the above described named entity grammars can be utilized in Heart of Gold through the generic SproutModule. We take morphology as example.

Although the morphological analysis of *SProUT* is utilized in most of the existing named entity grammars, the morphological analysis feature structures are normally not copied to the output, as only matched named entities should be part of the output structure of a named entity grammar.

Using a single *SProUT* (*XTDL*) grammar rule, it is possible to make the morphology analyses also available in Heart of Gold[13]:

```
morphcopy :> morph & #1 -> #1.
```

As *morph* inherits from *token*, the output also includes tokenization information. We exemplify the analysis for the input sentence 'Wie geht es dem Papst?', transcribed to AVM notation below.

[13] Alternatively, copying morphology information to the output can also be configured in the *SProUT* grammar-specific configuration file as an interpreter option.

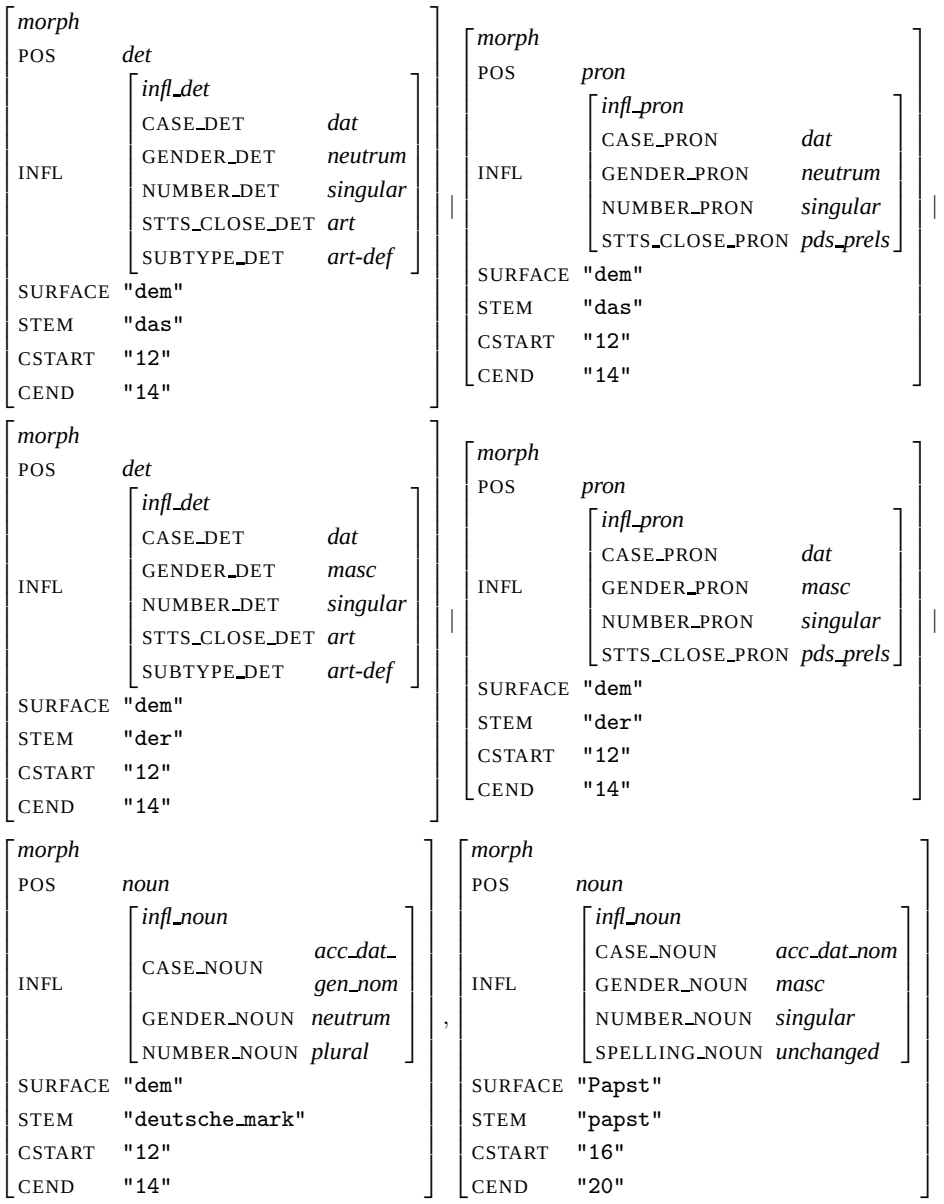
It has to be noted that the morphology output is not disambiguated, but reflects the complete content of the *SProUT* morphology and lexicon component for German, in this case based on MMORPH (Petitpierre and Russell, 1995).

```
[ morph
  POS     adverb
  INFL    [ infl_adverb
            SPELLING_ADVERB   unchanged
            STTS_OPEN_ADVERB  adv ]
  SURFACE "Wie"
  STEM    "wie"
  CSTART  "0"
  CEND    "2" ]
|
[ morph
  POS     conj
  INFL    [ infl_conj
            SPELLING_CONJ     unchanged
            STTS_CLOSE_CONJ   kokom ]
  SURFACE "Wie"
  STEM    "wie"
  CSTART  "0"
  CEND    "2" ] ,

[ morph
  POS     verb
  INFL    [ infl_verb
            MODE_VERB         imperative
            NUMBER_VERB       plural
            PERSON_VERB       p2
            SPELLING_VERB     unchanged
            STTS_OPEN_VERB    vvimp
            TENSE_VERB        present
            VFORM_VERB        fin ]
  SURFACE "geht"
  STEM    "gehen"
  CSTART  "4"
  CEND    "7" ]
|
[ morph
  POS     verb
  INFL    [ infl_verb
            MODE_VERB         indicative
            NUMBER_VERB       plural
            PERSON_VERB       p2
            SPELLING_VERB     unchanged
            STTS_OPEN_VERB    vvfin
            TENSE_VERB        present
            VFORM_VERB        fin ]
  SURFACE "geht"
  STEM    "gehen"
  CSTART  "4"
  CEND    "7" ]
|

[ morph
  POS     verb
  INFL    [ infl_verb
            MODE_VERB         indicative
            NUMBER_VERB       singular
            PERSON_VERB       p3
            SPELLING_VERB     unchanged
            STTS_OPEN_VERB    vvfin
            TENSE_VERB        present
            VFORM_VERB        fin ]
  SURFACE "geht"
  STEM    "gehen"
  CSTART  "4"
  CEND    "7" ] ,

[ morph
  POS     pron
  INFL    [ infl_pron
            CASE_PRON         acc_nom
            GENDER_PRON       neutrum
            NUMBER_PRON       singular
            PERSON_PRON       p3
            STTS_CLOSE_PRON   pper ]
  SURFACE "es"
  STEM    "es"
  CSTART  "9"
  CEND    "10" ] ,
```



Similarly, output of other *SProUT* components such as coreference matcher or compound segmentation could be made available in Heart of Gold, using SproutModule plus an appropriate configuration file for a *SProUT* grammar instance.

9.5.4.2 LingPipeModule

LingPipe[14] is a statistical named entity recognition system including coreference resolution with existing trained models for various languages (incl. English, German, Spanish) and special domains such as genomes. LingPipeModule (default

[14] Carpenter (2005), `http://www.alias-i.com/lingpipe/`

depth: 40) runs the original LingPipe Java distribution that is available as open source code, and transforms the native MUC-like XML output format (cf. Chapter 7) into both the RMRS and the PET input chart format using XSLT available via the Heart of Gold `TransformationService`. This is similar to SproutModule, but instead of automatically generated transformation stylesheets, simple fixed stylesheets for the few MUC named entity types are provided.

9.5.5 Deep Parsing: The PetModule

9.5.5.1 Introduction

PET is a highly efficient deep parser for HPSG grammars. It has been developed in C and C++ at Saarland University and DFKI (Callmeier, 2000). In addition to the version that has been used in the WHITEBOARD integration, PET has been augmented with the RMRS output module from LKB (Copestake, 2002) jointly developed by Stephan Oepen, Ann Copestake, Dan Flickinger, Ulrich Callmeier and Bernd Kiefer. Through this extension, PET can natively output RMRS markup.

Furthermore, Bernd Kiefer added the PET input chart XML parser for incorporating output of shallow preprocessors that we will describe below. The PET input chart replaces the WHAT-based programming interface described in Chapter 8, providing more flexibility by moving the annotation transformation out of the PET system into the architecture. In this sense, PET becomes an NLP component as the other components in Heart of Gold, with XML input and output interfaces.

Because one of the envisaged improvements of Heart of Gold over WHITEBOARD was to add multilinguality support, the whole module code that wraps the core PET component in the architecture has been made highly configurable. Similar to other modules such as SproutModule, configuration is separated into two steps. One is the configuration for the resource (i.e., the grammar) with depending settings for the parser, and one is configuration for the module instance, including pre- and post-processing stylesheets, encoding settings, and the link to the resource-specific configuration file.

Following is a sample configuration for PetModule with the English Resource Grammar (ERG; Flickinger 2002). Dependencies of TnT and *SProUT* preprocessing via PET input chart are configured as well as RMRS output (maximally 3 readings sorted according to a parse selection model) including fragments (in case deep parsing did not find a fully spanning analysis) that are sorted according to their lengths and cut to the maximally 5 RMRSes with the longest span. Configuration details are explained as comments in the following example.

```
module.name=PET
module.depth=100
module.language=en
module.rootelement=pet
#
# path to cheap binary
```

```
pet.binary=components/pet/bin/cheap
#
# additional library search path for cheap
pet.libs=components/pet/lib
#
# working directory (where the grammar is)
pet.grammardir=components/pet/german
#
# prefix for grammar file
pet.grammarprefix=german
#
# command line options for cheap
pet.options=-tok=xml_counts -mrs=xml -default-les -limit=70000 \
    -results=3 -packing -partial
#
# character set encoding for PET input
pet.inputencoding=UTF-8
#
# character set encoding for PET output
pet.outputencoding=UTF-8
#
# input annotation(s), comma-separated
# use either "rawtext" or PET XML input chart (e.g. LingPipepiXML)
# formats in accordance with pet.options
pet.inputannotation=TnTpiXML,SProUTpiXML
#
# stylesheet for XML input chart combination
pet.combinestylesheet=xsl/pic/combinepixml.xsl
#
# stylesheet for preprocessing the PET input chart
# There will be no transformation/filtering if this option is unset
pet.preprocstylesheet=xsl/pic/remove-subspan-items.xsl
#
# stylesheet for post-processing (filtering) of partial RMRSes
# return only the n longest fragments
# return all (=no stylesheet application) if unset
pet.postprocstylesheet=xsl/rmrs/extract-longest-fragment.xsl
#
# stylesheet parameter: number of fragments to return
# return all (=no stylesheet application) if unset
pet.postprocfragments=5
```

Besides ERG, many other available HPSG grammars have been integrated through specific configurations, most of them during the DEEPTHOUGHT project, e.g. German (Crysmann, 2003), Jacy for Japanese (Siegel and Bender, 2002), Greek (Kordoni and Neu, 2004) and Norwegian (Hellan *et al.*, 2004).

9.5.5.2 PET input chart format: interface between shallow preprocessing and deep parsing

The PET input chart XML format (cf. DTD Appendix on page 292) has been designed and implemented to provide a generic input format for all kinds of preprocessing components that could contribute to useful information for deep parsing. This includes not only part-of-speech tagging (with ranked readings to guide the deep parser faster to probable analyses) and named entity recognition as already realized in the WHITEBOARD integration, but also larger, arbitrary structures based on multiple tokens, also overlapping and in parallel.

Introduction of information from shallow preprocessing into the deep parser's chart is realized by specifying the input items including a character position (or count). Several components can add items with concurrent character spans, provided that the underlying 'characterization' is identical, as already discussed above. This corresponds to standoff markup unified in a single annotation, i.e., the input chart must consist of a single XML document. There is a preprocessing XSLT stylesheet shipped with Heart of Gold that concatenates multiple XML input chart documents into a single one (see below).

In addition, the PET input chart format also supports 'injection' of feature values under arbitrary feature paths. Using these, it is possible to insert chunk, phrase or subclause boundaries as well as morphological or other structured information from outside into the HPSG grammar.

Following is a sample PET input chart for the sentence 'Did Bernd Kiefer present a paper at ACL 2003?'. This is one possibility to encode the input chart, treating multi-word named entities as single, not further structured objects as it is done by TntModule and SproutModule; a second one would be to introduce `<ne>` tags referring to the `<w>` tokens for multi-token expressions such as named entities, phrases *etc*.

```
<?xml version="1.0"?>
<pet-input-chart>
  <w id="TNT0" cstart="0" cend="2">
    <surface>Did</surface>
    <pos tag="VBD" prio="1.000000e+00"/>
  </w>
  <w id="TNT1" cstart="4" cend="8">
    <surface>Bernd</surface>
    <pos tag="NNP" prio="1.000000e+00"/>
  </w>
  <w id="TNT2" cstart="10" cend="15">
    <surface>Kiefer</surface>
    <pos tag="NNP" prio="1.000000e+00"/>
  </w>
  <w id="TNT3" cstart="17" cend="23">
    <surface>present</surface>
    <pos tag="VB" prio="6.187941e-01"/>
    <pos tag="JJ" prio="2.305826e-01"/>
```

```
      <pos tag="VBP" prio="5.459771e-02"/>
      <pos tag="RB" prio="5.275342e-02"/>
      <pos tag="NN" prio="4.327213e-02"/>
    </w>
    <w id="TNT4" cstart="25" cend="25">
      <surface>a</surface>
      <pos tag="DT" prio="1.000000e+00"/>
    </w>
    <w id="TNT5" cstart="27" cend="31">
      <surface>paper</surface>
      <pos tag="NN" prio="1.000000e+00"/>
    </w>
    <w id="TNT6" cstart="33" cend="34">
      <surface>at</surface>
      <pos tag="IN" prio="1.000000e+00"/>
    </w>
    <w id="TNT7" cstart="36" cend="38">
      <surface>ACL</surface>
      <pos tag="NNP" prio="1.000000e+00"/>
    </w>
    <w id="TNT8" cstart="40" cend="43">
      <surface>2003</surface>
      <pos tag="CD" prio="1.000000e+00"/>
      <typeinfo id="TYI8" baseform="no">
        <stem>$generic_number</stem>
      </typeinfo>
    </w>
    <w id="TNT9" cstart="44" cend="44" constant="yes">
      <surface>?</surface>
      <pos tag="?" prio="1.0"/>
    </w>
    <w id="SPR2.1" cstart="4" cend="15" constant="yes" prio="0.5">
      <surface>Bernd Kiefer</surface>
      <typeinfo id="TIN2.1" baseform="no">
      <stem>$generic_name</stem>
      </typeinfo>
    </w>
    <w id="SPR3.1" cstart="36" cend="43" constant="yes" prio="0.5">
      <surface>ACL 2003</surface>
      <typeinfo id="TIN3.1" baseform="no">
      <stem>$generic_name</stem>
      </typeinfo>
    </w>
  </pet-input-chart>
```

The generated input chart consists of words tagged by TnT, concatenated (XML-wise) with named entities recognized by *SProUT*, according to the PET configuration file shown above (configuration line `pet.inputannotation`). 'piXML' stands for PET input chart XML. The stylesheet configured under `pet.combine-`

`stylesheet` combines (XML-wise concatenates) arbitrarily many input annotations specified by their annotation IDs. The stylesheet source can be found in the XSLT Appendix on page 296.

In the above example, multiple information is given for the named entities *Bernd Kiefer* and *ACL 2003*, one from the tagger TnT (`id` attributes `TNT1`, `TNT2`, `TNT7` and `TNT8`), and one from the named entity component *SProUT*, (`ids` `SPR2.1` and `SPR3.1`). Resolution of the multiple information is left to the deep parser by passing both to it via the XML input chart.

A further, optional preprocessing stylesheet may remove overlapping or conflicting readings, e.g. to override tags from the tagger that have a reliable named entity alternative. The decision which reading to eliminate could be based on confidence values assigned by the producing component. The example stylesheet on page 297 in the XSLT Appendix may illustrate how this can be done in XSLT.

The mapping to HPSG types is part of the input chart (viz. the stylesheets that generate them), with the exception of part-of-speech tags. The mapping between PoS tags and corresponding HPSG types has to be part of the PET configuration. Here is an example for a mapping from TnT PoS tag names (left) to ERG PoS types (right).

```
posmapping :=
  JJ $generic_adj
  JJR $generic_adj_compar
  JJS $generic_adj_superl
  NN $generic_sg_noun
  NN $generic_mass_noun
  NNS $generic_pl_noun
  NNPS $generic_pl_noun
  NNP $generic_name
  FW $generic_mass_noun
  RB $generic_adverb
  VB $generic_trans_verb_bse
  VBD $generic_trans_verb_past
  VBG $generic_trans_verb_prp
  VBN $generic_trans_verb_psp
  VBP $generic_trans_verb_presn3sg
  VBZ $generic_trans_verb_pres3sg.
```

All other mappings, as shown e.g. for the *SProUT* integration in Section 9.5.4.1, should take place when the PET input chart format is generated, e.g. in mapping stylesheets or auxiliary mapping tables read by the stylesheets.

The benefits of the shallow preprocessing just described in terms of increased coverage will be discussed in the evaluation section.

There is ongoing research and development under the umbrella of the DELPH-IN collaboration (cf. Section 9.12) with the goal to come up with a uniform, generalized preprocessor framework for both the LKB and the PET system. The reason for this standardization effort is that resource and application development would

benefit from providing identical deep-shallow integration behavior in both development (LKB) and runtime (PET/Heart of Gold) framework, which is currently not the case as LKB currently comes with a different shallow preprocessing interface (SPPP, cf. Section 9.11.9).

The idea is that both HPSG parsing frameworks should share a common XML format, briefly sketched in Waldron *et al.* (2006) that is based on explicit lattice as in the morpho-syntactic annotation format (MAF; Clément and Villemonte de la Clergerie 2005), and also allows for typed feature structure input instead of the set of feature path-value 'injections' into chart elements provided by the PET input chart. This would also include the input of RMRS XML which, however, can be represented as TFS and is thus only a syntactic variant.

The envisaged new format is largely just additional syntactic sugar as compared to the PET input chart format, but with the additional benefit that it could become input to both LKB and PET.

Thanks to the felicitous design of the PET system, PetModule is largely independent of the statistical parse ranking, automatically acquired lexicon extensions, the TSDB test interface (Oepen, 2001) and extensions for performance improvements such as packing. Most of these features can be configured as part of the command line options or the PET configuration file(s). As a consequence, almost all of these features can hence also be used when PET is embedded in a Heart of Gold-based application.

We finally exemplify RMRS output for the sentence 'Did Bernd Kiefer present a paper at ACL 2003?' in Figure 9.12, returned by the hybrid analysis through PetModule.

We see that both named entities occurring in the sentence, each consisting of two words, are treated as single *named_rel*s here. Later, we will see examples of how these EPs can be enriched with further structured output from a *SProUT* named entity grammar augmented with ontology information (Section 9.7).

9.5.5.3 Pre- and Post-processing Stylesheets

A very flexible mechanism in PetModule (also shown in the above configuration example) is the configurable pre- and post-processing stylesheet option. A preprocessing stylesheet can be used to introduce additional knowledge about resources or components (or configuration-related combinations thereof) and their specific integration behavior. It is e.g. possible to filter or re-rank conflicting chart elements to reduce ambiguity or correct predictable errors in components based on input context. An example has already been presented above for the case of filtering unwanted overlapping analyses from different components.

PET can also output RMRS fragments in case no full parse could be computed for the whole input sentence. Such partial output can be handled in the post-processing stylesheet. This is e.g. useful to sort and filter fragments in order to reduce the (potentially huge) output size and restrict it to the most *n* promising ones. These could by an application be treated as if they were full parses and thus

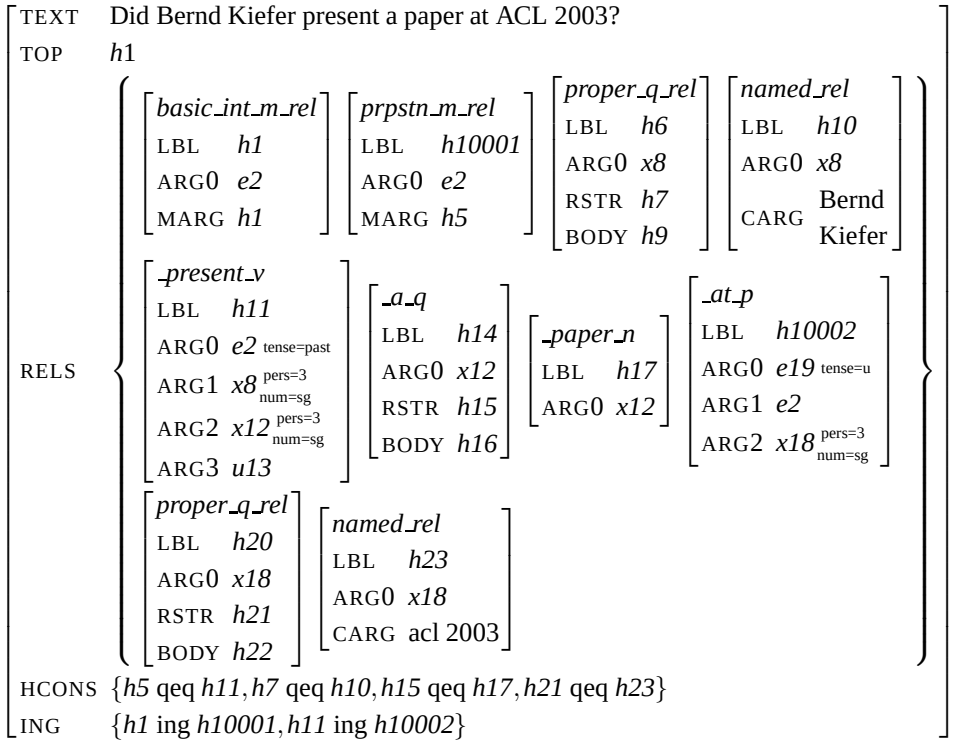


Figure 9.12: RMRS generated by hybrid parsing using PetModule

provide a further important means to increase robustness of deep parsing. A simple implementation of such a fragment filter is shown in the XSLT Appendix on page 298.

9.5.6 Further Integrated NLP Components

There are further integrated NLP components for which Heart of Gold module classes have been implemented. We will not describe them here in detail as they are currently not so relevant for deep-shallow integration, and similar to other described modules. Details on DTDs and configuration can be found in Schäfer (2005).

PicModule (default depth: 10)
PicModule is a very simple module that generates Pet Input Chart format (Section 9.5.5) from raw input text. It has been developed for the integration of the Modern Greek named entity grammar (*SProUT*) with PET. As there was no PoS tagger for Modern Greek, a module had to be developed that generates the in-

put chart from raw text in order to provide XML input for those tokens that are no named entities (and that are therefore omitted in the *SProUT* system output). However, thanks to Unicode, the module is independent of Greek and could also be used for other languages.

TreeTaggerModule (default depth: 20)
TreeTagger (Schmid, 1994) is a statistical PoS tagger with supported language resources for German, English, Spanish, Italian (others could be trained). In Heart of Gold, it is primarily used as tagger for Sleepy (see below).

CorcyModule (default depth: 45)
Corcy is a coreference resolver for English implemented in Python by Özgür Demir on the basis of a paper of Cardie and Wagstaff (1999). Corcy uses a heuristic clustering algorithm to determine coreference relationships. It uses TnT, Chunkie and WordNet (Miller *et al.*, 1993). Both LingPipe and *SProUT* provide further algorithms for coreference resolution.

SleepyModule (default depth: 50)
Sleepy (Dubey and Keller, 2003) is a probabilistic shallow parser for German implemented in OCaml. SleepyModule uses output of the TreeTaggerModule.

9.5.7 Sub-Architectures with the Generic SdlModule

9.5.7.1 Motivation

The SDL module enhances Heart of Gold with a compilable NLP module control flow for sub-architectures, i.e., enabling declarative specification of modules that are composed of other modules. An application are e.g. cascades of (shallow) NLP modules and XSL transformations.

Although the described mainly sequential control flow approach in Heart of Gold for NLP modules by defining a depth and canonical processing order based upon, augmented with potentially multiple input and multiple output annotations in each processing step, was flexible enough for deep-shallow integrations for many languages, it turned out that some envisaged, RMRS-related shallow processing applications required additional features such as loops and parallelism – which SDL supports.

SDL (System Description Language) has been developed independently of Heart of Gold by Krieger (2003). SDL generates Java code for declaratively defined architectures of NLP systems obeying a class interface imposed by the SDL framework. The initial intention was to be able to declaratively define cascaded *SProUT* grammars, e.g. for shallow chunk parsing.

In this section, we briefly introduce SDL, describe how the generic SdlModule has been integrated to provide the possibility to define sub-architectures for Heart

of Gold, and finally present two implemented instances, namely RMRS construction from shallow chunks, and XSLT cascades for combination of multiple RMRS annotations from different modules.

9.5.7.2 SDL

The idea of SDL (Krieger, 2003) is to declaratively specify a flow of information (input and output) between NLP modules or, more generally, software modules. Together, these modules form a *system* (or architecture) with an overall input and output that is compiled into a single Java class definition[15]. To implement a concrete system, the modules need to fulfill a Java interface, with basic operations for setting input, clearing internal state, starting computation, and setting output.

For maximal flexibility, mediators can be defined that are responsible for the communication between modules.

The declarative specification of the architecture is a single expression consisting of symbolic module names connected via operators, plus assignment of these symbolic module names to Java class names, constructor arguments, and some processing options.

The basic connecting operators that can be used to define an architecture are

- $+$ (*sequence*): one module starts after the previous module has finished, taking its output as own input. The input of the sequence is input to the first module. Instead of a single module, each element in the sequence could also be a complex description. A mediator can be defined optionally for encapsulating communication between different modules. The default mediator `seq()` implements the identity function.

- $|$ (*parallelism*): multiple modules or complex descriptions thereof are executed in parallel in separate threads in Java. A mediator can be defined that collects the outputs and combines them into a single output object, which then becomes the output of the whole expression. The default mediator `par()` combines the objects in an array.

- $*$ (*unrestricted iteration*): a module or a complex description is executed in a loop until its output remains unchanged. This can be used to implement a kind of 'fixpoint computation'. A default `fix()` mediator is defined for fixpoint computation that can also be replaced by a module-specific implementation.

Formally, a set of syntactically well-formed module descriptions D in SDL is inductively defined based on an initial set M of (atomic) modules as follows.

- $m \in M \Rightarrow m \in D$

[15] which is then in Heart of Gold executed as a separate, configurable sub-architecture embedded in the generic SdlModule.

- $d_1, d_2 \in D \Rightarrow d_1 + d_2 \in D$
- $d_1, \ldots, d_k \in D \Rightarrow (|\ d_1, \ldots, d_k) \in D$
- $d \in D \Rightarrow (*d) \in D$

The prefix notation with parentheses is used for the $|$ and $*$ operator to avoid grouping ambiguities.

Based on the inductively defined syntax for SDL descriptions, a precise formal semantics can be assigned, using functional composition for the sequence operator, Cartesian product for parallelism, and fixpoint semantics for the iteration operator. Details as well as the full SDL BNF syntax are presented in Krieger (2003).

Compilation of an SDL description into a Java class is done by simply calling the compiler class `de.dfki.lt.sdl.Sdl` with the SDL description file name as argument. The SDL compiler generates a Java source file which can then be compiled into a Java class.

9.5.7.3 SdlModule (Heart of Gold)

SdlModule is a generic wrapper for SDL sub-architectures that can be plugged in into the Heart of Gold. SdlModule acts as any other Heart of Gold module in that it takes a (configurable) XML annotation as input, and returns an output annotation.

The name of the embedded SDL Java class containing the compiled architecture description (previous section) is part of the SdlModule configuration (option name `sdlclassname`). The class is compiled at compile time with the Heart of Gold build tool[16], and executed at runtime in the SdlModule code using Java reflection. The following code fragment is part of the generic SdlModule.

```
// initialize SDL cascade
this.sdlObject=(Modules)Class.forName(sdlclassname).newInstance();
```

Heart of Gold modules are different from SDL modules in that they support more flexible runtime configuration, e.g. the module order is determined at runtime and not at compile time as in SDL. Moreover, the explicit support of multiple input and output annotations is less strict than foreseen in SDL.

This, and the fact that Heart of Gold had been augmented later with SdlModule, is the reason why SDL modules are not the same as Heart of Gold modules. They do not need be, as there is also a facility that can be used to easily include any annotation computed by other Heart of Gold components in SDL modules via XSLT and the below described HoG URI, i.e. without having to implement the module as SDL module.

SDL modules have been defined so far for *SProUT* and XSLT – *SProUT* because it is highly flexible and can be used to implement rule-based transformation

[16] The `ant` target `sdl` executes the SDL compiler.

and annotation enrichment based on typed feature structures with efficiently encoded type hierarchies, and XSLT because it is, as already motivated also in previous chapters, a powerful language for accessing, transforming and combining XML NLP annotations.

Through SDL descriptions, complex sub-architectures for combining and enriching annotations can be defined declaratively, consisting of two base modules, SDL *SProUT* module and SDL XSLT module.

SDL *SProUT* module The SDL *SProUT* modules execute the *SProUT* runtime system as does the Heart of Gold SproutModule. Although they are contained in different classes, the runtime system of the Heart of Gold SproutModule can share its external resources with the *SProUT* SDL module. The configuration mechanism for *SProUT* resources and its components is the same (single configuration file in module description), e.g.

```
de.dfki.lt.sdl.sprout.SproutModulesTextDom("sproutproject.cfg")
```

Depending on the *SProUT* runtime project configuration, the module either takes raw text or XML annotation in the *SProUT*put format as input and in any case returns XML *SProUT*put format (cf. DTD Appendix on page 288, example on page 210).

SDL XSLT module This module applies a configurable XSLT stylesheet to the input annotation and returns the transformation result.

Parameters can be passed to the stylesheet as part of the SDL description (initialization parameter for the SDL XSLT module), e.g.

```
de.dfki.lt.sdl.xslt.XsltModulesDomDom("stylesheet.xsl",
                                      "param1", "value1")
```

In the `Encapsulated` variants (cf. next paragraph), the special parameter `"aid"` (for annotation ID) can be passed as parameter to specify an annotation name (Heart of Gold module name). The `TransformationService` complements this ID to a full HoG URI of the form `hog://sid/acid/aid` by adding session and annotation collection ID from the current context. This mechanism provides a powerful means to incorporate or combine other annotations dynamically created during the active Heart of Gold session by other components. Moreover, the SDL XSLT module can be used to transform any XML format into the typed feature structure XML format and vice versa (e.g. as first and last stage of an SDL cascade).

AnnotationEncapsulator SDL modules for *SProUT* and XSLT use an auxiliary object `AnnotationEncapsulator` to encapsulate Heart of Gold annotation plus metadata. The reason for this is that SDL components can take only a single object

as input parameter, while access to metadata, session context and other facilities provided by Heart of Gold is necessary in addition to access the annotation.

An example for important necessary information is the session and annotation collection context as well as access to the `TransformationService` object using the `hog://` URI syntax described above and in Section 9.3.9. Both SdlModule sample applications we will show later make use of this URI. Using this facility, it is also possible to access annotation computed by components for which no SDL integration exists, the only prerequisite is that they have to be executed before the SDL sub-architecture runs (e.g. Chunkie in the ChunkieRMRS cascade example below).

The `AnnotationEncapsulator` also provides abstraction from the annotation representation format. Both the *SProUT* and the XSLT engines support XML represented as DOM and as String. Subclasses of the `Modules` implementations support both variants, also mixed for input and output[17].

```
de.dfki.lt.sdl.xslt.XsltModulesDomDomEncapsulated
de.dfki.lt.sdl.xslt.XsltModulesDomStringEncapsulated
de.dfki.lt.sdl.xslt.XsltModulesStringDomEncapsulated
de.dfki.lt.sdl.xslt.XsltModulesStringStringEncapsulated
```

However, because of the quite huge and relatively slow DOM model currently used in standard Java, there is no significant difference in performance between using DOM and XML as String representations. This may change when switching to a different XSLT/XML implementation.

Besides these extensions, the encapsulated XSLT SDL modules for Heart of Gold behave in the same way the normal, Heart of Gold-independent XSLT modules behave that we have implemented for SDL (in fact they inherit from them).

9.5.7.4 Example 1: Architecture of the Chunkie RMRS Cascade.

The idea of the Chunkie RMRS cascade developed in Frank *et al.* (2004) is to 'raise' shallow, statistically computed chunks, combined with morphological information, to RMRS structures using an elegant, unification-based approach.

As motivated in Section 9.4, RMRS representations from shallow components can be used as fall-back analyses in case deep parsing fails. However, the output of a shallow chunker basically only extends to syntactically classifying a group of words, as it is based on part-of-speech-tagged input. What is missing is functional information that could be used to identify arguments e.g. of a verb. However, in case-marking languages such as German, and to a lesser extent also English, morphological information can be used to disambiguate constituents.

The idea of the *SProUT*-XSLT cascade is to combine the chunk analyses from a probabilistic chunker with morphological information using typed feature structure

[17] The SDL *SProUT* module variants additionally support also raw input text in addition to annotations which themselves correspond to the XML TFS input mode of *SProUT*.

unification, i.e., a unification (or constraint)-based approach. Through agreement constraints, chunks can be morphologically disambiguated, and thus exactly the partial (underspecified) argument identification that is only expressible in RMRS (in contrast to MRS) is performed.

Using *SProUT*'s XTDL syntax, morphological agreement can be elegantly and compactly stated using disjunction as is the following, very general projection principle.

```
agr :>   lex & [NODE [M-ID #mid]]*
       ( lex & [NODE [M-ID #mid], M-SYN [CAT nn, AGR #agr]] |
         lex & [NODE [M-ID #mid], M-SYN [CAT adja, AGR #agr]] |
         lex & [NODE [M-ID #mid], M-SYN [CAT art, AGR #agr]] )
         lex & [NODE [M-ID #mid]]*
   -> phrase & [NODE [ID #mid], M-SYN [AGR #agr]].
```

The example rule matches a sequence of lexical nodes, and establishes agreement between a single daughter node (right hand side of the rule) and its mother node (left hand side, indicated via the `M-` feature name prefix) for one of the three categories `nn`, `adja` or `art`.

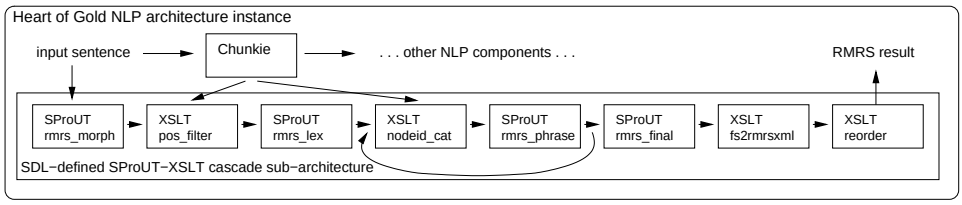


Figure 9.13: *SProUT* XSLT cascade in a Heart of Gold architecture instance

The implemented cascade, displayed in Figure 9.14 (with Heart of Gold integration illustrated graphically in Figure 9.13), consists of four *SProUT* grammar instances with four interleaved XSLT transformations.

```
chunkiermrs = ( sprout_rmrs_morph + xslt_pos_filter + sprout_rmrs_lex
              + (* xslt_nodeid_cat + sprout_rmrs_phrase )
              + sprout_rmrs_final + xslt_fs2rmrsxml + xslt_reorder )

sprout_rmrs_morph  = SproutModulesTextDom("rmrs-morph.cfg")
xslt_pos_filter    = XsltModulesDomDom("posfilter.xsl", "aid", "Chunkie")
sprout_rmrs_lex    = SproutModulesDomDom("rmrs-lex.cfg")
xslt_nodeid_cat    = XsltModulesDomDom("nodeinfo.xsl", "aid", "Chunkie")
sprout_rmrs_phrase = SproutModulesDomDom("rmrs-phrase.cfg")
sprout_rmrs_final  = SproutModulesdDomDom("rmrs-final.cfg")
xslt_fs2rmrsxml    = XsltModulesDomDom("fs2rmrsxml.xsl")
xslt_reorder       = XsltModulesDomString("reorderrmrsdtrs.xsl")
```

Figure 9.14: SDL definition of the *SProUT* XSLT cascade (chunkiermrs.sdl)

The first step, `sprout_rmrs_morph`, runs the *SProUT* interpreter configured for morphological analysis on the raw text input. The result contains the morphology information in XML-encoded typed feature structures such as those shown on page 214. These are input to step 2, `xslt_pos_filter`, which adds the Chunkie category for each word within a chunk span. The XML-encoded chunk analyses are accessed through the above described Heart of Gold `AnnotationEncapsulator` and `TransformationService` from ChunkieModule (cf. Section 9.5.3.1), using a Heart of Gold URI with `aid="Chunkie"`.

The merged result is input to the next *SProUT* cascade, `sprout_rmrs_lex`, which maps PoS-specific encodings of agreement features to a general agreement feature, and defines basic lexical semantics (relation, arg0) for the RMRS generation. Step 4, `xslt_nodeid_cat`, inserts node IDs (defined by spans), mother node ID and category as imposed by the chunker. `sprout_rmrs_phrase` implements RMRS semantics composition for NP and PP chunks. The recursive application of phrase composition rules is defined by means of the SDL star operator.

Step 6, `sprout_rmrs_final`, collects the elements of the semantic sets of the (chunk) daughters into the semantic sets of the phrase using the *SProUT* `collect` operator. Step 7 and 8 are auxiliary XSL transformations. `xslt_fs2rmrsxml` transforms the raw RMRS encoded as typed feature structures (for compatibility with *SProUT* I/O) into the RMRS DTD format, and finally `xslt_reorder` rearranges the RMRS elements as imposed by the RMRS DTD.

The chunkiermrs.sdl file shown in Figure 9.14 is compiled using the SDL compiler to a Java class. The following configuration file chunkiermrs.cfg for the Heart of Gold module then mainly contains the name of the Java class with the compiled cascade code (configuration line `sdl.classname`).

```
module.name=ChunkieRmrs
module.depth=35
module.language=en
module.rootelement=chunkiermrs
#
# name of input annotation (raw text for first cascade/SProUT)
sdl.inputannotation=rawtext
# class name of compiled SDL definition
# (same as class name at beginning of .sdl file)
# can be compiled using 'ant sdl'
sdl.classname=de.dfki.lt.hog.sdlgen.chunkiermrs_en
```

To give a small example for German (the cascade is defined analogously to English), we show Chunkie XML output below and the generated Chunkie RMRS for the sentence 'Florian liebt den grünen Frosch.' in Figure 9.15.

```
<s id="S0" cstart="0" cend="30">
  <w pos="NN" cstart="0" cend="6">Florian</w>
  <w pos="VVFIN" cstart="8" cend="12">liebt</w>
  <chunk cat="NP" cstart="14" cend="30">
```

```
    <w pos="ART" cstart="14" cend="16">den</w>
    <w pos="ADJA" cstart="18" cend="23">grünen</w>
    <w pos="NN" cstart="25" cend="30">Frosch</w>
  </chunk>
</s>
```

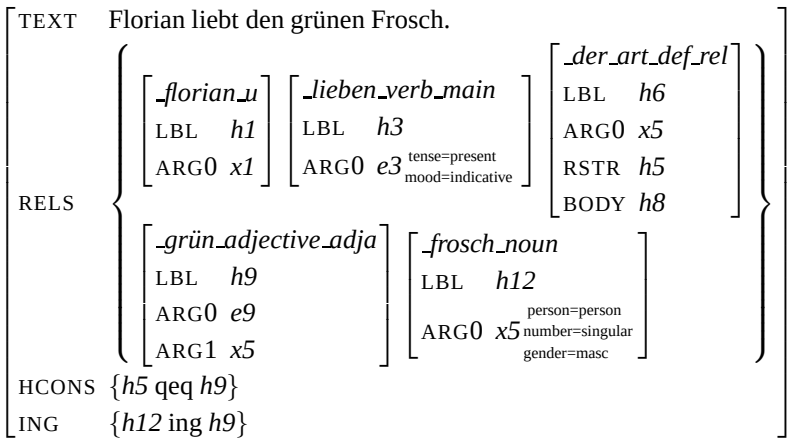


Figure 9.15: RMRS generated with ChunkieRMRS

9.5.7.5 Example 2: Architecture of the RMRSmerge Cascade

Another instance of an SDL-based sub-architecture is the RMRSmerge cascade for combining two RMRSes. The cascade consists of 5 XSL transformations mainly developed by Anette Frank. It merges a secondary RMRS annotation, typically from a named entity recognition component such as *SProUT* or LingPipe, into a configurable primary RMRS annotation, typically from RASP or PET, by using character span information and adjusting RMRS variables.

The result is a single RMRS combining all information obtained from the input RMRSes. The primary annotation is configured in the SdlModule configuration file rmrsmerge.cfg, the secondary annotation is configurable as parameter in the SDL definition rmrsmerge.sdl (parameter to the merging stylesheet).

The following examples illustrate merging of PET and *SProUT* RMRSes. We define a depth of 110 to ensure that the RMRSmerge module is started after PET (default depth 100) in the module configuration file rmrsmerge.cfg:

```
module.name=RmrsMerge
module.depth=110
module.language=en
module.rootelement=merged-rmrs
# ----- common modules settings end here -----
# name of input annotation (e.g. PET or RASP)
```

```
sdl.inputannotation=PET
# class name of compiled SDL definition
# (same as class name at beginning of .sdl file)
# can be compiled using 'ant rmrsmerge'
sdl.classname=de.dfki.lt.hog.sdlgen.rmrsmerge
```

The SDL definition (rmrsmerge.sdl) for the XSLT cascade is

```
rmrsmerge = ( rmrs_ep_rargs2rels + adjust_nespans + merge_ne2petrasp +
              rmrs_rels2ep_rargs + reorder_rmrs_dtrs )
rmrs_ep_rargs2rels = XsltModulesStringDom("rmrs_ep_rargs2rels.xsl")
adjust_nespans = XsltModulesDomDom("adjust_nespans.xsl","aid","Sprout")
merge_ne2petrasp = XsltModulesDomDom("merge-ne-to-rasp.xsl","aid","Sprout")
rmrs_rels2ep_rargs = XsltModulesDomDom("rmrs_rels2ep_rargs.xsl")
reorder_rmrs_dtrs = XsltModulesDomString("reorderrmrsdtrs.xsl",
                                         "aid","xmltext")
```

Figure 9.16 depicts an example of a merged RMRS for the sentence 'Did Bernd Kiefer present a paper at ACL 2003?'. The first two rows of RELS contain the deep RMRS, the third row contains the fine-grained RMRS produced by *SProUT* for the person name, the last row contains the RMRS for 'ACL 2003'. The *SProUT* RMRSes are linked with the deep RMRS via *x*8 and *x*18.

9.6 Sample configuration scenarios for robust deep-shallow integration

In this section, we describe three sample Heart of Gold session configurations of robust deep-shallow integration, for German, English and Japanese. We do not list the details for the respective module configurations, as these are roughly (modulo language-specific variations) those we presented in the module descriptions above, and differ only in minor details such as language code. In particular, the configured module depths that determine the processing order are identical with the default values shown in the previous sections. Similar configurations have been used for the evaluations we will describe in Section 9.9.

9.6.1 Sample Configuration for German

The MoCoMan configuration file for the robust deep-shallow integration workflow illustrated in Figure 9.17 contains the following lines (file conf/de/deepshallow.cfg):

```
de.dfki.lt.hog.modules.JTokModule=conf/de/jtok.cfg
de.dfki.lt.hog.modules.TnTModule=conf/de/tnt.cfg
de.dfki.lt.hog.modules.ChunkieModule=conf/de/chunkie.cfg
de.dfki.lt.hog.modules.SdlModule=conf/de/chunkiermrs.cfg
de.dfki.lt.hog.modules.SproutModule=conf/de/sprout.cfg
de.dfki.lt.hog.modules.PetModule=conf/de/pet.cfg
de.dfki.lt.hog.modules.SdlModule=conf/de/rmrsmerge.cfg
```

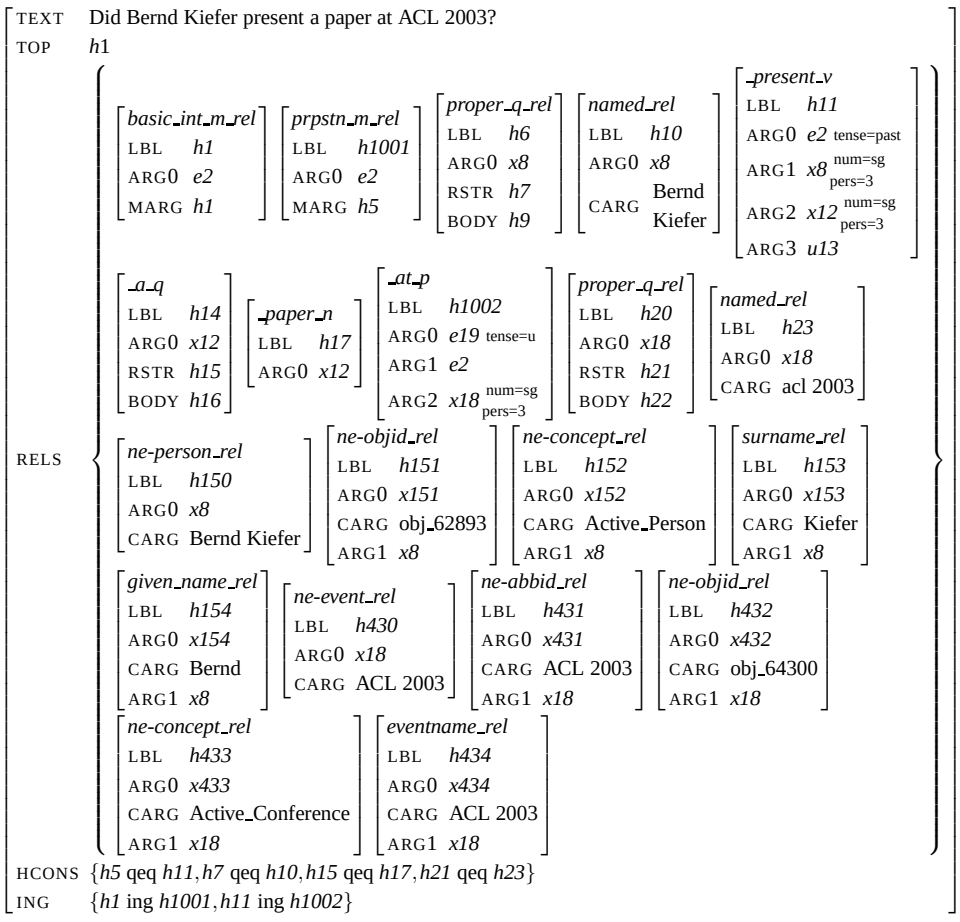


Figure 9.16: RMRS merged using the RmrsMerge module

The raw input sentence text is sent to the JTok tokenizer (depth 10) and *SProUT* (depth 40), because *SProUT* comes with its own tokenizer with a finer-grained token classification. Chunkie (depth 30) and TnT (depth 20) use the tokenized output from JTok as input, Chunkie output (proprietary XML format) is used as secondary input for the ChunkieRMRS cascade (depth 35), which in turn consists of 4 *SProUT* instances and 4 XSLT stylesheets as explained in Section 9.5.7.4. The output of the cascade (RMRS format) can be used as shallow fall-back result in case the deep parser fails to parse the input sentence.

TnT output as well as the recognized named entities from *SProUT* are transformed into the PET input chart format. The configuration which PET input charts to merge (using the default combinepixml.xsl stylesheet from page 296) is part of pet.cfg:

```
pet.inputannotation=TnTpiXML,SProUTpiXML
```

PET has depth 100, and because the RMRS merging cascade module consisting of the 5 XSLT stylesheets explained in Section 9.5.7.5 has depth 110, it can take the RMRS outputs of PET and *SProUT* and generate a unified RMRS of them as described in the previous section.

An application will then get either the merged PET/*SProUT* RMRS or as fallback the shallow ChunkieRMRS. Whether PET returns fragments or not, is a matter of configuration in pet.cfg. The overall system instance thus provides a much more robust analysis than the deep parser alone and allows for a wide range of possible combinations just as a matter of configuration (cf. also the evaluation Section 9.9).

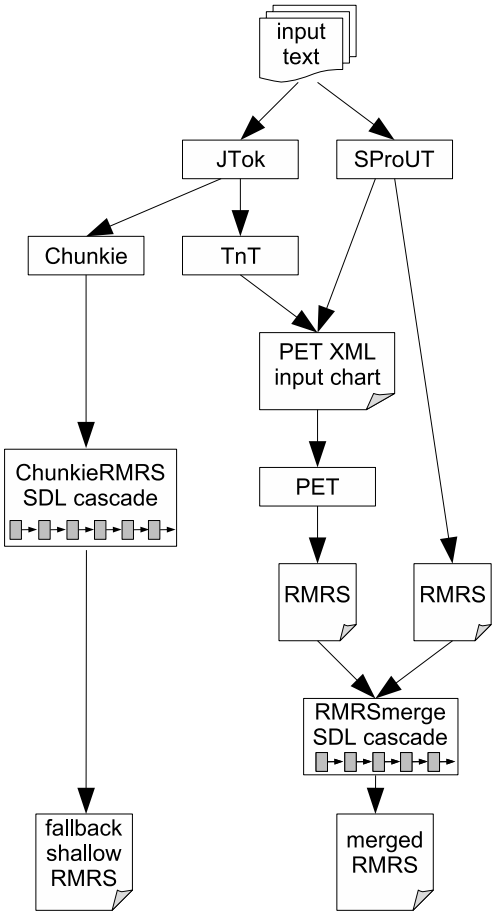


Figure 9.17: Sample configuration of deep-shallow integration for German

9.6.2 Sample Configuration for English

The configuration for English (Figure 9.18) centered around the English Resource HPSG Grammar (ERG) is exactly analogous to German[18], except that there is an additional RASP module (depth 50) with native RMRS output that can be used as shallow fall-back as alternative to ChunkieRMRS. As explained in Section 9.5.3.2, RASP can, in contrast to ChunkieRMRS, deliver filled predicate-argument structures in the RMRS.

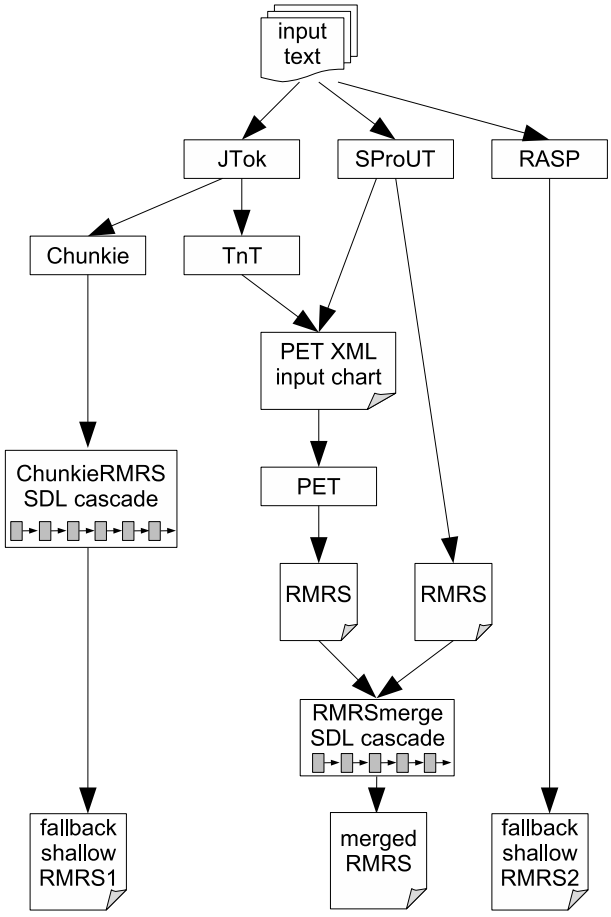


Figure 9.18: Sample configuration of deep-shallow integration for English

The configuration file is conf/en/deepshallow.cfg:

```
de.dfki.lt.hog.modules.JTokModule=conf/en/jtok.cfg
de.dfki.lt.hog.modules.TnTModule=conf/en/tnt.cfg
de.dfki.lt.hog.modules.ChunkieModule=conf/en/chunkie.cfg
de.dfki.lt.hog.modules.SdlModule=conf/en/chunkiermrs.cfg
de.dfki.lt.hog.modules.SproutModule=conf/en/sprout.cfg
```

[18] i.e. same modules and components with lingware for English instead of German.

```
de.dfki.lt.hog.modules.RaspModule=conf/en/rasp.cfg
de.dfki.lt.hog.modules.PetModule=conf/en/pet.cfg
de.dfki.lt.hog.modules.SdlModule=conf/en/rmrsmerge.cfg
```

9.6.3 Sample Configuration for Japanese

As there is no chunker, no shallow parser and a different tagger available for Japanese, the configuration (Figure 9.19) is quite different from that for English and German. However, as a Japanese named entity recognition grammar exists for *SProUT*, the *SProUT* module can operate in the same way the English and German configurations do, delivering PET input chart format through transformation.

ChaSen acts as segmentizer and tagger, and via configuration directly produces the PET input chart format. As in the English and German configurations, ChaSen output is concatenated with *SProUT* NER output before being parsed with PET running the Jacy HPSG grammar (Siegel and Bender, 2002) using the following line in the pet.cfg configuration file.

```
pet.inputannotation=ChaSen,SProUTpiXML
```

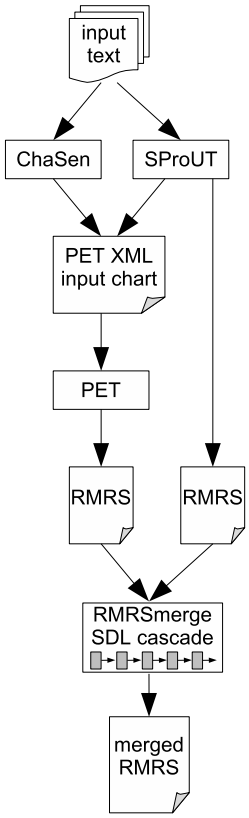


Figure 9.19: Sample configuration of deep-shallow integration for Japanese

Similarly, the final module is the RMRSmerge cascade with depth 110, combining *SProUT* and HPSG analyses. The session configuration file conf/ja/deep-shallow.cfg for Japanese is defined as follows.

```
de.dfki.lt.hog.modules.ChasenModule=conf/ja/chasen.cfg
de.dfki.lt.hog.modules.SproutModule=conf/ja/sprout.cfg
de.dfki.lt.hog.modules.PetModule=conf/ja/pet.cfg
de.dfki.lt.hog.modules.SdlModule=conf/ja/rmrsmerge.cfg
```

As already mentioned in Section 9.5.1.2, character position recalculation provided by the ChasenModule is crucial for correct alignment and merging of the different RMRSes from ChaSen, *SProUT* and PET.

9.7 Interfacing Ontologies

This section is dedicated to an extension of the Heart of Gold (in particular, *SProUT*) lingware with ontology information, an important feature when aiming at high precision and recall in domain-specific texts and Semantic Web applications.

The extended lingware can be used to improve hybrid processing in Heart of Gold by combining named entity recognition and information extraction for recognizing domain-specific names, terms and expressions with a general, open-domain broad-coverage HPSG grammar.

We describe an implemented process we call OntoNERdIE (Schäfer, 2006b) that maps OWL/RDF-encoded ontologies with large, dynamically maintained instance data to named entity recognition (NER) and information extraction (IE) engine resources, preserving hierarchical concept information and links back to the ontology concepts and instances.

Applications of the approach are e.g. ontology-based hybrid question analysis (described in Section 9.10.2 viz. Frank *et al.* (2006)), automatic typed hyperlinking of instances and concepts occurring in documents along the lines in (Busemann *et al.*, 2003), or other innovative applications that combine Semantic Web and language technology.

In any case, the links from recognized instances back to entries in the ontology can be used for advanced navigation and queries in the domain modeled by the ontology. The NER/IE resources are kept up-to-date and in sync with the growing ontology (instance) data.

The approach has been implemented for the ontology on language technology that works at the back-end of the LT WORLD web portal[19] (Uszkoreit *et al.*, 2003), but could be easily adapted to other domains, ontologies and systems, because it is already almost fully automated.

LT WORLD is an ontology-based virtual information center on the wide spectrum of Human Language Technology (HLT), providing information about people, technologies, products, resources, projects, and organizations in this area. The

[19] http://www.lt-world.org

service is free and is provided by the German Research Center for Artificial Intelligence (DFKI) to the R&D community, potential users of language technologies, students and other interested parties.

We use *SProUT* (Chapter 7) as named entity recognition and information extraction tool because it comes with (1) a type system and typed feature structures as basic data structures with a closed type world and strict welltypedness and appropriateness conditions, (2) a powerful and declarative rule mechanism with regular expressions over typed feature structures, (3) a gazetteer module with fine-grained, customizable classification of recognized entities. Moreover, *SProUT* comes with additional, configurable modules such as tokenizer and morphology, that can be exploited in the rule system, e.g. to use context or morphological variation for improved NER.

The *SProUT* runtime component, extended with the ontology information as we will describe below, has been integrated as NER and IE component into the Heart of Gold framework.

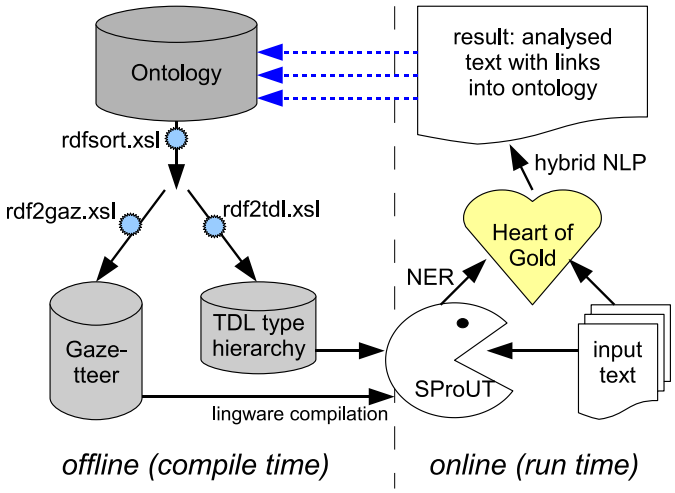


Figure 9.20: OntoNERdIE flow of information

9.7.1 OntoNERdIE

In the following, we describe the processing steps of OntoNERdIE. Following the general motivation presented in Chapter 5 resp. Schäfer (2003), the approach heavily relies on XSLT transformation of the XML representation formats. However, this time, the use of XSLT is restricted to offline processing only.

9.7.1.1 Making Implicitly Encoded Inheritance Information in the Ontology Explicit

Starting from the OWL representation of the ontology, forward-chaining inference rules are applied through the open source RDF database Sesame[20] in order to make all `subClassOf` relations explicit (details in Frank *et al.* 2006). This is only done in order to avoid inferences for subtype queries that would otherwise have to be performed presumably less efficiently at later processing stages, e.g. through XSLT transformations.

9.7.1.2 Merging rdf:Descriptions

The resulting RDF file is processed with a small but sophisticated XSLT stylesheet (`rdfsort.xsl`, in the XSLT Appendix on page 299) merging `rdf:Descriptions` that are distributed over the file but belong together. This is a necessary prerequisite for the subsequent extraction steps, and, as it cannot be implemented by a simple `xsl:sort` statement, has to be coded as a proper, dedicated stylesheet.

The result for a single instance, `obj_89404`, is shown in Figure 9.21. The `rdfsort.xsl` stylesheet make use of the XSLT `key` declaration and of the `generate-id`() method to look up and merge all descriptions with the same `about` or `nodeid` attribute.

9.7.1.3 Extracting Inheritance Statements and Converting to TDL Definitions

The second stylesheet (`rdf2tdl.xsl`) converts the RDF `subClassOf` statements from the output of the previous step into a set of TDL type definitions that can be immediately imported by the *SProUT* NER grammar, e.g. currently 1260 type definitions for the same number of `subClassOf` statements in the LT WORLD ontology.

Following are two type definitions (out of 1260) that have been generated from the OWL input file using the `rdf2tdl.xsl` stylesheet.

```
Active_Conference := Conferences & Backend_Events.
Natural_Language_Parsing := Written_Language & Language_Analysis.
```

This is of course a lossy conversion because not all relations supported by an OWL (DL or full) ontology such as `unionOf`, `disjointWith`, *etc* are mapped. However, for named entity (NE) classifications, the `subClassOf` taxonomy mappings are sufficient. Moreover, the efficiently encoded type hierarchy in *SProUT* makes the subclass queries very fast at runtime. Other relations could be formulated as direct (though slower) ontology queries from the obtained NLP analysis results (RMRSes) using the `OBJID` mechanism described in the next step.

[20] `http://www.openrdf.org`

```
<rdf:Description rdf:about="http://www.lt-world.org/ltw.owl#obj_89404">
  <rdf:type rdf:resource="http://www.lt-world.org/ltw.owl#
    Active_Conference"/>
  <dc_keyword rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    Machine Translation</dc_keyword>
  <homepageURL rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    http://www.lrec-conf.org/lrec2006/</homepageURL>
  <dc_keyword rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    NLP Tools</dc_keyword>
  <dateStart rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    2006-05-24</dateStart>
  <paperDeadline rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    2005-10-14</paperDeadline>
  <eventNameVariant rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    LREC 2006</eventNameVariant>
  <dc_keyword rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    Semantic Web</dc_keyword>
  <takesPlaceInCountry rdf:resource="http://www.lt-world.org/ltw.owl#
    lt-world_Individual_334"/>
  <eventNameVariant rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    5th Conference on Language Resources and Evaluation</eventNameVariant>
  <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    5th Conference on Language Resources and Evaluation</name>
  <locatedIn rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    Magazzini del Cotone Conference Center, Genoa</locatedIn>
  <eventName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    5th Conference on Language Resources and Evaluation</eventName>
  <dc_language rdf:resource="http://www.lt-world.org/ltw.owl#lt-world_
    Individual_105"/>
  <dateEnd rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    2006-05-26</dateEnd>
  <eventNameAbbreviation rdf:datatype="http://www.w3.org/2001/XMLSchema#
   string">LREC 2006</eventNameAbbreviation>
</rdf:Description>
```

Figure 9.21: LT WORLD ontology entry for LREC 2006 (shortened)

9.7.1.4 Generating Gazetteer Entries

Another stylesheet (`rdf2gaz.xsl`) selects statements from the RDF input file about instances of relevant concepts via `rdf:type` and converts them to gazetteer source files for the *SProUT* gazetteer compiler. In the following example, two of the approx. 20000 converted entries for LT WORLD are shown.

```
Martin Kay | GTYPE: lt_person | SNAME: "Kay" | GNAME: "Martin" |
             CONCEPT: Active_Person | OBJID: "obj_65046"
LREC 2006  | GTYPE: lt_event | GABBID: "LREC 2006" |
             CONCEPT: Active_Conference | OBJID: "obj_89404"
```

The attribute CONCEPT contains the TDL type from the previous step. For convenience, several ontology concepts are mapped manually (as part of the configuration of the stylesheet) to only a few NE classes (under attribute GTYPE), namely person, organization, event, project, product and technology plus some properties for LT WORLD.

This has the advantage that NER context rules from existing *SProUT* grammars can be re-used[21] for better robustness and disambiguation, e.g. to recognize not only Martin Kay, but also Prof. Kay, Dr. Kay, Mr. Kay with or without firstname and including morphological variation.

The following *SProUT* rule (*XTDL* syntax) simply copies the slots of a matched gazetteer entry for events (e.g. a conference) to the output as a recognized named entity.

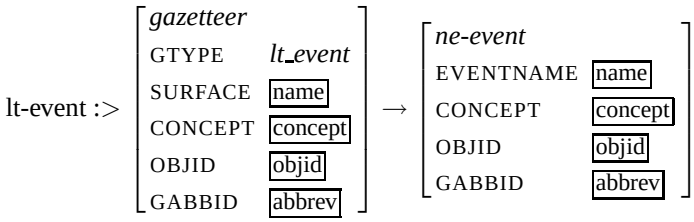


Figure 9.22: A simple *SProUT* rule that copies gazetteer output

OBJID contains the object identifier of the instance in the ontology. This can be used as link back to the full knowledge stored in the ontology, e.g. for subsequent queries such as 'Who else participated in project [with OBJID obj_4789]?' *etc*, formulated in an ontology query language. How such natural language questions can be automatically translated to ontology queries will be addressed in Section 9.10.2.

9.7.1.5 Named Entity Recognition at Runtime

The output of *SProUT* for a recognized NE is a typed feature structure (e.g. in XML format) containing the RHS of the rule as shown in the previous step with the copied gazetteer data shown there (Figure 9.22) plus some additional meta-information such as character span, NE type *etc*.

A mapping to a deep HPSG grammar for hybrid processing can be performed by means of the XSLT stylesheet automatically generated from the *SProUT* type hierarchy as shown in Section 9.5.4.1.

At run time, the generated stylesheet would e.g. produce the following item for LREC 2006 on the deep parser's input chart (HPSG type $generic_event).

```
<w id="SPR3.1" cstart="48" cend="56" constant="yes">
   <surface>LREC 2006</surface>
```

[21] Alternatively, a fully automatic, but maybe too fine-grained 1:1 mapping of all concepts could be performed.

```
    <typeinfo id="TIN3.1" baseform="no">
      <stem>$generic_event</stem>
    </typeinfo>
  </w>
```

The transformation output then contains only the NER information that is required by the deep parser (character span and generic HPSG type for a chart item to be generated). Alternatively (e.g. if no hybrid deep-shallow processing is performed), the full output from a *SProUT* runtime system could be used instead.

9.7.1.6 Information Extraction at Runtime

Similar to the NER mapping from the previous section, Heart of Gold can also automatically generate XSLT stylesheets that produce a richer, robust semantics representation format (RMRS, Section 9.4) at runtime from the *SProUT* results. An example is shown in Figure 9.23. Here, *objid*, *surname*, *given_name* and other structured information from the ontology instance is preserved in the representation. The advantage is that this RMRS can also be combined *ex post* with analyses from other deep or shallow NLP components (cf. Section 9.5.7.5), e.g. partial analyses when a full parse fails.

$$\begin{bmatrix}
\text{TEXT} & \text{Prof. Martin Kay} \\
\text{TOP} & h0 \\
\text{RELS} & \left\{ \begin{bmatrix} \text{REL} & \textit{ne-person} \\ \text{LBL} & h0 \\ \text{ARG0} & x0 \\ \text{CARG} & \text{Prof. Martin Kay} \end{bmatrix} \begin{bmatrix} \text{REL} & \textit{concept} \\ \text{LBL} & h6 \\ \text{ARG0} & x6 \\ \text{CARG} & \text{Active_Person} \\ \text{ARG1} & x0 \end{bmatrix} \begin{bmatrix} \text{REL} & \textit{objid} \\ \text{LBL} & h7 \\ \text{ARG0} & x7 \\ \text{CARG} & \text{obj_65046} \\ \text{ARG1} & x0 \end{bmatrix} \right. \\
 & \left. \begin{bmatrix} \text{REL} & \textit{surname} \\ \text{LBL} & h8 \\ \text{ARG0} & x8 \\ \text{CARG} & \text{Kay} \\ \text{ARG1} & x0 \end{bmatrix} \begin{bmatrix} \text{REL} & \textit{given_name} \\ \text{LBL} & h9 \\ \text{ARG0} & x9 \\ \text{CARG} & \text{Martin} \\ \text{ARG1} & x0 \end{bmatrix} \begin{bmatrix} \text{REL} & \textit{title} \\ \text{LBL} & h10 \\ \text{ARG0} & x10 \\ \text{CARG} & \text{Prof.} \\ \text{ARG1} & x0 \end{bmatrix} \right\} \\
\text{HCONS} & \{\} \\
\text{ING} & \{\}
\end{bmatrix}$$

Figure 9.23: RMRS generated from *SProUT* output in Heart of Gold

An example of a full, hybrid RMRS generated with input from OntoNERdIE and merged deep and shallow RMRS is reproduced in Figure 9.16.

9.7.1.7 Summary

We have described OntoNERdIE, an XSLT-based procedure that maps ontology instances and concepts to NER and IE resources. The process is fully automatic for

instances and only requires manual filtering of interesting concepts and properties. The key features and advantage we see are that lingware resources are kept in sync with information from dynamically expanding ontologies such that instances from the ontology are precisely and efficiently recognized in NER and IE.

Existing multilingual NER and IE grammars can be (re-)used that exploit context information for augmented precision. The ontology concept information can be exploited in NER and IE rules. Links from recognized instances back into the ontology are preserved for subsequent ontology queries in the applications. The improved NER can be utilized for hybrid deep-shallow analysis in Heart of Gold, and richer, IE-like information can be provided in a structured, robust semantics output format.

9.8 Visualization

Information visualization is indispensable for the inspection of complex NLP analyses. Although RMRS already is a gross simplification of the complex nested feature structures that are returned by the HPSG parser, and abstracts from syntactic details that contributed to the analysis, at the same time providing the semantic details that might be of interest to an application, the variable- and handle-based structure is still somewhat confusing for a human reader.

Therefore, we developed a stylesheet that transforms an RMRS into an HTML with Javascript document in the MRS look (with EP args grouped in AVM-like matrices) that highlights corresponding variable/handle occurrences and the corresponding part of the input text when the user moves the mouse cursor over an RMRS region (Figure 9.24). The same transformation service that is used for annotation transformation thus also is responsible for visualization transformation (cf. Section 9.3.9).

Furthermore, stylesheets for general XML document visualization (taken from the Apache Tomcat project) and a Swing-based Java browser applet for visualizing typed feature structures encoded in XML are part of the Heart of Gold system.

Finally, stylesheets that produce LaTeX code for AVM-like visualizations from RMRS and TFS-XML can be used for written documentations (e.g. used in this thesis; Schäfer 2004a).

9.9 Evaluation

At the end of the DEEPTHOUGHT project, an evaluation of hybrid processing in Heart of Gold has been conducted. This evaluation has been documented in Beermann *et al.* (2004), from which most of the content of this section is taken. The hybrid parsing evaluation concentrated on the Heart of Gold RMRS output alone, with robustness in focus, while the application evaluation included an analysis of the usefulness of Heart of Gold output for NLP-based applications (business intelligence and automatic email response management).

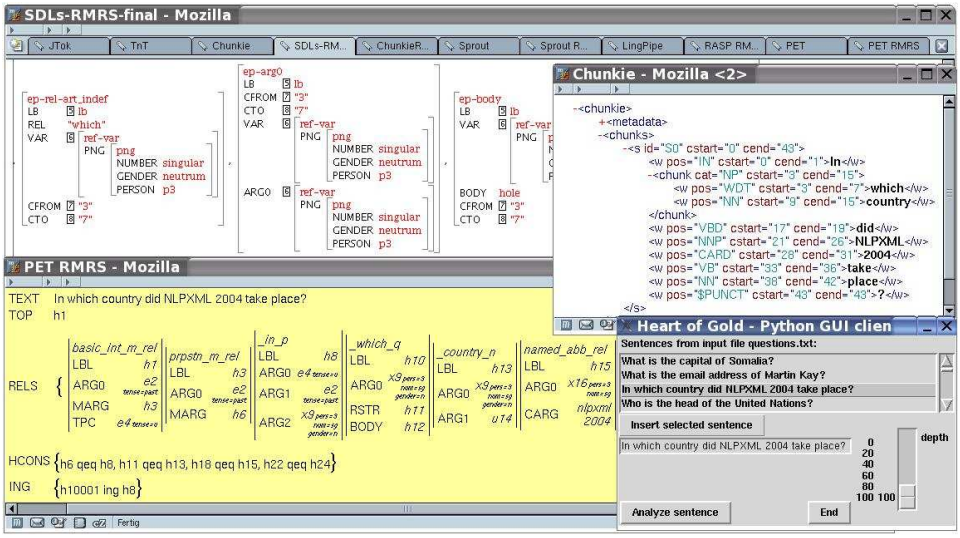


Figure 9.24: Analysis results in GUI with specialized XML visualizations

9.9.1 Hybrid Parsing Evaluation

The hybrid parsing evaluation employed and compared three Heart of Gold configuration modes, with and without partial, deep analyses[22], and using or not using part-of-speech tagging and named entity recognition.

- *Configuration 1.* Heart of Gold configured to provide the deepest result possible for each sentence. PET uses both part-of-speech tags delivered by TnT and named entities detected by *SProUT* as input and delivers partial parses in case no spanning analysis is available.
- *Configuration 2.* Heart of Gold configured to provide the deepest result possible for each sentence. PET uses both part-of-speech tags delivered by TnT and named entities detected by *SProUT* as input, but does not deliver partial parses, i.e. no success in case no spanning analysis is available.
- *Configuration 3.* Heart of Gold configured to provide only complete analyses from PET and RASP alone. No information from shallower modules (part-of-speech tagging, named entity recognition) is used.

These configurations correspond to subsets of the sample deep-shallow integration scenarios presented in Section 9.6, but without the RMRSmerge and

[22] Full analyses covering the complete input sentences are called *spanning* analyses in the following.

ChunkieRMRS modules that were not yet implemented at the time this evaluation was performed.

9.9.1.1 Evaluation on PASCAL Challenge Data (English)

The training data of the PASCAL challenge (Ireson *et al.*, 2005) contains declarative English sentences of various domains. 581 test sentences of the PASCAL corpus were sent to Heart of Gold using configuration 1 as described above. Table 9.2 shows the coverage of PET and RASP. A RASP result is only returned as fall-back answer, i.e., when the deep parser does not return an analysis.

configuration	sentences	PET	spanning PET	RASP fall-back	total results
1 1	581 100%	442 76.06%		139 23.92%	581 100%
2 2	581 100%		134 23.06%	447 76.94%	581 100%
3 3	581 100%	37 6.37%	14 2.41%	544 93.63%	581 100%

Table 9.2: Evaluation on PASCAL data

As expected, annotations could be computed for all sentences, either by PET or RASP. RASP is very robust and able to produce analyses for all sentences for which PET does not return a result (e.g. because of missing lexicon entries for some verbs). In this domain, which is quite diverse in lexical choices, it could be shown that the usage of default lexicon entries for recognized part-of-speeches and named entities heavily influences the performance of the deep linguistic processing. Without these, PET delivers results in only 6.37% of the sentences and spanning results in only 2.41%, while with the input, the coverage of PET rises up to 76.06% for partial analyses and 23.06% for spanning results.

9.9.1.2 Mobile Phone Product Description Corpus

692 sentences (many of them fragments and lists) of mobile phone product descriptions collected from Internet sites were sent to the Heart of Gold using the three configurations described above. Results are shown in Table 9.3.

The lexicons were tuned to this domain, such that PET was more successful than in the former domain, in the cases of using or not using default lexicon entries. Still, it could be shown that the usage of PoS and NER information from shallower modules increases the performance of PET enormously from 9.36% to 20.23%. The data contains many lists and tables the deep HPSG grammar is not quite prepared for. It shows how the overall processing gains from being able to fall back to partial parses or (underspecified) RASP results.

configuration	sentences	PET	spanning PET	RASP fall-back	total results
1 1	692 100%	631 91.18%		61 8.82%	692 100%
2 2	692 100%		140 20.23%	552 79.77%	692 100%
3 3	692 100%	296 42.77%	65 9.39%	396 57.23%	692 100%

Table 9.3: Evaluation on the mobile phone descriptions corpus

9.9.1.3 Newspaper Corpus

48 sentences of a (business news) article in the San Francisco Chronicle of 2004-07-27 were sent to the Heart of Gold. The results are listed in Table 9.4.

configuration	sentences	PET	spanning PET	RASP fall-back	total results
1 1	48 100%	31 64.58%		17 35.42%	48 100%
2 2	48 100%		6 12.50%	42 87.50%	48 100%
3 3	48 100%	5 10.42%	1 2.08%	43 89.58%	48 100%

Table 9.4: Evaluation on San Francisco Chronicle articles

This text is completely out of the training domain and therefore significantly shows the effect of default lexicon entries.

9.9.1.4 All Corpora

Tables 9.5, 9.6 and 9.7 show the three corpora and the coverage of all three configurations in sum.

9.9.1.5 Conclusions

First of all, the strategy to use the deepest available result delivered by the Heart of Gold core architecture guarantees results for all sentences in different domains. These results are comparable and compatible to each other because they are formulated in the same framework, RMRS. It therefore seems useful to combine very robust modules such as RASP with deeper modules such as PET.

corpus	sentences	PET	RASP fall-back
Pascal	581	442	139
Mobile Phone	692	631	61
Newspaper	48	31	17
All	1321 100%	1104 83.57%	217 16.43%

Table 9.5: Evaluation results using configuration 1

corpus	sentences	spanning PET	RASP fall-back
Pascal	581	134	447
Mobile Phone	692	140	552
Newspaper	48	6	42
All	1321 100%	280 21.20%	1041 78.80%

Table 9.6: Evaluation results using configuration 2

In different domains, closer and farther away from the development domain in lexicon as well as syntactic structures, it could be shown that the depth of results increases enormously when using the results of PoS tagging and named entity recognition in deep linguistic processing. Over all domains, spanning HPSG (PET) processing increased from 6.06% up to 21.20%.

corpus	sentences	PET	spanning PET	RASP fall-back
Pascal	581	37	14	544
Mobile Phone	692	296	65	396
Newspaper	48	5	1	43
All	1321 100%	338 25.59%	80 6.06%	983 74.41%

Table 9.7: Evaluation results using configuration 3

9.9.1.6 Hybrid Processing and the (Early) German HPSG Grammar

The large-scale HPSG grammar for German (Müller and Kasper, 2000; Crysmann, 2003) developed at DFKI has been integrated into the Heart of Gold during the 2nd quarter of 2004. This evaluation is also described in Beermann *et al.* (2004). The main task of this integration effort was the adaptation of the semantic output to current (R)MRS standards. Furthermore, interface types and mappings have been provided to integrate shallow NLP analyses into the deep parser, thereby ensuring robustness. As for English, named entities recognized by *SProUT* and PoS information from TnT are used to address the unknown-word problem.

In order to assess the gains in robustness offered by the integrated deep-shallow processing adopted by Heart of Gold, an experiment on unseen data, measuring the coverage obtained with and without deep-shallow integration, was run. As test data, 200 questions from the German section of the CLEF 2003 multilingual question answering competition, were used. The corpus was parsed both by a stand-alone PET and by the version integrated into Heart of Gold.

The standalone PET system (baseline) was able to deliver a full parse for 34 sentences only (17%). Inspection of the error log revealed that the most common source for parse failure was lexical in nature: in 77.5% of the input sentences, at least one lexical item was unknown. Abstracting away from the problems of lexical coverage, syntactic coverage was around 80% (34/42), although these figures are certainly not reliable, owing to the size of the data set.

Deep-shallow integration drastically improved on these figures: by feeding NER and PoS tag information into the deep parser, coverage went up to 73% (146/200), a figure comparable to those achieved on corpora for which the grammar had been optimized (e.g. VERBMOBIL data: VM-CD01: 74.1%; VM-CD15: 78.4%).

The results obtained by the German Heart of Gold also compare well to a simulation of an 'ideal' NER component. Manual substitution of NEs resulted in an overall coverage of 56.5% (113/200). Owing to the fact that substitution was restricted to named entities, lexical coverage was still an issue, accounting for 30% (60/100) of parse failures. Relative to the 140 sentences without lexical errors, we measured a syntactic coverage of around 80%.

To conclude, the integrated shallow-deep approach embodied by Heart of Gold, and, most notably, the combination of NER and PoS mappings, proves to be highly successful in improving the robustness of the deep parser for German as well.

9.9.2 Evaluation in Application Context

9.9.2.1 Evaluation of the Auto-Response Application (German, English)

The underlying scenario targets at email response management for customer relationship management. The application developed and evaluated by the project partner Xtramind GmbH provided information extraction functionalities for the following scenarios in the content domain of a mobile phone provider: product

ordering, mix-ups in deliveries of products, and replacement of defective products. It took as input one or more e-mails (German and/or English) and delivered filled scenario templates as output.

These templates were the result of several hybrid processing steps such as named entity recognition, shallow and deep analysis, coreference resolution, mapping of results from the preceding analysis on domain specific templates, and merging operations on the partially filled templates resulting in filled scenario templates. The final scenario templates were of the following types: (1) Exchange (2) Ordering (3) Mix-up. In cases where no merging operations can take place, the partially filled templates were presented to the user.

An email corpus was constructed using relevant anonymized customer emails. Since the evaluation of the component was performed manually, the data set used for the evaluation was rather limited: 87 emails for German and 84 for English. On average, each email contained 4 sentences. Hence, the German data set consisted of 348 sentences and the English data set of 336 sentences.

Input e-mails were processed by the system that returned filled scenario templates as output. The system identified the customer using the predicate-argument structure from the deep analysis and by performing a domain-specific coreference resolution between certain pronouns and potential antecedents. A person writing an e-mail, e.g. referring to herself or himself by 'I' or 'me' *etc* presumably mentions her name either in the complimentary close or in the address part of the e-mail. Products were identified by predicate-argument relations and named entity recognition. The predicates triggered the process of choosing the correct scenario template.

The evaluation compared two different result sets for German and English using two different preprocessing levels configured in the Heart of Gold. The deep processing used in the application and for evaluation corresponds to configuration 2 described in Section 9.9.1 above. Since the application had to be applied in real world contexts, robustness was a necessary precondition, and configuration 3 (no shallow preprocessing for deep parsing) was not considered at all, nor was configuration 1 (including also partial results from deep processing in case of parsing failure).

The usage of part-of-speech tags delivered by TnT and named entities detected by *SProUT* as input guaranteed the robustness requirement. Therefore the application has been evaluated with PET using part-of-speech tags and named entities for preprocessing in Heart of Gold. For German, the chunk tagger Chunkie has been used as shallow fall-back component. For English, the robust shallow parser RASP has been used as fall-back. Both have been integrated using Heart of Gold as well.

The configurations that have been used for the evaluation were (1) only deep analysis, (2) deep analysis with shallow fall-back. Precision, recall, and f-score were measured for the scenario templates delivered by the system by manually comparing them against gold standard template annotations in the email corpus mentioned above.

Two different types of evaluation were performed: a template-based evaluation

and a feature-based evaluation. During template-based evaluation, a template was judged correct if and only if all required template features were correctly filled and the type of the template was correct. During feature-based evaluation, all feature values were evaluated separately. Each single slot was judged either as correct or false. The relevant features for all three template types were the following: template type, product list (each product counting singly if more than one), product feature list (each feature counting singly if more than one), customer, provider.

First experiment (German) The first experiment shows figures using mainly deep analysis and, as fall-back solution, shallow processing (configuration 2; Table 9.8).

	Precision	Recall	f-score
Template-based evaluation	50.35 %	45.74 %	47.93 %
Feature-based evaluation	60.46 %	56.38%	58.34 %

Table 9.8: First experiment German; results using configuration 2

Second Experiment (German) The second experiment uses only deep analysis as preprocessing (configuration 1; Table 9.9).

	Precision	Recall	f-score
Template-based evaluation	62.25 %	36.95 %	46.30 %
Feature-based evaluation	68.43 %	46.65 %	54.76 %

Table 9.9: Second experiment German; results using configuration 1

First Experiment (English) The first experiment shows figures using mainly deep analysis and, as fall-back solution, shallow processing (configuration 2; Table 9.10).

	Precision	Recall	f-score
Template-based evaluation	57.25 %	30.58 %	39.86 %
Feature-based evaluation	83.19 %	47.13 %	60.17 %

Table 9.10: First experiment English; results using configuration 2

Second Experiment (English) The second experiment uses only deep analysis as preprocessing (configuration 1; Table 9.11).

The evaluators concluded that precision and recall values for feature-based evaluation were always higher than those for template-based evaluation due to

	Precision	Recall	f-score
Template-based evaluation	48.13 %	38.52 %	67.56 %
Feature-based evaluation	75.45 %	61.17 %	42.70 %

Table 9.11: Second experiment English; results using configuration 1

the fact that in many cases templates contained only one or two incorrect feature values. During template-based evaluation these templates were regarded as completely incorrect. This fact explained the difference between accuracy values considering whole templates and feature values, respectively.

The precision value when using only deep analysis was higher than the precision value when combining deep and shallow analysis, whereas the higher f-score in the first experiment for both evaluation types indicated that the combined approach delivered better results altogether.

The usage of shallow preprocessing mainly supported the identification of ordering templates. This was mainly due to the difficulty of recognizing templates of type exchange or mix-up when using only shallow processing. In these cases accurate recognition of predicate argument structure was a necessary precondition for making decisions such as 'What are the features of the product?' or 'Which product has been ordered and which product must be replaced?'.

Moreover, the relevant agreement features were not available in the domain-specific coreference resolution between pronouns and customer names as potential antecedent, or nouns and product named entities as potential antecedent. On the other hand, shallow processing delivered correct templates in some cases (mainly order templates) for which the deep analysis could not provide a template at all.

9.9.2.2 Summary

The application-oriented evaluations of the hybrid parsing configuration of Heart of Gold performed during the DEEPTHOUGHT project showed promising results. However, as in the latter case (email response application), the baseline had already been a hybrid system and not a deep parser. The reason for this is that the advantages of the hybrid approach are so evident that it seems (from an application point of view) to be useless to still consider isolated deep parser evaluations.

On the other hand, all evaluations cited here were performed at relatively early stages of grammar development (esp. German HPSG) and also shallow components (e.g. *SProUT* grammars) are much more developed now, and the good results from Chapter 7 did not enter in these earlier hybrid evaluations. To sum up, there is much evidence from this evaluation that hybrid processing in Heart of Gold considerably improves robustness of NLP processing for applications, and that the results would be much better now than they had been during DEEPTHOUGHT. The next section also illuminates recent developments in various other applications.

9.10 Further Applications Based on Heart of Gold

The integration of deep and shallow processing opens potential for a wide range of applications. We give an overview of implemented and envisaged applications based on the presented integration scenarios in Heart of Gold. We start with two quite detailed application descriptions, namely creativity support in document production (Section 9.10.1) and question answering from structured knowledge sources (Section 9.10.2). Further applications, conducted by colleagues in Saarbrücken and at other sites, will be briefly presented in Section 9.11.

9.10.1 Creative Authoring Support

The research performed in the DEEPTHOUGHT project aimed at demonstrating the potential of deep linguistic processing if added to existing shallow methods ensuring robustness. The approach has been used to demonstrate the feasibility of three ambitious applications, of which we have already described one in the previous section. The creative authoring application could not be evaluated because of its nature and time limits in the project, but as it constitutes an original approach that had been implemented in a fully functional prototype using Heart of Gold, we will briefly describe it in this section.

The aim of the application is to support *creative document production* (Uszkoreit *et al.*, 2004). To this end, it combines functionality for document editing with advanced semantic information retrieval and question answering. We describe the prototype and the methodology developed for combining the respective virtues of different processing methods. Using some examples, we will illustrate the collaboration of NLP components on the basis of Heart of Gold.

9.10.1.1 Motivation

When new ideas are produced, discussed, and presented, a large proportion of the effort goes into looking up and combining existing pieces of information. The reasons for this are simple: (i) the completely new ideas and facts only constitute a tiny fraction of the total content and (ii) we cannot keep all the cited facts and sources in our memory.

If the lookup of facts, sources, references, pictures can be performed with greater ease and speed, the creative process gains immensely in efficiency. If the authors are not constantly interrupted by searches and if they can spend more time on the truly creative portions of the task, the quality of the results will also improve.

Everyone who has ever authored a document remembers the numerous disruptions in situations when information is missing and it has to be looked up. Only a few years ago, one had to consult books, journals, and archives to find the required data. Today, much of the lookup can be done on the Internet or on other electronically accessible repositories. Nevertheless, any lookup is disruptive.

Experienced writers do not stop the creative process each time some piece of information is missing, but rather insert a note for later lookup. The basic idea is that this lookup can be performed automatically. This can happen while the author continues to write, or even after hours. While search and presentation are automated, the selection and the actual creative tasks are left to the human author.

The need for looking up information also occurs when complex charts or other figures are composed. Only in rare cases does the author really need to draw the pictorial elements from scratch. Today, symbols, icons or other graphical elements are readily available in clip art collections, graphics archives, or on the web. Again, one can delay the search for missing elements by inserting a dummy shape such as an empty rectangle or a circle together with a note.

9.10.1.2 Sample Scenario

In a creative meeting, the participants collectively develop a marketing plan for mobile phones. The moderator stands in front of the group entering the contributions onto an electronic flip chart (e.g. a SmartBoard) by means of electronic pen and microphone. She or he might want to insert information about the functionality of a Nokia 8890 and – using her microphone – dictates the question 'Does the Nokia 8890 possess Bluetooth?' to the application and then pushes the button for 'search'.

While the discussion continues, the system searches for the answer. Whenever a search is completed, the question will turn into green or red, depending whether an answer has been found or not. If a green question is clicked, a menu appears that lists the most highly ranked answer candidates. The answers contain links to their source, such that a browser window can be opened that contains more information about the topic (say, in this case, a web page on the features of Nokia 8890).

Next, the moderator may want to insert a picture of the phone set on the flip chart. In this case, the analyzed query is compared with analyses of natural language descriptions of pictures (such as 'this is a picture of the Nokia 8890'), and the best matches will be in the menu to choose.

9.10.1.3 Linguistic Challenges

The described application opens up a bag of challenges to linguistic processing. Answering questions requires information of varying granularity. On the one hand, the analysis of query and possible answers must be robust. It may contain named entities, which requires more robust processing than can be provided by deep parsing.

As also speech input is allowed, the processing must be able to deal with spoken language and recognition errors. On the other hand, the analysis must be as exact as possible. Recognition of negation scope and predicate-argument relations is necessary in more complex queries such as 'I want a picture of a Nokia phone, but not the Nokia 8890' or 'show me a picture of the Nokia 8890 and a table of the

features of the Siemens S55'. Modification anchors are needed to decide, if in the case of 'I want a large picture of the Nokia 8890' the user wants a large picture or a picture of the large phone.

To account for robustness and exactness of analysis the machinery that processes queries and answers uses an intelligent combination of deep and shallow NLP modules as well as standard web-based QA systems as a fall-back strategy. AnswerBus (Zheng, 2002) has been used for the latter purpose.

9.10.1.4 Application Architecture and Implementation with Heart of Gold

The key idea to overcome the outlined problem is the integrated exploitation of linguistic components that allow analysis at different levels of granularity. In this way robustness and efficiency of shallow processing is combined with the increased accuracy provided by deep analysis. The integration is facilitated by the choice of RMRS as common semantic representation language that allows flexibility in the level of detail that is specified.

Heart of Gold is used with the configuration and components as described in Section 9.6.2 to reassemble partial output from multiple components into one coherent representation.

Once a query has been entered by the user, it is sent to the Heart of Gold. From there, the search engine gets back the RMRS-annotated query. Using the repository of RMRS annotated texts, pictures and graphics, it extracts similar annotations and composes the result for presentation. It then sends the result to the text and presentation editor module via the application server.

The Creative Authoring Support Application consists of the following main modules:

- An editor for text and graphics display and input, request sending functionality and information insertion functionality, as well as speech recognition interaction.
- A server hosting the application logic.
- An information search engine with the functionality of information extraction from RMRSes and interaction with the linguistic core machine and the stored annotated texts, graphics and pictures.
- A connection to a speech recognition system.

A schematic overview of the overall architecture is depicted in Figure 9.25.

The application uses a client-server architecture, where the user client, implemented using Macromedia Flash, is usable in any common web browser via the network. The application server and the search engine have been implemented in Python. Two ways of connecting speech recognition to the system are supported.

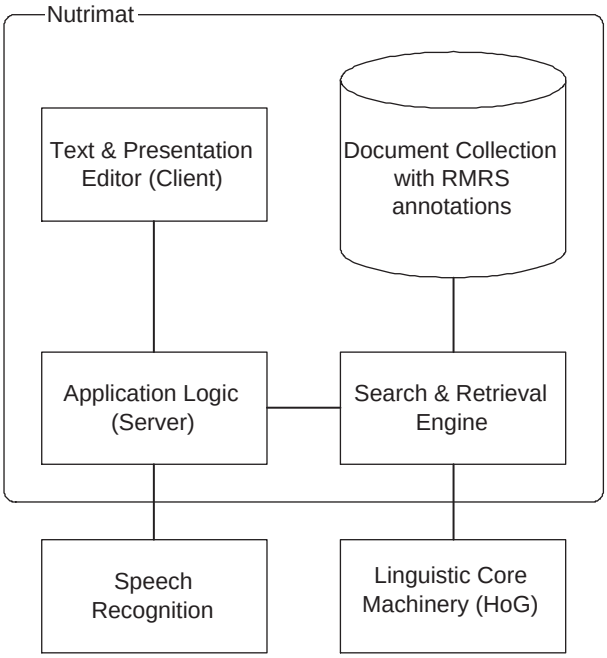


Figure 9.25: The software architecture for creative authoring

The straightforward setup uses a client-side recognition engine, e.g. a dictation system that is installed on the user's machine. Speech input without any local installation is possible using a server-side recognition engine, where the audio signal is transmitted to the server and handed to the recognition module, as displayed in the diagram.

The information search is based on a collection of texts that have been RMRS-annotated through hybrid analysis in Heart of Gold, pictures and graphics. The query is sent to Heart of Gold and returned with RMRS annotation. Based on this annotation, a search on the RMRS-annotated text, pictures and graphics is performed, using information extraction techniques.

When a search is initiated in the user interface through marking a text and pushing a search button, the query is sent to the application logic server, which in turn interacts with the search engine, and sends the query to the search engine, accompanied by query context and requested result types (pictures, texts or links). Search results can be texts, pictures or documents. They are annotated with a description (string), and a URL. They are presented to the user for selection in a pull-down menu.

Figure 9.26: The software prototype for creative authoring

9.10.1.5 Conclusion

A new application for creativity support in document authoring has been implemented with Heart of Gold as backbone. The user is assisted in composing a text, possibly enriched with pictures taken from a local repository or the web. Domain-specific questions and commands related to the content of the document can be answered on the basis of the repository, thus helping the user to perform the authoring task faster and with fewer disruptions.

The application uses robust semantic representation (RMRS) gained from a hybrid combination of deep and shallow NLP components. The application benefits from robustness and efficiency of the shallow components, as well as from increased accuracy provided by the deep HPSG parser.

The underlying XML-based, network-enabled Heart of Gold architecture is open and generic, and can be used to integrate additional NLP components and build the foundation for various other applications. In combination with ontologies, the existing framework could be extended and form the basis for further challenging applications in the context of the Semantic Web, one of which will see in the following section.

9.10.2 Question Answering from Structured Knowledge Sources

Heart of Gold is used to provide deep question analysis in the QUETAL cross-lingual question answering (QA) system for structured knowledge sources in restricted domains.

In this section, we report on how Heart of Gold, basically with the standard configurations for German and English as described in Section 9.6.1 and 9.6.2, plus some domain-specific extensions, automatically incorporated using the OntoNERdIE approach described in Section 9.7, is employed in a complex question answering system. The system is described in more detail in Frank *et al.* (2005, 2006), from which most of the following sections are extracted. Again, our contribution is the architecture for hybrid deep-shallow analysis on the basis of Heart of Gold, and our aim is to show within an elaborated scenario how RMRS output from Heart of Gold can be utilized in an NLP-based application.

9.10.2.1 Motivation and Overview

There is increasing need for question answering in restricted domains, due to several reasons: First, where open-domain QA exploits the wealth of information on the Web, it is also confronted with the problem of reliability: information on the Web may be contradictory, outdated, or utterly wrong. Second, the utilization of formalized knowledge in a restricted domain can improve accuracy, since both questions and potential answers may be analyzed w.r.t. to the knowledge base (cf., e.g. Fleischman *et al.*, 2003). Third, there is a need for accurate specialized information management solutions in both business intelligence and public administration.

QA systems for restricted domains may be designed to retrieve answers from so-called unstructured data (free texts), semi-structured data (such as XML-annotated texts), or structured data (databases). Whenever structured data can be exploited, this option offers clear advantages over open text QA. However, despite a tendency towards deeper analysis, current techniques in QA are still knowledge-lean, in exploiting data redundancy and paraphrasing techniques. That is, textual QA works on the assumption that the answer to a question is explicitly stated in some textual passage, which is typically not the case in restricted domains.

Question answering applied to restricted domains is therefore interesting and challenging in two important respects. Restricted domains tend to be small and stable enough to permit careful knowledge and data modeling in terms of structured knowledge bases, and can therefore serve as certified information sources. More importantly though, QA in restricted domains requires techniques that crucially differ from the techniques that are currently applied in open-domain textual QA. Since document volumes tend to be small, textual QA techniques cannot exploit data redundancy. Further, both in domain-restricted textual QA and QA from structured knowledge sources, we cannot expect the answer to a given question to be explicitly stated.

Since the question is the primary source of information to direct the search for the answer, a careful and high-quality analysis of the question is of utmost importance in the area of domain-restricted QA. Most importantly, since questions shall be answered where the answer is not literally stated in the underlying document or knowledge base, a semantic interpretation of the question is needed that can be tightly connected to the domain knowledge sources and the process of answer extraction.

The approach to domain-restricted QA from structured knowledge sources presented here starts from these considerations. We focus on a deep linguistic analysis of the question, with a conceptual-semantic interpretation of the question relative to the chosen application domain. By relying on Heart of Gold as core linguistic processing architecture, and RMRS as common semantic representation format, the approach naturally extends to multilingual QA scenarios and provides a natural interface to the underlying knowledge bases, enabling flexible strategies for answer extraction.

In this section, we present an overview of the architecture and the base components of the domain-restricted QA system, and the overall system architecture, the QUETAL QA system, in which the domain-restricted architecture is embedded.

We then introduce the main aspects of domain modeling for two application domains: Nobel prizes and Language Technology. We then describe hybrid question analysis based on Heart of Gold. We start from HPSG analyses of questions, which are enriched with a conceptual-semantic representation that can be further modified by domain-specific inference rules, and are extended to multilingual question answering.

Subsequently, we briefly describe the interface between question interpretation and domain ontologies for query processing. A mapping is defined between the domain-specific concepts used in semantic question interpretation and the concepts in the underlying domain ontology. This mapping is used to extract so-called *proto queries* from the semantic representation of the question. Proto queries are abstract query patterns in a higher-level query language that are translated to concrete database or ontology query language constructs in the answer extraction phase. Finally, we report on an evaluation of the prototype system.

9.10.2.2 Architecture for Domain-Restricted QA

The Architecture for domain-restricted QA is a sub-system of a more general, hybrid question answering architecture that incorporates both open-domain question answering on unstructured text (e.g. from the web; Neumann and Sacaleanu 2004), and closed-domain question answering on structured knowledge sources (Figure 9.27).

The hypothesis underlying the QUETAL architecture design is that QA systems perform best if they combine virtues of domain-specialized and open-domain QA, accessing structured, semi-structured, and unstructured knowledge bases.

The core idea is that, instead of providing specific information portals (with

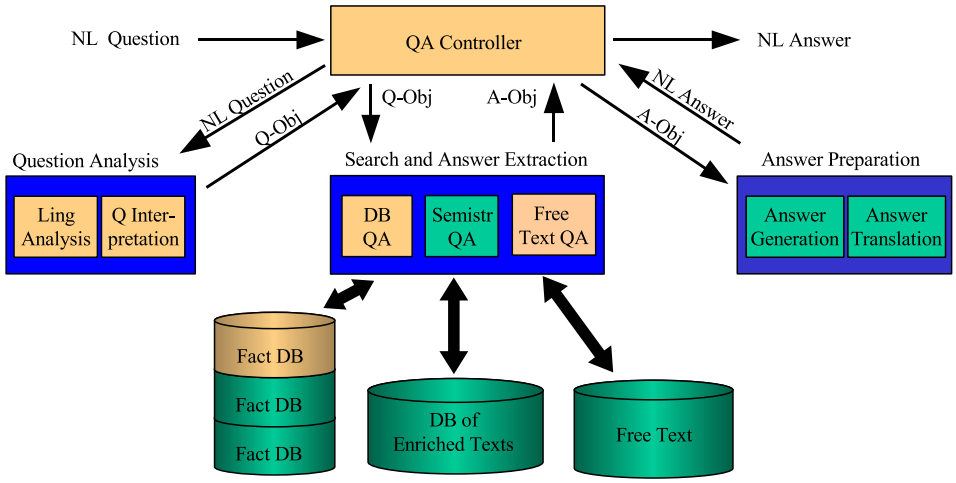


Figure 9.27: Hybrid, overall Quetal architecture

system-specific user interfaces), the Quetal system provides a single and uniform natural language-based access to different information sources that exhibit different degrees of structuring.

The sub-system for domain-restricted QA performs a deep, robust question analysis on the basis of Heart of Gold.

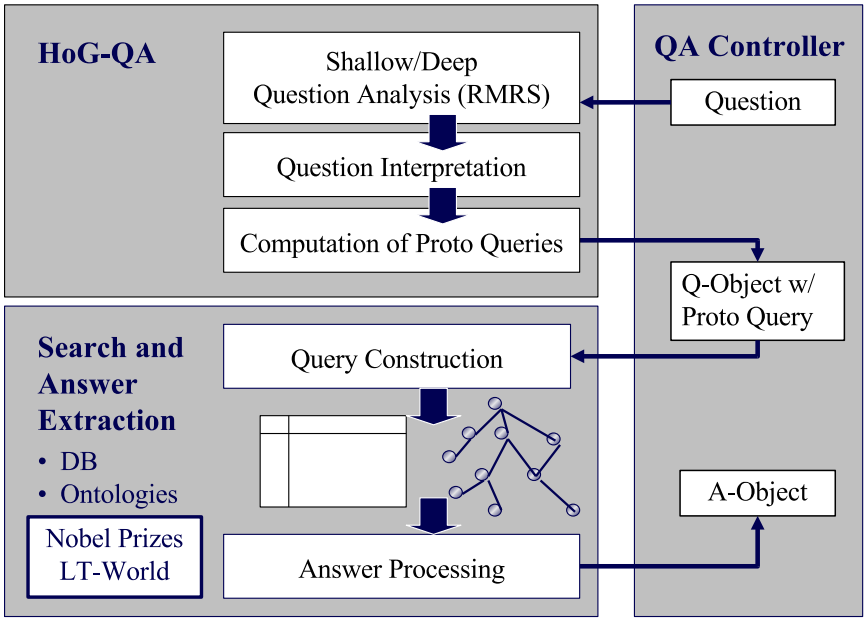


Figure 9.28: Architecture of Heart-of-Gold-based query analysis

The semantic representations generated by the Heart of Gold are then interpreted and a question object is generated that contains a proto query (cf. Figure 9.28). This proto query can be viewed as an implementation-independent, 'higher-level' representation of a database or ontology query. From this, an instance of a specific database or ontology query is constructed and sent to the database or ontology. From the result(s) returned by the queried information source, an answer object is generated which forms the basis for subsequent natural language answer generation (which we will not describe here as it is currently based on template-based shallow generated not related to or integrated in Heart of Gold).

9.10.2.3 Ontologies as Structured Knowledge Sources

Domain ontologies play a crucial role in the structured QA approach. They are used as the interface between question analysis and answer extraction and also form the formalized, queryable knowledge source itself – not only through concepts stored in the ontology, but also through instance data such as people's names, events, locations *etc.*

To demonstrate flexibility and modularity of the approach, two ontologies, LT WORLD on language technology and a considerably smaller Nobel prize ontology, have been chosen for the prototype system.

The Nobel prize ontology has been designed by Feiyu Xu. It contains simple concepts such as `prize`, `laureate`, `prize-area`, `organization`, `monetary value`, `person`, `prize-area`, `location`, `date`, `time`, `nobel-prize-winning`, `nobel-prize-awarding`, `nobel-prize-nomination`, plus various sub-concepts.

The LT WORLD domain is much bigger, containing more than 600 concepts (classes), 200 properties, and 20000 instances (approx. unique 400000 RDF triples in sum), of which parts have been extracted semi-automatically from diverse web sites and data bases. The LT WORLD ontology has originally been developed for a comprehensive web portal (Uszkoreit *et al.* 2003; `http://www.lt-world.org`) on Human Language Technology, providing information about people, technologies, products, resources, projects, and organizations in this area.

Entries for real projects, person names, events or organization are *instances* of the ontology concepts. For example, people actively working in Language Technology are modeled in the ontology as instances of the concept `Active_Person`. `Active_Person` is a subclass of `Players_and_Teams` which has further subclasses such as `Projects` or `Organizations`.

The employed LT WORLD ontology is encoded in the Web ontology language OWL (Bechhofer *et al.*, 2004). OWL makes use of constructs from RDF (Klyne and Carroll, 2004) and RDFS (Brickley and Guha, 2004) such as `rdf:resource`, `rdfs:subClassOf`, or `rdfs:domain`, but its two variants OWL Lite and OWL DL restrict the expressive power of RDFS, thereby ensuring decidability. What makes OWL unique (as compared to RDFS) is the fact that it can describe resources in more detail and that it comes with a well-defined model-theoretical semantics, based on description logic (Baader *et al.*, 2003).

9.10.2.4 Querying the Ontology

Because of the large amount of instance data, the LT WORLD ontology was stored in an RDF database system (Guo *et al.*, 2004; Haase *et al.*, 2004). The basic idea is that even though OWL ontologies are employed, the information that is stored is still RDF on the syntactic level. Therefore, a good candidate is an RDF database system which can interpret the semantics of OWL and RDFS constructs such as `rdfs:subClassOf` or `owl:equivalentClass`.

For the structured QA system, the LT WORLD ontology has thus been stored in Sesame (`http://www.openrdf.org`), an open-source middleware framework for storing and retrieving RDF data (Broekstra *et al.*, 2002; Aduna B.V., 2004). Sesame partially supports the semantics of RDFS and OWL constructs via entailment rules that compute 'missing' RDF triples in a forward-chaining style at compile time.

These predefined rules can be altered and the XML rule file can be extended, according to the users' needs. Termination is guaranteed as (long as) no new classes or instances are introduced.

Since sets of RDF statements represent RDF graphs, querying information in an RDF framework means to specify path expressions. Sesame comes with a powerful query language, SeRQL (Broekstra and Kampman, 2003), which turned out to be sufficiently powerful in order to retrieve the right objects from the LT WORLD ontology. The query syntax and structure is similar to relational database query languages (cf. examples later).

9.10.2.5 Hybrid NLP for Question Analysis

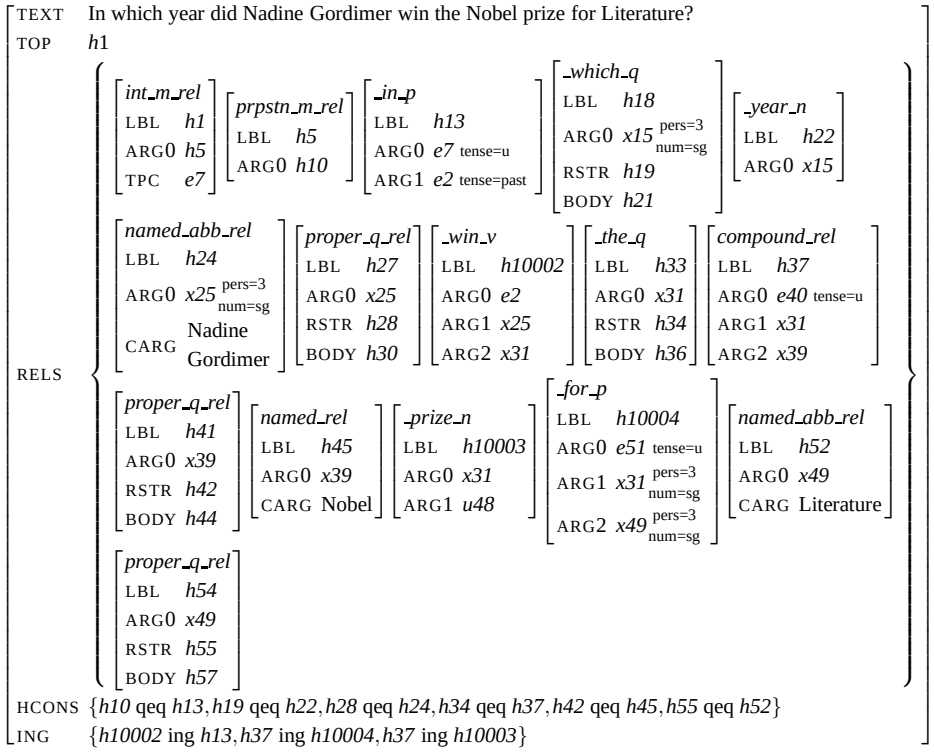
Natural language question processing starts with generic syntactic and semantic analysis on the basis of HPSG parsing in the Heart of Gold architecture. For increased robustness, the HPSG parser is seamlessly integrated with the information extraction system *SProUT* (Drożdżyński *et al.*, 2004).

SProUT performs named entity recognition (NER) on the basis of unification-based finite-state transduction rules and gazetteers (standard NE grammars for English and German). It provides structured representations both for general named entity classes and domain-specific terms and named entities.

The lingware resources for *SProUT* were extended by the automatic OntoNERdIE ontology mapping (Schäfer, 2006b) as described in Section 9.7 with concepts and instances from LT WORLD, where recognized instances also return an object identifier as unique pointer into the ontology (for an example cf. Section 9.7).

Furthermore, the part-of-speech tagger TnT helps to guess the word class of unknown words other than named entities. The hybrid integration scenario used roughly corresponds to the configurations described in Section 9.6 for German and English.

HPSG parsing in Heart of Gold delivers semantic representations in the for-



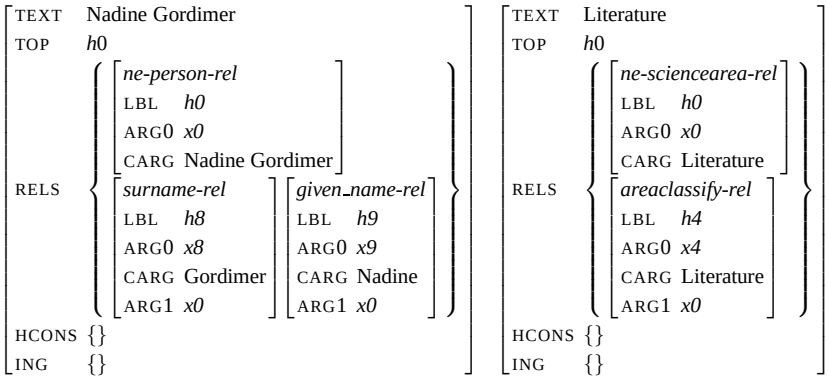


Figure 9.29: RMRS of HPSG analysis (top) and *SProUT* NE recognition (bottom)

malism of Robust Minimal Recursion Semantics (RMRS; Copestake 2003), as also already described in Section 9.4.

We exemplify an RMRS generated by Heart of Gold of a question in Figure 9.29. It can, roughly, be read as an interrogative proposition (*int_m_rel*) with a wh-quantified modifier '*in which year*', where the modified event *e2* is a winning relation, its logical subject ARG1 refers to an individual *x25*, with proper name '*Nadine Gordimer*', and whose logical object *x31* is represented as a definite quantified (*_the_q*) compound noun (*compound_rel*) composed of a head noun relation *_prize_n* and a proper name relation ('*Nobel*').

The former is modified by the PP '*for Literature*', where the preposition's ARG1 refers to the variable of the modifié (*x31*), and its ARG2 to the variable for *'Literature'*, which is recognized, by NE recognition, as a proper name in the domain of Nobel prizes. As seen in the bottom structures, NE recognition delivers EPs for the main NE relation types (here, *ne-person-rel* and *ne-sciencearea-rel*), together with more fine-grained information, such as *surname* and *given_name* relations. The latter are represented as modifiers, taking the ARG0 variable of the main relation as value of their ARG1 argument.

The RMRSes of the *SProUT* NER component are available as highly structured, IE-like NE representations, decomposing, for instance, a complex person name into *surname* and *given_name* relations. The identified NE classes are further mapped to coarse-grained HPSG NE-types (see *named_abb_rel*), which are directly delivered to the HPSG parser to enhance robustness.

Both these highly structured RMRS representations and the coarse-grained HPSG types are produced at runtime by XSLT stylesheets that are automatically generated at compile time from the output structure specifications of *SProUT* NE types as part of the Heart of Gold framework as described in Section 9.5.4.1.

9.10.2.6 Question Interpretation

The challenge of the question interpretation phase is to extract from a general semantics representation of the question encoded in RMRS the *queried variable* along with sortal information for this queried variable, the *expected answer type (EAT)*, for later database or ontology query construction.

The RMRS analysis of questions as delivered by HPSG parsing marks the proposition with the semantic relation *int_m_rel* for interrogative message type (Figure 9.29). In wh-questions, interrogative pronouns introduce sortal relations for the queried constituent, such as *person_rel* (who), *thing_rel* (what), *time_rel* (when), *etc*. For wh-phrases with nominal heads, the semantic relation introduced by the noun constrains the semantic type of the queried constituent (see *_year_n* in Figure 9.29). Imperative sentences such as *'List all persons who work on Information Extraction.*' introduce an imperative message type *imp_m_rel*.

The question interpretation module provided by Anette Frank takes as input the RMRS representations of the question as delivered by hybrid analysis in the Heart of Gold: the RMRS produced by the English or German HPSG parsers, and the

RMRSes for recognized named entities (Figure 9.30).

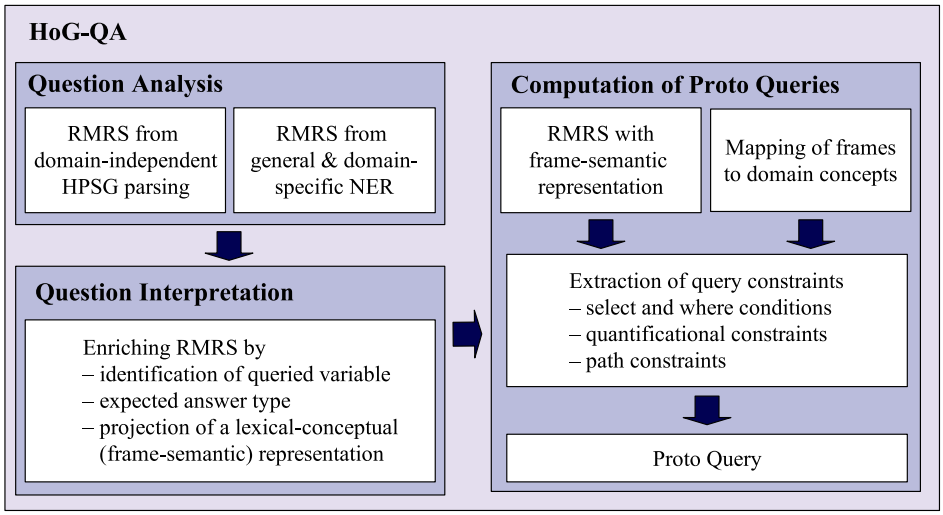


Figure 9.30: Question interpretation in HoG-QA

Interpretation rules are formulated and applied using the term rewriting system of Crouch (2005) that refer to (partial) argument structures in the RMRS in order to identify and mark the queried variable *q_var* of the question.

Furthermore, the ontological type of the queried variable is computed, which provides important semantic constraints for answer extraction. Pronominal wh-phrases introduce a semantic relation for the queried variable, such as *person*, *location*, or *reason*. For these general concepts, as well as for wh-phrases headed by common nouns, a concept lookup is performed, either by mapping a general ontological class using WordNet (Miller *et al.*, 1993), or by directly mapping the lexeme to its corresponding domain concept.

For the example displayed in Figure 9.29, this yields the additional semantic constraints: *q_var(x15)* and *EAT(x15, 'year')*, with *x15* the variable corresponding to *'year'*. These additional constraints are encoded in the RMRS by way of elementary predications (EPs) *q_focus* and *EAT_rel*, as seen below. In both EPs the value of the ARG0 feature identifies the queried variable. *EAT_rel* in addition encodes the feature SORT, which takes as value the sortal type determined for the queried variable.

$$\begin{bmatrix} \text{REL} & \textit{q\_focus} \\ \text{ARG0} & \textit{x15} \end{bmatrix} \begin{bmatrix} \text{REL} & \textit{EAT\_rel} \\ \text{ARG0} & \textit{x15} \\ \text{SORT} & \textit{year} \end{bmatrix}$$

The RMRS as logical form of the question now explicitly encodes the queried

variable, along with ontological restrictions as additional sortal constraints. The remaining EPs in the RMRS define relational constraints on the requested information. In our example, we are looking for the time when a Nobel prize was won, by a person named '*Nadine Gordimer*', where the area was '*Literature*'. These are the key relational constraints that need to be satisfied when retrieving the answer from the underlying knowledge base.

It is the task of question interpretation to identify these relational constraints on the basis of the semantic representation of the question. These constraints can then be translated to a search query in the formal query language of the underlying knowledge base. We perform this task in three steps: We first enrich the RMRS with a frame-based lexical-conceptual representation.

On the basis of a pre-defined set of domain-relevant frames and roles we extract from this enriched representation relational constraints for query construction. These relational constraints, defined in a so-called *proto query*, can then be translated to a search query with corresponding domain-specific concepts and properties, to retrieve the requested information from the knowledge base.

The motivation for this approach is two-fold: First, the projection of a frame-based lexical-conceptual structure yields a normalized semantic representation that naturally accounts for linguistic variants, or paraphrases of questions. It further constitutes a natural approach for multilingual and cross-lingual question answering in restricted domains.

Second, by defining a set of domain-relevant frames and roles we can establish a modular interface between the linguistically determined lexical-conceptual representation of the question and the concepts of the underlying knowledge bases. On the basis of a mapping between domain-relevant frames and corresponding concepts in the domain ontologies, we can efficiently identify and extract the domain-relevant constraints from the semantic representation of the question.

These constraints are encoded in a proto query that is handed over to the answer extraction process. The use of abstract proto queries gives us a clean interface that abstracts away from the syntax and functionality of the back-end query languages.

9.10.2.7 Projection of a Frame-Semantic Representation

The RMRS of the question is enriched with a lexical-conceptual projection, following the theory of Frame Semantics, as pursued in the FrameNet project (Baker *et al.*, 1998). FrameNet is building a lexical database of frame-semantic descriptions for English verbs, nouns, and adjectives.

A *frame* models a conceptual situation with a number of concept-specific roles that identify the participants in the situation. Each frame lists a number of *lexical units* that can evoke the corresponding frame.

An important motivation for using a frame-semantic projection is that – due to their design as lexical-conceptual semantic structures – frames account very naturally for the normalization of paraphrases. For illustration, consider the semantically equivalent paraphrases in (a), which are all very typical expressions for

requesting information from a database about Nobel prizes, e.g. in questions such as *When did Marie Curie win the Nobel prize for Physics?*

a. *(win / be awarded / obtain / get / be winner of) a prize*

HPSG semantic representations in terms of (R)MRS, however, are tailored to account for structural semantic properties such as quantifier scoping and predicate-argument structure, and thus still reflect the various different argument structures involved, as illustrated in (b).

b. Different argument structures in RMRS representation

$$\left(\begin{bmatrix} \text{REL} & \textit{win/get/obtain} \\ \text{ARG0} & \textit{e1} \\ \text{ARG1} & \textit{x1} \\ \text{ARG2} & \textit{x2} \end{bmatrix} \vee \begin{bmatrix} \text{REL} & \textit{award} \\ \text{ARG0} & \textit{e1} \\ \text{ARG1} & \textit{u1} \\ \text{ARG2} & \textit{x2} \\ \text{ARG3} & \textit{x1} \end{bmatrix} \vee \begin{bmatrix} \text{REL} & \textit{winner} \\ \text{ARG0} & \textit{x1} \\ \text{ARG1} & \textit{x2} \end{bmatrix} \right) \begin{bmatrix} \text{REL} & \textit{prize} \\ \text{ARG0} & \textit{x2} \end{bmatrix}$$

Following related work in Frank and Erk (2004), the RMRS representations are enriched with a frame-semantic projection, by mapping the different argument structures of verbs or nouns to their corresponding frame structure, which states the name of the frame and its frame-specific roles. An example of such a frame assignment rule is given in (c).

c. RMRS-based frame assignment rule

$$\begin{bmatrix} \text{REL} & \textit{win} \\ \text{ARG0} & \textit{e1} \\ \text{ARG1} & \textit{x1} \\ \text{ARG2} & \textit{x2} \end{bmatrix} \begin{bmatrix} \text{REL} & \textit{prize} \\ \text{ARG0} & \textit{x2} \end{bmatrix} \Rightarrow \begin{bmatrix} \text{GETTING} & \textit{e1} \\ \text{SOURCE} & \textit{u1} \\ \text{THEME} & \textit{x2} \\ \text{RECIPIENT} & \textit{x1} \end{bmatrix} \begin{bmatrix} \text{AWARD} & \textit{x2} \\ \text{LAUREATE} & \textit{x1} \\ \text{DOMAIN} & \textit{u3} \end{bmatrix}$$

(d) displays the frame-semantic representation obtained for the partial RMRS variants in (b).

d. Conceptual (frame-semantic) representation

$$\begin{bmatrix} \text{GETTING} & \textit{e1} \\ \text{SOURCE} & \textit{u1} \\ \text{THEME} & \textit{x2} \\ \text{RECIPIENT} & \textit{x1} \end{bmatrix} \begin{bmatrix} \text{AWARD} & \textit{x2} \\ \text{LAUREATE} & \textit{x1} \\ \text{DOMAIN} & \textit{u3} \end{bmatrix}$$

The frame-semantic representations are further enriched by applying forward-chaining inference rules. The purpose is to fill gaps between the (generic) frames and some domain-specific knowledge, but also to assign non-instantiated arguments, and furthermore to correct mismatches between the generic, domain-independent linguistic analysis and the structure of the underlying knowledge base.

We only give a simple, illustrative example here. More details and further examples can be found in Frank *et al.* (2006).

The sample rule below maps the temporal modifier of the winning event to the TIME role of the GETTING frame, i.e., the time of receiving an award is equal to the time (attribute) of the award. This information would otherwise not be explicit in the frame-enriched representation of the question.

$$\begin{bmatrix} \text{GETTING} & e1 \\ \text{THEME} & x2 \\ \text{TIME} & t1 \end{bmatrix} \begin{bmatrix} \text{AWARD} & x2 \\ \text{TIME} & u2 \end{bmatrix} \Rightarrow \begin{bmatrix} \text{AWARD} & x2 \\ \text{TIME} & t1 \end{bmatrix}$$

9.10.2.8 Proto Query Construction

Besides the frame-enriched and by inference rules augmented RMRS, a further mapping from domain-relevant frames and roles to concepts in the underlying knowledge bases is necessary and performed using the already mentioned term-rewriting system. These additional constraints will become constraints for the query construction.

From the frame-semantic structure and the query constraints, proto queries will be constructed. A basic distinction for the construction of structured query terms is the distinction between queried vs. constraining concepts and attributes.

For the extraction of *queried concepts*, we select those domain-relevant frames and roles that correspond to the queried variable (*q_var*) in the logical form, represented as the ARG0 argument of the *q_focus* relation (cf. Section 9.10.2.6).

We further extract the corresponding ontological restrictions encoded as the expected answer type in *EAT_rel*. We extract all remaining (non-queried) domain-relevant frames and roles, which provide additional constraints on the queried concepts. Again, we extract ontological restrictions, here in terms of their named entity type, as encoded by the RMRS structures provided by named entity recognition in the Heart of Gold.

Subsequent rules further identify the value of the constraint, in general the main predicate (relation) or CARG (constant name) associated with the role's variable, such as 'Marie Curie' in (b), or time constants for temporal constraints.

Proto queries may be complex, that is, they may be decomposed into individual sub-queries with specially marked dependencies. Therefore, all conditions that pertain to a single sub-query are marked by a common sub-query index (`qid`).

We exemplify a short proto-query for the question *In which areas did Marie Curie win a Nobel prize?*

```
<PROTO-QUERY id="1">
  <SELECT-COND qid="0" rel="award" attr="domain"
               sort="FieldofStudy">
  <WHERE-COND  qid="0" rel="award" attr="laureate" netype="person"
               val="Marie Curie">
</PROTO-QUERY>
```

Quantificational questions Question answering from structured knowledge bases is particularly well suited to answer questions for which the answer is not explicitly represented in the document or knowledge base, but must be inferred from the available information. Prime examples are cardinality, quantificational or comparative questions, as below.

1. *How many researchers won a Nobel prize for Physics before 1911?*
2. *Which institution has published most papers between 2000 and 2004?*
3. *Which nation has won more Nobel prizes in Physics than the U.S.?*

To account for these quantificational aspects, we employ special proto query conditions `OP-COND` and `QUANT-COND`. These constructs go beyond the formal power of ontology query languages such as SeRQL, but can be translated to special post-processing operations in the answer extraction phase.

The quantificational conditions are strongly determined by the semantic representation of the question, e.g. for the scope of a *how many* question. It is for this reason that the computation of proto queries is tightly integrated with question interpretation (see Figure 9.30, 9.31).

```
<PROTO-QUERY id="1">
  <SELECT-COND qid="0" rel="award" attr="laureate" sort=""/>
  <WHERE-COND  qid="0" rel="award" attr="time" netype=""
               valfunc="before" valarg="1911"/>
  <WHERE-COND  qid="0" rel="award" attr="domain"
               netype="sciencearea" val="Physics"/>
  <OP-COND oprel="card" domain-type="answer" domain-id="0"/>
</PROTO-QUERY>
```

Figure 9.31: Proto query for *How many researchers won a Nobel prize for Physics before 1911?*

9.10.2.9 Multi- and Cross-Linguality

The frame- and RMRS-based approach to question interpretation naturally extends to multilingual and cross-lingual QA scenarios. Since frames are defined as lexical-conceptual structures, they are to a large extent language-independent. Thus, question interpretation in terms of a frame-semantic representation effectively implements a kind of 'interlingua' approach for question answering: the frame-semantic representations serve as a language-independent interface to the underlying knowledge bases.

As illustrated in Figure 9.32, HPSG grammars for different languages – in our scenario, English and German – provide semantic structures in a uniform formalism, (R)MRS. The language-specific relations in these semantic forms are

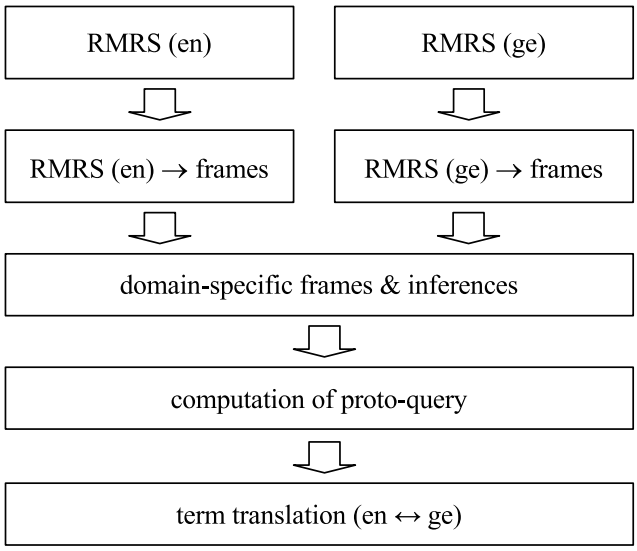


Figure 9.32: Natural language utterances (language-specific) to proto-queries: frame-semantic representations as interlingua for multilingual QA

translated by language- and lexeme-specific frame projection rules to a common, language-independent frame-semantic representation.

The remaining parts of the question interpretation and answer extraction processes are then uniform across languages. Domain-specific inference rules refer to the common frame-semantic representations, thus they are applied to the same type of intermediate structures in question interpretation, irrespective of whether they were produced by German or English HPSG grammars. Similarly, the subsequent rules for the extraction of proto queries uniformly operate on the language-independent frame-semantic representations (see Section 9.10.2.8).

For cross-lingual QA from structured knowledge sources, we perform term translation for instances (named entities) and domain-specific terms of the knowledge base that can appear as values in search queries constructed from the question's representation.

Actually, very moderate development effort was required when porting the question processing module from English to German.

9.10.2.10 Answer Extraction

The answer extraction phase, provided by Hans-Ulrich Krieger, mainly consists of translating the proto query to a database or ontology query language expression and sending it to the database or ontology.

For the database case, the answer extraction took place on a relational database and the translation of the proto queries was quite straightforward from the similar constructs, e.g. the SELECT-COND corresponding to a SQL SELECT statement, and the WHERE-COND corresponding to a SQL WHERE clause.

We will therefore skip the SQL translation part which is described in more detail in Frank *et al.* (2006), and turn to the more challenging ontology query language part, again skipping some details that can be found in the article, by showing how a proto query can be mapped to an expression in the query language SeRQL of Sesame.

Based on the mapping from domain-specific frames and roles in the proto query conditions to domain concepts and properties (see Section 9.10.2.8), we first perform a translation of the values of `rel`, `attr`, and `path` attributes to the corresponding domain concepts and attributes of the LT World ontology. Thus, each relation (value of `rel`) now denotes a concept in the ontology and each attribute (value of `attr`) denotes an OWL property.

In a SeRQL query, instances of a concept are identified by variables in the subject position of an RDF triple. The concept itself is stated in the object position, and subject and object are connected by `rdf:type` – this is exactly the way how instances of a specific concept are represented in the RDF base of Sesame. For example,

```
<SELECT-COND qid=".." rel="Organizations" attr="locatedIn" ... />
```

leads to the introduction of the following RDF triple (`_r` is a fresh variable, `ltw` the LT WORLD namespace):

```
{_r} rdf:type {ltw:Organizations}
```

Since attributes such as `locatedIn` refer to properties of a concept, we obtain a further triple:

```
{_r} ltw:locatedIn {_q}
```

The property `locatedIn` connects instances of the main concept `Organizations` via the root variable `_r` with the queried information. The queried information is bound to a new question variable `_q` that will be returned. It is marked by the SELECT clause in a SeRQL query:

```
SELECT {_q}
FROM {_r} rdf:type {ltw:Organizations},
     {_r} ltw:locatedIn {_q} ...
```

Figure 9.33 contains the main principles of the transformation from proto queries to SeRQL queries. To illustrate the transformation principles, we consider the question *'Who is working in the Quetal project?'*, with its (simplified) proto query that contains a SELECT and a single WHERE condition:

(1)	for each SELECT-COND and WHERE-COND
	– each relation denotes a concept
	– each attribute denotes a property
	– each unique relation introduces a new *root* variable
(2)	each SELECT-COND introduces a new *query* variable
(3)	each WHERE-COND introduces a new *local* variable
(4)	guarantee that the RDF triples form a connected graph
	– if path constraints are specified, link the root variables
	– otherwise, introduce new *property* variables linking the roots
(5)	finally apply OP-COND to the result table (if needed)

Figure 9.33: Principles for transformation of proto queries to SeRQL queries

```
<PROTO-QUERY>
  <SELECT-COND rel="Active_Person" attr=""/>
  <WHERE-COND rel="Active_Project" attr="projectName" val="Quetal"/>
</PROTO-QUERY>
```

Given this proto query, the following SeRQL query is generated:

```
SELECT DISTINCT _q0
FROM {_r1} rdf:type {ltw:Active_Person},
     {_r2} rdf:type {ltw:Active_Project},
     {_r1} ltw:name {_qo},
     {_r2} ltw:projectName {_l3},
     [ {_r1} _p4 {_r2} ],
     [ {_r2} _p5 {_r1} ]
WHERE (NOT (_p4 = NULL) AND (_p5 = NULL)) AND (_l3 LIKE "Quetal")
```

Further details and optimizations are described in Frank *et al.* (2006).

The answers from the database or ontology are encoded in a structured *answer object*, similar to the query object that embodies the original proto query (Figure 9.28).

An answer object refers to the query id of the query object and distinguishes between potential conflicting answers (several VALUES tags) and list-based answers (a single result, consisting of several pieces; several VALUE tags). Similar to question objects, the answer objects serve as XML interchange structures in the QA architecture. That is, the same type of structure is returned by MySQL for the Nobel prize domain and Sesame in the LT WORLD domain.

As an example, the answer object generated for the question *'Who is working in the Quetal project?'* from the LT WORLD ontology is

```
<AOBJ id="id18" msg="answer" query-id="Q01" lang="EN">
  <ANSWER type="complex" score="1.0">
```

```
    <VALUES>
      <VALUE>Anette Frank</VALUE>
      <VALUE>Berthold Crysmann</VALUE>
      <VALUE>Bogdan Sacaleanu</VALUE>
      <VALUE>Feiyu Xu</VALUE>
      <VALUE>Günter Neumann</VALUE>
      <VALUE>Hans Uszkoreit</VALUE>
      <VALUE>Hans-Ulrich Krieger</VALUE>
      <VALUE>Ulrich Schäfer</VALUE>
    </VALUES>
  </ANSWER>
</AOBJ>
```

9.10.2.11 Evaluation

In this section, we report on an initial evaluation of the prototype system for domain-restricted QA from structured knowledge sources. A system-internal evaluation assesses the quality and efficiency of question interpretation and answer extraction. In addition, a comparative evaluation of the domain-restricted system to the web-based open-domain textual QA system AnswerBus (Zheng, 2002) has been performed. This, in conjunction with a detailed classification of question types, allows to assess the added value of a specialized, domain-restricted QA component in a hybrid system architecture [23].

The comparative evaluation to AnswerBus restricts us to questions in English; we further chose the Nobel prize domain, as information about this domain is appropriately covered by the WWW. We compiled a set of 100 English questions about the Nobel prize domain, in part adapted or inspired from the FAQ sections of Nobel prize web portals.

Question classification The question types in the test set range from factual and list questions to different types of cardinality and quantificational questions. Table 9.12 shows a detailed overview of the different question types and their distribution over the sample set, along with a classification of the questions' expected answer types, again with quantitative distribution (the types are overlapping, so the figures do not sum up to 100%).

The questions are varied in terms of paraphrases (verbal and nominal paraphrases, interrogative, non-interrogatives or embedded questions, such as '*Give me a list of...*', '*Could you tell me in which year...*'), and according to different types of constraints to be used in question interpretation and answer extraction, such as

[23] The textual QA system of QUETAL obtained the best results in the 2004 cross-lingual CLEF task (Neumann and Sacaleanu, 2004). However, at the time of evaluation, it was not yet extended to web-based QA. Since we do not yet have access to appropriately large document bases for our two domains, it seemed most appropriate to choose an independent open-source web-based QA system and to perform the comparative evaluation in the Nobel prize domain, for which enough information can be found on the Web.

question type	in %	expected answer type	in %
factual	58	time	13
list	15	person, organization	54
definition	2	currency	3
cardinality (how many)	22	prize area	14
quantificational (most)	24	nation	12
event quantification	2	achievement	1
embedded questions	17		

Table 9.12: Distribution of question types and expected answer types

(relational) temporal constraints (*in/before/since/after 1999*), gender (*female prize winners*), prize areas, as well as countries, locations, and affiliations.

Question processing and interpretation For the 100 questions the average runtime (real time) per question was 3.74 seconds for full online processing from text input to answer object output, on a Intel Xeon 2.5 GHz Linux machine. Answer extraction (ontology query in Sesame with the complete LT WORLD data) alone took 125 milliseconds per query object on average. For four questions, the linguistic analysis failed, i.e. Heart of Gold could not return a deep analysis.

The question sample contains 18 questions that instantiate two types of event quantification which are not yet accounted for by the question interpretation module. For the qualitative evaluation, we accordingly distinguish between the full question sample as basis for evaluation, displayed in the first row of Figure 9.13, while the figures in the second row are computed on the basis of the 82 questions that can currently be considered as in-scope phenomena.

The HPSG grammars used were equipped with stochastic models for parse selection (Oepen *et al.*, 2002a) that are, however, general and not trained for the specific domain or question answering as such. In the current set-up, the question interpretation module applies to the three highest-ranked semantic HPSG analyses, and delivers a separate question object for each of them.

Table 9.13 contains an overview of the distribution of correct proto queries over the highest-ranked parses (columns 2-4), as well as the overall number of correct proto queries across the three highest-ranked analyses (columns 5-7). Of the overall set of 100 questions, 46% return the correct proto query for the highest-ranked parse, 41% and 32%, respectively, for the second and third ranks; restricted to the in-scope phenomena, the figures raise to 56.1%, 50% and 39%, respectively.

In many cases, question interpretation extracts a correct proto query from more than one of the three best parses: For 18% of all questions (22% of the in-scope samples) we obtain the correct proto query from all three parses considered. 24% (29.3%) return two correct proto queries; for 17% (20.7%), we obtain a single

	correct pq in n^{th} parse (in %)			number of correct pq's per q (in %)			overall proto query results (with voting)			
	1st	2nd	3rd	3	2	1	corr.	uncert.	wrong	no pq
full sample	46.0	41.0	32.0	18.0	24.0	17.0	58.0	3.0	15.0	24.0
in-scope only	56.1	50.0	39.0	22.0	29.3	20.7	69.5	2.4	7.3	19.5

Table 9.13: Evaluation of question interpretation (pq=proto query)

correct proto query from the three highest-ranked analyses.

Since we are considering the three best parses, we can apply a voting scheme to determine which one of possible alternative proto queries to send to the answer extraction module. In cases of non-conflicting multiple results, voting is not necessary. However, we often obtain proto queries that are partial, or less specific than another proto query result for the same question, which hence could lead to wrong answers.

In those cases where the partial query is subsumed by both alternative analyses, or by a single alternative analysis out of two resulting proto queries, we ignore the partial query, in favor of the more specific one. In 67,9% of all cases that involve partial queries (28 on the full corpus), this strategy yields a correct proto query.

In cases where a proto query is subsumed by only one of two alternative proto queries, we mark the result as uncertain. This occurs in 3% (2.4%) of cases. For 24% (19.5%) of the questions, all analyses return an empty proto query, and are thus to be regarded as out of system coverage. These cases are either due to problems in the semantic analysis (failed or wrong parses or parse selection), or in the question interpretation process.

As seen in Figure 9.13, the overall ratio of correct proto queries that result from the voting and filtering process is 58% (69.5%). With 15% (7.3%), we achieve a moderate error rate, opposed to a higher rate of cases where the system signals that it is uncertain (3%/2.4%) or unable to answer the question (24%/19.5%).

Overall, then, the system features relatively high precision that is balanced against a low error rate and reduced recall. This tendency is especially welcome for a domain-restricted QA system that is confronted with high user expectations regarding the reliability of the answers delivered.

Another outcome of the evaluation is that a high percentage of the unanswered questions failed because the correct parse was not promoted high enough by stochastic parse selection. This could be improved by retraining the stochastic disambiguation model on typical question samples instead of the used QA-unrelated corpus.

Answer extraction We evaluated the answer extraction module on the basis of the 58 correct proto queries that were selected by the voting procedure. Table 9.14 presents the results: for 74.1% of the proto queries the correct answer was returned;

	correct	incorrect	no answer
abs. #	43	4	11
in %	74.1	6.9	19.0

Table 9.14: Evaluation of answer extraction (based on 58 correct proto queries)

question type	in % (in-scope)	expected answer type	in %
factual	53.4 (64.6)	time	46.2
list	40.0 (42.0)	person, organization	40.7
definition	100.0 (100.0)	currency	0.0
cardinality (how many)	18.2 (22.0)	prize area	42.9
quantificational (most)	25.0 (54.5)	nation	58.3
event quantification	100.0 (100.0)	achievement	0.0
embedded questions	47.1 (47.1)		

Table 9.15: Distribution of correct answers over question types and expected answer types

in 6.9% the answer was wrong; for 19%, finally, no answer was returned.

Error analysis for the 4 incorrect answers yielded a single minor cause of error (wrong answer type identification). For missing answers we identified several causes that need to be adjusted: mismatches of concept-database mappings, wrong table selection and out of scope phenomena.

Table 9.15 details the distribution of correct answers over different question types, with restriction to in-scope phenomena in parentheses.

Comparison to AnswerBus In order to assess the added value of a domain-restricted QA component, we compare the results of our current prototype system to the results delivered by the open-domain textual QA system AnswerBus (Zheng, 2002). We collected the three highest-ranked answers returned by AnswerBus, and evaluated the returned answers (Table 9.16).

The coverage on our 100 question sample is rather poor: for only 15% of the questions it delivered a correct answer within the first three ranks. Detailed

	correct for top n results				correct for question types			
	1st	2nd	3rd	overall	fact	card	quant	embedded
in %	9.0	8.0	8.0	15.0	22.4	4.5	4.2	5.9

Table 9.16: Distribution of correct answers (AnswerBus)

analysis of the distribution of results over question types shows that AnswerBus fares moderately well for factual questions, but shows poor performance for other question types, such as cardinality, quantificational, or embedded questions. Of the remaining question types, none could be answered.

9.10.2.12 Conclusion

We have described a novel approach for domain-restricted QA from structured knowledge sources, building on deep semantic analysis of the question provided by Heart of Gold, with a modular interface between linguistically motivated semantic representations and domain-specific models, in terms of ontologies or domain databases. The architecture embodies a flexible interface to various types of knowledge storage devices and their corresponding query languages.

Compared to earlier work on research in natural language interfaces to databases (cf. Copestake and Sparck Jones (1990) for an overview), using an *existing* broad-coverage HPSG grammar for deep question analysis, frame-semantic structures, proto-query construction, ontologies and ontology query languages in a modular architectures is new and adds some new, also engineering-oriented advantages (reusability, portability to new domains and knowledge sources *etc*).

Heart of Gold plays a crucial role not only for the multilingual, broad-coverage deep analysis of questions which is domain-independent on purpose, but also for the flexible, automatic integration of domain-specific knowledge resources via e.g. named entity recognition modules (*SProUT* with lingware resources derived from ontologies using OntoNERdIE in the presented scenario).

The evaluations, although performed on a preliminary, prototypical system, show clear improvements compared to classical shallow QA techniques.

9.11 Further Applications

In this section, we briefly describe further applications of Heart of Gold that have been conducted (or are still being conducted) by colleagues, and other external institutions.

9.11.1 Learning Transfer Rules for Machine Translation

Michael Jellinghaus at Saarland University has started work on using Heart of Gold for generating RMRS structures from multi-lingually transcribed European Parliament speeches. The RMRSes (German and English translations of the same speeches) are to be used for machine learning of transfer rules for machine translation based on RMRSes. Given the powerful, multilingual middleware architecture of Heart of Gold, the automatic annotation of the speech transcriptions can be, at least technically, be regarded as a simple by-product. Challenges for this task are the open domain and relatively long sentences of the transcribed speeches that will presumably result in many readings per analyzed sentence.

9.11.2 Parsing Japanese Dictionary Definition Sentences

Basically using ChaSen, the Japanese HPSG grammar JACY (Siegel and Bender, 2002) and other tools form Heart of Gold, a group of researchers at NTT parsed a large amount of Japanese definition sentences from a machine-readable dictionary with considerably high coverage (Nichols *et al.*, 2005). Possible applications are question answering, information extraction, ontology population, *etc.*

9.11.3 Trailfinder (Travel Information Application, Norwegian)

NTNU (University) Trondheim and Businesscape (company), Trondheim have developed an application using the Norwegian HPSG grammar (NorSource, also available from the DELPH-IN repository, cf. Section 9.12) aiming at extracting information from hiking route descriptions and supplying it for a web portal. The Heart of Gold machinery produces RMRSes which are mapped onto standardized information matrices. The input grammar has a specially developed semantics coping with aspects of paths and movement, extending the original core grammar. The application is described in Hellan *et al.* (2004) and has been developed further since then.

9.11.4 Soccer SmartWeb

SmartWeb is a multi-modal dialog system deriving answers from unstructured resources such as the Web, from automatically acquired knowledge bases and from web services. Heart of Gold is used in SOBA, the SmartWeb Ontology-Based Annotation system (Buitelaar *et al.*, 2006). SOBA automatically populates a knowledge base by information extraction from soccer match reports as available on the web. The extracted information is defined with respect to SWIntO, the underlying SmartWeb Integrated Ontology (Oberle *et al.*, 2006) in order to be smoothly integrated into the rest of the system.

The current implementation uses Heart of Gold in a configuration consisting of *SProUT* with a specialized soccer information extraction grammar, as well as TreeTagger and Sleepy for German shallow parsing. Figure 9.34 (courtesy of Anette Frank) depicts a schema of the workflow of the overall extraction process and where Heart of Gold comes into play. In this context, Gregory Gulrajani has written a SOAP binding for the MoCoMan class (in analogy to the XML-RPC server class) that is used to provide a SOAP-based web service.

9.11.5 RMRS Chatterbot

Tina Klüwer at University of Cologne has developed an interactive chatterbot on the basis of RMRS generated by Heart of Gold (Klüwer, 2006). Heart of Gold is embedded in a J2EE (Java 2 Enterprise Edition) application server that connects to Heart of Gold via XML-RPC. Hybrid RMRSes, produced by Heart of Gold in

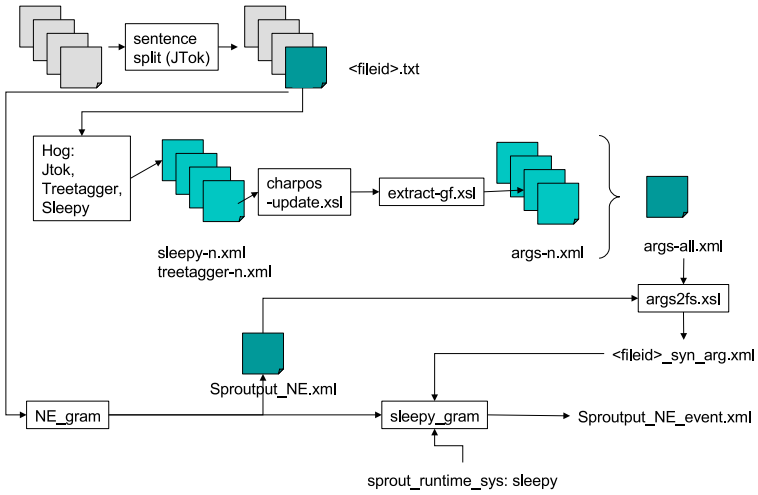


Figure 9.34: SmartWeb soccer information extraction architecture

a configuration as described in Section 9.6.2 for English, is used for an AIML interpreter (Program D[24]). AIML is an XML format for describing complex patterns and templates for simulating a human discourse.

The advantage of using RMRS instead of plain (English) text in the AIML patterns – as they are typically used in current AIML applications – is that they provide a further level of abstraction through the semantics representation, including also morphological information. Because the architecture Tina Klüwer has developed is general, as is AIML, the system could presumably also used to implement RMRS-based chatterbots in other languages for which Heart of Gold integrations exist such as German, Japanese, *etc*.

9.11.6 Training

The shallow only part of Heart of Gold (mainly taggers and chunkers for English and German) have been successfully used for training in a student camp in summer 2005 in Switzerland, by Manfred Pinkal, Aljoscha Burchardt (Saarland University) and Michael Kohlhase (International University, Bremen). The author has also used Heart of Gold for an invited course on XML-based integration of natural language processing components at the 12th European Summer School on Language and Speech Communication, Information Fusion in Natural Language Systems, in Hamburg, July 2006.

[24] `http://www.alicebot.org`

9.11.7 Anaphora, Coreference Resolution in Discourse

Nuria Bertomeu at Saarland University is using Heart of Gold for PhD research on anaphora resolution and handling elliptical questions in Question Answering systems. The hog://-URI mechanism of Heart of Gold can be easily employed to refer back to previous annotations, e.g. the sentence prior to the current sentence can be reached via `hog://sid/{acid-1}/aid`, where `acid-1` is the annotation collection ID minus one. If numerical annotation collection IDs (or at least collection IDs ending with an ordered number), increased by 1 for each sentence, are used, then the expression exactly refers to the targeted sentence, where e.g. antecedents of anaphora could be found.

9.11.8 Modern Greek Grammar

Julia Gorius (née Neu) at Saarland University has developed a Modern Greek named entity grammar in *SProUT* as well as, in cooperation with Valia Kordoni, a deep HPSG grammar for Modern Greek. Both resources have been integrated using Heart of Gold (Neu, 2004; Kordoni and Neu, 2004), i.e., the configuration consists of *SProUT*, PET and a simple tokenizer that produces PET XML input chart items from Greek words for compatibility with *SProUT* output in PET input chart format.

9.11.9 Spanish HPSG Grammar with Shallow Preprocessing

Montserrat Marimon at Universitat Pompeu Fabra, Barcelona, is currently migrating her large-scale HPSG grammar of Spanish originally developed for ALEP (Marimon, 2002b) to LKB/PET and integrating it with shallow pre-processors (FreeLing; Atserias *et al.* 2006). We have implemented a preliminary version of a FreeLingModule to integrate these resources in Heart of Gold.

The main difference to the other HPSG grammars integrated so far is that morphological analysis is performed in the shallow preprocessing phase (FreeLing) as well, whereas it is part of the HPSG grammars in all other languages integrated so far.

Through the PET input chart, the information is given to the deep grammar. As the FreeLing interface has been written first for the LKB system and its *Simple Pre-Processing Protocol* (SPPP, cf. DTD Appendix on page 293), an XSLT stylesheet (`sppp2pic.xsl`, cf. XSLT Appendix on page 300) translates into the PET input chart format.

SPPP example (English):

```
<segment>
  <token form="kim" from="0" to="2">
    <analysis stem="kim"/>
  </token>
  <token form="sleeps." from="4" to="10">
```

```
      <analysis stem="sleep">
        <rule id="plur_noun_infl_rule" form="sleeps"/>
        <rule id="punct_period_rule" form="sleeps."/>
      </analysis>
      <analysis stem="sleep" probability="0.5" pos="VP">
        <rule id="third_sg_fin_verb_infl_rule" form="sleeps"/>
        <rule id="punct_period_rule" form="sleeps."/>
      </analysis>
    </token>
  </segment>
```

9.11.10 Parsing Debian Linux User Forum Discussions (English)

Timothy Baldwin (University of Melbourne) has started a project that aims at parsing Linux User Forum Discussions on the web using Heart of Gold (with ERG and shallow preprocessing for increased robustness) to automatically provide better searchability and answers to questions already discussed.

9.11.11 SciBorg

The project SciBorg ('Extracting the Science from Scientific Publications'), started in October 2005 at the University of Cambridge in cooperation with the Nature Publishing Group, the Royal Society of Chemistry, and the International Union of Crystallography. The project aims at automatic knowledge extraction form scientific publications, in particular chemistry. According to Copestake *et al.* (2005b), Heart of Gold, at least some of the already integrated components such as RASP and PET, is under consideration as integration platform which will also and anyway be based on RMRS. In the course of the project, the implementation of the common preprocessor format for PET and LKB, as briefly sketched in section 9.5.5.2, is also foreseen.

9.12 Heart of Gold in International Collaboration

Heart of Gold has been made publicly available as open source tool under the umbrella of the DELPH-IN collaboration (Oepen *et al.*, 2002b).

DELPH-IN (Deep Processing with HPSG initiative[25]) has been founded in the context of the EU-funded project DEEPTHOUGHT by its project partners, and currently consists of various computer science and computational linguistics research labs at Cambridge University (UK), DFKI Saarbrücken, Kyung Hee University (Korea), LORIA Nancy, NTT Communication Science Laboratory (Japan) Norwegian University of Science and Technology, Saarland University, Stanford University (US), Tokyo University, University of Oslo, University of Sussex (UK), and University of Washington.

[25] `http://www.delph-in.net`, `http://wiki.delph-in.net`

The areas of collaborative research comprise robustness, disambiguation and specificity of HPSG processing, and multilingual grammar engineering.

Besides the application-oriented runtime middleware Heart of Gold, a comprehensive suite of open source grammar development system (LKB), HPSG parser (PET), HPSG grammars for various languages, evaluation, machine translation and visualization tools as well as corpora and treebanks is published under open source licenses and at the same time under continuous further development by the member institutions.

9.13 Related Work

We have already discussed NLP architectures and *ad hoc* deep-shallow integrations in Section 6.2. Except some general language technology architectures already mentioned above, there is currently no NLP architecture comparable to Heart of Gold. While Heart of Gold is unique in that it provides a highly configurable, network-enabled middleware for the integration of XML standoff annotation, provides facilities for XSL transformation and RMRS support as well as a fully integrated efficient HPSG parser, other solutions are either inflexible *ad hoc* integrations for hybrid deep-shallow processing (mostly only up to PoS tagging fed into a deep parser), or they do not support deep parsing and are hence not suitable for hybrid processing.

Löwe and Noga (2002) present a further framework that deserves mention here. It has been developed completely independently of language technology, but bears some similarity with the Heart of Gold middleware we propose[26]. Löwe and Noga (2002) describe a generic XML-based, network-enabled middleware architecture for re-usable components that explicitly makes use of XSLT as adapter language between components. It has been proposed as a generic middleware in the spirit of CORBA, DCOM or EJB. As the flexible configuration aspect for NLP components is not foreseen, nor is the concept and support of standoff annotation, it cannot be really judged as a 'concurrent' approach. However, it can well be conceived as a supporting, independent argument that the XML and XSLT-based middleware approach makes sense in the software architecture landscape.

9.14 Outlook and Future Work

The Heart of Gold middleware, as described in this chapter, although only started three years ago, has quickly grown to now constitute a considerably powerful framework for flexible integration of numerous deep and shallow processing components for various languages.

Moreover, the implementation of innovative new applications on the basis of Heart of Gold at many different sites all over the world as well as, in most cases,

[26] In fact, it has been developed almost in parallel with Heart of Gold, and at the time Heart of Gold was designed and implemented, we were not aware of this framework.

application-oriented evaluations have shown the usefulness of the approach mainly for increased robustness and uniform semantics output of hybrid NLP analyses.

The idea is that Heart of Gold, as one of the open source tools downloadable from the DELPH-IN web page, will become a standard framework for application-oriented deep-shallow integration. In this section, we will briefly sketch some ideas and plans for future developments.

First of all, one goal is the integration of new languages and NLP components as in the ongoing effort for integration the Spanish HPSG grammar as described in Section 9.11.9. Further next candidates could be e.g. French and Chinese as soon as mature HPSGs exist for these languages. The idea is that also other languages than German, English and Japanese, for which the robustness-oriented integration is already quite advanced, could benefit from lessons learned and methods and combinations developed for these three pioneering languages.

As sketched in Section 9.5.5.2, there are plans to replace the current PET input chart format by a (mainly) syntactic variant, in order to provide a common, general interface format for both LKB as development and PET as runtime system. In most cases, this will be possible via adding or adapting existing XSL transformation stylesheets. The benefit of the uniform preprocessor interface is mainly for grammar development in LKB, that currently makes testing and development on unseen text (without full lexicon coverage) uncomfortable.

A further interesting extension, though largely language resource-dependent, is adding a kind of divide and conquer strategy for structuring long sentences, e.g. contained subclauses, before sending them to the deep parser. This could for German be performed by re-using and adapting the WHITEBOARD topoparser integration in Heart of Gold (through cascades of largely re-used XSLT stylesheets), and adapting structural mappings to the current German HPSG grammar.

For other languages, however, the appropriate shallow preprocessors, lingware resources and mappings will still have to be developed or identified. We expect this to be a promising approach to increase coverage of deep parsing on very long sentences as they occur in some newspapers, literature or scientific works such as the present one.

Post-parsing reassembling (or repair) of fragmentary analyses is another extension that could further improve robustness for applications. The currently implemented, simple approach uses an XSLT stylesheet for sorting fragments output by the deep parser in the RMRS format according to their length. It returns the *n* longest fragments as a first quick solution, but shallow techniques and heuristics could also be used to try to reconstruct what a deep parser would ideally do.

Knowledge from ontologies, injected via the described OntoNERdIE mapping using *SProUT*, into the deep parser's input chart could also be employed in grammars as additional source for disambiguation. This would extend grammar semantics by both world knowledge and domain-specific knowledge and may help to reduce ambiguity and increase precision of the analysis output.

Similarly and at the same time, this would develop Heart of Gold further towards a platform for the Semantic Web. A further byproduct application of Heart

of Gold could be the automatic generation of (e.g. RMRS-) annotated corpora, optionally supported by the database interface, e.g. for machine learning and machine translation respectively.

Chapter 10

Conclusion

In this thesis, we have described our contributions to frameworks for integrating shallow and precision-oriented deep natural language processing components.

We have addressed this complex problem by introducing an abstraction layer through XML. Using XML technology, we are able to employ manifold available tools including XML transformation and query languages.

We have examined related XML corpus query languages and developed a novel framework for XSLT-based *online* integration of NLP components, as started in the WHITEBOARD annotation transformer (WHAT) and continued and streamlined in the *Heart of Gold* transformation service.

The major result of the synergy gained through hybrid deep-shallow integration are *increased robustness* and *performance*. We could show that the coverage of pre-existing HPSG grammars could be more than doubled through shallow preprocessing by part-of-speech tagging and named entity recognition. This result of course varies depending on the domain, status of grammars and resources *etc*. In WHITEBOARD, it could be shown that using shallow topological parsing as preprocessing (for German) can speed up deep parsing by a factor of 2.25.

Besides the immediately measurable gains, the XML- and XSLT-based integration frameworks also improve soft factors such as flexibility and modularity of component integration and maintainability and re-usability of lingware resources.

While the WHITEBOARD architecture mainly provided an API for access to results of NLP components organized in a strict sequence, the processing model has been made much more flexible in the successor framework *Heart of Gold* that forms a highly configurable *middleware* in between NLP components and NLP-based applications.

Furthermore, our XML approach also drastically eases multilingual NLP processing and *standoff markup* handling through its standardized Unicode-based character model.

Heart of Gold is superior to other NLP integration frameworks in that it supports configurable, multilingual web services and support for a common, though optional, semantic representation format, RMRS, that can also be used for post-

parsing integration of NLP analyses preserving and increasing robustness and precision.

Besides solving the central integration problem, we have also proposed generalizable solutions for lingware and component building and testing (*SProUT*omat), ontology lingware integration (OntoNERdIE) and XML-based information visualization tools for complex NLP representations.

Finally, we would like to stress the role of *SProUT* in conjunction with Heart of Gold. In many implemented applications, *SProUT* is utilized as one of the key components of Heart of Gold, sometimes even multiple *SProUT* grammars for different NLP purposes at the same time for a single language. We see both frameworks together as a strong couple that supports rapid development of application-oriented component integrations.

While *SProUT* named entity and information extraction grammars allow to quickly add domain-specific knowledge (domain modeling is further eased by domain ontology import via the OntoNERdIE tool), the general, domain-independent language engineering part can be handled by HPSG grammars that have, e.g. for English, German and Japanese, reached a quite mature state, and possibly assisted by a statistical part-of-speech tagger that helps to identify unknown words.

The presented frameworks layed the foundations for a new generation of XML-based application-oriented NLP integration scenarios, including advanced information extraction and question answering. Domain-adaptivity, the economic advantage of sharing resources such as shallow and deep lexica and the surplus of getting rich and robust semantic analyses with high precision constitute the potential benefit for further novel applications.

It has to be pointed out that much more is possible both on the processing strategy and on the application side than has been tried so far and is described here. Many more combinatory variants are possible even on the basis of the components and resources that have been integrated so far, but also by including additional ones. From the evaluations and results obtained so far, one can extrapolate that is will be a promising way to go.

The main scientific contribution of our thesis is the development of generic framework that serves as a complex research instrument for experimenting with novel processing strategies combining deep and shallow methods. Moreover, it also supports the development of new applications that make use of instantiations of the implemented processing strategies.

A scientific contribution is also the fact that the architecture may form a common basis and scientific platform for comparing and replicating results that have been achieved by experimenting with various combinations of hybrid processing instances.

With the advancement of knowledge technologies (knowledge representation, ontologies, ontology databases, inference engines) in the context of the Semantic Web, there will be increasing need for deep, accurate semantic analysis of natural language. Robust hybrid NLP can help to brigde the gap between natural language and semantic processing.

Appendix A

DTDs

In this appendix, we display some DTDs, mainly those of XML formats produced or used by components integrated in Heart of Gold (Chapter 9).

A.1 ACE DTD Fragment

Following is a fragment of the ACE DTD discussed in Chapter 4, with standoff pointers for character span, two-dimensional bounding boxes and timespan.

```
<!-- extract from ACE RDC DTD by John C. Henderson of MITRE
     http://www.nist.gov/speech/tests/ace/resources/ace-rdc.v2.0.1.dtd -->

<!ELEMENT name             (bblist|charspan|charseq|timespan)>

<!-- The extent is the maximal subset of the signal permitted in
     judging correctness, and the head is the minimal subset. -->

<!ELEMENT extent           (bblist|charspan|charseq|timespan)>
<!ELEMENT head             (bblist|charspan|charseq|timespan)>

<!-- A list of bounding boxes is needed to describe wrapped words in
     an image. -->

<!ELEMENT bblist           (pixelboundingbox)+>

<!-- Alternate habits for describing bounding boxes.
     Both can be supported because the tags wrap the elements.
     (x1,y1) will presumably be upper left point and
     (x2,y2) will be lower right point (suggested by English
     reading order). -->

<!ELEMENT pixelboundingbox (x1,((x2,y1,y2)|(y1,x2,y2)))>

<!-- A character SPAN (charspan) is a pair of indices that wraps
     the signal being annotated in text.   This means that the first
     index points to the imaginary gap *before* the first character
     and the second index points to the imaginary gap *after* the
```

```
     final character in the span.

     A character SEQUENCE (charseq) is a pair of indices pointing to
     the first and last character of the text being annotated.  This
     means that the first index points to the first character of the
     text being annotated (which is the same as pointing to the
     imaginary gap *before* the first character), and the second index
     points to the last character in the annotated text (the imaginary
     gap *before* the last character in the annotated text). -->

<!ELEMENT charspan         (start,end)>
<!ELEMENT charseq          (start,end)>
<!ELEMENT timespan         (start,end)>

<!ELEMENT x1               (#PCDATA)>
<!ELEMENT x2               (#PCDATA)>
<!ELEMENT y1               (#PCDATA)>
<!ELEMENT y2               (#PCDATA)>
<!ELEMENT start            (#PCDATA)>
<!ELEMENT end              (#PCDATA)>
```

A.2 TFS DTD

A minimalistic, recursive DTD for encoding typed features structures in XML as discussed in Section 5.4.1, Structure and names of element and attributes are similar to the TEI feature structure DTD (Langendoen and Simons, 1995).

```
<?xml version="1.0" ?>
<!-- minimal typed feature structure DTD -->

<!-- an FS (feature structure) node may contain features -->
<!-- atoms have an empty feature list -->
<!ELEMENT FS ( F* ) >
<!ATTLIST FS type  NMTOKEN #IMPLIED
             coref NMTOKEN #IMPLIED >

<!-- a feature has a name (attribute) and FS (feature structure) -->
<!-- as value -->
<!ELEMENT F  ( FS ) >
<!ATTLIST F  name  NMTOKEN #REQUIRED >
```

A.3 *XTDL*

This is the DTD of the internal *SProUT XTDL* grammar formalism representation, e.g. used for grammar compilation and compile time type check (page 129). The DTD is largely isomorphic to the XTDL BNF shown in Figure 7.2 on page 121.

```
<?xml version="1.0"?>
<!-- SPROUT Grammar XML DTD Version 2004
   AUTHOR : {krieger,scherf,uschaefer,witold}@dfki.de
```

```
   VERSION: 2.1
   DATE:    2003-12-19
   NOTES: CFS is restricted by the parser in the following way.
   Negation and seek are allowed only on the toplevel and only on the LHS.
   Sets are allowed only as values of features, but not on the toplevel.
   COLLECT with no type attribute is only allowed on the LHS of a rule,
   COLLECT with type set or list is only allowed on the RHS of a rule.
   Elements SEEK, TYPE, FN and F are augmented by the XTDL parser with an
   additional attribute pos containing "beginLine beginColumn endLine
   endColumn" of the name of the rule, type, function and feature in the
   XTDL source code. The same attribute is also generated for the RULE
   element. Here, the attribute indicates start and end position of the
   complete rule definition.   -->

<!ENTITY % rvalue "DISJ | CONCAT | N-TIMES | RANGE | STAR | PLUS |
                   ZERO-ONE | CFS | SEEK" >

<!ELEMENT SPROUT-GRAMMAR ( RULES ) >

<!ELEMENT RULES ( RULE | DUMMY_RULE )+ >

<!ELEMENT RULE  ( LHS, RHS?, FNCN? ) >
<!ATTLIST RULE  name NMTOKEN #REQUIRED
                pos  NMTOKENS #IMPLIED >

<!ELEMENT DUMMY_RULE  ( LHS, RHS?, FNCN? ) >
<!ATTLIST DUMMY_RULE  name NMTOKEN #REQUIRED
                    pos  NMTOKENS #IMPLIED >

<!ELEMENT LHS   ( %rvalue; )+ >

<!ELEMENT RHS   ( CFS ) >

<!ELEMENT FNCN  ( FN )+ >

<!ELEMENT FN    ( CFS )+ >
<!ATTLIST FN    name  NMTOKEN #REQUIRED
                coref NMTOKEN #IMPLIED
                pos   NMTOKENS #IMPLIED >

<!ELEMENT DISJ  ( ( %rvalue; ), ( %rvalue; )+ ) >

<!ELEMENT CONCAT ( ( %rvalue; ), ( %rvalue; )+ ) >

<!ELEMENT N-TIMES ( %rvalue; ) >
<!ATTLIST N-TIMES num NMTOKEN #REQUIRED >

<!ELEMENT RANGE ( %rvalue; ) >
<!ATTLIST RANGE start NMTOKEN #REQUIRED
                end   NMTOKEN #REQUIRED >

<!ELEMENT STAR  ( %rvalue; ) >

<!ELEMENT PLUS  ( %rvalue; ) >
```

```
<!ELEMENT ZERO-ONE ( %rvalue; ) >

<!ELEMENT SEEK  ( CFS? ) >
<!ATTLIST SEEK  name NMTOKEN #REQUIRED
                pos  NMTOKENS #IMPLIED >

<!ELEMENT SET   ( CFS | SET )+ >

<!ELEMENT CFS   ( TYPE | FS | SET | COREF | COLLECT )* >

<!ELEMENT FS    ( F )* >
<!ATTLIST FS    neg  (true) #IMPLIED >

<!ELEMENT F     ( CFS ) >
<!ATTLIST F     name NMTOKEN #REQUIRED
                pos  NMTOKENS #IMPLIED >

<!ELEMENT TYPE  ( #PCDATA ) >
<!ATTLIST TYPE  neg (true) #IMPLIED
                pos NMTOKENS #IMPLIED >

<!ELEMENT COREF ( #PCDATA ) >
<!ATTLIST COREF dct NMTOKEN #IMPLIED
                pos NMTOKENS #IMPLIED >

<!ELEMENT COLLECT ( #PCDATA ) >
<!ATTLIST COLLECT type (list|set) #IMPLIED >
```

A.4 *SProUT*put

The generic output format of the *SProUT* interpreter is the typed feature structure DTD (TFS DTD, cf. Appendix A.2), augmented with set-valued feature values, and embedded in meta-information element `MATCHINFO`, containing token and character span of the matched input sequence as well as the name of the *XTDL* rule that generated the output (Section 7.3).

```
<?xml version="1.0"?>
<!-- Sproutput DTD Version 2004
     AUTHOR : uschaefer@dfki.de
     VERSION: 2.1
     DATE:    2004-01-21   -->

<!ELEMENT SPROUTPUT ( DISJ )* >

<!ELEMENT DISJ ( MATCHINFO )+ >
<!ATTLIST DISJ id ID >

<!ELEMENT MATCHINFO ( FS ) >
<!ATTLIST MATCHINFO id     ID      #IMPLIED
                    rule   NMTOKEN #IMPLIED
```

```
                  cstart NMTOKEN #IMPLIED
                  cend   NMTOKEN #IMPLIED
                  start  NMTOKEN #IMPLIED
                  end    NMTOKEN #IMPLIED >

<!ELEMENT FS ( F )* >
<!ATTLIST FS type  NMTOKEN #REQUIRED
             coref NMTOKEN #IMPLIED >

<!ELEMENT F  ( FS | SET ) >
<!ATTLIST F  name NMTOKEN #REQUIRED >

<!ELEMENT SET ( FS | SET )* >
<!ATTLIST SET coref NMTOKEN #IMPLIED >
```

A.5 JTok

JTok comes with a native XML output functionality that is used by JTokModule (Section 9.5.1.1) for generating the module output, augmented with the `<metadata>` element.

```
<?xml version="1.0"?>
<!-- JTok DTD -->

<!ELEMENT jtok ( metadata p* ) >

<!-- paragraphs -->
<!ELEMENT p ( tu )+ >

<!-- text units, i.e., sentences etc. -->
<!ELEMENT tu ( Token )+ >
<!ATTLIST tu id ID >

<!-- token information -->
<!ELEMENT Token EMPTY >
<!ATTLIST Token string CDATA #REQUIRED
                type NMTOKEN #REQUIRED
                offset NMTOKEN #REQUIRED
                length NMTOKEN #REQUIRED >
```

A.6 TnT

As the statistical tagger TnT Brants (2000) does not produce XML output natively, the following PoS tag DTD is implemented in the TnTModule of Heart of Gold (Section 9.5.2.1).

```
<?xml version="1.0"?>
<!-- TnT DTD -->

<!ELEMENT tnt ( metadata tokens ) >
```

```
<!ELEMENT tokens ( w )* >

<!-- each token <w> may have several PoS tags <p> assigned -->
<!ELEMENT w ( p )* >
<!ATTLIST w str CDATA #REQUIRED
            cstart NMTOKEN #REQUIRED
            cend NMTOKEN #REQUIRED >

<!-- PoS tag, p attribute contains probability -->
<!ELEMENT p EMPTY >
<!ATTLIST p pos NMTOKEN #REQUIRED
            p   CDATA #REQUIRED >
```

A.7 Chunkie

The statistical chunker Chunkie (Skut and Brants, 1998) itself does not deliver XML output. The DTD below is generated by the Heart of Gold ChunkieModule (XML example on page 71).

```
<?xml version="1.0"?>
<!-- Chunkie DTD -->

<!ELEMENT chunkie ( chunks ) >

<!ELEMENT chunks ( s )* >

<!-- <s>entence -->
<!ELEMENT s ( w | chunk )* >
<!ATTLIST s id ID
            cstart NMTOKEN #REQUIRED
            cend   NMTOKEN #REQUIRED >

<!-- <w>ord/token with PoS tag -->
<!ELEMENT w ( #PCDATA ) >
<!ATTLIST w pos    NMTOKEN #REQUIRED
            cstart NMTOKEN #REQUIRED
            cend   NMTOKEN #REQUIRED >

<!-- <chunk> -->
<!ELEMENT chunk  ( w )+ >
<!ATTLIST chunk cat    NMTOKEN #REQUIRED
                cstart NMTOKEN #REQUIRED
                cend   NMTOKEN #REQUIRED >
```

A.8 RMRS

This is a snapshot of the RMRS DTD published by Ann Copestake taken from `http://lingo.stanford.edu:8000/rmrs.dtd` (as of 2004-07-21). The ad-

mitted values list of the `cogn-st` attribute of the `var` element is shortened here. RMRS is described in Section 9.4.

```
<?xml version="1.0"?>
<!-- RMRS DTD -->

<!ELEMENT rmrs-list (rmrs)*>

<!ELEMENT rmrs (label, (ep|rarg|ing|hcons)*)>
<!ATTLIST rmrs
          cfrom CDATA #REQUIRED
          cto   CDATA #REQUIRED >

<!ELEMENT ep ((realpred|gpred), label, var)>
<!ATTLIST ep
          cfrom CDATA #REQUIRED
          cto   CDATA #REQUIRED >

<!ELEMENT realpred EMPTY>
<!ATTLIST realpred
          lemma CDATA #REQUIRED
          pos (v|n|j|r|p|q|c|x|u) #REQUIRED
          sense CDATA #IMPLIED >
<!ELEMENT gpred (#PCDATA)>
<!ELEMENT label EMPTY>
<!ATTLIST label
          vid CDATA #REQUIRED >

<!ELEMENT var EMPTY>
<!ATTLIST var
          sort (x|e|h|u|l) #REQUIRED
          vid  CDATA #REQUIRED
          num  (sg|pl|u) #IMPLIED
          pers (1|2|3|1-or-3|u) #IMPLIED
          gender (m|f|n|m-or-f|u) #IMPLIED
          divisible (plus|minus|u) #IMPLIED
          cogn-st (type-id|uniq-id|fam|activ|...|u) #IMPLIED
          tense (past|present|future|non-past|u) #IMPLIED
          telic (plus|minus|u) #IMPLIED
          protracted (plus|minus|u) #IMPLIED
          stative (plus|minus|u) #IMPLIED
          incept (plus|minus|u) #IMPLIED
          imr (plus|minus|u) #IMPLIED
          boundedness (plus|minus|u) #IMPLIED
          refdistinct (plus|minus|u) #IMPLIED >

<!ELEMENT rarg (rargname, label, (var|constant))>
<!ELEMENT rargname (#PCDATA)>
<!ELEMENT constant (#PCDATA)>
<!ELEMENT ing (ing-a, ing-b)>
<!ELEMENT ing-a (var)>
<!ELEMENT ing-b (var)>
<!ELEMENT hcons (hi, lo)>
<!ATTLIST hcons
```

```
          hreln (qeq|lheq|outscopes) #REQUIRED >
<!ELEMENT hi (var)>
<!ELEMENT lo (label|var)>
```

A.9 PET Input Chart DTD

The PET input chart XML format is explained with examples in Section 9.5.5.2.

```
<?xml version="1.0"?>
<!-- PET input chart DTD  Version 2004-12-21 -->
<!-- {Bernd.Kiefer,Ulrich.Schaefer}@dfki.de  -->

<!ELEMENT pet-input-chart ( w | ne )* >

<!-- base input token constant "yes" means: do not analyze,
     i.e., if the tag contains no typeinfo, no lexical item
     will be built by the token-->
<!ELEMENT w ( surface, path*, pos*, typeinfo* ) >
<!ATTLIST w         id ID      #REQUIRED
                cstart NMTOKEN #REQUIRED
                  cend NMTOKEN #REQUIRED
                  prio CDATA   #IMPLIED
              constant (yes | no) "no" >

<!-- The surface string -->
<!ELEMENT surface ( #PCDATA ) >

<!-- numbers that encode valid paths through the input graph (optional) -->
<!ELEMENT path EMPTY >
<!ATTLIST path     num NMTOKEN #REQUIRED >

<!-- every typeinfo generates a lexical token -->
<!ELEMENT typeinfo ( stem, infl*, fsmod* ) >
<!ATTLIST typeinfo   id ID     #REQUIRED
                   prio CDATA  #IMPLIED
               baseform (yes | no) "yes" >
<!-- Baseform yes: lexical base form; no: type name -->

<!-- lexical base form or type name -->
<!ELEMENT stem ( #PCDATA ) >

<!-- type name of an inflection rule-->
<!ELEMENT infl  EMPTY >
<!ATTLIST infl    name CDATA   #REQUIRED >

<!-- put type value under path into the lexical feature structure -->
<!ELEMENT fsmod  EMPTY >
<!ATTLIST fsmod   path CDATA   #REQUIRED
                 value CDATA   #REQUIRED >

<!-- part-of-speech tags with priorities -->
<!ELEMENT pos  EMPTY >
```

```
<!ATTLIST pos      tag CDATA   #REQUIRED
                  prio CDATA   #IMPLIED >

<!-- structured input items, mostly to encode named entities -->
<!ELEMENT ne  ( ref+, pos*, typeinfo+ )  >
<!ATTLIST ne        id ID      #REQUIRED
                  prio CDATA   #IMPLIED >

<!-- reference to a base token -->
<!ELEMENT ref  EMPTY >
<!ATTLIST ref      dtr IDREF   #REQUIRED >
```

A.10 Simple PreProcessor Protocol (SPPP) DTD

The SPPP format has been implemented in LKB as preprocessor format. XML documents with this DTD can be transformed by the XSLT stylesheet from Appendix B.6 into the PET input chart format (Appendix A.9). An example is presented in Section 9.11.9.

```
<?xml version="1.0"?>
<!-- SPPP DTD -->

<!ELEMENT segment ( token )* >
<!ELEMENT token ( analysis )* >
<!ATTLIST token form CDATA #REQUIRED
                from NMTOKEN #REQUIRED
                to NMTOKEN #REQUIRED >
<!ELEMENT analysis ( rule )* >
<!ATTLIST analysis stem CDATA #REQUIRED
                   probability NMTOKEN #IMPLIED
                   pos NMTOKEN #IMPLIED >
<!ELEMENT rule EMPTY >
<!ATTLIST rule id NMTOKEN #REQUIRED
               form CDATA #REQUIRED >
```

Appendix B

XSLT Stylesheets

This appendix contains some of the discussed XSLT stylesheets from Chapter 9. The complete files (plus additional ones) are also contained in the Heart of Gold source code distribution in the `xsl` subdirectory.

B.1 Automatically Generated *SProUT* to RMRS Stylesheet

This is a fragment of the code generated by Heart of Gold for the single attribute PRODUCT-NAME of the named entity grammar for products. The XSLT code is taken from the automatically generated stylesheet `en_types-sprout2rmrs.xsl`. The complete stylesheet comprises approx. 8700 lines of code (Section 9.5.4.1).

```
<xsl:variable name='PRODUCT-NAME'
     select='FS[@type="ne-product"]/F[@name="PRODUCT-NAME"]/FS/@type'/>
<xsl:if test='($PRODUCT-NAME!="string") and ($PRODUCT-NAME!="")'>
  <xsl:element name='ep'>
    <xsl:attribute name='cfrom'>
      <xsl:value-of select='$cfrom'/>
    </xsl:attribute>
    <xsl:attribute name='cto'>
      <xsl:value-of select='$cto'/>
    </xsl:attribute>
    <xsl:attribute name='surface'>
      <xsl:value-of select='$rsurface'/>
    </xsl:attribute>

    <xsl:element name='gpred'>
      <xsl:value-of select='"product-name_rel"'/>
    </xsl:element>
    <xsl:element name='label'>
      <xsl:attribute name='vid'>
         <xsl:value-of select='$offset + 10'/>
      </xsl:attribute>
    </xsl:element>
    <xsl:element name='var'>
      <xsl:attribute name='sort'>
```

```
        <xsl:value-of select='"x"'/>
      </xsl:attribute>
      <xsl:attribute name='vid'>
        <xsl:value-of select='$offset + 10'/>
      </xsl:attribute>
    </xsl:element>
  </xsl:element>

  <xsl:element name='rarg'>
    <xsl:element name='label'>
      <xsl:attribute name='vid'>
        <xsl:value-of select='$offset + 10'/>
      </xsl:attribute>
    </xsl:element>
    <xsl:element name='rargname'>
      <xsl:value-of select='"CARG"'/>
    </xsl:element>
    <xsl:element name='constant'>
      <xsl:value-of select="translate($PRODUCT-NAME, '&quot;', '')"/>
    </xsl:element>
  </xsl:element>

  <xsl:element name='rarg'>
    <xsl:element name='label'>
      <xsl:attribute name='vid'>
        <xsl:value-of select='$offset + 10'/>
      </xsl:attribute>
    </xsl:element>
    <xsl:element name='rargname'>
      <xsl:value-of select='"ARG1"'/>
    </xsl:element>
    <xsl:element name='var'>
      <xsl:attribute name='sort'>
        <xsl:value-of select='"x"'/>
      </xsl:attribute>
      <xsl:attribute name='vid'>
        <xsl:value-of select='$offset'/>
      </xsl:attribute>
    </xsl:element>
  </xsl:element>
</xsl:if>
```

B.2 Combining Input Annotations

The `combinepixml.xsl` stylesheet is called from the PetModule (Section 9.5.5). It XML-wise concatenates multiple PET input chart documents that can be specified as a list of HoG URIs (or XML file names) in the global stylesheet parameter `urilist`.

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

```
  <!--Combine different PET XML input chart documents into one document,
      omitting metadata element and its children. Within a living HoG,
      the parameter urilist should countain a comma-separated
      list of HoG annotation names e.g. hog://session1/acol1/TnTpiXML
  -->

  <xsl:output method="xml"/>
  <xsl:strip-space elements="*"/>

  <xsl:param name="urilist">TnTpiXML.xml,SProUTpiXML.xml</xsl:param>

  <xsl:template match="/">
    <xsl:element name="pet-input-chart">
      <xsl:for-each select="@*">
        <xsl:copy-of select="."/>
      </xsl:for-each>
      <xsl:call-template name="insert-documents">
        <xsl:with-param name="urilist" select="$urilist"/>
      </xsl:call-template>
    </xsl:element>
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template name="insert-documents">
    <xsl:param name="urilist"/>
    <xsl:choose>
      <xsl:when test="contains($urilist,',')">
        <xsl:variable name="car" select="substring-before($urilist,',')"/>
        <xsl:copy-of select="document($car)/pet-input-chart/w"/>
        <xsl:copy-of select="document($car)/pet-input-chart/ne"/>
        <xsl:call-template name="insert-documents">
          <xsl:with-param name="urilist"
                          select="substring-after($urilist,',')"/>
        </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
        <xsl:copy-of select="document($urilist)/pet-input-chart/w"/>
        <xsl:copy-of select="document($urilist)/pet-input-chart/ne"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

  <xsl:template match="text()"/>

</xsl:stylesheet>
```

B.3 Removing Conflicting Items in the PET Input Chart

As the previous one, this stylesheet can be called from the PetModule (Section 9.5.5) as a preprocessing filter.

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

  <!-- Removal of PET input chart items in contained spans -->
  <!-- (to overcome bug in cheap with PET input chart -->

  <xsl:output method="xml"/>
  <xsl:strip-space elements="*"/>

  <xsl:template match="/pet-input-chart">
    <xsl:element name="pet-input-chart">
      <xsl:for-each select="@*">
        <xsl:copy-of select="."/>
      </xsl:for-each>
      <xsl:apply-templates select="w"/>
    </xsl:element>
  </xsl:template>

  <xsl:template match="w">
    <xsl:variable name="cstart" select="@cstart"/>
    <xsl:variable name="cend" select="@cend"/>
    <!-- remove Token if start or end is between start and end of a
         right NE -->
    <xsl:if test="not(following-sibling::w[(@cstart <= $cstart)
                                       and ($cstart <= @cend)] or
                      following-sibling::w[(@cstart <= $cend) and
                                           ($cend <= @cend)])">
      <xsl:copy-of select="."/>
    </xsl:if>
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="text()"/>

</xsl:stylesheet>
```

B.4 Sorting and Filtering Longest RMRS Fragments

This stylesheet takes fragmentary RMRS output from PET where the parser could not compute a full spanning analysis for a sentence. It sorts the fragments according to their length, and outputs only the *n* longest. Discussion on page 221.

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

  <!-- Extract longest RMRS fragment -->
  <!-- parameter fragments: number of fragments to return -->
  <!-- Created 2005-04-25 -->

  <xsl:output method="xml"/>
```

```
  <xsl:param name="fragments" select="5"/>

  <xsl:template match="/pet">
    <xsl:copy>
      <xsl:for-each select="@*">
        <xsl:copy-of select="."/>
      </xsl:for-each>
      <xsl:apply-templates select="metadata"/>
      <xsl:for-each select="rmrs">
        <xsl:sort select="number(ep/@cto) - number(ep/@cfrom)"
                  data-type="number" order="descending"/>
        <xsl:if test="position() <= $fragments">
          <xsl:copy-of select="."/>
        </xsl:if>
        <xsl:apply-templates/>
      </xsl:for-each>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="metadata">
    <xsl:copy-of select="."/>
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="text()"/>

</xsl:stylesheet>
```

B.5 Sorting and Merging RDF Descriptions

The following stylesheet, part of the initial OntoNERdIE offline transformation tool (Section 9.7.1.2) sorts and merges `rdf:descriptions` distributed over an RDF file, but bearing the same rdf:about (and rdf:nodeID) attributes. The input RDF file must contain unabbreviated RDF (without QName abbreviations).

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

  <!-- Date: 2005-10-07 -->
  <!-- Author: Ulrich.Schaefer@dfki.de -->
  <!-- Description: rdfsort.xsl sorts and merges distributed
       rdf:Description
       elements with same rdf:about (and rdf:nodeID) attributes
       Input DTD: RDF input file without QName abbreviations
       Output: RDF (with grouped Descriptions) -->

  <xsl:output method="xml"/>

  <xsl:key name="aboutkeys" match="rdf:Description" use="@rdf:about"/>
  <xsl:key name="nodekeys"  match="rdf:Description" use="@rdf:nodeID"/>
```

```
  <!-- root template -->
  <xsl:template match="/rdf:RDF">
    <xsl:copy>
      <!-- copy namespace and other top attributes -->
      <xsl:copy-of select="@*"/>

      <!-- walk through rdf:Description elements with rdf:about
           attributes -->
      <xsl:for-each select="rdf:Description[generate-id(.)=
                            generate-id(key('aboutkeys', @rdf:about)[1])]">
        <xsl:sort select="@rdf:about"/>
        <xsl:copy>
          <xsl:copy-of select="@*"/>
          <xsl:for-each select="key('aboutkeys', @rdf:about)">
            <xsl:copy-of select="*"/>
            </xsl:for-each>
        </xsl:copy>
      </xsl:for-each>

      <!-- walk through rdf:Description elements with rdf:nodeID
           attributes -->
      <xsl:for-each select="rdf:Description[generate-id(.)=
                            generate-id(key('nodekeys', @rdf:nodeID)[1])]">
        <xsl:sort select="@rdf:nodeID"/>
        <xsl:copy>
          <xsl:copy-of select="@*"/>
          <xsl:for-each select="key('nodekeys', @rdf:nodeID)">
            <xsl:copy-of select="*"/>
          </xsl:for-each>
        </xsl:copy>
      </xsl:for-each>
    </xsl:copy>
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="text()"/>

</xsl:stylesheet>
```

B.6 SPPP to PIC

This stylesheet transforms the LKB Simple PreProcessor Protocol (SPPP) XML format into the PET input chart, used for morphological preprocessing of the Spanish HPSG grammar (cf. Section 9.11.9).

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

  <!-- Transform LKB sppp format (http://wiki.delph-in.net/moin/LkbSppp)
  <!-- to PET XML input chart format
```

```
<!-- (http://wiki.delph-in.net/moin/PetInput, XML input)
<!-- Version: $Id: sppp2pic.xsl,v 1.3 2006/06/29 13:04:02 uschaefer $ -->

<xsl:output method="xml"/>
<xsl:strip-space elements="*"/>
<xsl:param name="metadata"><metadata/></xsl:param>

<xsl:template match="/">
  <xsl:element name="pet-input-chart">
    <xsl:copy-of select="$metadata"/>
    <xsl:for-each select="@*">
      <xsl:copy-of select="."/>
    </xsl:for-each>
    <xsl:apply-templates/>
  </xsl:element>
</xsl:template>

<xsl:template match="token">
    <xsl:apply-templates select="analysis">
      <xsl:with-param name="cstart" select="@from"/>
      <xsl:with-param name="cend" select="@to"/>
      <xsl:with-param name="tokenno" select="position()"/>
      <xsl:with-param name="surface" select="@form"/>
    </xsl:apply-templates>
</xsl:template>

<xsl:template match="analysis">
  <xsl:param name="cstart" select="-1"/>
  <xsl:param name="cend" select="-1"/>
  <xsl:param name="tokenno" select="0"/>
  <xsl:param name="surface" select="'SURFACE'"/>
  <xsl:param name="idsuffix" select="concat($tokenno,'.',position())"/>
  <w id="{concat('FLW', $idsuffix)}" cstart="{$cstart}" cend="{$cend}">
    <xsl:choose>
      <xsl:when test="count(*)=0 and not(@pos)">
        <xsl:attribute name="constant">
          <xsl:value-of select="'yes'"/>
        </xsl:attribute>
        <surface><xsl:value-of select="$surface"/></surface>
        <typeinfo id="{concat('FLT',$tokenno,'.',position())}"
                  baseform="no">
          <stem><xsl:value-of select="@stem"/></stem>
        </typeinfo>
      </xsl:when>
      <xsl:otherwise>
        <surface><xsl:value-of select="$surface"/></surface>
      </xsl:otherwise>
    </xsl:choose>
    <xsl:if test="@tag">
      <pos tag="{@tag}">
        <xsl:if test="@probability">
          <xsl:attribute name="prio">
            <xsl:value-of select="@probability"/>
          </xsl:attribute>
```

```
          </xsl:if>
        </pos>
      </xsl:if>
      <xsl:apply-templates select="rule">
        <xsl:with-param name="idsuffix" select="$idsuffix"/>
      </xsl:apply-templates>
    </w>
  </xsl:template>

  <xsl:template match="rule">
    <xsl:param name="idsuffix" select="0"/>
    <xsl:param name="stem" select="NOSTEM"/>
    <typeinfo id="{concat('FLR',$idsuffix, '.', position())}" baseform="no">
      <stem><xsl:value-of select="@form"/></stem>
      <infl name="{@id}"/>
    </typeinfo>
  </xsl:template>

  <xsl:template match="text()"/>

</xsl:stylesheet>
```

Bibliography

Serge Abiteboul. Querying semi-structured data. In *ICDT*, pages 1–18, 1997.

Steven Abney. Parsing by chunks. In Robert C. Berwick, Steven P. Abney, and Carol Tenny, editors, *Principle-Based Parsing: Computation and Psycholinguistics*, pages 257–278. Kluwer Academic Publishers, Boston, 1991.

Steven Abney. Partial parsing via finite-state cascades. In *Proceedings of the ESSLLI '96 Robust Parsing Workshop*, 1996.

Douglas Adams. *The Hitchhiker's Guide to the Galaxy*. Ballantine Books, London, UK, 1979.

Aduna B.V. *User Guide for Sesame*, 2004. http://www.openrdf.org.

Hassan Aït-Kaci and Roger Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3(3):185–215, 1986.

Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, January 1989.

Hassan Aït-Kaci. *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Types*. PhD thesis, University of Pennsylvania, Philadelphia, PA, 1984.

Jan W. Amtrup. ICE–INTARC communication environment users guide and reference manual version 1.3. Technical report, Universität Hamburg, 1995. Verbmobil Technisches Dokument 14.

Douglas E. Appelt and David Israel. *Introduction to information extraction technology*. IJCAI-99 Tutorial, Stockholm, Sweden, 1999.

Masayuki Asahara and Yuji Matsumoto. Extended models and tools for high-performance part-of-speech tagger. In *Proceedings of COLING-2000*, 2000.

Jordi Atserias, Bernardino Casas, Eli Comelles, Meritxell González, Lluís Padró, and Muntsa Padró. FreeLing 1.3: Syntactic and semantic services in an open-source NLP library. In *Proceedings of the 5th International Conference on Lan-*

guage Resources and Evaluation LREC-2006, pages 2281–2286, Genoa, Italy, 2006.

Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.

Collin F. Baker, Charles J. Fillmore, and John B. Lowe. The Berkeley FrameNet project. In *Proceedings of COLING-ACL 1998*, Montréal, Canada, 1998.

Jon Barwise and John Perry. *Situations and Attitudes*. MIT Press, Cambridge, MA, 1983.

Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL web ontology language reference, W3C recommendation, 2004. World Wide Web Consortium, http://w3c.org/TR/owl-ref/.

Markus Becker and Anette Frank. A Stochastic Topological Parser of German. In *Proceedings of COLING 2002*, pages 71–77, Taipei, Taiwan, 2002.

Markus Becker, Witold Drożdżyński, Hans-Ulrich Krieger, Jakub Piskorski, Ulrich Schäfer, and Feiyu Xu. SProUT - shallow processing with typed feature structures and unification. In *Proceedings of the International Conference on Natural Language Processing (ICON 2002)*, Mumbai, India, 2002.

Dorothee Beermann, Berthold Crysmann, Petter Haugereid, Lars Hellan, Dario Gonella, Daniela Kurz, Giampaolo Mazzini, Oliver Plaehn, and Melanie Siegel. DEEPTHOUGHT deliverable 5.10. Technical report, The DEEPTHOUGHT consortium, 2004.

Emily Bender, Dan Flickinger, Frederik Fouvry, and Melanie Siegel, editors. *Proceedings of the ESSLLI 2003 Workshop on Ideas and Strategies for Multilingual Grammar Development*, Vienna, Austria, 8 2003.

Christian Bering and Ulrich Schäfer. JTaCo & SProUTomat: Automatic testing and evaluation of multilingual language technology resources and components. In *Proceedings of the Workshop on Quality Assurance and Quality Measurement for Language and Speech Resources in Conjunction with LREC-2006*, pages 42–47, Genoa, Italy, 5 2006.

Christian Bering, Witold Drożdżyński, Gregor Erbach, Clara Guasch, Petr Homola, Sabine Lehmann, Hong Li, Hans-Ulrich Krieger, Jakub Piskorski, Ulrich Schäfer, Atsuko Shimada, Melanie Siegel, Feiyu Xu, and Dorothee Ziegler-Eisele. Corpora and evaluation tools for multilingual named entity grammar development. In *Proceedings of Multilingual Corpora Workshop at Corpus Linguistics 2003*, pages 42–52, Lancaster, 3 2003.

Christian Bering. JTaCo user guide. Technical report, Saarland University, Computational Linguistics Department, Saarbrücken, Germany, 2004.

Steven Bird and Mark Liberman. A formal framework for linguistic annotation. *Speech Communication*, 33(1,2):23–60, 2001.

Steven Bird, Peter Buneman, and Wang-Chiew Tan. Towards a query language for annotation graphs. In *Proceedings of LREC-2000*, pages 807–814, Athens, Greece, 2000.

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)*, pages 35–46, Long Beach, California, 1 2005.

Torsten Bittner. Performance evaluation for XSLT processing, 2004. Studienarbeit, University of Rostock, Germany.

Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language, W3C candidate recommendation, 2006. World Wide Web Consortium, http://w3c.org/TR/xquery/.

Christian Boitet and Mark Seligman. The ”Whiteboard” architecture: a way to integrate heterogeneous components of NLP systems. In *Proceedings of the 15th Conference on Computational Linguistics*, pages 426–430, Morristown, NJ, USA, 1994. Association for Computational Linguistics.

Francis Bond, Stephan Oepen, Melanie Siegel, Ann Copestake, and Dan Flickinger. Open source machine translation with DELPH-IN. In *Proceedings of the Open-Source Machine Translation Workshop at the 10th Machine Translation Summit*, pages 15–22, Phuket, Thailand, September 2005.

Gosse Bouma and Geert Kloosterman. Querying dependency treebanks with XML. In *Proceedings of LREC-2002*, pages 1686–1691, Las Palmas, Gran Canaria, 2002.

Ronald J. Brachman and James G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216, 1985.

Ronald J. Brachman. On the epistomological status of semantic networks. *Associative Networks*, pages 3–50, 1979.

Thorsten Brants. TnT - A Statistical Part-of-Speech Tagger. In *Proceedings of Eurospeech*, Rhodes, Greece, 2000.

Christian Braun. Flaches und robustes Parsen deutscher Satzgefüge. Master’s thesis, Saarland University, 1999.

Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0 second edition, 1998. World Wide Web Consortium, http://w3c.org/TR/REC-xml/.

Andrew Bredenkamp, Thierry Declerck, Frederik Fouvry, and Bradley Music. Efficient integrated tagging of word constructs. In *Proceedings of COLING-96*, pages 1028–1031, Copenhagen, Denmark, 1996.

Andrew Bredenkamp, Bernd Kiefer, Stefan Müller, Günter Neumann, Jakub Piskorski, Melanie Siegel, and Hans Uszkoreit. WHITEBOARD – multi-level annotation for dynamic free text processing, project proposal, 1999.

Chris Brew, David McKelvie, Richard Tobin, Henry Thompson, and Andrei Mikheev. *The XML Library LT XML. User documentation and reference guide*. LTG, University of Edinburgh, 2000.

Dan Brickley and Ramanathan V. Guha. RDF vocabulary description language 1.0: RDF Schema, W3C recommendation 10, 2004. World Wide Web Consortium, http://w3c.org/TR/rdf-schema/.

Eric Brill and Mitch Marcus. Tagging an unfamiliar text with minimal human supervision. In *Proceedings of the AAAI Symposium on Probabilistic Approaches to Natural Language*, pages 10–16. American Association for Artificial Intelligence (AAAI), 1992.

Eric Brill. A simple rule-based part-of-speech tagger. In *Proceedings of ANLP-92, 3rd Conference on Applied Natural Language Processing*, pages 152–155, Trento, Italy, 1992.

Edward J. Briscoe and John Carroll. Robust accurate statistical annotation of general text. In *Proceedings of LREC-2002*, pages 1499–1504, Las Palmas, Gran Canaria, 2002.

Jeen Broekstra and Arjohn Kampman. The SeRQL query language, 2003. http://www.openrdf.org/doc/SeRQLmanual.html.

Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *Proceedings ISWC 2001*, pages 54–68. Springer, 2002.

Paul Buitelaar, Thomas Eigner, Greg Gulrajani, Alexander Schutz, Melanie Siegel, Nicolas Weber, Philip Cimiano, Günter Ladwig, Matthias Mantel, and Honggang Zhu. Generating and visualizing a soccer knowlegde base. In *Proceedings of the EACL-2006 Demo Sessions*, Trento, Italy, 4 2006.

Felix Burkhardt, Joachim Stegmann, and Markus van Ballegooy. A voiceportal enhanced by semantic processing and affect awareness. In *Proceedings of GI Jahrestagung*, 2005.

Stephan Busemann and Hans-Ulrich Krieger. Resources and Techniques for Multilingual Information Extraction. In *Proceedings of the 4th International Conference on Language Resources and Evaluation (LREC)*, pages 1923–1926, 2004.

Stephan Busemann, Witold Drożdżyński, Hans-Ulrich Krieger, Jakub Piskorski, Ulrich Schäfer, Hans Uszkoreit, and Feiyu Xu. Integrating information extraction and automatic hyperlinking. In *Proceedings of ACL-2003, Interactive Posters/Demonstrations*, pages 117–120, Sapporo, Japan, 2003.

Joe Calder. *Thistle: Diagram Display Engines and Editors*. HCRC, University of Edinburgh, 2000.

Chris Callison-Burch and Miles Osborne. *Statistical Natural Language Processing*, chapter 7. CSLI Publications, 2003.

Ulrich Callmeier, Andreas Eisele, Ulrich Schäfer, and Melanie Siegel. The DeepThought core architecture framework. In *Proceedings of LREC-2004*, pages 1205–1208, Lisbon, Portugal, 2004.

Ulrich Callmeier. PET – A platform for experimentation with efficient HPSG processing techniques. *Natural Language Engineering*, 6(1):99 – 108, 2000.

Ulrich Callmeier. Efficient parsing with large-scale unification grammars. Master's thesis, Saarland University, Computer Science Department, 2001.

Claire Cardie and Kiri Wagstaff. Noun phrase coreference as clustering. In *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Very Large Corpora, Association for Computational Linguistics*, pages 82–89, 1999.

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*, London and NY, 2002. Continuum International.

Jean Carletta, Stefan Evert, Ulrich Heid, Jonathan Kilgour, Judy Robertson, and Holger Voormann. The NITE XML toolkit: flexible annotation for multi-modal language data. *Behavior Research Methods, Instruments, and Computers, special issue on Measuring Behavior*, pages 353–363, 2003.

Bob Carpenter. *The Logic of Typed Feature Structures*. Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1992.

Bob Carpenter. LingPipe 2.0 API documentation, 2005. http://www.alias-i.com/lingpipe/.

Steve Cassidy and Jonathan Harrington. EMU: An enhanced hierarchical speech data management system. In *Proceedings of the 6th Australian International Conference on Speech Science and Technology*, 1996.

Steve Cassidy. XQuery as an annotation query language: a use case analysis. In *Proceedings of LREC-2002*, pages 2055–2060, Las Palmas, Gran Canaria, 2002.

Don Chamberlin, Peter Fankhauser, Daniela Florescu, Massimo Marchiori, and Jonathan Robie. XML query use cases, W3C working draft, 2006. World Wide Web Consortium, http://w3c.org/TR/xquery-use-cases/.

Noam A. Chomsky. *Aspects of the Theory of Syntax*. MIT Press, 1965.

Noam A. Chomsky. *Lectures on Government and Binding*. Foris, Dordrecht, 1981.

James Clark and Steve DeRose. XML path language (XPath) version 1.0, W3C recommendation, 1999. World Wide Web Consortium, http://w3c.org/TR/xpath/.

James Clark. Associating style sheets with XML documents version 1.0, W3C recommendation, 1999. World Wide Web Consortium, http://w3c.org/TR/xml-stylesheet/.

James Clark. XSL transformations (XSLT) version 1.0, W3C recommendation, 1999. World Wide Web Consortium, http://w3c.org/TR/xslt/.

Lionel Clément and Éric Villemonte de la Clergerie. MAF: a morphosyntactic annotation framework. In *Proceedings of the 2nd Language & Technology Conference (LT'05)*, pages 90–94, Poznan, Poland, 2005.

Ann Copestake and Karen Sparck Jones. Natural language interfaces to databases. *Knowledge Engineering*, 5(4):225–249, 1990.

Ann Copestake, Dan Flickinger, Rob Malouf, Susanne Riehemann, and Ivan Sag. Translation using minimal recursion semantics. In *Proceedings of the Sixth International Conference on Theoretical and Methodological Issues in Machine Translation*, Leuven, Belgium, 1995.

Ann Copestake, Dan Flickinger, Rob Malouf, Susanne Riehemann, and Ivan Sag. Translation using Minimal Recursion Semantics. In *Proceedings of the 6th International Conference on Theoretical and Methodological Issues in Machine Translation (TMI-95)*, Leuven, Belgium, July 1995.

Ann Copestake, Dan Flickinger, Ivan A. Sag, and Carl Pollard. Minimal recursion semantics: an introduction. *Journal of Research on Language and Computation*, 3(2–3):281–332, 2005.

Ann Copestake, Simone Teufel, Peter Murray-Rust, and Andy Parker. Extracting the science from scientific publications, SciBorg project proposal, public version, 2005. http://www.cl.cam.ac.uk/˜aac10/escience/public.pdf.

Ann Copestake. *Implementing Typed Feature Structure Grammars*. CSLI publications, Stanford, CA, 2002.

Ann Copestake. Report on the design of RMRS. Technical Report D1.1b, University of Cambridge, Cambridge, UK, 2003.

Richard Crouch. Packed rewriting for mapping semantics to KR. In *Proceedings IWCS*, Tilburg, The Netherlands, 2005.

Berthold Crysmann, Anette Frank, Bernd Kiefer, Stefan Müller, Jakub Piskorski, Ulrich Schäfer, Melanie Siegel, Hans Uszkoreit, Feiyu Xu, Markus Becker, and Hans-Ulrich Krieger. An Integrated Architecture for Deep and Shallow Processing. In *Proceedings of ACL 2002*, Philadelphia, PA, 2002.

Berthold Crysmann. On the efficient implementation of German verb placement in HPSG. In *Proceedings of RANLP-2003*, Borovets, Bulgaria, 2003.

Hamish Cunningham, Kevin Humphreys, Robert Gaizauskas, and Yorick Wilks. Software infrastructure for natural language processing. In *Proceedings of the 5th Conference on Applied Natural Language Processing*, pages 237–244, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

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*, 2002.

Hamish Cunningham. *Software Architecture for Language Engineering*. PhD thesis, University of Sheffield, 2000.

Michael Daum, Kilian Foth, and Wolfgang Menzel. Constraint Based Integration of Deep and Shallow Parsing Techniques. In *Proceedings of EACL 2003*, Budapest, 2003.

Ferdinand de Saussure. *Cours de linguistique générale*. Payot, Lausanne et Paris, 1916. Translation by Roy Harris: Course in General Linguistics, 1983, London, Duckworth.

Thierry Declerck and Heinz-Dieter Maas. The integration of a part-of-speech tagger into the ALEP platform. In *Proceedings of the 3rd ALEP User Group Workshop*, Saarbrücken, Germany, 1997.

Steve DeRose, Eve Maler, and David Orchard. XML linking language (XLink), 2001. World Wide Web Consortium, http://w3c.org/TR/xlink/.

Steve DeRose, Eve Maler, and Ron Daniel Jr. XML pointer language (XPointer), 2002. World Wide Web Consortium, http://w3c.org/TR/xptr/.

Peter Dienes and Amit Dubey. Deep syntactic processing by combining shallow methods. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics, ACL-2003*, Sapporo, Japan, 2003.

Witold Drożdżyński, Hans-Ulrich Krieger, Jakub Piskorski, Ulrich Schäfer, and Feiyu Xu. Shallow processing with unification and typed feature structures – foundations and applications. *Künstliche Intelligenz*, 2004(1):17–23, 2004.

Amit Dubey and Frank Keller. Parsing German with sister-head dependencies. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, Sapporo, Japan, 2003.

Olivier Dubuisson. *ASN.1 Communication between Heterogeneous Systems*. Morgan Kaufmann, 2000.

Patrick Durusau and Matthew Brook O'Donnell. Concurrent markup for XML documents. In *Proceedings of XML Europe 2002*, 2002.

Marc Dymetman. A simple transformation for offline-parsable grammars and its termination properties. In *COLING*, pages 1226–1230, 1994.

Martin Emele. Unification with lazy non-redundant copying. In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics*, pages 323–330, 1991.

Katrin Erk and Sebastian Padó. A powerful and versatile XML format for representing role-semantic annotation. In *Proceedings of LREC-2004*, pages 799–802, Lisbon, Portugal, 2004.

Michael Fleischman, Eduard Hovy, and Abdessamad Echihabi. Offline strategies for online question answering: Answering questions before they are asked. In Erhard Hinrichs and Dan Roth, editors, *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pages 1–7, 2003.

Dan Flickinger. On building a more efficient grammar by exploiting types. In Dan Flickinger, Stephan Oepen, Hans Uszkoreit, and Jun-ichi Tsujii, editors, *Collaborative Language Engineering. A Case Study in Efficient Grammar-based Processing*, pages 1–17. CSLI Publications, 2002.

Mary Ellen Foster and Michael White. Techniques for text planning with XSLT. In *Proceedings of the 4th NLPXML Workshop*, Barcelona, Spain, 2004.

Anette Frank and Katrin Erk. Towards an LFG syntax-semantics interface for Frame Semantics annotation. In A. Gelbukh, editor, *Computational Linguistics and Intelligent Text Processing*. LNCS, Springer, 2004.

Anette Frank, Markus Becker, Berthold Crysmann, Bernd Kiefer, and Ulrich Schäfer. Integrated shallow and deep parsing: TopP meets HPSG. In *Proceedings of ACL-2003*, pages 104–111, Sapporo, Japan, 2003.

Anette Frank, Kathrin Spreyer, Witold Drożdżyński, Hans-Ulrich Krieger, and Ulrich Schäfer. Constraint-based RMRS construction from shallow grammars.

In Stefan Müller, editor, *Proceedings of the HPSG-2004 Conference, Center for Computational Linguistics, Katholieke Universiteit Leuven*, pages 393–413. CSLI Publications, Stanford, CA, 2004.

Anette Frank, Hans-Ulrich Krieger, Feiyu Xu, Hans Uszkoreit, Berthold Crysmann, Brigitte Jörg, and Ulrich Schäfer. Querying structured knowledge sources. In *Proceedings of AAAI-05. Workshop on Question Answering in Restricted Domains*, page 10, Pittsburgh, Pennsylvania, 7 2005.

Anette Frank, Hans-Ulrich Krieger, Feiyu Xu, Hans Uszkoreit, Berthold Crysmann, and Ulrich Schäfer. Question answering from structured knowledge sources. *Journal of Applied Logic, Special Issue on Questions and Answers: Theoretical and Applied Perspectives*, 4(3), 2006. doi:10.1016/j.jal.2005.12.006.

Manuel Früh, Philipp Häuser, and Michael Marks. Bewertung von XSLT-Prozessoren. Technical report, Institut für Parallele und Verteilte Systeme (IPVS), Abteilung Anwendersoftware (AS), Universität Stuttgart, Stuttgart, Germany, 2004. Fachstudie Nr. 32.

Gerald Gazdar, Ewan Klein, Geoffrey Pullum, and Ivan Sag. *Generalized Phrase Structure Grammar*. Harvard University Press, Cambridge, MA, 1985.

Kurt Godden. Lazy unification. In *Proceedings of the 28th Annual Meeting of the Association for Compulational Linguistics (ACL)*, pages 180–187, Pittsburgh, PA, 1990.

Charles F. Goldfarb. *The SGML Handbook*. Clarendon Press, Oxford, 1990.

Joris Graaumans. *Usability of XML Query Languages*. PhD thesis, Dutch Research School for Information and Knowledge Systems, Utrecht, The Netherlands, 2005. SIKS Dissertation Series No. 2005-16.

Gregory Grefenstette and Pasi Tapanainen. What is a word, what is a sentence? problems of tokenization. In *Proceedings of the 3rd International Conference on Computational Lexicography*, Budapest, Hungary, 1994.

Gregory Grefenstette. Light parsing as finite state filtering. In *Workshop on Extended finite state models of language, ECAI-1996*, Budapest, Hungary, 1996.

Ralph Grishman and Beth Sundheim. Message understanding conference - 6: A brief history. In *Proceedings of COLING-96*, pages 466–471, 1996.

Ralph Grishman. TIPSTER text architecture design document version 3.2. Technical report, DARPA, 1997. http://www-nlpir.nist.gov/related_projects/tipster/docs/arch31.doc.

Claire Grover and Alexis Lascarides. XML-based data preparation for robust deep parsing. In *Proceedings of ACL/EACL 2001*, pages 252–259, Toulouse, France, 2001.

Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. An evaluation of knowledge base systems for large OWL datasets. In *Proceedings of ISWC 2003*. Springer, 2004.

Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. A comparison of RDF query languages. In *Proceedings ISWC 2003*, pages 502–517. Springer, 2004.

Birgit Hamp and Helmut Feldweg. GermaNet - a lexical-semantic net for German. In *Proceedings of ACL workshop Automatic Information Extraction and Building of Lexical Semantic Resources for NLP Applications*, Madrid, 1997.

Marti A. Hearst. Untangling text data mining. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics*, University of Maryland, Australia, June 1999.

Ulrich Heid, Holger Voormann, Jan-Torsten Milde, Ulrike Gut, Katrin Erk, and Sebastian Padó. Querying both time-aligned and hierarchical corpora with NXT search. In *Proceedings of the 4th International Conference on Language Resources and Evaluation LREC-2004*, pages 1455–1458, Lisbon, Portugal, 2004.

Lars Hellan, Dorothee Beermann, Jon Atle Gulla, and Atle Prange. Trailfinder: A case study in extracting spatial information using deep language processing. In Ton van der Wouden, Michaela Poß, Hilke Reckman, and Crit Cremers, editors, *Computational Linguistics in the Netherlands 2004: Selected papers from the fifteenth CLIN meeting*, pages 121–131, Utrecht, Netherlands, 2004.

Gerd Herzog, Heinz Kirchmann, Stefan Merten, Alassane Ndiaye, and Peter Poller. MULTIPLATFORM Testbed: An integration platform for multimodal dialog systems. In *Proceedings of the HLT-NAACL 2003 Workshop on Software Engineering and Architecture of Language Technology Systems (SEALTS)*, pages 75–82, Edmonton, Canada, 2003.

Gerd Herzog, Heinz Kirchmann, Stefan Merten, Alassane Ndiaye, Peter Poller, and Tilman Becker. Large-scale software integration for spoken language and multimodal dialog systems. *Natural Language Engineering*, 10(3/4):283–305, 2004.

Jerry Hobbs, Douglas Appelt, John Bear, David Israel, Megumi Kameyama, Mark Stickel, and Mabry Tyson. Fastus: A cascaded finite-state transducer for extracting information from natural-language text. In E. Roche and Y. Schabes, editors, *Finite State Devices for Natural Language Processing*. MIT Press, 1997.

Tilman Höhle. Topologische Felder. Unpublished manuscript, University of Cologne, http://www.linguistik.uni-tuebingen.de/hoehle/manuskripte/Topologische_Felder.pdf, 1983.

Nancy Ide and Laurent Romary. A common framework for syntactic annotation. In *Proceedings of ACL-2001*, pages 298–305, Toulouse, 2001.

Nancy Ide and Laurent Romary. Standards for language resources. In *Proceediungs of LREC-2002*, pages 59–65, Las Palmas, Gran Canaria, 2002.

Nancy Ide, Patrice Bonhomme, and Laurent Romary. XCES: An XML-based encoding standard for linguistic corpora. In *Proceediungs of LREC-2000*, pages 825–830, Athens, Greece, 2000.

Nancy Ide, Adam Kilgarriff, and Laurent Romary. A formal model of dictionary structure and content, 2000.

Nancy Ide. Encoding linguistic corpora. In *Proceedings of the 6th Workshop on Very Large Corpora*, pages 9–17, 1998.

Nancy Ide. The XML framework and its implications for the development of natural language processing tools. In *Proceedings of the COLING Workshop on Using Toolsets and Architectures to Build NLP Systems*, Luxembourg, 2000.

Neil Ireson, Fabio Ciravegna, Mary Elaine Califf, Dayne Freitag, Nicholas Kushmerick, and Alberto Lavelli. Evaluating machine learning for information extraction. In *Proc. Int. Conf. Machine Learning*, 2005.

ISO/IEC. ISO/IEC 10179:1996. document style semantics and specification language (DSSSL), 1996. International Standard, International Organization for Standardization, International Electrotechnical Commission.

Ray Jackendoff. *X'-Syntax: A Study of Phrase Structure*. MIT Press, Cambridge, Massachusetts, 1977.

Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987.

Aravind K. Joshi and Phil Hopely. A parser from antiquity. *Natural Language Engineering*, 2(4):291–294, 1996.

Aravind K. Joshi, K. Vijay-Shanker, and David Weir. The convergence of mildly context-sensitive grammar formalisms. In P. Sells, Shieber S. M., and T. Warsow, editors, *Foundational Issues in Natural Language Processing*, pages 31–81. MIT Press, Cambridge, MA, USA, 1991.

Hans Kamp and Uwe Reyle. *From Discourse to Logic*. Kluwer, Dordrecht, 1993.

Ronald Kaplan and Joan Bresnan. Lexical-functional grammar: A formal system for grammatical representation. In Joan Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173–281. MIT Press, Cambridge, Mass, 1982.

Ronald M. Kaplan and Martin Kay. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378, 1994.

Ronald M. Kaplan and Tracy Holloway King. Low-level markup and large-scale LFG grammar processing. In *Proceedings of the LFG03 Conference*, pages 238–249. CSLI publications, 2003.

Ronald M. Kaplan, Tracy Holloway King, and John Maxwell. Adapting existing grammars: The XLE experience. In *Proceedings of the COLING-2002 Workshop on Grammar Engineering and Evaluation*, pages 29–35, 2002.

Ronald M. Kaplan, John T. Maxwell III, Tracy Holloway King, and Richard Crouch. Integrating finite-state technology with deep LFG grammars. In *Proceedings of the ESSLLI 2004 workshop on Combining Shallow and Deep Processing for NLP*, pages 11–20, Nancy, France, 2004.

Lauri Karttunen and Martin Kay. Structure sharing with binary trees. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, pages 133–136A, Chicago, Illinois, USA, 1985.

Lauri Karttunen, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328, 1996.

Lauri Karttunen. KIMMO: A general morphological processor. *Texas Linguistic Forum*, 22:163–186, 1983.

Robert T. Kasper and William C. Rounds. The logic of unification in grammar. *Linguistics and Philosophy*, 13:35–58, 1990.

Walter Kasper, Bernd Kiefer, Hans-Ulrich Krieger, Christopher J. Rupp, and Karsten L. Worm. Charting the depths of robust speech parsing. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (ACL'99), June 20–26*, pages 405–412, University of Maryland, College Park, USA, 1999.

Walter Kasper, Jörg Steffen, Jakub Piskorski, and Paul Buitelaar. Integrated language technologies for multilingual information services in the MEMPHIS project. In *Proceedings of LREC-2004*, Lisbon, Portugal, 2004.

Martin Kay. Functional grammar. In C. Chiarello et al., editor, *Proceedings of the 5th Annual Meeting of the Berkeley Linguistics Society*, pages 142–158, Berkeley, Cal, 1979.

Bernd Kiefer, Hans-Ulrich Krieger, John Carroll, and Rob Malouf. A bag of useful techniques for efficient and robust parsing. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics, ACL-99*, pages 473–480, 1999.

Bernd Kiefer, Hans-Ulrich Krieger, and Mark-Jan Nederhof. Efficient and robust parsing of word hypotheses graphs. In Wolfgang Wahlster, editor, *Verbmobil: Foundations of Speech-to-Speech Translation*, Artificial Intelligence, pages 280–295. Springer, Berlin, Germany, 2000.

Simon Kirby. *Function, Selection and Innateness – The emerge of Language Universals*. PhD thesis, University of Edinburgh, 1996.

Tina Klüwer. Semantische Auszeichnungen in sprachverarbeitenden Prozessketten-systemen. Magister artium thesis, Institut für sprachliche Informationsverarbeitung, Universität Köln, 2006.

Graham Klyne and Jeremy J. Carroll. Resource description framework (RDF): Concepts and abstract syntax, W3C recommendation, 2004. World Wide Web Consortium, http://w3c.org/TR/rdf-concepts/.

Donald E. Knuth. A characterization of parenthesis languages. *Information and Control*, 11(3):269–289, 1967.

Kiyoshi Kogure. Strategic lazy incremental copy graph unification. In *Proceedings of the 13th International Conference on Computational Linguistics, COLING-90*, pages 223–228, 1990.

Valia Kordoni and Julia Neu. Deep analysis of modern Greek. In *Proceedings of the 1st International Joint Conference on Natural Language Processing (IJCNLP-04)*, Hainan Island, China, 2004.

Kimmo Koskenniemi. Two-level model for morphological analysis. In A. Bundy, editor, *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 683–685, Karlsruhe, Germany, 1983.

Hans-Ulrich Krieger and Ulrich Schäfer. TDL – A type description language for constraint-based grammars. In *Proceedings of the 15th International Conference on Computational Linguistics, COLING-94*, pages 893–899, 1994. An enlarged version of this paper is available as DFKI Research Report RR-94-37.

Hans-Ulrich Krieger and Feiyu Xu. A type-driven method for compacting MMorph resources. In *Proceedings of RANLP 2003*, pages 220–224, 2003.

Hans-Ulrich Krieger, Witold Drożdżyński, Jakub Piskorski, Ulrich Schäfer, and Feiyu Xu. A bag of useful techniques for unification-based finite-state transducers. In *Proceedings of KONVENS-2004*, pages 105–112, Vienna, Austria, 9 2004.

Hans-Ulrich Krieger. *TDL – A Type Description Language for Constraint-Based Grammars. Foundations, Implementation, and Applications*. PhD thesis, Universität des Saarlandes, Department of Computer Science, Saarbrücken, 1995.

Hans-Ulrich Krieger. SDL – A description language for building NLP systems. In *Proceedings of the HLT-NAACL Workshop on the Software Engineering and Architecture of Language Technology Systems, SEALTS*, pages 84–91, 2003.

Taku Kudo and Yuji Matsumoto. Use of support vector learning for chunk identification. In *Proceedings of CoNLL-2000*, Lisbon, Portugal, 2000.

Anna Kupść, Malgorzata Marciniak, Agnieszka Mykowiecka, and Jakub Piskorski. Intelligent content extraction from Polish medical reports. In *International Workshop on Intelligent Media Technology for Communicative Intelligence, Warsaw, Poland*, 9 2004.

Catherine Lai and Steven Bird. Querying and updating treebanks: A critical survey and requirements analysis. In *Proceedings of the Australasian Language Technology Workshop*, pages 139–146, Sydney, Australia, 2004.

D. Terence Langendoen and Gary F. Simons. A rationale for the TEI recommendations for feature structure markup. In Nancy Ide and Jean Veronis, editors, *Computers and the Humanities 29(3)*. Kluwer Acad. Publ., The Text Encoding Initiative: Background and Context, Dordrecht, 1995. Reprint.

Kiyong Lee, Lou Burnard, Laurent Romary, Eric de la Clergerie, Ulrich Schäfer, Thierry Declerck, Syd Bauman, Harry Bunt, Lionel Clément, Tomaz Erjavec, Azim Roussanaly, and Claude Roux. Towards an international standard on feature structure representation (2). In *Proceedings of the LREC-2004 workshop on A Registry of Linguistic Data Categories within an Integrated Language Resources Repository Area*, pages 63–70, Lisbon, Portugal, 2004.

Evan Lenz. XQuery: Reinventing the wheel?, 2003. http://www.xmlportfolio.com/xquery.html.

Christian Lieske, Susan McCormick, and Gregor Thurmair. The open lexicon interchange format (OLIF) comes of age. In *Machine Translation Summit VIII*, 2001.

Kaiying Liu. Research of automatic Chinese word segmentation. In *International Workshop on Innovative Language Technology and Chinese Information Processing (ILT&CIP-2001)*, 2001.

Welf Löwe and Markus L. Noga. A lightweight XML-based middleware architecture. In *Proceedings of IASTED AI 2002*, Innsbruck, Feb 2002. ACTA Press.

Robert Malouf, John Carroll, and Ann Copestake. Efficient feature structure operations without compilation. *Natural Language Engineering*, 6(1):29–46, 2000. Special Issue on Efficient Processing with HPSG.

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, 1994.

Montserrat Marimon. Integrating shallow linguistic processing into a unification-based Spanish grammar. In *19th International Conference on Computational Linguistics (COLING-2002)*, Taipei, Taiwan, 2002.

Montserrat Marimon. *On Distributing the analysis process of a broad-coverage unification-based grammar of Spanish*. PhD thesis, Institut de ciències de l'educació, Universitat Politècnica de Catalunya, Barcelona, Spain, 2002.

Jonathan Marsh and David Orchard. XML inclusions (XInclude) version 1.0, 2001. World Wide Web Consortium, http://w3c.org/TR/xinclude/.

Bruce Martin. Distributed Xbean applications. In *Proceedings of the 2nd International Symposium on Distributed Objects and Applications*. IEEE Computer Society Press, September 2000.

Neil Mayo, Jonathan Kilgour, and Jean Carletta. Towards an alternative implementation of NXT's query language via XQuery. In *Proceedings of the EACL-2006 Workshop on Multi-dimensional Markup in Natural Language Processing*, pages 27–34, Trento, Italy, 4 2006.

David McKelvie, Chris Brew, and Henry Thompson. Using SGML as a basis for data-intensive natural language processing. *Computers and the Humanities*, 31(5), 1998.

Robert McNaughton. Parenthesis grammars. *J. ACM*, 14(3):490–500, 1967.

Andreas Mengel and Wolfgang Lezius. An XML-based representation format for syntactically annotated corpora. In *Proceediungs of LREC-2000*, pages 121–126, Athens, Greece, 2000.

George A. Miller, Richard Beckwith, Christiane Fellbaum, Derek Gross, and Katherine J. Miller. Five papers on WordNet. Technical report, Cognitive Science Laboratory, Princeton University, 1993.

Stefan Müller and Walter Kasper. HPSG analysis of German. In Wolfgang Wahlster, editor, *Verbmobil: Foundations of Speech-to-Speech Translation*, Artificial Intelligence, pages 238–253. Springer, Berlin, Germany, 2000.

Stefan Müller. *Deutsche Syntax deklarativ. Head-Driven Phrase Structure Grammar für das Deutsche*. Max Niemeyer Verlag, Tübingen, 1999.

Christoph Müller. A flexible stand-off data model with query language for multi-level annotation. In *Proceedings of the ACL-2005 Interactive Poster and Demonstration Sessions*, pages 109–112, Ann Arbor, 2005.

Stefan Müller. *Head-Driven Phrase Structure Grammar – Eine Einführung*. Stauffenburg Verlag, Tübingen, 2007. to appear.

Bernhard Nebel and Gert Smolka. Representation and reasoning with attributive descriptions. *Sorts and Types in Artificial Intelligence*, 418:112–139, 1990.

Julia Neu. Deep and shallow processing of modern Greek in a multilingual context. Master's thesis, Saarland University, Computational Linguistics Department, Saarbrücken, Germany, 2004.

Günter Neumann and Jakub Piskorski. A shallow text processing core engine. *Journal of Computational Intelligence*, 18(3), 2002.

Günter Neumann and Bogdan Sacaleanu. Experiments on Robust NL Question Interpretation and Multi-layered Document Annotation for a Cross-Language Question/Answering System. In *Proceedings of the Working Notes for the CLEF-2004 Workshop*, Bath, UK, 2004.

Günter Neumann and Ulrich Schäfer. WHITEBOARD - eine XML-basierte Architektur für die Analyse natürlichsprachlicher Texte. In Stefan Jänichen, editor, *Proceedings of Online 2002, 25th European Congress Fair for Technical Communication Düsseldorf*, volume C, pages 635.01–635.12, Düsseldorf, Deutschland, 2002. Online GmbH Kongresse und Messen für technische Kommunikation.

Günter Neumann and Feiyu Xu. Mining answers in German web pages. In *Proceedings of IEEE/WIC WI-2003*, Halifax, Canada, 10 2003.

Günter Neumann, Rolf Backofen, Judith Baur, Markus Becker, and Christian Braun. An information extraction core system for real world German text processing. In *5th International Conference of Applied Natural Language*, pages 208–215, 1997.

Günter Neumann. *A Uniform Computational Model for Natural Language Parsing and Generation*. PhD thesis, Universität des Saarlandes, Saarbrücken, 1994.

Eric Nichols, Francis Bond, and Daniel Flickinger. Robust ontology acquisition from machine-readable dictionaries. In *Proceedings of the International Joint Conference on Artificial Intelligence IJCAI-2005*, pages 1111–1116, Edinburgh, 2005.

Daniel Oberle, Anupriya Ankolekar, Pascal Hitzler, Philipp Cimiano, Michael Sintek, Malte Kiesel, Babak Mougouie, S. Vembu, S. Baumann, Massimo Romanelli, Paul Buitelaar, R. Engel, Daniel Sonntag, Norbert Reithinger, Berenike Loos, Rainer Porzel, H.-P. Zorn, V. Micelli, C. Schmidt, Moritz Weiten, F. Burkhardt, and J. Zhou. DOLCE ergo SUMO: On foundational and domain models in SWIntO (SmartWeb integrated ontology)). Technical report, AIFB, University of Karlsruhe, 7 2006.

Stephan Oepen and Ulrich Callmeier. Measure for measure: Parser cross-fertilization. Towards increased component comparability and exchange. In *Proceedings of the 6th International Workshop on Parsing Technology (IWPT '00), February 23–25*, pages 183–194, Trento, Italy, 2000.

Stephan Oepen, Ezra Callahan, Dan Flickinger, Christopher D. Manning, and Kristina Toutanova. LinGO Redwoods: A rich and dynamic treebank for HPSG. In *Beyond PARSEVAL Workshop at the 3rd International Conference on Language Resources and Evaluation, LREC-2002*, Las Palmas, Spain, 2002.

Stephan Oepen, Dan Flickinger, Jun-ichi Tsujii, and Hans Uszkoreit, editors. *Collaborative Language Engineering. A Case Study in Efficient Grammar-based Processing*. CSLI Publications, Stanford, CA, 2002.

Stephan Oepen. [incr tsdb()] – competence and performance laboratory. User manual. Technical report, Computational Linguistics, Saarland University, Saarbrücken, Germany, 2001.

National Institute of Standards and Technology. A universal transcription format (UTF) annotation specification for evaluation of spoken language technology corpora, 1998. http://www.nist.gov/speech/tests/bnr/bnews_99/utf-1.0-v2.ps.

Garance Paris. Interaction between tag set design and multilingual information extraction. Bachelor's thesis, Computational Linguistics Department, Saarland University, 2002.

Jon Patrick. Text mining for finanical scams on the Internet. In S.J. Simoff and G.J. Williams, editors, *Proceedings of the 3rd Australasian Data Mining Conference*, pages 33–38, 2004.

Fernando C.N. Pereira and Stuart M. Shieber. The semantics of grammar formalisms seen as computer languages. In *Proceedings of the 10th International Conference on Computational Linguistics*, pages 123–129, 1984.

Fernando C.N. Pereira and David H.D. Warren. Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

Fernando C.N. Pereira and David H.D. Warren. Parsing as deduction. In *Proceedings of the 21th Annual Meeting of the Association for Computational Linguistics*, pages 137–144, Cambridge, MA, 6 1983. MIT.

Georgios Petasis, Vangelis Karkaletsis, Georgios Paliouras, Ion Androutsopoulos, and Constantine D. Spyropoulos. Ellogon: A New Text Engineering Platform. In *Proceedings of LREC-2002*, pages 72–78, Canary island, Spain, 2002.

Dominique Petitpierre and Graham Russell. MMORPH – the multext morphology program. Technical report, ISSCO, University of Geneva, 1995.

Jakub Piskorski and Günter Neumann. An intelligent text extraction and navigation system. In *Proceedings of the 6th International Conference on Computer-Assisted Information Retrieval RIAO-2000*, Paris, France, 2000.

Jakub Piskorski, Petr Homola, Małgorzata Marciniak, Agnieszka Mykowiecka, Adam Przepiórkowski, and Marcin Woliński. Information extraction for Polish using the SProUT platform. In *Proceedings of Intelligent Information Systems*, Zakopane, Poland, 2004.

Jakub Piskorski. Named-entity recognition for Polish with SProUT. In Leonard Bolc, Zbigniew Michalewicz, and Toyoaki Nishida, editors, *Lecture Notes in Computer Science Vol 3490 / 2005: Intelligent Media Technology for Communicative Intelligence: 2nd International Workshop, Warsaw, Poland, September 13–14, 2004. Revised Selected Papers*, pages 122–133. Springer, 10 2005.

Carl Pollard and Ivan A. Sag. *Information-Based Syntax and Semantics. Vol. I: Fundamentals*. CSLI Lecture Notes, Number 13. Center for the Study of Language and Information, Stanford, CA, 1987.

Carl Pollard and Ivan A. Sag. *Head-Driven Phrase Structure Grammar*. Studies in Contemporary Linguistics. University of Chicago Press, Chicago, 1994.

Robbert Prins and Gertjan van Noord. Unsupervised pos-tagging improves parsing accuracy and parsing efficiency. In *Proceedings of IWPT*, Beijing, 2001.

Robbert Prins and Gertjan van Noord. Reinforcing parser preferences through tagging. *Journal Traitement Automatique des Langues, Special Issue on Evolutions in Parsing,*, 44(3):121–139, 2003.

Frank Richter. *A Mathematical Formalism for Linguistic Theories with an Application in Head-Driven Phrase Structure Grammar*. PhD thesis, Universität Tübingen, 2000.

Douglas L. T. Rohde. *TGrep2 User Manual version 1.15*. MIT, Cambridge, MA, 2005. http://tedlab.mit.edu/~dr/Tgrep2/tgrep2.pdf.

William C. Rounds and Alexis Manaster-Ramer. A logical version of functional grammar. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, pages 89–96, 1987.

Ulrich Schäfer and Daniel Beck. Automatic testing and evaluation of multilingual language technology resources and components. In *Proceedings of the 5th International Conference on Language Resources and Evaluation LREC-2006*, pages 173–178, Genoa, Italy, 5 2006.

Ulrich Schäfer. Parameterized type expansion in the feature structure formalism TDL. Master's thesis, Universität des Saarlandes, Saarbrücken, 1995.

Ulrich Schäfer. WHAT: An XSLT-based infrastructure for the integration of natural language processing components. In *Proceedings of the Workshop on the Software Engineering and Architecture of LT Systems (SEALTS), HLT-NAACL03*, pages 9–16, Edmonton, Canada, 2003.

Ulrich Schäfer. *Typesetting XTDL Grammars and Typed Feature Structures with FS2LaTeX*. DFKI GmbH, Language Technology Lab, Saarbrücken, Germany, 11 2004. User manual, http://www.dfki.de/~uschaefer/fs2latex/.

Ulrich Schäfer. Using XSLT for the integration of deep and shallow natural language processing components. In *Proceedings of the ESSLLI 2004 workshop on Combining Shallow and Deep Processing for NLP*, pages 31–40, Nancy, France, 2004.

Ulrich Schäfer. *Heart of Gold – an XML-based middleware for the integration of deep and shallow natural language processing components, User and Developer Documentation*. DFKI Language Technology Lab, Saarbrücken, Germany, 2005. http://heartofgold.dfki.de/doc/heartofgolddoc.pdf.

Ulrich Schäfer. Middleware for creating and combining multi-dimensional NLP markup. In *Proceedings of the EACL-2006 Workshop on Multi-dimensional Markup in Natural Language Processing*, pages 81–84, Trento, Italy, 4 2006.

Ulrich Schäfer. OntoNERdIE – mapping and linking ontologies to named entity recognition and information extraction resources. In *Proceedings of the 5th International Conference on Language Resources and Evaluation LREC-2006*, pages 1756–1761, Genoa, Italy, 5 2006.

Helmut Schmid. Probabilistic part-of-speech tagging using decision trees. In *Proceedings of International Conference on New Methods in Language Processing*, Manchester, UK, 1994.

Helmut Schmid. *LoPar: Design and Implementation*. IMS, University of Stuttgart, Stuttgart, 2000. Arbeitspapiere des SFB 340, Nr. 149.

Marc Schröder and Stefan Breuer. XML representation languages as a way of interconnecting TTS modules. In *Proceedings of ICSLP 04*, Jeju, Korea, 2004.

Stuart M. Shieber, Hans Uszkoreit, Fernando Pereira, Jane Robinson, and Mabry Tyson. The formalism and implementation of PATR-II. In Barbara J. Grosz and Mark E. Stickel, editors, *Research on Interactive Acquisition and Use of Knowledge*, pages 39–79. AI Center, SRI International, Menlo Park, CA, November 1983.

Stuart M. Shieber, Gertjan van Noord, Fernando C.N. Pereira, and R.C. Moore. Semantic-head-driven Generation. *Computational Linguistics*, 16(1):30–42, 1990.

Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar*. CSLI Lecture Notes, Number 4. Center for the Study of Language and Information, Stanford, CA, 1986.

Stuart M. Shieber. A Uniform Architecture for Parsing and Generation. In *Proceedings of the 12th International Conference on Computational Linguistics*, pages 614–619, 1988.

Stuart M. Shieber. *Constraint-Based Grammar Formalisms*. MIT Press, Cambridge, MA, 1992.

Melanie Siegel and Emily M. Bender. Efficient deep processing of Japanese. In *Proceedings of the 3rd Workshop on Asian Language Resources and International Standardization. COLING 2002 Post-Conference Workshop, August 31*, Taipei, Taiwan, 2002.

Melanie Siegel, Feiyu Xu, and Günter Neumann. Customizing Germanet for the Use in Deep Linguistic Processing. In *Proceedings of the NAACL Workshop Wordnet and Other Lexical Resources*, Pittsburgh, USA, 2001.

Neil K. Simpkins. An open architecture for language engineering. In *Proceedings of the 1st Language Engineering Convention*, Paris, France, 1994.

Wojciech Skut and Thorsten Brants. Chunk tagger: statistical recognition of noun phrases. In *ESSLLI-1998 Workshop on Automated Acquisition of Syntax and Parsing*, Saarbrücken, Germany, 1998.

Wojciech Skut, Brigitte Krenn, Thorsten Brants, and Hans Uszkoreit. An annotation scheme for free word order languages. In Paul Jacobs, editor, *Proceedings of the 5th Conference on Applied Natural Language Processing ANLP-97*, Washington, DC, 1997. Morgan Kaufmann Publishers.

Wojciech Skut, Thorsten Brants, and Hans Uszkoreit. A linguistically interpreted corpus of german newspaper text. In *Proceedings of the ESSLLI Workshop on Recent Advances in Corpus Annotation*, Saarbrücken, Germany, 1998.

Gert Smolka. Feature constraint logic for unification grammars. IWBS Report 93, IWBS, IBM Germany, Stuttgart, November 1989. Also in Journal of Logic Programming, 12:51–87, 1992.

C. Michael Sperberg-McQueen and Lou Burnard, editors. *Guidelines for Electronic Text Encoding and Interchange*. Text Encoding Initiative, Chicago and Oxford, 1994.

Claire Louise Taylor. XSLT as a linguistic query language, 2003. Honours thesis, Department of Computer Science and Software Engineering, University of Melbourne, Australia.

Elke Teich, Silvia Hansen, and Peter Fankhauser. Representing and querying multi-layer annotated corpora. In *Proceedings of the IRCS Workshop on Linguistic Databases*, pages 228–237, Philadelphia, 2001.

Henry S. Thompson and David McKelvie. Hyperlink semantics for standoff markup of read-only documents. In *Proceedings of SGML-EU-1997*, 1997.

Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures, Second Edition, 2004. World Wide Web Consortium, http://w3c.org/TR/xmlschema-1/.

Erik Tjong Kim Sang and Sabine Buchholz. Introduction to the CoNLL-2000 shared task: Chunking. In *Proceedings of CoNLL-2000*, Lisbon, Portugal, 2000.

Hideto Tomabechi. Quasi-destructive graph unification. In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics*, pages 315–322, 1991.

Hideto Tomabechi. Quasi-destructive graph unification with structure-sharing. In *Proceedings of the 14th COLING*, pages 440–446, 1992.

Hans Uszkoreit, Rolf Backofen, Stephan Busemann, Abdel Kader Diagne, Elizabeth Hinkelman, Walter Kasper, Bernd Kiefer, Hans-Ulrich Krieger, Klaus Netter, Günter Neumann, Stephan Oepen, and Stephen P. Spackman. DISCO – an HPSG-based NLP system and its application for appointment scheduling. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING'94), August 5–9*, volume 1, pages 436–440, Kyoto, Japan, 1994.

Hans Uszkoreit, Brigitte Jörg, and Gregor Erbach. An ontology-based knowledge portal for language technology. In *Proceedings of ENABLER/ELSNET Workshop*, Paris, 2003.

Hans Uszkoreit, Ulrich Callmeier, Andreas Eisele, Ulrich Schäfer, Melanie Siegel, and Jakob Uszkoreit. Hybrid robust deep and shallow semantic processing for creativity support in document production. In *Proceedings of KONVENS-2004*, pages 209–216, Vienna, Austria, 9 2004.

Hans Uszkoreit. New Chances for Deep Linguistic Processing. In *Proceedings of COLING 2002*, pages xiv–xxvii, Taipei, Taiwan, 2002.

Marcel P. van Lohuizen. Memory-efficient and thread-safe quasi-destructive graph unification. In *Proceedings of ACL-2000*, pages 352–359, 2000.

Gertjan van Noord and Dale Gerdemann. Finite state transducers with predicates and identity. *Grammars*, 4(3):263–286, 2001.

Gertjan van Noord. *Reversibility in Natural Language Processing*. PhD thesis, University of Utrecht, 1993.

Sreeni Viswanadha and Sriram Sankar. Java compiler compiler (JavaCC) – The Java parser generator, 2002. Sun Microsystems, http://javacc.dev.java.net.

Wolfgang Wahlster, editor. *Verbmobil: Foundations of Speech-to-Speech Translation*. Artificial Intelligence. Springer, Berlin, Germany, 2000.

Wolfgang Wahlster, editor. *SmartKom: Foundations of Multimodal Dialogue Systems*. Springer, Berlin, 2006.

Ben Waldron, Ann Copestake, Ulrich Schäfer, and Bernd Kiefer. Preprocessing and tokenisation standards in DELPH-IN tools. In *Proceedings of the 5th International Conference on Language Resources and Evaluation LREC-2006*, pages 2263–2268, Genoa, Italy, 5 2006.

Oliver Wauschkuhn. Ein Werkzeug zur partiellen syntaktischen Analyse deutscher Textkorpora. In D. Gibbon, editor, *Proceedings of the 3rd KONVENS Conference*, pages 356–368, Berlin, 1996. Mouton de Gruyter.

Graham Wilcock. Pipelines, templates and transformations: XML for natural language generation. In *Proceedings of the 1st NLP and XML Workshop, NLPRS-2001*, pages 1–8, Tokyo, Japan, 2001.

Andreas Witt. TEI-based XML-applications: Transcriptions. In *Joint Conference of the ALLC and ACH (ALLCACH98)*, Debrecen, Hungary, 1998.

Mary McGee Wood. *Categorial Grammars*. Routledge, London, 1993.

David A. Wroblewski. Nondestructive graph unification. In *Proceedings of the 6th AAAI Conference*, pages 582–589, 1987.

Feiyu Xu and Hans-Ulrich Krieger. Integrating shallow and deep NLP for information extraction. In *Proceedings of RANLP 2003*, pages 513–517, Borovets, Bulgaria, 2003.

Rémi Zajac, Mark Casper, and Nigel Sharples. An open distributed architecture for reuse and integration of heterogeneous NLP components. In *Proceedings of the 5th Conference on Applied Natural Language Processing*, pages 245–252, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

Rémi Zajac. Annotation management for large-scale NLP. In *Proceedings of ESSLLI-98 Workshop on Recent Advances in Corpus Annotation*, Saarbrücken, Germany, 1998.

Zhiping Zheng. AnswerBus question answering system. In *Human Language Technology Conference (HLT-2002)*, San Diego, CA, 2002.