# INTEGRATION OF LIGHT-WEIGHT SEMANTICS INTO A SYNTAX QUERY FORMALISM

## AN EXTENSION OF THE TIGER QUERY LANGUAGE

TORSTEN MAREK

A thesis for the degree of

*Master of Science*

in

*Language Science & Technology*

at the

*Department of Computational Linguistics*

of

*Saarland University*

March 2009

Torsten Marek: *Integration of Light-weight Semantics into a Syntax Query Formalism,* Master of Science, © March 2009

## ERKLÄRUNG

Hiermit erkläre ich an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

*Saarbrücken, den 23. März 2009*

Torsten Marek

## ABSTRACT

In the Computational Linguistics community, much work is put into the creation of large, high-quality linguistic resources, often with complex annotation. In order to make these resources accessible to non-technical audiences, formalisms for searching and filtering are needed.

The TIGER query language can, by describing partial structures, be used to search treebanks with syntactic annotation. Recently, augmented treebanks have been published, including the SALSA corpus which features frame semantic annotation on top of syntactic structure. Query languages, however, need to keep up with newly introduced annotation, allowing it to be searchable and easy to access.

We design an extension for the TIGER language which allows searching for frame structures along with syntactic annotation. To achieve this, the TIGER object model is expanded to include frame semantics, while remaining fully backwards-compatible.

Finally, these extensions have been added to our own implementation of TIGER, which includes novel indexing features not found in the original work of Lezius (2002a).

## ZUSAMMENFASSUNG

Ein großer Teil der Arbeit in der Computerlinguistik wird auf die Erstellung hochqualitativer linguistischer Resourcen mit oft komplexer Annotation verwendet. Damit diese Resourcen auch dann noch von nicht-technischen Benutzern verwendet werden können, wenn sie eine gewisse Größe überschritten haben, sind Formalismen zum Durchsuchen und Filtern von großer Wichtigkeit.

Zum Durchsuchen von Baumbanken mit syntaktischer Annotation kann die TIGER-Abfragesprache benutzt werden, die die Beschreibung partieller Strukturen ermöglicht. In den letzten Jahren wurden jedoch erweiterte Baumbanken erstellt, so zum Beispiel das SALSA-Korpus, das zusätzlich zur syntaktischen auch Annotation von semantischen Frames enthält. Abfragesprachen wie TIGER müssen mit der Erweiterung der Annotation mithalten, da diese sonst nicht durchsuchbar und somit nur schwer zugänglich ist.

Wir entwickeln eine Erweiterung für die TIGER-Abfragesprache, die zusätzlich zur Suche über Syntax auch Frame-Strukturen unterstützt. Um dies zu erreichen, erweitern wir unter Erhaltung vollständiger Rückwärtskompatibilität das TIGER-Objektmodell mit neuen Typen für die Frame-Semantik.

Darüber hinaus haben wir diese Erweiterungen im Rahmen unser eigenen TIGER-Implementation realisiert, die Methoden zur Graph-Indizierung benutzt, welche über die ursprüngliche Arbeit von Lezius (2002a) hinausgehen.

*Beautiful is better than ugly.*
*Explicit is better than implicit.*
*Simple is better than complex.*
*Complex is better than complicated.*
*Flat is better than nested.*
*Sparse is better than dense.*
*Readability counts.*
*Special cases aren't special enough to break the rules.*
*Although practicality beats purity.*
*Errors should never pass silently.*
*Unless explicitly silenced.*
*In the face of ambiguity, refuse the temptation to guess.*

— from *The Zen of Python*, by Tim Peters

## ACKNOWLEDGMENTS

First of all, I want to thank my supervisor *Caroline Sporleder* for her support during my thesis work, her comments on all stages of the language design and implementation, literature support and her extensive feedback on different drafts of this text.

*Ines Rehbein* and *Josef Ruppenhofer* both answered my questions on frame semantics and provided helpful requests, comments and feedback on the interface and syntax of the query language.

*Professor Manfred Pinkal* strongly supported my thesis idea from the very beginning and included me in his research group.

I owe special thanks to *Professor Martin Volk*, who has mentored me since I first met him in the fall of 2005 and with whom I hope to continue the work that has only just begin with this thesis.

As far as the practical part of my work is concerned, I am deeply indebted to *Johannes Stiehler*. The lessons I learned while working with him are invaluable in their contribution to my software engineering skills. The TIGER implementation in the TreeAligner would not be what it is today without him.

My friends and former fellow students *Martin Lazarov* and *Armin Schmidt* proof-read this thesis and provided helpful comments and corrections. More importantly, they both accompanied me since we started studying Computational Linguistics in Tübingen back then, for which I am very grateful.

In the time this thesis was written, I shared many wonderful hours with *Jennifer Moore*. I am thankful for her love, her unbounded cheerfulness, her perfect coffee and her work as the principal tie-breaker for all my questions regarding the English language.

# CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

# 1 INTRODUCTION

The goal of this thesis is to extend the TIGER language, a query formalism for syntactically annotated corpora, to treebanks which also contain light-weight semantic annotation in the form of frame semantics. In this chapter, we give a brief introduction to frame semantics and some available resources. With an example of a linguistic query, we motivate our work and properly define its goal and scope. The last section contains a brief outline of the thesis and the relations between the individual chapters.

## 1.1 FRAME SEMANTICS

Frame semantics (Fillmore, 1976, 1985) is a formalism that aims to represent predicates and their arguments as conceptual structures. These conceptual structures, called *frames*, represent prototypical situations. As such, they provide an abstraction layer on the concrete syntactic realization of the predicate and its arguments and also disambiguate potentially polysemous words. Rather than describing the grammatical function of a phrase, frames relate phrases to predicates based on their semantic function, the *role*. The predicates which can evoke a frame are called *lexical units* and may be verbs, nouns, adjectives or adverbs. The actual instance of a lexical unit in the text is referred to as the *target* of a frame. Examples 1.1 to 1.3 show three instances of the frame SENDING, all evoked by the lexical unit *send.v*, but with differing target realizations. The target material is underlined, with the frame name in the subscript, material of roles is enclosed square brackets and the subscript contains the role name.

(1.1) [Alice]$_{\text{SENDER}}$ $\underline{\text{sent}}_{\text{SENDING}}$ [Bob]$_{\text{RECIPIENT}}$ [a message]$_{\text{THEME}}$.

(1.2) [Alice]$_{\text{SENDER}}$ $\underline{\text{sends}}_{\text{SENDING}}$ [a message]$_{\text{THEME}}$ [to Bob]$_{\text{RECIPIENT}}$.

(1.3) [To Bob]$_{\text{RECIPIENT}}$, [the message]$_{\text{THEME}}$ had been $\underline{\text{sent}}_{\text{SENDING}}$ [by Alice]$_{\text{SENDER}}$.

While the syntactic structure varies from example to example, all sentences describe the prototypical situation of SENDING a THEME (the message) from a SENDER (Alice) to a RECIPIENT (Bob). In all cases, the frame SENDING is evoked by the lexical unit *send*, independent of its tense and diathesis. Verbal alternations, as between example 1.1 and example 1.2, do not influence the frame structure.

The shallow analysis provided by frame semantics helps to generalize across syntactic alterations that fully or nearly preserve the meaning and provide a more abstract level than a syntactic analysis can, but at the same time is more robust than a full (deep) semantic analysis. On the other hand, frame semantics ignores phenomena like negation or scope ambiguity, which results in example 1.4 having the exact same frame semantic analysis as examples 1.1 to 1.3.

(1.4) [Alice]$_{\text{SENDER}}$ did not $\underline{\text{send}}_{\text{SENDING}}$ [a message]$_{\text{THEME}}$ [to Bob]$_{\text{RECIPIENT}}$.

While a full semantic representation of a text, for instance for Natural Language Understanding, must encode such information, frame semantics is mainly a formalism for lexical semantics. In this context, it is concerned with representing valence and polysemy of predicates rather than fully capturing the meaning of a sentence.

### 1.1.1 *Frame Nets*

Each frame has its own set of locally defined roles. Their interpretation is always linked to the frame, since the specification of universal roles that apply to all frames is problematic at best. This implies that roles with the same name on different frames do not necessarily reflect the same concept and should not be treated as such (cf. Burchardt et al., 2006a, sec. 2).

In order to allow for a generalization, relations between frames and roles are defined, which creates a *frame net*. An important relation is inheritance, which defines a conceptual hierarchy of frames and their roles. This relation encodes role equivalences explicitly, without the need to rely on a limited set of universal roles.

### 1.1.2 *Multilinguality*

Because of its generality, frame semantics is especially interesting in a multilingual context. A number of frames are language-independent and can be used as a light-weight interlingua representation, other frames are tied to individual languages. Even in the case of differing frames, sets of closely related frames should be identifiable across languages (Ruppenhofer et al., 2006, ch. 6).

## 1.2 RESOURCES FOR FRAME SEMANTICS

Several projects have developed frame semantic resources for different languages. The creation of a frame net, which contains the definitions of all frames for a given language, along with their roles, lexical units and frame-to-frame relations is usually accompanied by the annotation of one or more corpora with instances of frames.

### 1.2.1 *Berkeley FrameNet*

The Berkeley FrameNet project (Baker et al., 1998) at the International Computer Science Institute at UC Berkeley has created a frame net for English. In the latest release 1.3, it contains 795 frame descriptions, which define the meaning of each frame, its roles and lexical units.

These descriptions also contain annotated example sentences, using a plain-text format that is similar to the format used in examples 1.1 to 1.4. Additionally, a list of sentences is given for each lexical unit, with more formal annotation using an XML format. The examples are taken from the British National Corpus, but their usage as a real corpus resource is hampered by the fact that only some sentences are taken, which are scattered all over the corpus. They form a collection of sentences, but not a coherent collection of texts.

In newer versions, the FrameNet distribution also contains some texts from the PropBank (Palmer et al., 2005) and the AQUAINT program with exhaustive frame semantic annotation. Roles and targets are annotated on the raw surface text, the syntactic structure outside of these parts of the sentence is not known, since the corpus does not contain a full phrase structure annotation.

### 1.2.2 *TIGER & SALSA*

Based on the Berkeley FrameNet, the SALSA project at Saarland University started annotating the TIGER corpus (Brants et al., 2002), a corpus of 1.5 million words of German newspaper text, with frame semantics. A first version of this augmented corpus was eventually released as the SALSA corpus in 2007 (Erk et al., 2003; Burchardt et al., 2006a).

In contrast to the corpora included in FrameNet, the TIGER corpus has phrase structure annotation and roles and targets are not annotated on the surface text, but on the syntactic structure. This results in several interacting structural layers, also known as multi-level annotation. This is a new feature not found in earlier treebanks, which only contain one structural layer and probably several flat layers on the surface text.

To support the annotators, a new annotation tool has been developed (SALTO, Burchardt et al., 2006b). Roles and targets of frame instances can be connected directly to the syntactic material in the phrase structure tree of the sentence, which is presented to the annotators in a graphical user interface.

### 1.2.3 *Other Projects*

Other frame semantic resources exist for Spanish (Subirats and Sato, 2004), developed at the University of Barcelona, and Japanese (Ohara et al., 2003). Both projects work in collaboration with the Berkeley FrameNet project.

The Spanish frame net also contains a corpus of 350 million words, which is currently under construction. Apart from the frame information, it contains POS tag and phrase chunk annotations.

## 1.3 MOTIVATION

Corpora such as TIGER or SALSA are important resources for Natural Language Processing, as the annotation can be used to train machine-learning based systems. These systems usually process the whole or a contiguous part of the corpus, since their usefulness results from a broad coverage of existing phenomena. In contrast to that, linguistic interest is often limited to a single, specific phenomenon at a time, resulting in an exploratory rather than exhaustive usage pattern of the corpus.

The TIGER corpus query language (Lezius, 2002a) is a part of the TIGER project and has been developed to support exactly this kind of linguistically motivated text exploration. The language is implemented in TIGERSearch (Lezius, 2002b), a graphical tool for browsing and querying syntactically annotated treebanks.

With the TIGER language, linguistic queries like in example 1.5 can be expressed compactly, as shown in the query in example 1.6. Lezius

(2002a) also describes an implementation, so that annotated corpora of the size of TIGER can be searched within a matter of seconds, even for complex structures.

(1.5) *Find all sentences which have a coordinated noun phrase for a subject.*

(1.6) `[cat="S"] >SB [cat="CNP"]`

A TIGER query consists of node descriptions and constraints on the syntactic relations between them. In this case, a sentence has the category (`cat`) S, and the relation constraint `>SB` expresses that the node on the right hand side, the coordinated noun phrase with the category CNP, must be the subject of the sentence.

*Expressive Inadequacy*

With the availability of a large corpus with frame semantic *and* syntactic annotation, it is natural to formulate queries like the one in example 1.7, matching the sample sentence in example 1.8.

(1.7) *Find all sentences where the role* TOPIC *in the frame* STATEMENT *is realized by a PP with the preposition "über".*

(1.8) [Hotels und Gaststätten]$_{\text{SPEAKER}}$ <u>klagen</u>$_{\text{STATEMENT}}$ [über knauserige Gäste]$_{\text{TOPIC}}$.

[Hotels and restaurants]$_{\text{SPEAKER}}$ <u>complain</u>$_{\text{STATEMENT}}$ [about stingy guests]$_{\text{TOPIC}}$.

The part of the query which describes the syntactic realization of the role can be expressed in a simple way, shown in example 1.9.

(1.9) `[cat="PP"] >AC [word="über"]`

The search for a frame and its role instances and, even more important, the connection between a role and its syntactic material is inexpressible within the existing TIGER query language. For the SALSA corpus, the corpus encoding format was extended (Erk and Padó, 2004) and the new elements and relations, described in section 4.1, are not represented in the query language. This reduces the usefulness of the SALSA corpus to linguists—while it is still possible to visualize the corpus with SALTO, the new annotation can neither be searched nor browsed in TIGERSearch.

## 1.4 THESIS GOALS

In the course of this work, we are going to extend the TIGER language to include elements for querying frame semantic annotation and especially the interface between frame semantics and syntax. As a result, it will be possible to write a query that matches the graphs informally described in example 1.7.

### 1.4.1 *Requirements*

The extended language must provide a *complete* formalization of the new annotations and allow queries for all its parts:

- Instances of frames

- Instances of roles
- Instances of lexical units/target lemmas
- Roles and targets occurring in frames
- Syntactic realizations of roles and targets

It should be possible to freely mix frame semantic elements, syntactic phrases and constraints on their relations within a single query. All new elements must be well-defined and keep the language *sound*; a new element may not introduce meaningless or ambiguous relationships or elements without the query being rejected as invalid.

### 1.4.2 *Implementation and Context*

Along with the inclusion of frame semantics into the definition of the language, we will add the extensions to our own implementation of TIGER, which was originally introduced in the TreeAligner (Volk et al., 2007; Mettler, 2007) for querying parallel treebanks. Over the last year, it has been reimplemented to cover almost all features of the TIGER query language, and also contains some experimental extensions which try to remedy some of TIGER's shortcomings (cf. section 3.3.5).

### 1.4.3 *Limitations*

The focus of our work is on those relations which are expressible within TIGER without fundamentally changing the formalism and object model. While the frame semantic annotation introduces a new structural layer, the extended TIGER formalism remains a language to describe partial structures completely contained in a single graph.

The domain of our extension only contains instances of frames and roles. The abstract frame and role descriptions and the relations defined between them are included into the query language if they are useful, but only as a means to generalize over annotated instances. It cannot be used to query the frame database itself; a solution for this use case already exists with FrameSQL (Subirats and Sato, 2004).

### 1.4.4 *Thesis Outline*

Chapter 2 contains a review of work that is related to our own. We will list several corpora with multiple layers of annotation and investigate new, post-TIGER query formalisms for searching in these resources.

#### *Design*

Chapter 3 contains an introduction to the TIGER query language as developed in Lezius (2002a), with particular focus on the duality of TIGER as a corpus description on the one hand and a query language on the other hand. We will show how the tree structure of syntactic annotation is encoded using a directed acyclic graph and explain the basic node relations, along with several examples of queries. This chapter summarizes the most important background information needed for the comprehension of our extensions

In chapter 4, we present our extension of the query language. We first describe the structure of frame semantic annotation as encoded by the TIGER/SALSA XML format (Erk and Padó, 2004). Section 4.2 contains an extensive discussion on the design of single aspects of the new extension. The discussion is guided by several important factors; ease of use, formal correctness with regard to the original language specification and exact representation of the annotation structure are carefully weighted against each other in order to find a suitable solution. For some elements, an exact representation is not possible. In these cases, we try to sketch possible solutions for the problem at hand to be implemented in upcoming extensions or new query languages.

Section 4.3 contains the final description of all new query language elements. In contrast to the discussion section, we only present the final extensions without possible alternatives or motivation for any particular design. The contents serve as a basis for the implementation as well as a documentation for query authors. Appendix B contains a condensed version of this section, to be used as a quick reference.

### *Implementation*

The technical realization of the new elements in our implementation is explained in chapter 5. We give an introduction to the architecture of the query evaluator and describe where new language elements are included. In some cases, the implementation needs to be extended to support a new feature, which we discuss, too. An example of the query evaluation process based on a query that combines syntactic structure and frame semantics is given in section 5.3. Based on this example, we explain the inner workings of the constraint checker and demonstrate some of the techniques that enable more efficient query evaluation.

The technical descriptions are on an abstract level and independent of the programming language used for the implementation. An overview of the source code distribution is given in appendix A. It contains a walk-through of the classes in the query module, more details on the formal properties of relation constraints and pointers on how to execute the included automatic test suites.

### *Summary & Outlook*

In chapter 6, we provide a summary of our contribution. We also outline some of the future work to be done, which is needed to support the parts of the annotation that were found to be unrealizable within the current framework and which will ultimately create a much more powerful annotation formalism and query language.

# 2 RELATED WORK

In this chapter, we will give an overview of annotation and query formalisms which are related to our work. We limit the overview to query languages that support more than just queries on flat annotation layers and a single hierarchical structure and examine several implementations that support conflicting hierarchies, the combination of hierarchical and time-sequential annotation or an arbitrary number of overlapping flat annotation layers.

## 2.1 MULTI-LEVEL ANNOTATION

The availability of large annotated corpora as resources for data-driven Natural Language Processing has led to advancements in many fields. Treebanks like the Penn Treebank (Marcus et al., 1994) greatly helped the development of better statistical parsers (Charniak, 1997), another example is the EuroParl corpus (Koehn, 2005) and its role as the most important training data for statistical Machine Translation systems. Since then, innovations in corpus creation have gone into two main directions:

1. Additional or new types of annotation

   - **PropBank** (Palmer et al., 2005)
     A version of the Penn Treebank with predicate-argument relations
   - **MuLi** (Baumann et al., 2004)
     Subset of the TIGER corpus with prosody, information structure and augmented syntactic annotation
   - **Potsdam Commentary Corpus** (Stede, 2004)
     Newspaper commentary with morphological annotations, discourse connectives and rhetorical structure.
   - **SALSA Corpus** (Burchardt et al., 2006a)
     TIGER corpus with frame semantics

2. Multilingual corpora

   - **SMULTRON** (Samuelsson and Volk, 2006)
     Parallel corpus with syntactic structure and phrase alignments

These corpora are usually created to be used as training material in machine-learning processes, but are also of general interest to many researchers outside the Computational Linguistics community, for example Linguistics or Liberal Arts. In connection to that, the *sustainability* of linguistic resources (Dipper et al., 2006) has received increasing amounts of attention. While corpora are usually released under semi-restrictive licenses which allow free use for research purposes, but no modification or redistribution and are encoded with easily processable markup languages like XML, usability for non-technical audiences is severely limited by the lack of proper graphical user interfaces and methods of exploration and searching for interesting phenomena.

## 2.2 MULTI-LEVEL SEARCHING

To store these corpora and make them accessible to a large audience, a number of new object models and query languages have been developed, which we will shortly describe in this section.

### 2.2.1 *NQL*

The NITE object model (Carletta et al., 2003) defines an annotation model for multi-modal corpora. In multi-modal corpora, a timeline common to all recorded phenomena, like speech, gestures or mimics, serves as a connection layer. More complex structures, like hierarchical annotation are added on top of a transcription layer, which is linked to the base timeline. The NITE object model allows for any number of input and annotation layers, which again can be the basis for an arbitrary number of possibly intersecting hierarchies. The annotation data model of NITE is an directed acyclic graph similar to TIGER (cf. chapter 3), but in contrast to it, a node can have any number of incoming edges. Other relationships are modeled using pointers, which represent directed edges between two nodes that do not allow for transitive closure and may therefore introduce cycles.

NQL (Evert and Voormann, 2003) is a query language on top of the NITE object model that can be used to search both structural and timeline-based annotations in multi-modal corpora. NQL queries are similar to queries in languages like SQL or XQuery[1]. Variables are declared, bound by quantifiers and constrained in expressions.

The TIGER query from example 1.9 is equivalent to the NQL query found in example 2.1, given a corpus with the same feature names. A notable difference is the lack of labeled dominance edges, this is approximated by the `@gf` attribute of the child node, '`^1`' is the operator for immediate dominance.

(2.1)
```
($pp phrase) ($w word):
  $pp ^1 $w && $pp@cat == "PP" && $w@gf = "AC"
```

Heid et al. (2004) have shown that it is possible to encode the SALSA corpus in the NITE object model. This makes it possible to formulate moderately complex linguistic queries like 2.2 as NQL queries, as shown in example 2.3.

(2.2) *Find words or syntactic categories which are the target of different semantic frames or which have more than one role, each role belonging to a different frame.*

(2.3)
```
($f1 frame) ($f2 frame)
(exists $phrase syntax)
(exists $target word):
  $f1 >"target" $target &&
  $f2 >"target" $target &&
  $f1 ^ $phrase &&
  $f2 ^ $phrase &&
  $f1 != $f2
```

Frames are implemented as a new type of annotation, and they are connected to their targets using pointers, which are queried by the use

1 XQuery 1.0: An XML Query Language, http://www.w3.org/TR/xquery/

of the `>"target"` operator. For the syntactic material inside frames, generic dominance is used, which is not distinguishable from the dominance between syntactic elements. The authors do not give a complete definition of frame annotation structure as represented in the NITE object model, but only present a proof of the general concept.

NXT Search, the sample implementation of NQL, is freely available from the University of Stuttgart[2], but has not seen any updates since 2003.

### 2.2.2 *MMAXQL*

MMAX2 (Müller, 2006) is an annotation tool for corpora with several layers of flat annotation. It uses *stand-off* representation, in which the actual annotation, whether flat or hierarchical, is stored at a different location than the base material (flat text or time series). It is not an object model *per se*, but simply a means to allow serialization of conflicting or overlapping hierarchies in XML.

The important difference between MMAX2 and graph- or tree-based annotation object models is that annotation layers in MMAX2 are independent of each other. Each layer is defined independently and connected only to the base data, any relation between two layers, if meaningful, can only be established through it. This format is optimized for superimposed flat annotation layers, but is problematic in case of hierarchical structures. Since the layers need to be known beforehand, graph-like annotation is limited to a fixed depth[3] and the relations between layers have to be specified in order to allow meaningful queries on syntactic structure.

The specification of elements in the base data is left to individual corpus designers. While words (lexical tokens) are a good default for text-based corpora, it is also possible to use single characters or other units. All other layers contain *markables*, which reference complete base units, i. e. the granularity of the base layer must be chosen accordingly so that no markable might ever need to reference only parts of base elements.

Simplified MMAXQL (Müller, 2005) is a query language for the MMAX annotation object model. A query is composed from query tokens for either base elements (word) or markables and relations between these query tokens. Relations are all defined with regard to the base layer. Sequential relations describe the position of two query tokens on the base layer, hierarchical relations use containment.

Example 2.4 contains an MMAX query that approximates the TIGER query from example 1.9. It searches for the word "über" contained in the grammatical function GF that starts at the same point as a phrase of category PP.

(2.4) `'über' in (/gf=AC starts /cat=PP)`

MMAX has been applied in the context of coreference annotation (Chiarcos et al., 2008) and transcription of speech in multi-modal corpora, but to our knowledge not for corpora with syntactic annotation and frame semantics. The definition of dominance by containment of markables is rather implicit and the spreading of similar categories

2 http://www.ims.uni-stuttgart.de/projekte/nite/
3 Given a sufficiently high number of layers or a flat syntax formalism, this should not pose a problem *in practice*.

over several layers when used for hierarchical structures makes it awkward to implement. Also, it is not immediately possible to constrain the number of children of nodes, since this requires an expensive search over several annotation layers.

The ability to relate arbitrary layers by their reference to a common timeline is the key strength of MMAX2 and makes it very suitable for several layers of flat or fixed-depth annotation like noun chunks or named entities, but turns out to be complex in heavily structured annotation, where the relations between elements are strictly defined.

### 2.2.3 *ANNIS*

ANNIS (Götze and Dipper, 2006) is a complete system for storage and retrieval of annotated corpora. It uses PAULA-XML (Dipper, 2005), a stand-off XML format that is meant to be an *interlingua* between all existing file formats for encoding linguistic annotation on corpora. PAULA-XML supports an arbitrary number of structural layers on the token layer. Token layers themselves are based on the original text, which is stored as plain text. This allows several different tokenizations.

In the ANNIS system, a corpus read from PAULA-XML files is stored in a relational database management system and made accessible for browsing and querying via a web interface. The ANNIS query language can be used to search through the database. Example 2.5 shows the ANNIS QL equivalent of the query in example 1.9.

(2.5)
```
word="über" & cat="PP"
#2 >[tiger:func="AC"] #1
```

As opposed to TIGER or NQL, nodes in ANNIS QL are named implicitly, based on their order of appearance in the query string. The operators for structural queries are taken over from TIGER and complemented by sequential relation operators. The ANNIS query language has been successfully employed for examining non-canonical constructions in the TIGER corpus (Chiarcos et al., 2008), which was extended with coreference annotations. Structures of interest were then explored with a combination of sequential and hierarchical relation constraints available in ANNIS QL. So far however, the query language has not been applied to corpora with frame semantic annotation.

In Dipper et al. (2007), the authors also evaluate the applicability of XQuery-based searches on PAULA-XML corpora. Its generality and the current state of XQuery implementations make it infeasible for larger data sets. Queries similar to the ones listed in Lai and Bird (2004), written in XQuery and executed on the TIGER corpus encoded in the PAULA-XML stand-off format could not be evaluated due to memory limitations. A condensed representation, which abandons the advantages of the stand-off format in exchange of speed, still took at least 1 min even for simple queries like *Find all sentences with the word "kam"*.

### 2.2.4 *DDDquery*

For the DDD corpus (Lüdeling et al., 2004), a diachronic corpus of German texts ranging from Old High German to Modern German, the query language DDDquery has been developed (Vitt, 2005). Similar to ANNIS QL and NQL, it supports both hierarchical and sequential queries, but also supports layers that are word-aligned to the original text

DDDquery takes over syntax elements from XPath[4], similar to LPath (Bird et al., 2005). The TIGER query in example 1.9 can be written as in example 2.6, where `$w` defines a variable binding.

(2.6)
```
//PP/word$w/"über" & $w/attribute::GF=="AC"
```

Internally, DDDQuery is based on LoToS (Faulstich and Leser, 2005), a system that translates predicate logic expressions to SQL queries that can be evaluated by a relational database.

While the structure of DDDQuery should allow for queries on corpora with syntactic and frame semantic annotation, no such application has been tried out yet. In contrast to the other query languages, no publicly accessible implementation exists, an independent evaluation of DDDquery for these purposes is therefore not possible.

### 2.2.5 *SPLICR*

SPLICR (Rehm et al., 2008b) is a platform for archiving and accessing a huge collection of corpora that have been created by several research groups at the universities of Tübingen, Hamburg, Potsdam and Berlin.

The object model used is comparable to the NITE object model, since it allows multiple layers of annotation with conflicting hierarchies. The query system is based on AnnoLab (Rehm et al., 2008a), which uses XQuery to evaluate linguistic queries. An approximate translation of 1.9 is given in example 2.7[5]

(2.7)
```
declare namespace leveler="urn:xmlns:sfb441:leveler";
element result {
  let $w := ds:layer('Lexical')//tok[
       . &= 'über']
  let $pp := ds:layer('Phrase')//[
       @leveler:category='PP']
  let $res := seq:containing($pp, $w)
  for $s in $res
  return element match { $s }
}
```

Similar to Dipper et al. (2007), Rehm et al. (2008a) report comparably long run times for the evaluation of simple linguistic queries. Furthermore, even very simple queries result in complex expressions, which facilitates the need for a simpler interface on top. In the case of SPLICR, a web interface is used that contains a graphical editor for linguistic queries.

Due to the fact that AnnoLab supports multi-layer corpora, it can be used to formulate queries on corpora with syntactic and frame semantic annotation, but so far no such work has been carried out.

4 XML Path Language (XPath) Version 1.0: http://www.w3.org/TR/xpath
5 Due to the scarcity of documentation, we cannot guarantee the correctness of this example.

## 2.3 CONCLUSION

Apart from Heid et al. (2004), no work has been done to create a system that fully integrates queries on syntactic and frame semantic annotation, and our work is a novel contribution, especially in the context of the TIGER query formalism.

For research purposes, we think that it is of highest importance that systems created at public universities are freely accessible to other researchers or commercial users, without any restrictions on modification or redistribution and that systems are available not only as binaries, but also in source code. This is the case for our query module.

In the case of NQL, which has a free implementation in the form of NXT Search, the problem is that the query language has not seen active work in six years and the latest release of the search tool is as old. In an unmaintained state, its usefulness to the research community is diminished since even existing bugs will not fixed.

# 3 THE TIGER CORPUS QUERY LANGUAGE

This chapter contains an overview of the TIGER corpus query language, which serves as a basis for the novel work described in chapter 4. We will shortly present the syntax and semantics of the query language, how it is connected to the object model of the TIGER corpus annotation formalism and outline its origin, scope and limitations.

## 3.1 OVERVIEW

TIGER is a description and query language for syntactically annotated corpora created by Lezius (2002a). It has been developed for the TIGER treebank (Brants et al., 2002), a corpus of approximately 50,000 sentences of manually annotated German newspaper text. The reference implementation is TIGERSearch (Lezius, 2002b), a powerful tool for browsing and querying treebanks with support for quantitative statistics on result sets. The usage is not limited to the TIGER treebank, it can be used with any corpus that is representable in TIGER-XML, an application of XML for the serialization of TIGER annotation graphs.

## 3.2 CORPUS DESCRIPTION LANGUAGE

In contrast to other tools for retrieval of syntactically annotated text, like TGrep2 (Rohde, 2005) or LPath (Bird and Lee, 2006; Bird et al., 2005), TIGER is a formalism for description of the annotation as well as a query language. Lezius (2002a) uses the example of TGrep (the predecessor of TGrep2), which operates on syntax trees in common *bracketed notation* as found in the Penn Treebank. TGrep queries are given in a special syntax which is different from the syntax for the annotation. While the queries are evaluated on the class of tree structures, the syntax of the language is not designed to describe the full trees.

The design of TIGER is based on tree description languages (Rogers and Vijay-Shanker, 1992; Blackburn et al., 1993). The syntactic annotation of a parse tree is modeled by the means of a directed graph, in which nodes and edges are strongly typed. The words of the sentences are in the leaf nodes of the graphs, which are called *terminals* and have the type T. Phrases are represented as internal nodes with the type NT for *nonterminals*. A corpus is defined as a set of graphs, and each graph by a list of all its nodes and relations between them.

### 3.2.1 *Feature Records*

The supertype for nonterminal and terminal nodes is FREC, the generic *feature record*. Each node can be seen as a shallow typed feature structure, also known as attribute-value matrix. While the features need to be defined explicitly in a concrete corpus, the TIGER formalism does

| TYPE | FEATURE | FEATURE NAME |
|---|---|---|
| T | Surface string | `word` |
| T | Lemma | `lemma` |
| T | Word category / Part of speech | `pos` |
| T | Morphological information | `morph` |
| NT | Phrasal category | `cat` |

Table 3.1: Features commonly used in TIGER-encoded corpora

not dictate which features have to be defined for which node types[1]. Usually, corpora adhere to a quasi-standard for feature names, to increase interoperability between different corpora. Some of the commonly used features are listed in table 3.1. Since there is no theoretical limit on the number of features, a corpus can have any number of flat annotation layers on the terminal nodes, which makes TIGER useful for corpora without any syntactic annotation, too.

In the linguistic literature, typed feature structures are written using a special layout. A feature structure for a T node representing the word "runs", containing lemma and POS tag information, is given in example 3.1.

(3.1) $$\begin{bmatrix} \text{WORD} & \text{runs} \\ \text{LEMMA} & \text{run} \\ \text{POS} & \text{VBZ} \end{bmatrix}_{\text{T}}$$

This layout is certainly not suited for computational uses, especially since the same formalism is used for queries, which are typed in by users. The keyboard-compatible version of the feature record from 3.1 is given in example 3.2. In the remainder of the text, we will refer to these structures as node descriptions.

(3.2) `[T & word="runs" & lemma="run" & pos="VBZ"]`

A simple node description is a conjunction of type (`T`) and feature constraints (`word="runs"`). The type constraint is effectively superfluous; since all features are defined for specific node types, the type of a node description can usually be inferred from the feature constraints.

In a graph description, nodes are referenced at more than one place. Although it is possible that two node descriptions refer to the same node when the type and feature constraints are identical, it is often needed to state several facts about exactly the same node, especially for the node relations introduced in the next section. This requires that it is possible to refer to node descriptions by a name. The definition of a node variable, which allows multiple references to the same node, is shown in example 3.3.

(3.3) `#a:[T & ...]`

The variable `#a` can be used at any place in the graph description where a node description can occur.

1 Because of this, it can be used to encode any kind of tree structure, not just phrase trees.

### 3.2.2 *Node Relations*

With node descriptions, all features of a node can be defined, but no connections between nodes exist so far. The following binary *relation constraints* are used to define the relationship between two nodes, the conjunction of all constraints forms the overall structure of the annotation graph.

#### *Dominance*

The phrase structure of the parse tree is described by means of the dominance relation between two nodes. A parent node of type NT can dominate any number of child nodes, which can be either terminal or nonterminal. Terminal nodes cannot dominate other nodes[2]. The edges representing the dominance relation may not introduce a cycle in the graph and each node may have at most one incoming edge, i. e. one parent. A valid syntax graph must also have exactly one root node, a node without a parent, from which all other nodes in the graph are reachable by traversing the directed dominance edges. The dominance relation is irreflexive, asymmetric and transitive.

Dominance edges are labeled, which are typically used to annotate the grammatical function of the child node. If such information is not required, a neutral label is used, usually represented by "`--`".

(3.4) `[cat="S"] >SBJ [cat="NP"]`

In example 3.4, the noun phrase with category NP is the subject (`SBJ`) of its parent node, a sentence, with '`>`' being the dominance operator.

#### *Linear Precedence*

Although the structure defined by the dominance relation is a proper tree (acyclic, directed, single root, at most one parent per node), we always called the structures *graphs*. In a tree, the linearization of the leaves is normally given by the linear ordering of internal nodes, which is often defined implicitly. In TIGER, the linear precedence of T nodes is encoded explicitly in the graph description. A linear ordering of NT nodes is then reduced to a graph layout problem, since it can be deduced from the terminals and does not need to be encoded explicitly. Linear precedence is asymmetric, irreflexive and transitive.

The graph description fragment in example 3.5 defines that the word "The" precedes the word "dog", using the operator '`.`' for linear precedence.

(3.5) `[T & word="The" & ... ] . [T & word="dog" & ...]`

Most importantly, explicit linear ordering of the leaf nodes can be used to model crossing dominance edges. In the NEGRA annotation scheme (Brants et al., 1997, 1999), which is used for the TIGER corpus and many other German treebanks, the main finite verb is always directly dominated by its sentence, using the label `HD` (for *head*) on the dominance edge. In sentences like the one shown in fig. 3.1, this leads to a crossing edge.

2 Annotation of the internal structure of morphologically complex words must be done using a feature with a specialized syntax.

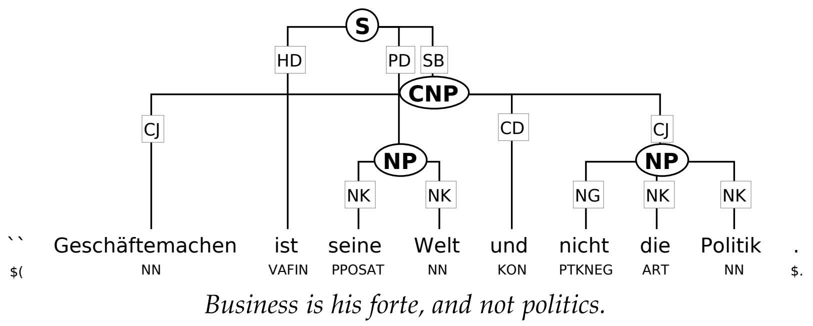


Figure 3.1: Syntax graph of a German sentence, with crossing edges

*Secondary Edges*

The secondary edge relation is used to model structure-sharing. Since it is not possible that a tree branch can have more than one parent, ellipsis[3] is modeled by connecting the omitted material with a secondary edge. Secondary edges are labeled and can be used to encode other linguistic phenomena, like anaphoric relations, too. They are directed and can exist between any combination of two nodes, regardless of their type, effectively creating a second structural layer on top of the syntactic annotation. The constraint operator is '`>~l`', where `l` is the label of the edge.

### 3.2.3 *Graph Descriptions*

A description of the graph in fig. 3.2 is given in example 3.6, which is a conjunction of node descriptions and relation constraints.

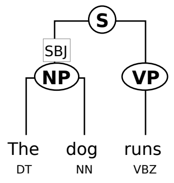


Figure 3.2: Example graph, with lemmas and neutral edge labels omitted

(3.6)
```
#n1:[cat="S"] & #n2:[cat="NP"] & #n3:[cat="VP] &
#n4:[word="The" & lemma="the" & pos="DT"] &
#n5:[word="dog" & lemma="dog" & pos="NN"] &
#n6:[word="runs" & lemma="run" & pos="VBZ"] &

#n1 >SBJ #n2 & #n1 >-- #n3 &
#n2 >-- #n4 & #n2 >-- #n5 & #n3 >-- #n6 &

#n4 . #n5 & #n5 . #n6
```

While this description encodes the graph in fig. 3.2, it does not specify that this is the complete material of the graph—all other graphs

3 For example, when a noun phrase is part of several sentences, but occurs only once in the surface.

which contain the nodes and fulfill the relation constraints from example 3.6 match the description, too. This open-endedness is important for the query language, but for a graph description, it is undesired. The arity[4] of all nonterminal nodes must be fixed and the root node defined explicitly, effectively preventing the addition of any other material. In practice, this is done with *predicates*, the third top-level element of the TIGER language. Together, examples 3.6 and 3.7 form the complete description.

(3.7)
```
root(#n1) & arity(#n1, 2) &
arity(#n2, 2) & arity(#n3, 1)
```

## 3.3 QUERY LANGUAGE

In the corpus description, special emphasis is put on the fact that only the material specified, *and nothing else*, is part of the graph. Using underspecified descriptions, which is done by simply leaving out certain parts of the graph material, the same formalism can be used as a corpus query language. The result set of a query will contain only those graphs from the corpus that contain the described nodes and fulfill the relation constraints and predicates.

In order to make the query language more user-friendly and to allow for the expression of more general structures, more elements are included in the language.

### 3.3.1 *Complex Node Descriptions*

The basis of all queries are the node descriptions. They can contain arbitrary boolean expressions for both feature values and feature constraints to allow for more compact expression of alternatives.

(3.8)
```
[pos=("NN"|"NNS") | cat="NP"]
```

The node description from example 3.8 matches terminals whose POS tag is either `NN` or `NNS` or nonterminals with category NP. It also shows that a single node description can refer to nodes of different types.

Additionally to the literal feature values shown so far, the query language also allows the usage of regular expressions[5] in feature constraints. When a value on the right-hand side of a feature constraint is surrounded by forward slashes (`/`) instead of double quotes, it is interpreted as a regular expression, which is evaluated against the feature values of nodes.

Boolean expressions in node descriptions also allow negation to match all nodes whose feature does not have a certain value or match a regular expression. For instance, the node description in example 3.9 matches all terminal nodes which are not tagged as nouns.

(3.9)
```
[pos=!("NN"|"NNS")]
```

4 The number of outgoing dominance edges.

5 Regular expressions in TIGER use a subset of the ubiquitous Perl regular expression syntax.

### 3.3.2 *Constraint Modifiers*

With the relations defined in section 3.2.2, all possible graph structures can be described, the query language is complete and reduced to three basic, well-defined notions. In practice, however, such a language would be cumbersome to use and some queries range between impractical and impossible to express. When two nodes are supposed to be in a transitive rather than direct dominance or precedence relation, a query would have to specify a disjunction of paths of all possible lengths between the two nodes, which can, in theory, be infinitely many.

To allow expression of queries like this, the semantics of relation constraint operators can be changed using *modifiers*. Transitivity, which applies to both dominance and precedence, is expressed using the asterisk symbol '`*`'.

(3.10) `#n1:[cat="S"] >* #n2:[word="house"]`

The query in example 3.10 matches all graphs containing a node `#n1` with category S that transitively dominates a node where the value of the feature *word* is "house". While `#n1` has to be an NT node, because only nonterminal nodes can have outgoing dominance edges, the type of `#n2` is entirely up to the corpus definition[6].

With the transitivity modifier, the length of the path between two nodes is unrestricted. If the path length should have a fixed or fall into an interval, it can be suffixed to the operator symbol. The operator '`.4`', for example, matches all pairs of terminals which have three intervening other terminals.

Relation operators can also be modified by omission. Both the dominance and secondary edge relation can be used without the label, which only states the existence of an edge between two nodes, but leaves the labeling unspecified.

### 3.3.3 *Derived Relations*

*Siblings*

Two nodes with the same parent are commonly called siblings. Since so far the tree hierarchy is only defined in terms of dominance, a query which requires two nodes to be siblings always needs to mention the parent node as well. If there are no constraints on the features of the parent, it can be left empty, but not dropped. Since the sibling relation is quite common, a derived relation with the constraint operator '`$`' is introduced, shown in example 3.11.

(3.11) `[pos="DT"] $ [pos="NN"]`

*Corners*

The corner relation constraint matches on left- ('`>@l`') or rightmost ('`>@r`') terminal successors of nonterminal nodes. It is a derived relation which combines both basic relations, dominance and linear precedence. It is also used to define the precedence between NT nodes.

6 The use of the feature `word` suggests the type T, however.

### 3.3.4 *Predicates*

The predicates `root` and `arity` have already been introduced for graph descriptions. The query language also includes the following predicates:

`tokenarity(#v, n)` true iff the number of terminals dominated by `#v` is $n$.

`continuous(#v)` true iff the terminal successors of `#v` are a substring of the annotated sentence, i. e. there are no holes.

`discontinuous(#v)` the negation of `continuous`

### 3.3.5 *Negation*

All constraints in modified or basic form can be negated by prefixing the operator symbol with a `'!'`. Since each node description implies an existential quantification, the result set includes all graphs that contain nodes which fulfill the constraints.

(3.12) `[cat="S"] !>* [cat="PP"]`

As an example, the query in example 3.12 matches all graphs that contains a node with category PP that is not dominated by one with category S. Negation in TIGER cannot be used to express the complete absence of a certain node, because it lacks universal quantification. This is a restriction of the underlying logical formalism.

*Partial Support for Universal Quantifiers*

Marek et al. (2008) describe a new addition to the TIGER query language which introduces a light version of universal quantification. The query in example 3.13 only matches those graphs which contain an S node that does not dominate any PP node. Universal quantification is signaled by the use of `%` as a variable prefix.

(3.13) `[cat="s"] !>* %s:[cat="pp"]`

## 3.4 FURTHER READING

In this chapter, we have only given a very short introduction of the TIGER query language. A complete overview and discussion of all its features along with the manual of TIGERSearch and a description of the corpus storage format TIGER-XML is available online[7].

The complete formal background with the description of the reference implementation is available in Lezius (2002a), which is written in German. A shorter English overview is König and Lezius (2003).

Lai and Bird (2004) compare TIGERSearch with other query formalisms for treebanks. This paper also has a discussion of the negation restrictions and its implications on linguistic queries. Solutions for some of these problems are discussed in Marek et al. (2008).

7 http://www.ims.uni-stuttgart.de/projekte/TIGER/TIGERSearch

# 4 DESIGN OF THE QUERY LANGUAGE EXTENSION

In this chapter, we will design an extension for the TIGER corpus query language which combines the existing features for querying syntactic with frame semantic annotations. We will review the extensions Erk and Padó (2004) added to TIGER-XML in order to store this type of annotation and transfer them back into the query language. This results in a change of the node type system and introduces a list of new basic and derived node relations and predicates.

## 4.1 ANNOTATION OF FRAME SEMANTICS

In the previous chapter, we described the TIGER corpus query language, which can be used to search for syntactic structures in treebanks. The SALSA project at Saarland University[1] has added frame semantic annotation to Release 1 of the TIGER corpus, which has subsequently been released as the SALSA corpus (Burchardt et al., 2006a). It is stored using the TIGER/SALSA XML format (Erk and Padó, 2004), which extends TIGER-XML by new elements for the annotation of frame semantics. In this section, we will describe the structure and features of frame semantic annotation.

The SALSA corpus uses the frame descriptions from versions 1.2 and 1.3 of the Berkeley FrameNet project (Ruppenhofer et al., 2006), but also extends it with a number of new frames which are specific to German, and therefore not part of the English frame net.

### 4.1.1 *New Annotation Elements*

The annotation of frame instances forms a new structural layer on top of the existing syntax annotation. Each instance forms a small tree on its own, with a fixed depth and fixed node types on each level. The root is the frame instance itself, which has one exactly target (or frame-evoking element, FEE) and any number of roles (or frame elements, FE). FEE and FE nodes reference syntactic material, which can consist of terminals T and nonterminals NT. They form the connection layer between the semantics and syntactic annotation. The generalized structure of a single frame annotation is shown in fig. 4.1.

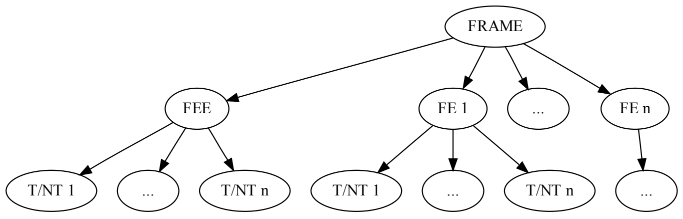


Figure 4.1: Annotation structure of a frame instance

1 http://www.coli.uni-saarland.de/projects/salsa/

The syntax part of the annotation graph is not modified by the new structural layer and still a single-rooted tree with ordered leaves, as introduced in the previous chapter. If the new frame semantic elements from a TIGER/SALSA XML corpus are removed, it becomes a normal TIGER-XML corpus.

### *Frames*

A frame node has several features, with the *name* being the most important one. Using this feature, a frame instance can be connected to its formal description in the frame database, which contains the lexical units, additional ontological information on the frames and sample sentences.

Although frames have unique numerical IDs, their names are used as the globally unique identifiers. The SALSA corpus makes use of frames from several different FrameNet versions, across which uniqueness of IDs cannot be guaranteed. The name is also used in the frame-to-frame relationships, which are part of the frame database.

A frame can also have flags, which are assigned by annotators and mark the presence certain linguistic phenomena such as metaphors.

### *Frame-Evoking Elements*

A frame is evoked by the occurrence of one of its associated lexical units in the annotated sentence. The nodes from the syntax graph that make up the lexical unit are referenced by the target node in the frame instance.

While the target is always a single concept, it can happen that it is realized by several phrases whose terminals are non-contiguous. In German, this occurs when the frame is evoked by a finite particle verb, as shown in fig. 4.2.

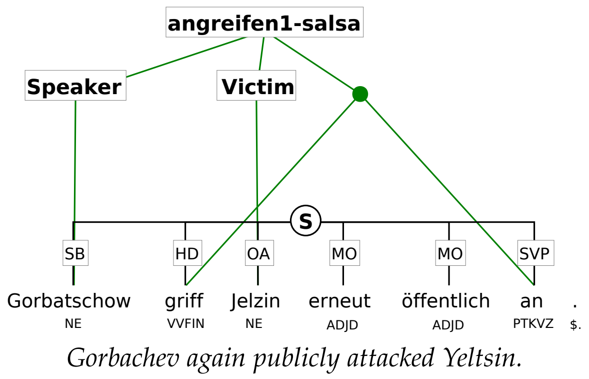


Figure 4.2: Example of a frame-evoking element with two referenced nodes

The target element has a feature *lemma*, which contains the normalization of the words which evoked the frame. In the graph from fig. 4.2, the lemma is *angreifen*. A simple single-word lemma will usually be an element of the list of lexical units in the frame database. They are not limited to verbs, but can also be nouns or adjectives.

In some cases, the frame-evoking element is a complex idiomatic phrase, also called multi-word units, as shown in example 4.1.

(4.1) Wann [der Waffenstillstand]$_{\text{ACCORD}}$ <u>in Kraft treten</u>$_{\text{TRETEN1-SALSA}}$ soll, wurde zunächst nicht mitgeteilt.

It was not said when [the ceasefire]$_{\text{ACCORD}}$ is supposed to come into effect$_{\text{TRETEN1-SALSA}}$.

Each multi-word unit has a normalized form, which is taken as the value for the lemma. The syntactic head of the multi-word unit is annotated in the feature *headlemma*. In example 4.1, the normalization of the lemma is "treten in Kraft", its head is "treten". If the lemma is simple, the head is the same as the lemma and is omitted from the annotation.

*Roles*

The main feature of a role is its name, which is also part of the frames' list of roles in the frame database and only these roles are allowed to be instantiated on a frame. In the example in fig. 4.2, the roles SPEAKER and VICTIM of the frame ANGREIFEN1-SALSA are filled.

Just like frame-evoking elements, roles reference syntactic material. If a role is not filled by any material, it is not added to the annotation, even if it is assumed to be instantiated by default. Similar to frame nodes, roles can carry flag values.

*Splitwords*

As mentioned in section 3.2.2, the TIGER corpus formalism does not have any direct means to encode the structure of morphologically complex words. Compound nouns, for example, are realized as a single terminal. In some cases, it happens that only a part of the compound is referenced by a target or role. Example 4.2 contains such a sentence. Since TIGER does not provide any means to reference substrings of features, it is not possible to represent this phenomenon in the annotation.

(4.2) [Ein 18jähriger Deutscher]$_{\text{VICTIM}}$ war dabei [durch einen [Messer]$_{\text{INSTRUMENT}}$stich]$_{\text{CAUSE}}$ getötet$_{\text{KILLING}}$ worden.

During this, [an 18-year-old German]$_{\text{VICTIM}}$ had been killed$_{\text{KILLING}}$ by [a [knife]$_{\text{INSTRUMENT}}$ stab]$_{\text{CAUSE}}$.

In order to annotate the role VICTIM in the sentence from example 4.2, the graph needs to contain additional information about the structure of the T node "Messerstich".

This is provided by non-exhaustive morphosyntactic annotation of complex terminals, called *splitwords*. While TIGER/SALSA XML puts this annotation into the semantics parts, it is strictly part of the syntax annotation. It introduces new nodes below the terminals, which can be referenced by roles or targets just like any other syntactic node.

### 4.1.2 *Underspecification*

In some cases, annotators may be unsure as to which frame should be used for a given target, or which role a certain phrase fills. In this case, *underspecification blocks* are introduced. These blocks are sets of either only frame or role nodes. It is not annotated whether a certain block is to be interpreted as a conjunction or a disjunction of its elements.

## 4.2 INTEGRATION INTO THE QUERY LANGUAGE

In this section, we reintegrate the elements from TIGER/SALSA XML frame semantics annotation into the query language. In some cases, representation is problematic or can be realized in several different ways. In all cases, we based our design decisions on the following guidelines:

1. A syntax-only query on a corpus with frame semantic annotation must yield the same results as if the corpus consisted of only syntactic annotation.
2. Any extension or change to the query language should integrate as seamlessly and with as few extensions as possible, to limit the amount of additional knowledge a user has to acquire in order to use the new elements.
3. The introduction of new types and relations may not interfere with the existing parts of the query language, i. e. an implementation should not have to radically change its behavior based on the presence of frame semantics annotation in the corpus.

The complete extension of the query language, with all new node types and their features, new node relation constraints and predicates will be summarized in section 4.3.

### *Definitions*

When we define new node relation constraints and their applicable types, we use the following syntax:

$\text{LTYPE}_1\ R\ \text{RTYPE}_1|\ldots|\text{LTYPE}_n\ R\ \text{RTYPE}_n$

A relation constraint definition is a disjunction of atomic definitions, because a constraint can be defined on different pairs of types. $R$ is the operator symbol which represents the constraint in the query language, usually without any modifiers. Example definitions for the basic relations described in chapter 3 are shown in table 4.1. The root type is FREC and in the original language, refers to the union of terminals and nonterminals.

| RELATION | DEFINITION |
|---|---|
| Dominance | NT > FREC |
| Precedence | T . T |
| Secondary edge | FREC >~ FREC |

Table 4.1: Type definitions for syntactic relations from chapter 3

For predicates, we use a similar syntax. A predicate $P$ is applicable to a node type TYPE and can have a number of additional arguments $a_i$. So far all predicate arguments are integer numbers, but new predicates could also take string arguments.

$P(\text{TYPE}, a_1, \ldots, a_n)$

### 4.2.1 *Node Types for Frame Semantics*

We need to extend the type system of nodes in annotation graphs. Since all new node types are feature records as well, FREC remains at the root of the type hierarchy. The fact that their features are fixed by the annotation format and not part of the corpus definition is only interesting for an implementation, but not important for the type system itself.

#### *Restructuring the Syntactic Types*

In TIGER, all node types are syntactic types. In order to make a fundamental distinction between syntax and semantics annotation nodes possible, we create a new type SYNTAX, derived from FREC, and use it as the new supertype for T and NT. With this change alone, the definition of dominance becomes NT $>$ SYNTAX.

#### *Integration of Splitwords*

As already said in section 4.1, splitwords are a new type of annotation that contain the possibly incomplete structural analysis of complex terminals. Parts of splitwords can be referenced from both roles and targets.

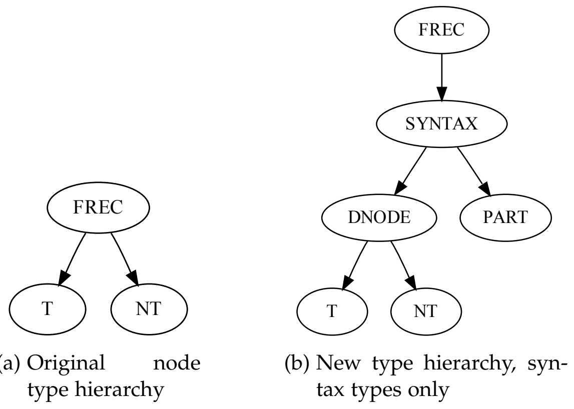
Most splitwords are compounds, which is a productive formation process in German and part of the morphology rather than the syntax. With regard to the semantic annotation, word parts behave exactly like terminals or nonterminals, and are only annotated when needed. This justifies to integrate them into the syntax part of the graph.

A straightforward solution for integration is to convert terminals with several parts into nonterminals and add their parts as new terminals below them. This approach has several problematic implications:

1. Word parts have a feature set that is completely different from terminals. If word parts are converted into terminals, the name of the feature containing the surface string needs to be guessed (*word* would be a safe choice) or specified by the user.
   Since TIGER requires all features of a node to be defined, the remaining features must be inferred, which is only possible in some cases. Morphological features of a compound word are determined by its head, which is not annotated explicitly and implementation can only try to guess it based on the surface strings. This still leaves the morphological features of the modifiers unfilled, which have to be filled using an electronic dictionary.
   While this could be possible for some features like the POS tag, an implementation needs to know the exact semantics of each feature in each corpus, which is infeasible.
2. The conversion of terminals into nonterminals removes all features and makes it impossible to query for the original terminal string, which has to be added as a new feature on the nonterminal. More importantly, this change introduces nonterminals whose interpretation differs from all other NT nodes, since they do not represent linguistic phrases but complex words. This certainly breaks the third design rule.
3. The new terminals and nonterminals "leak" into the syntactic structures and possibly create new matches for certain queries. This is in violation with the first design rule.

A safer way is to create a new node type for word parts as a subtype of SYNTAX. This node type is called PART and contains only the features defined in the annotation. This leaves the question unanswered whether word parts should be integrated into the dominance hierarchy. If we want them to be part of it, we need to change the definition of dominance to allow T nodes to have children. Again, this violates the design guidelines, because it involves far-reaching modifications of the existing language.

Instead of trying to find a workaround for this problem, it is important to look at the arguments why word parts should be in the dominance hierarchy. In general, we can assume words to be opaque with regards to syntactic structure. Linguistic phrases and parts of morphologically complex words do not interact directly, therefore it is not needed to have a direct relation between them or put them into the same annotation layer.



(a) Original node type hierarchy

(b) New type hierarchy, syntax types only

Figure 4.3: The old and new hierarchies of the syntax node types

Based on the previous arguments, we introduce a new type DNODE (short for *dominance node*[2]), which is a subtype of SYNTAX and the supertype of all nodes which can be part of the syntactic dominance hierarchy. The relationship between a terminal and its splitwords is a special morphological relation and realized using a new basic node relation. Figure 4.3b contains the graphical representation of the revised type system so far.

### *Node Types for Semantic Annotation*

Similar to the SYNTAX type, we introduce a new type SEMANTICS, derived from FREC, as the base type of all new node types for frame semantics annotation.

Since each frame has exactly one corresponding target element, one could argue that frames and targets should be merged into a single node. In the annotation format, target elements do not even have their own IDs, hence frame instances and targets cannot be separated from each other. But even given this strong connection, there are several reasons to keep frames and targets as two separate nodes with different types in the annotation graph:

2 We did not use the more adequate name DOMNODE, because "DOM" is a common acronym for **D**ocument **O**bject **M**odel, a model for structured documents like XML, which also features nodes.

- Targets behave the same way like roles, since they connect the frame instance to syntactic material. The fact that the connection has a different interpretation is aptly represented by roles and targets having different types.
- If the target is a separate node, a frame only has child nodes which connect it to syntactic material. Otherwise, its children nodes are a mixture of syntax and semantics node.
- A frame can be evoked by a number of different lexical units, which means there are many different concrete target realizations for the same frame.
- Frames and targets have an associated semantic type. Although the new frames introduced by the SALSA corpus do not have semantic types and consequently are not used in the SALSA corpus, the two values could theoretically conflict in a corpus which has this kind of information. If the target is merged with the frame node, these values need to be put into different features. Even more confusing, it yields a node with two different semantic types.

We therefore introduce a new type FRAME for frame instances and new types FE for roles and FEE for targets. The latter two types have a different feature set and a different interpretation with regard to their containing frame instance. But as argued earlier, both node types are members of frames and reference syntactic material. This is the reason for the introduction of a new intermediate type SYNSEM (short for *syntax-semantics connector*), from which we derive FE and FEE, instead of making them a direct subtype of SEMANTICS. This new type allows for more concise definitions of the new node relation constraints. Figure 4.4 contains the full extended type hierarchy as developed in this section[3].

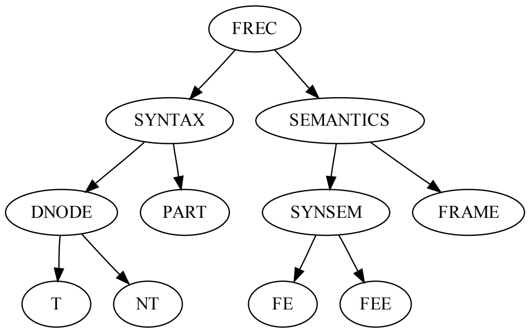


Figure 4.4: The final extended type hierarchy

### 4.2.2 *Syntactic Considerations*

All features in a TIGER corpus have to be defined for a specific node type, usually called the domain of the feature. So far there was no need to allow features with the same name for different node types—if a feature was applicable to T and NT, it could simply be defined for the root type FREC.

3 This type hierarchy is identical to the node type hierarchy in the implementation, cf. section A.3.3.

With the introduction of the new node types, avoiding name clashes among the features is more difficult. The set of features used by the frame semantics annotation is fixed and, in contrast to syntax nodes, not defined in the corpus metadata. The way feature values for nodes are annotated is also different from syntax nodes. In the syntax part, annotation elements simply carry whichever features were defined earlier. For semantics elements, the feature values are assigned based on the annotation structure, changing them means changing the implementation. In the SALSA corpus, we would therefore have a name clash between the feature *lemma* on T nodes and feature with the same name on FEE nodes.

It is theoretically possible to retroactively add feature definitions for the new types to the corpus definition, although there is no way to connect them to the semantics annotation. This alone, however, does not avoid the problem of name clashes. The problem can be solved by requiring the features of SEMANTICS nodes to have a special prefix, and making it illegal for freely defined features from the corpus metadata to begin with this string. This effectively introduces name spaces, and makes feature names longer and more difficult to remember.

In order to use elegant and fitting feature names on the new types, we extend the feature system such that two types are now allowed to have a feature with the same name, as long as none is the supertype of the other type.

Allowing features with the same name on different node types can lead to additional matches given certain queries, which is in conflict with the first design guideline. In the SALSA corpus for instance, the query in example 4.3 will now return matches for both terminals and frame-evoking elements. In order to constrain the result set, explicit type constraints need to be added, as shown in example 4.4.

(4.3) `[lemma="run"]`

(4.4) `[lemma="run" & T]`

These type constraints are strictly optional and node descriptions can refer to both syntactic and semantic types. This is unwanted behavior, since apart from all nodes being feature records, there is little behavior shared between SYNTAX and SEMANTICS. The query language does not gain any power by allowing to mix fundamentally different node types, it only becomes more confusing. Any new relation constraints between SEMANTICS and SYNTAX nodes, as well as the existing ones, are not affected by this, simply because the nodes cannot occur in the same place.

Because of this, and the design guideline that any extension does not change result sets for queries concerning the syntactic material only, we need to find another query syntax. Adding the requirement for seemingly superfluous type constraints to queries is a burden on the user, who should not have to be familiar with the intricacies of the underlying type system.

In order to overcome this problem, but also to support users with visual cues when writing queries that combine syntactic and semantic material, we extend the syntax of TIGER. Node descriptions with a type that is a subtype of SEMANTICS must be surrounded by curly braces instead of square brackets. In example 4.5, it is clear that the node description can only match an FEE and not a T node.

(4.5) `{lemma="run"}`

We say that the upper type boundary of nodes described with `[]` is SYNTAX, of the one of `{}` is SEMANTICS. This is equivalent to adding a top-level type constraint to node descriptions based on the surrounding brace type. Assuming that <> is brace type for node descriptions with the upper type boundary FREC, the two queries in example 4.6 are equivalent to each other, as are the two in example 4.7. The different brace types are reduced to syntactic sugar[4].

(4.6)
```
[lemma="run"]
<SYNTAX & (lemma="run")>
```

(4.7)
```
{lemma="run"}
<SEMANTICS & (lemma="run")>
```

If a user writes a query in which frame semantics nodes are referenced with square brackets (or vice versa), the implementation can spot the mistake by checking the types of the features used in the description against the upper type boundary and fail with a meaningful error message.

### 4.2.3 *Basic Node Relations*

To allow expression of queries on the structure of the new annotation elements in TIGER/SALSA XML, we define several new basic relation constraints. The information for the constraints can be read off directly from the annotation.

#### *Frame Structure*

To describe the structure of a frame annotation (cf. fig. 4.1), we need to introduce two new basic relation constraints.

**Definition 1 (Frame Member)** FRAME $>$ SYNSEM
*A* SYNSEM *node is a member of a the* FRAME *node.*

**Definition 2 (Syntactic Material)** SYNSEM $>$ SYNTAX
*A* SYNTAX *node is referenced by a* SYNSEM *node, the connection layer between syntax and frame semantics.*

We decided to reuse the operator symbol '>' in both cases because it can be intuitively understood as a dominance.

#### *Underspecification*

A new relation is also needed to state that two nodes are part of the same underspecification block.

**Definition 3 (Underspecification)** FRAME $\sim$ FRAME|FE $\sim$ FE|FEE $\sim$ FEE
*Two nodes with exactly the same subtype of* SEMANTICS *the are both members of the same underspecification block.*

4 While this neutral node description syntax does not exist, the different brace types are in fact converted to upper type boundaries in the actual implementation.

In this case, we decided against reusing an existing operator. The sibling constraint '`$`' might be the closest fit, because it is symmetrical and expresses membership in a common structure, but we reserve this operator as a derived relation constraint for membership of two SYNSEM nodes in the same frame. Further, we do not want to reuse a symbol which represents a derived constraint for SYNTAX nodes as a basic constraint on SEMANTICS nodes.

The constraint for secondary edges does not fit, because secondary edges are directed and do not provide a symmetric relation. We also reserve the possibility to extend secondary edges to arbitrary nodes, not just between two nodes of type DNODE.

The definition states that all semantic types can be part of an underspecification block now, although only blocks for FRAME and FE are annotated. Underspecification on the frame always implies that the targets of the two frames reference the same syntactic material. If underspecification constraints are used with two FEE nodes, the constraint checks if the containing frames are in an underspecification relation.

It is tempting to simplify the definition to SEMANTICS $\sim$ SEMANTICS. This, however, is not possible, because an underspecification block can only contain nodes with the exact same type. The definition correctly requires the same types on both sides.

*Splitwords*

The result from section 4.2.1 was to introduce a new relation constraint between complex terminals and their parts. We use the symbol '`<`' as the operator.

**Definition 4 (Part-of)** PART $<$ T
*A* PART *node is a part of the terminal* T

The symbol choice reflects that the relation between a T and its parts is similar to dominance, but not part of the regular dominance hierarchy.

### 4.2.4 *Non-Local References*

A frame instance is always annotated in the same sentence as its target element, so the FRAME node and the FEE connected to it will always be in the same graph. In contrast to that, the syntactic material referenced by the roles does not always fall into the boundaries of a single sentence, since they may reference syntactic material from adjacent graphs (cf. Ruppenhofer et al., 2006, sec. 3.2.5). In example 4.8, the role MANUFACTURER exclusively references material from the preceding sentence, but a single role can also have syntactic material from several graphs. Example 4.9 shows that a role may also reference a complete graph.

(4.8) [Das Unternehmen]$_{\text{MANUFACTURER}}$ produzierte einschließlich Fremdfertigung mehr als 20 000 Fahrzeuge. Im Jahr zuvor waren [19 348 Autos]$_{\text{PRODUCT}}$ <u>vom Band gerollt</u>$_{\text{MANUFACTURING}}$.

[The company]$_{\text{MANUFACTURER}}$ produced 20 000 cars, including external production. In the previous year, [19 348 cars]$_{\text{PRODUCT}}$ <u>left the assembly line</u>$_{\text{MANUFACTURING}}$.

(4.9) [Der US-Delegationsleiter John Kornblum]$_{\text{SPEAKER}}$ reagierte$_{\text{STATEMENT}}$ mit den Worten: “[Wir sind nicht bereit, 100 Tage zu warten, absolut nicht]$_{\text{MESSAGE}}$.”

[US delegation chief John Kornblum]$_{\text{SPEAKER}}$ reacted$_{\text{STATEMENT}}$, saying: “[We are not willing to wait 100 days, absolutely not.]$_{\text{MESSAGE}}$.”

Overall, non-local references are quite rare, only 206 of 40,020 graphs in the SALSA corpus show at least one occurrence. Most external roles belong to graphs that contain quoted or indirect speech. The relatively high frequency of this phenomenon is explained by the fact that the text is taken from newspaper articles. Table 4.2 gives more detailed statistics about non-local references in the 20,727 frame instances of the SALSA corpus.

| TYPE | COUNT |
|---|---|
| Non-local | 216 |
| Local | 41,126 |
| Total | 41,324 |

(a) FE→SYNTAX edges

| TYPE | COUNT |
|---|---|
| Non-local + local | 93 |
| Non-local only | 123 |
| Local only | 39,937 |
| Total | 40,153 |

(b) Position of FE material

Table 4.2: Statistics on FE nodes and references to syntactic material in the SALSA corpus

So far, there is no node relation constraint in the TIGER language that connect nodes from several graphs. Even for secondary edges, which could be used to model anaphoric relations between phrases in different sentences, non-local references are not supported[5] and implementations of TIGER can make the fundamental assumption that all constraints are between two nodes of the same graph. If constraints between two nodes of different annotation graphs were to be allowed, the execution model of a query evaluation implementation would have to be fundamentally redesigned.

In the next section, we will discuss possible solutions and their relative merits. The final query language extension only include predicates to check for the presence or absence of non-local references on FE nodes, any information on the referenced SYNTAX nodes is discarded.

### *Integration Approaches*

For a possible extension of query evaluation that allows non-local references from FE to SYNTAX nodes, we assume, based on the statistics from table 4.2, that non-local references are very rare and only occur in a small fraction of all corpus graphs. Local evaluation of queries remains the default; only when an external reference is actually encountered, the context of the query is extended to cover the externally referenced material. Otherwise, the evaluation context is the single

5 The index of the secondary edges in the original implementation can only handle graph-local secondary edges (cf. Lezius, 2002a, pp. 194ff).

graph as described in the corpus. A query like the one given in examples 4.10 and 4.11 then can contain a non-local reference between `#fe` and `#n`.

(4.10) *Find all frame elements which contain an* NP *that transitively dominates a* PP.

(4.11) `#n:[cat="NP"] >* #p:[cat="PP"] &  #fe:{FE} > #n`

Normally, the order in which constraints are checked against the graph does not matter for the actual result, and implementations are free to reorder them to improve execution speed. If the evaluation context can be extended gradually, all constraints which may introduce non-local references need to be evaluated up front. If any of them actually has a reference to a node in another graph, the evaluation context needs to be extended to contain this graph as well.

In example 4.11, the structure `#n >* #p` might exist in a graph $g_n$, however if `#fe` contains a reference to nodes from the graph $g_{n-1}$, checking for it in $g_n$ first is premature.

The extension also removes the possibility of using an inverted index of node features to reduce the number of graphs to be checked early in the evaluation phase. Since all node variables are bound by implicit existential quantifiers (cf. section 3.3.5), a graph which does not contain any node with category PP can never be a match for the query in example 4.11. In the presence of non-local references, this assumption is invalid, since the node may now occur in any other graph. This results in a significant increase of the number of graphs that need to be checked.

For more efficient query evaluation, information about graphs which are connected by non-local references can be precomputed. The query evaluation must be changed accordingly to take this information into account. On the other hand, users cannot assume any more that all nodes in a query are local to the same graph. In the query from example 4.12, the frame elements `#f1` and `#f2` could end up being in different graphs, if they have non-local references.

(4.12) `#f1:{FE} > #n1 & #f2:{FE} > #n2 & #n1 . #n2`

While such a result is formally correct, it is also very confusing and unexpected. Since node locality should not be given up lightly as the default, one might think about explicitly allowing non-locality of constraints using a modifier and otherwise not considering them non-local references.

### *Provisional Support*

Although extending TIGER to support references between nodes in different graphs allows for modeling and querying many other phenomena as well, we consider this extension to be outside the scope of this work and will revisit the integration approaches from the previous section in future work, because such a change requires extensive changes to the query formalism as well as the implementation. The impact of this decision is lessened by the infrequence of non-local references in the SALSA corpus.

In the query language extension we implement in this work, non-local references are going to be dropped from the annotation structure while the corpus is prepared for query evaluation. In order to provide at least some provisional support and representation of this annotation, a flag is set on the modified FE node which can be queried with two new predicates:

- `has_external(FE)`: True iff the FE node has at least one non-local reference in the original annotation
- `no_external(FE)`: True otherwise

### 4.2.5 *Frame-to-Frame and Role Relationships*

Ruppenhofer et al. (2006, Chapter 6) defines a list of relations which describe the semantic connections between frames and optionally their roles. By linking a new frame to already existing ones, these frame-to-frame relations are meant to improve the comprehensibility of frames and increase robustness in the interaction between frame nets for different languages, cf. section 1.1.2.

| REFLEXIVE | ASYMMETRIC | ASYM. & TRANSITIVE |
|---|---|---|
| *CoreSet* | *Subframe*$^+$ | *Inheritance*$^+$ |
| *Excludes* | *Using*$^+$ | |
| *Requires* | *See_also*$^+$ | |
| | *ReFraming_Mapping*$^+$ | |
| | *Inchoative_of*$^+$ | |
| | *Causative_of*$^+$ | |
| | *Is_Superseded_By-salsa* | |
| | *Modifies-salsa* | |

Table 4.3: Frame-to-frame relations in SALSA

Table 4.3 gives an overview of all frame-to-frame relations which are defined in the frame database distributed along with the SALSA corpus. It is important to note that the formal of the relations properties differ. Some are reflexive, since they only define relations among the roles of a single frame, while all other ones are asymmetric, with some allowing transitivity, too.

These properties are not part of the formal description, but for a better overview, we have added them in the table. Some of the asymmetric frame relations also define relations between roles, which is indicated by a little + symbol following the name.

Before we discuss the applicability of individual frame-to-frame relations in the extended query language, it is important to note that these relations are defined for abstract frames, not frame instances. Although we sometimes use the terms interchangeably, the distinction is fundamental in this case.

Because of the difference between abstract frame descriptions in the database and concrete frame instances in the annotation, it is questionable at best to introduce new node relations for frame-to-frame relations. The exceptions where a new relation constraint between frame instances is most useful are those frame-to-frame relations that describe discourse structure. An example of that is *Precedes*, which is not part of the SALSA frame database. A constraint for this relation will also introduce non-local references, since the linked frame instances can and will most likely occur in different graphs. The problems of non-local references have been discussed extensively in section 4.2.4.

### *Inheritance*

An important and well-populated frame-to-frame relation is inheritance, which defines a hierarchy from general to more specific frames. A small excerpt from the frame hierarchy is shown in fig. 4.5. It states that the frame BUILDING inherits from INTENTIONALLY_CREATE, which itself inherits from INTENTIONALLY_ACT.

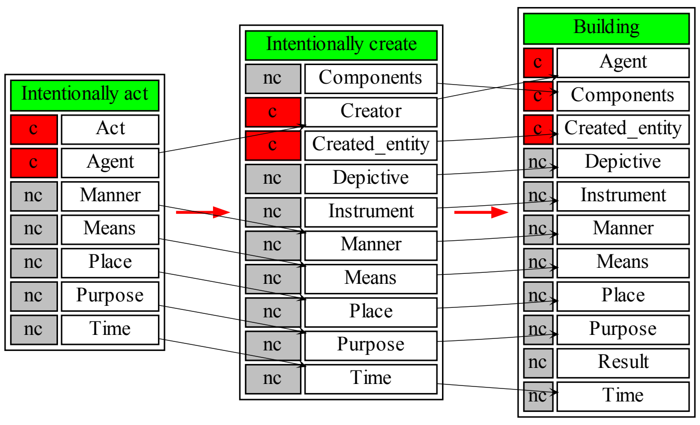


Figure 4.5: An excerpt from the inheritance hierarchy defined by the frame-to-frame relations[6]
Thick arrows encode inheritance of frames (Generic→Specific), thin ones inheritance of roles. Roles marked with *c* are core roles, *nc* roles are peripheral.

Since searching for frames given a more general frame is an interesting feature, we want to include a mechanism to support queries like the one in example 4.13.

(4.13) *Return all graphs which contain instances of a frame that inherits from* INTENTIONALLY_ACT.

The name of a frame is already a feature of the node, and using the name and the inheritance relations from the frame database, an implementation can build a hierarchy of frames to evaluate these kind of queries. Given the small hierarchy in fig. 4.5, the result set of the query in example 4.13 contains the instances of all three frames.

We need to extend the syntax for node descriptions to include a new comparison method for feature values, additionally to string and regular expression literals. Since this syntax is needed for semantic types as well, we will discuss it separately in Section 4.2.7.

The inheritance hierarchy also defines relations between the roles of two frames, as indicated in the excerpt. The hierarchy of roles will

6 Inspired by the FrameGrapher tool, `http://framenet.icsi.berkeley.edu/FrameGrapher/`

be included into the extended query language, to support queries for roles similar to 4.13. In this context, it is important to recall that two roles with the same name in different frames do not refer to the same entity. If they did, some frame hierarchies would introduce cycles in the role hierarchies, e. g. AGENT→CREATOR→AGENT in the hierarchy from fig. 4.5. To avoid this problem, role names need to be disambiguated transparently.

#### *Core Sets*

Each frame can have one or more core sets of roles (Ruppenhofer et al., 2006, sec. 3.2.2.1). If any of the elements of one core set is present on a frame instance, the frame is considered complete. For example, the frame BRINGING has the two possible core sets {SOURCE, PATH, GOAL} and {AGENT, CARRIER}. If any role from one of the sets is annotated for a given frame instance, the semantic valence of the frame is satisfied, i. e. the event is described in its entirety, although not all core roles occur. It is unusual that all members of a core set are annotated, but possible.

The *CoreSet* relation is a reflexive relation, since it is only concerned with the roles of a single frame. We introduce two new predicates for frames:

- `has_coreset(`FRAME`)`: True iff any member of at least one core set has been annotated on a frame instance
- `no_coreset(`FRAME`)`: True otherwise

We disregard that FE nodes can be checked for membership in a core set, a good approximation can already be achieved by using the *coretype* feature to be introduced on roles, cf. section 4.3.1. If the use case for exact membership in core sets arises, such predicates can be added.

#### *Excludes and Includes*

*Excludes* and *Includes* are both reflexive relations on frames which describe the dependencies between roles. While both can be realized with new basic relations, they would also both need predicates in case the role in- or excluded is not present on the frame instance, which is the default case for *Excludes*. Because of the comparatively small importance of the feature, we do not include it. Again, we point out that they can be included given the use case.

### 4.2.6 *Other Frame Relations*

None of the other frame relations listed in table 4.3 are included in the query language. Most of them do not provide any helpful features for the query language, define only very few actual relations or are only of interest to corpus or frame net authors. All of them require the introduction of new constraint relations or modifications of the node description syntax and therefore provide very little gain in query power for a moderate increase of language and implementation complexity.

As already mentioned, all these relations are defined on frames, not on frame instances, and only frame instances are members of the domain of the extended TIGER query language. Relations between the types of frames have to be encoded in a different way, should they be part of the query language, as in the example of inheritance.

### 4.2.7 *Type Hierarchies for Feature Values*

All node types for frame semantics have a feature for their semantic type. Semantic types are organized in a small ontology and describe possible fillers for roles and formal properties of frames. Similar to the frame and role hierarchies, it is natural to query for all roles whose semantic type is derived from a given type.

As mentioned in section 4.2.5, we need a new syntax element for type literals in feature constraints, which signals that a value is to be interpreted as a base type and not as a literal string like in query 4.14, which only matches the exact type SENTIENT.

(4.14) `{FE & semtype="Sentient"}`

The original TIGER language definition already contains support for type literals in node descriptions. If, for instance, the two POS tags `NE` and `NN` are defined to have the type NOUN, it is possible to write a query like the one in example 4.15[7].

(4.15) `[pos=noun]`

We decided against using this syntax[8] and instead introduce our own. If a feature value must inherit from a given feature, based on the hierarchy of feature values, a string on the right hand side of a feature constraint has to be enclosed in square brackets. It is also possible to use negation on the feature, in which case only types which are not subtypes are a match. A query is shown in example 4.16.

(4.16) `{FE & semtype=[Sentient]}`

There are several reasons for using a new syntax element:

- The type NOUN in example 4.15 is defined in an external type hierarchy. In our case, the feature values themselves (like SENTIENT) are types and organized in a hierarchy.
- Using completely undecorated strings, as shown in example 4.15, provides very little visual cues to distinguish type from normal string literals. Simply dropping the quotation marks does, in our opinion, not reflect the change in interpretation.
- Similar to regular expression literals, the change of which feature value is considered a match is reflected by a variation of the surrounding elements. This is simply a new manifestation of an abstract syntactic pattern present in the language.

An implementation has to make sure that type literals are used only in constraints for features which are in a hierarchy, otherwise it must fail with an error message. It also needs to obtain the feature hierarchies, since they are not included in the corpus definition, and store them to be able to resolve type queries at runtime.

7 Taken from section 8.2 of the online TIGERSearch manual at `http://www.ims.uni-stuttgart.de/projekte/TIGER/TIGERSearch/doc/html/QueryLanguage_Types_Definition.html`

8 As to this point, this syntax is not supported in our implementation.

### 4.2.8 *Backwards Compatibility*

It was part of our design guidelines not to break backwards compatibility with the syntax part of TIGER and we need to revisit the way empty node variables, i. e. variables that are not attached to any node description, are handled.

The goal of the new node description syntax introduced in section 4.2.2 was to prevent a mixture of SYNTAX and SEMANTICS nodes in the same variable. While it is not possible to write node descriptions with the upper type boundary FREC, it is possible to leave nodes completely undefined by only using a node variable, effectively achieving the same.

For all constraints and predicates, associated node types are known. An implementation can use *type inferencing* to identify the exact type of an "empty" variable, to ensure that SYNTAX and SEMANTICS nodes are kept separate. This type inference fails in some cases, since we decided to reuse the operator symbol '`>`', originally for dominance in the syntax part, for the frame semantics annotation as well. Example 4.17 shows a query where the type inferencer fails to identify variables as either having SYNTAX or SEMANTICS types only.

(4.17) `#a > #b`

In the extended query language, example 4.17 has the following interpretations:

1. *Find all* NT *nodes which dominate another* DNODE.
2. *Find all* FRAME *nodes which contain a* SYNSEM *node.*
3. *Find all* SYNSEM *nodes which reference a* SYNTAX *node.*

In the original TIGER query language, only the first item is a valid interpretation and our design guidelines specifically required the extensions not to introduce any changes to the result sets of syntax-only queries, which can be interpreted as syntax-only.

The reuse of operator symbols has caused a violation of said rule, but we reused operators in accordance with the second design rule, which requires the changes to the query language to be as small as possible. To keep the operators, we have to introduce one additional rule for the type inferencer, which will keep backwards compatibility with syntax-only queries at the expense of increased verbosity in some queries including frame semantics:

**Type Inference Rule 1** *If and only if, after considering all sources of type information (constraints, features, predicates), the set of possible types for a variable* `#v` *still includes at least one subtype of* SYNTAX *and one subtype of* SEMANTICS, *then the implementation should assume that* `#v` *is a subtype of* SYNTAX.

This makes it impossible for the query in example 4.17 to have anything but the first interpretation. If another interpretation is required, the node types of the variables have to be fixed using type constraints, as shown in examples 4.11 and 4.12.

### 4.2.9 *Final Remarks*

Syntax of formal, especially programming languages, is always a delicate matter and tends to lead to fundamental arguments over seemingly unimportant details. We do acknowledge that our decision to introduce a new brace type for frame semantics nodes might be seen as controversial and a further extension of TIGER to include a new kind of annotation would run short of new brace types.

Another decision we made early in the design phase, to reuse constraint operators for new basic node relations, forced us to integrate a new implicit rule in the type inferencer to keep backwards compatibility could be criticized, too.

We justify these decisions by the fact that our extension to the query language is a pragmatic inclusion of new features in order to allow working with existing resources. A query language that completely formalizes multi-level annotation and non-local references and does not need special rules for backwards compatibility requires a change in the annotation format, the object model and the query syntax. This is very well outside the scope of our work.

## 4.3 SUMMARY OF THE EXTENSIONS

This section contains the complete and definitive description of the query language extension to handle annotation of frame semantics as contained in the TIGER/SALSA XML format used for the SALSA corpus. It covers the extensions introduced in the design section, introduces several derived relation constraints and redefines existing query language elements to cover frame semantics annotation as well. This section serves as the basis of the implementation outlined in chapter 5.

A condensed version of this overview is included in appendix B, which is supposed to serve as a quick reference of the extensions when writing queries.

### 4.3.1 *Features*

For all new features added by TIGER/SALSA XML, we list the formal properties and describe how their values are obtained from the corpus.

#### *Common Features*

Two features are defined for the type SEMANTICS. Since this is the supertype for all nodes in frame semantics annotation, its features are inherited by the more specific types.

All semantic nodes have a feature *flag*. Flags are strings which are used to mark the existence of linguistic phenomena. Since no authoritative list of flags is given in the corpus metadata, the feature is treated like an open-class feature and values are taken directly from the corpus annotation. In TIGER/SALSA XML, targets cannot have flags. Therefore, FEE nodes from corpora encoded in this format will never have this feature filled with a value.

The feature *semtype* is present on all nodes, too. Since the SALSA corpus distribution does not include the hierarchy of semantic types, the one included in the Berkeley FrameNet distribution should be used instead. Only the types which are listed in this hierarchy are valid feature values, *semtype* is treated like a closed-class feature from the corpus metadata.

The semantic types are not directly included in the annotation but available in the frame database. When a corpus is prepared for query processing, semantic nodes need to be enriched with this information.

If the frame database does list a semantic type for an element, the *semtype* feature is set to the type *void*, which is provided by the implementation. Undefined semantic types are dropped from the annotation and produce a warning. This feature can be used with the new type literals; for a sample usage, see the query in example 4.16.

### *Frames*

Nodes with the type FRAME have the feature *frame*, which contains the name. The list of valid frame names is defined by the descriptions in the frame database, all other names are considered to be an error in the annotation. Based on the inheritance relations defined in the frame database, the *frame* feature also supports the use of type literals, as shown in example 4.18.

(4.18) `{frame=[Rewards_and_Punishment]}`

### *Frame-Evoking Elements*

Frame-evoking elements, type FEE, have two additional features, *lemma* and *head*. The features are filled by the lemma and head lemma given in the annotation. Both features are treated as open-class. Although the frame database specifies a list of possible lexical units for each frame, reconstructing it from the annotation might not be possible in all cases. The list can be incomplete, especially with regard to English frames reused by the SALSA project.

As mentioned in section 4.1.1, the head is only given if the lemma is a complex string. If the head lemma is empty, the lemma is takes as the value for both features.

### *Frame Elements*

Two more features are defined for frame elements, type FE. The feature which identifies a frame element is *role*. The description of a frame lists all possible roles, but since different frames can have roles with the same name, role named are made unique internally. The unique role names then form the list of valid feature values. If the role name is not found in the frame database, it is treated as a mistake in the annotation and dropped from the node.

The disambiguation of role names happens transparently for the user, the result set of query 4.19 will contain all graphs that contain a frame instance with a role named EVENT. If only roles from a certain frame should be matched, a constraint on the containing frame has to be added, as shown in example 4.20.

(4.19) `{role="Event"}`

(4.20) `{frame="Death"} > {role="Event"}`

The *role* feature supports usage of type literals, too, the hierarchy of roles is given by the hierarchy of frames.

The second feature is *coretype*, which describes the *coreness* of a role with regard to its containing frame, i. e. whether a role usually must be annotated in a frame instance, or simply serves as a modifier. The core types of roles are defined in the frame descriptions. Although there is no list of valid core types in the frame database, the implementation only accepts these four core types:

- Core
- Peripheral
- Extra-Thematic
- Core-Unexpressed

Like the *semtype* feature, the core type is added to FE nodes during corpus preparation.

*Word Parts*

PART nodes have the single feature *part*, which is taken directly from the splitword annotations in the corpus. Because this feature contains surface material, it is open-class.

### 4.3.2 *Relation Constraints*

Additionally to the basic relations that describe the structure of frame annotations, we introduce new derived constraints which allow more compact expression of certain queries. All constraints support negation using the standard '`!`' prefix modifier.

*Frame Members*

Syntax: FRAME > SYNSEM

The frame member constraint is true if a SYNSEM node is annotated in a FRAME node. A query is given in example 4.20.

Negation of the constraint suffers from the limitations discussed in section 3.3.5, but since our implementation contains support for simple universal quantification discussed in the same section, it is possible to search for frame instances that do not contain a specific role. The result set of the query in example 4.21 contains only instances of the frame DEATH in which the CAUSE role is not filled.

(4.21) `{frame="Death"} !> %s:{role="Cause"}`

*Frame Siblings*

Syntax: SYNSEM $ SYNSEM

Similar to the sibling constraint for DNODES, the derived frame siblings constraints is used for testing whether two SYNSEM nodes occur in the same frame. Using this constraint, the query in example 4.22 can be shortened to 4.23 without changing the result set.

(4.22) `#f:{FRAME} > {role="Speaker"} & #f > {role="Message"}`

(4.23) `{role="Speaker"} $ {role="Message"}`

*Syntactic Material*

Syntax: SYNSEM > SYNTAX

This constraint can be used to check for the syntactic material directly referenced by a target or a role, which is defined in the corpus annotation. Similar to the dominance constraint on syntax nodes, the constraint behavior can be changed using a number of modifiers, which can all be negated.

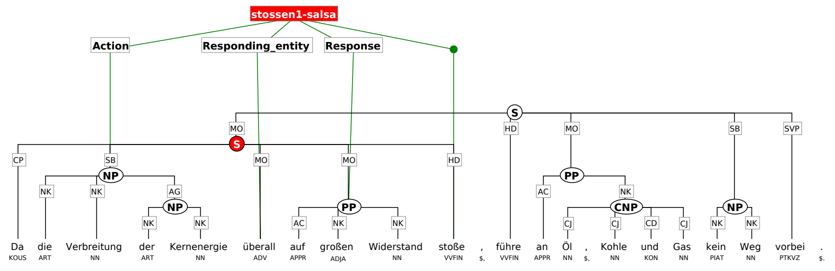


Figure 4.6: Match for the first common ancestor constraint from the query in example 4.25

$>^*$ true if the SYNTAX node is contained somewhere in the syntactic material, i. e. is either directly referenced or dominated by a directly referenced syntax node.

$>n$ same as above, but the distance between the SYNSEM and the SYNTAX node must be exactly $n$.

$>n, m$ same as above, but the distance between the SYNSEM and the SYNTAX node must lie between $n$ and $m$ inclusive.

From this basic relation, we derive a constraint that uses FRAME nodes on the left-hand side, resulting in FRAME > SYNTAX. This constraint is true if the SYNTAX node is referenced by the target or any role of the FRAME node. Example 4.24 shows a query which matches all graphs in which a terminal with the lemma string "Katze" occurs anywhere in a frame instance.

(4.24) `{FRAME} >* [lemma="Katze"]`

*First Common Ancestor*

Syntax: NT ^ SEMANTICS

The first common ancestor constraint is a combination of dominance and frame membership. It is met if an NT node is the lowest (as seen from the root node) node which transitively dominates all SYNTAX nodes referenced in a frame, target or role. For this constraint, PART nodes take the same position in the tree as the T node they belong to.

(4.25) `[NT] ^ {frame="stossen1-salsa"}`

Figure 4.6 shows a graph that is a match for query in example 4.25, the node which corresponds to `[NT]` is highlighted.

*Underspecification*

Syntax: FRAME ~ FRAME|FE ~ FE|FEE ~ FEEa

The underspecification constraint checks if two nodes are members of the same underspecification block. In the case of negation, the constraint is satisfied either if the nodes are in different underspecification blocks or at least one is not part of such a block at all.

*Part-of*

Syntax: PART $<$ T

The part-of relation connects a complex terminal of type T with its parts. Usually, only words which contain parts that are referenced in SYNSEM nodes are annotated as splitwords. This constraint therefore is of little help for morphological queries.

### 4.3.3 *Predicates*

Section 4.2 already introduced several predicates for the new node types. We also include a list of derived predicates and extend existing ones to cover the new node types as well. For most predicates, both positive and negative versions need to be defined, since TIGER does not include negation at the topmost expression level.

*Arity*

The predicate for arity of a node is extended to cover all new node types as well, which is in accordance with the usage of `arity` in graph descriptions, as discussed in section 3.2.3.

*Coresets*

For FRAME nodes, two predicates exist which can be used to check if any or no core set of roles has been instantiated completely. The two predicates are:

- `has_coreset(`FRAME`)`
- `no_coreset(`FRAME`)`

*Non-local References*

Non-local references from SYNSEM to SYNTAX nodes in other graphs are not supported, but two predicates can be used to check whether the original annotation contained any references of this kind that have been dropped:

- `has_external(`SYNSEM`)`
- `no_external(`SYNSEM`)`

The predicates do not distinguish between SYNSEM nodes that have only non-local and those that have some local references, too. Example 4.26 show a query that matches FE with only non-local references, which are not counted towards the node arity.

(4.26) `#f:{FE} & has_external(#f) & arity(#f, 0)`

*Underspecification*

Checks for membership in the same underspecification block can already be done using the new constraint ~, however no possibility exists to express that a certain node does not belong to any underspecification block at all. It should also be possible to state that a node is underspecified without the need to introduce another node variable. Two new predicates are added to address these requirements:

- `uspec`(SEMANTICS)
- `spec`(SEMANTICS)

`uspec` is simply an abbreviation of the underspecification constraint. Its counterpart `spec` cannot be expressed using the negated constraint because of the implicit existential quantification, which is the reason for its inclusion.

## 4.4 RESULT

With the definitions of node types and relations describing frame annotation structure, we can revisit the query from section 1.3 which served as the original motivation for the extension:

(4.27) *Find all sentences where the role* TOPIC *in the frame* STATEMENT *is realized by a PP with the preposition "über".*

To formulate this query, we need four of the new elements:

1. Find all instances of TOPIC:
   `#r:{role="Topic"}`
2. Find all instances of STATEMENT:
   `#f:{frame="Statement"}`
3. Role is a member of frame:
   `#f > #r`
4. Role is fully realized by a syntax node:
   `#r > #pp & arity(#r, 1)`

Putting all these parts together with the original syntax query fragment from example 1.9, we get the query in example 4.28, which is a correct formalization of example 4.27.

(4.28)
```
{frame="Statement"} > #r:{role="Topic"} &
#pp:[cat="PP"] >AC [word="über"] &
#r > #pp & arity(#r, 1)
```

To relax the constraint on the syntactic material of the TOPIC role, it is possible to drop the arity constraint. In this case, the role might have several more references.

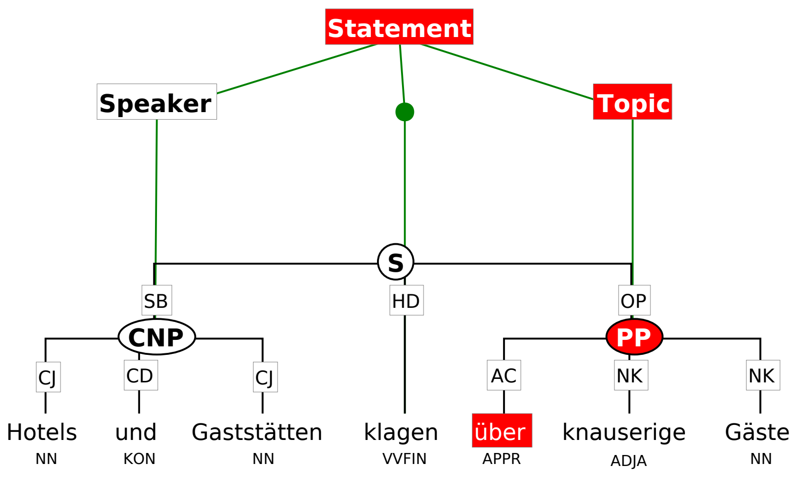


Figure 4.7: The annotated graph for the sentence from example 1.8

Figure 4.7 shows the annotation graph of the sentence in example 1.8 with the nodes bound to the variables in query 4.28 filled, which demonstrates the ability of the extended language to express queries as required in section 1.4.1.

# 5 IMPLEMENTATION OF THE EXTENSION

In this chapter, we will introduce the general architecture of our query evaluator implementation. In the architectural overview, we explain key algorithms and show where the new query language elements from chapter 4 are plugged into the existing hierarchy.

In the second part of the chapter, we show how the frame semantic annotations from TIGER/SALSA XML corpora are preprocessed, which kind of information is put into the index used for query evaluation and how this data is used in the constraint checks.

The descriptions in this chapter are as language-agnostic as possible. A walk-through of the implementation is given in appendix A.

## 5.1 ARCHITECTURE OF THE QUERY EVALUATOR

The new query elements for frame semantics annotation occur in different parts of our TIGER query evaluation engine. For an understanding of the implementation requirements, it is necessary to have an overview of the different subsystems in the query module and to know how these systems interact to create the result set for a TIGER query—the process which we refer to as *query evaluation*. An important part of the evaluation is the corpus index, which contains a representation of the corpus that is optimized for searching and which is created prior to any other operation.

### 5.1.1 *The Corpus Index*

TIGER-XML and its derivative allow a moderately compact and efficient representation of annotated corpora as XML document trees. XML is a generic and flexible markup language, originally for narrative documents, but it also became the quasi standard for structured data in recent years[1]. There are libraries for XML handling written for almost all modern general-purpose programming languages, and a large number of programs and tools.

The huge popularity of XML has also lead to its wide spread adoption in the NLP community, and many formats for storage of linguistically annotated text similar to TIGER-XML have been developed, like GrAF (Ide and Suderman, 2007), PAULA-XML (Dipper, 2005), EXMARaLDA (Schmidt, 2001), MMAX2 (Müller, 2006), XCES (Ide et al., 2000) or the NITE XML toolkit (Carletta et al., 2003), some of which are also queryable and have been discussed in 2

The technological ecosystem that grew around XML since its introduction in 1998 also contains a general-purpose query language for XML documents (XQuery) and a language for querying nodes in document trees (XPath). It is possible to extend XPath to cover linguistic

1 Although recent developments have lead to the introduction and adoption of more compact markup languages like JSON.

queries on XML documents encoding annotated text (Bird et al., 2005; Bird and Lee, 2006).

In the case of TIGER-XML corpora, using either query language directly is highly problematic. As explained earlier, TIGER annotation constitutes a graph with crossing edges, a structure which cannot be directly represented in the hierarchical structure of XML. TIGER-XML stores the graphs in a flat representation and makes use of the tree nature of XML for structuring of the corpus only, not for storing hierarchical annotation.

XML is also a very verbose formalism and TIGER-XML corpora are usually large. In canonicalized form[2], the SALSA corpus needs 144 MiB of storage. Storing this XML tree completely in-memory needs huge amounts of RAM, and parsing XML documents is a very time-consuming process.

In order to avoid these problems and to provide efficient evaluation of queries, TIGER-XML corpora are preprocessed and the actual work is done on a corpus index. This index contains the complete original annotation, but in a much more condensed representation, which is not designed for data interchange. For each node, information is precomputed to allow fast checking of relation constraints. This makes it possible to implement dominance checks, which have to be tested using a path search in the original annotation graph, as a simple sequence comparison operation. The corpus index is created only once, which usually takes several minutes, and stored along with the original corpus files.

#### *Index Contents*

The corpus index consists of several major parts which are used at the different stages of query evaluation:

- *Feature Definitions*
  Feature domains, lists of values and formal properties.
- *Inverted Features Indices*
  A lookup table which lists all nodes with a certain feature value, similar to a full-text index for texts.
- *Node Data*
  For each node in the corpus, a list of precomputed values for constraint and predicate checking. The actual fields in these entries depend on the concrete type of the node.

### 5.1.2 *Overview*

So far, we have used the term *query evaluation* through the work without defining it properly. By evaluation, we refer to searching each graph in a corpus index for the structure described by a TIGER query and identifying all distinct combinations of nodes that fulfill the constraints of the query. This process generates a *result set*, which contains all matches that were found in the corpus. For each match, the following information is given:

- the unique identifier of the graph the structure was found in
- for each variable defined in the query, the unique identifier of the node which was bound to it.

2 Canonical XML Version 1.0, `http://www.w3.org/TR/XML-c14n.html`

The process of searching the corpus index for matches involves several steps:

1. Parsing the query
2. Query analysis
3. Node candidate retrieval
4. Relation constraint checking
5. Result set creation

Each step is handled by its own sub-module, apart from the last two, which are merged. The sub-modules are oblivious of the inner workings of the preceding or following steps and only depend on the input data structures and the corpus index.

In the upcoming sections, we will visit each step briefly and explain its in- and outputs. A walkthrough of all these steps with a sample query is given in section 5.3.

### 5.1.3 *Query Parsing*

In the first step, the query string must be parsed into an internal data structure that can be easily processed and modified by later stages. In programming language and compiler theory, such a representation is usually called *abstract syntax tree* (AST). It abstracts away from the concrete textual representation, which includes insignificant white space, newlines or features which can be expressed in several different ways and represents it using a tree data structure with typed nodes for different syntactic elements. In this sense, an AST contains a normalization of the material found in the query string, but in a data structure which can be processed and modified more easily.

The query parser itself only checks for the well-formedness of the query, but no checks regarding the validity are done. This makes the parser completely independent from the corpus index.

### 5.1.4 *Query Analysis*

After the well-formedness of a query has been established and an AST created, it needs to be tested for validity. This is only possible in the context of a corpus; if the query contains an undefined feature, for example, it is invalid and evaluation therefore impossible.

In section 4.2.7, we also mentioned that the implementation has to make sure that type literals in feature constraints are only used for those features that define a hierarchy, which is also enforced in this step. Query analysis also introduces variables for those node descriptions that were not bound previously, thus making sure that each node can be referred to by a name during the evaluation phase.

The most important part of the analysis is the type checking, which tests if the features, constraints and predicates used for each variable allow a match at all, or introduce a conflict which prevents any match. This makes it possible to distinguish between queries which simply do not have any match, because the structure they describe never occurs in the corpus, or queries that cannot occur, because the structure they describe is impossible.

An invalid structure is shown in the query in example 5.1. The intersection of the node types T and NT is empty, no node can exist which has both types at the same time. This is in contrast to example 5.2, where the type constraint for DNODE is superfluous, because it is already implied by the more specific type NT.

(5.1) `[T & NT]`

(5.2) `[DNODE & NT]`

The type checking also has to inspect the constraints and compare them against their abstract type definitions. The query in example 5.3 is invalid, because a frame cannot contain another frame. The relation constraint FRAME $>$ FRAME is undefined.

(5.3) `{frame="Cure"} > {frame="Healer"}`

### *Type Inference*

The type, or set of types if the node description contains a disjunction, for each node variable has to be determined by combining all sources of type information,which are the features and type constraints used in the node descriptions, the predicates and the relation constraints. Type inference is especially important for *empty* node descriptions, because they can refer to any node in the graph, and any kind of information inferred about such nodes can drastically decrease the time needed to evaluate the query.

For each disjunct of a node description, the initial type is identified by creating the intersection of type constraints and feature domains. The intersection of two types A and B is always the more specific one (cf. fig. 4.4 for the type hierarchy). If the two types are not in an inheritance relation, the intersection is empty and the disjunct is marked as invalid. If all disjuncts are invalid, the query is reported as invalid, otherwise the invalid parts are discarded silently. If the variable does not have any node description, the type defaults to FREC.

These types are combined with the type definitions of the predicates applied to the node variables, if any. The last step involves checking the types against the definitions of the relation constraints. If after this step, the set of possible types for a variable still contains one subtype of SYNTAX and one subtype of SEMANTICS, the special rule for breaking these ties is applied, cf. section 4.2.8. At the end, if the set of possible types for any variable is empty, the query is rejected as invalid.

### *Query Objects*

For the actual evaluation, predicates and relation constraints are instantiated. In both cases, the parser only checked the well-formedness, but not if the combination of modifiers on a constraint was valid, if all arguments for a predicate were specified or if a predicate name was defined at all. The analysis module uses the registries of predicates and relation constraints and return concrete implementations, i. e. objects which contain the concrete code for the evaluation.

Each node variable is represented by a *node query* data structure, which combines its type, the associated node description, which is extracted from the AST, and all predicates. This data structure is used by the node candidate retrieval. The relation constraints along with the left and right operand variables are passed to the constraint checker.

### 5.1.5 *Node Candidate Retrieval*

For each node variable, all matching nodes from the whole corpus are searched. This is done by joining the feature constraints and predicates[3] into a single index query. The results for the single node variables have to be grouped together based on the graph. Since each node variable is bound by an existential quantifier, only those graphs for which each variable has at least one matching node are considered in the relation constraint checking phase. Relation checks are computationally expensive, the algorithm therefore tries to minimize the number of possible candidates for each node variable.

No changes are necessary to support new features in a corpus. Since feature definitions are part of the corpus index and not hard-coded, they just need to be added during the preprocessing phase. Because features are just string values, any new feature does not need special handling. In contrast to that, new predicates need an explicit implementation, since it is tightly coupled to the structure and type-dependent interpretation of the node data entries. The instantiation is generic, any predicate simply has to be registered under its name, it can then be created during the query analysis step.

#### *Type Literals*

The only extension which involves a substantial change to the node retrieval are the type literals in feature constraints, since they constitute a novel way of matching a feature value against a constraint. The query analysis module has made sure that type literals are only used for features which define a hierarchy, but the feature values cannot be checked using exact string matching any more, as there are many different feature values which fulfill the constraint.

If a type literal is encountered, the transitive closure over all children of the given value has to be created. In the query in example 5.4, all frames that inherit from EVENT node should be matched. For this, the hierarchy of frames is needed, which is read from the frame database and stored in the corpus index during the preprocessing.

(5.4) `{frame=[Event]}`

The actual search for matching nodes needs to be modified in such a way that any number of matching feature values can be specified, without hurting efficiency. In our implementation, there is a generic way of finding node candidates based on a list of values rather than a single value. This code was originally used for regular expression literals only, but could be generalized with little effort.

### 5.1.6 *Result Set Creation*

Relation constraint checking and result set generation are handled in a single step. After all matches have been identified, the result set creation is trivial and simply involves putting the results into a data structure that is more convenient for further processing.

Under the simplifying assumption that all node variables have the same number of candidates, there are $n^m$ possible matches for a query

3 Type constraints like `[T]` are handled as predicates internally.

in a graph, with $n$ being the number of node variables and $m$ the average number of candidates per node. Relation constraint checking is the most complex part of the query evaluation pipeline, and therefore the one which usually takes longest. The individual checks as well as the algorithm are well-optimized and also contain heuristics to minimize the number of checks that need to be performed.

The simplistic algorithm for constraint checking is short and follows a generate-and-test pattern. For the node candidates of a given graph, all possible combinations of assignments to node variables need to be done, since a node variable can only be bound to one node at the time. On each of these assignments, the relation constraint checks are performed. A particular assignment of nodes to variables that satisfies all constraints is considered a match and added to the result set.

The constraint checking algorithm is largely oblivious of the internals of individual constraints. Each relation constraint has a set of formal properties, which are used to improve evaluation efficiency, but the internal workings of the relation and which fields from the node data entries it needs is of little importance to the algorithm. Just as in the predicate handling, we need to implement the new constraints, or extend existing ones in case of operator reuse. They are associated with the specific AST nodes which are used to represent the operators in the query string. No changes to the core of the result building algorithm are needed for the new constraints.

#### *Evaluation Order and Parallelism*

Node candidate retrieval and result building is done for one graph at a time instead of processing the whole corpus in a single step. This has the advantage that matching graphs can be reported to a user as soon as possible. If a query takes very long to evaluate or the connection between the user and the query server has a high latency[4], a result can be presented as soon as the first match is found. This improves subjective performance and interactivity, at the expense of not knowing the final size of the result set in the beginning.

Since graphs are independent of each other, the checks of relation constraints can be performed in parallel. On computers with several independent processors or processor cores, the whole corpus is split into $n$ equally large parts[5] and the query is evaluated on each part in parallel, resulting in a substantial speed improvement.

## 5.2 PREPROCESSING FRAME SEMANTICS ANNOTATION

The corpus index is fundamental to efficient evaluation of even the most simplistic queries. Since it is a process which only needs to be done once, it almost always pays off to spend more time during the index creation and precompute more information, which just needs to be accessed instead of computed on-the-fly during evaluation.

The corpus index creation is graph-based. After the corpus metadata have been parsed and all features are known, each graph is processed by a number of steps:

1. Convert the graph description from the XML corpus file into a data structure.

4 When using a web query frontend, for instance.
5 $n$ should be proportional to the number of available CPUs.

2. Enrich the annotation graph with external features.
3. Index the feature values.
4. Precompute node data for constraints and predicates.

The data structure used during the preprocessing is a directed graph which contains all nodes with their features and encodes the relation as typed edges. It is not used during evaluation, but still needed for graph rendering.

For the frame semantics annotation, the features *coretype* and *semtype* have to be added to the respective nodes from the frame database, which is done in the second step. The feature value indexing algorithm is independent of the actual features, it simply covers all available features on each node. Because of that, the introduction of new nodes for frame semantics does not involve any changes to this part.

### 5.2.1 *Precomputed Information*

The structural information in the graph needs to be converted into a representation that allows efficient checks of relation constraints and predicates during evaluation. This representation is stored in the node data.

Each entry in the node data index, whether for SYNTAX or SEMANTICS nodes, contains a field which encodes its exact type. This field is used for type constraints in node descriptions. In the remainder of the section, we will explain the other data fields for SEMANTICS nodes.

#### *Frame Numbers*

In order to evaluate the constraint FRAME $>$ SYNSEM and its derivative SYNSEM \$ SYNSEM, each frame receives a numeric ID which is unique within a single graph. It is stored in the entries of the frame and all its direct SYNSEM children. The relation constraints for frame membership and frame siblings evaluate to true if the node operands on both sides have the same frame ID.

#### *Underspecification Blocks*

Each underspecification block receives a unique ID. This ID is stored in the entries of all nodes that are members of the block, usually only two. All nodes that are not part of any block at all receive a reserved value, which signals the absence of underspecification.

The underspecification constraint can then check if both operands have the same block ID, which has to differ from the reserved value. In the predicates, the block ID of the node is simply compared to the reserved value. If both are equal, `spec` is true, otherwise it is false. `uspec` is the negation of these results.

#### *Node Addresses*

The preprocessing for relations between SYNSEM nodes and their syntactic material relies on the way the dominance relation constraint is implemented. Each node in a graph is assigned a unique address, which is derived from the path between the root node and itself. The *Gorn address* of a node (Gorn, 1967) is created by the following algorithm:

- The address $G_r$ of the root node $r$ is the empty address $()$.
- All non-root nodes receive a numeric identifier $i_n$, which has to be unique at least among all its siblings.
- The address of a node $n$ is found by concatenating the address $G_m$ of its parent node $m$ with $i_n$, i. e. $G_n = G_m \oplus (i_n)$

Since the syntax part of the graph is a tree, there is only one such address for each node. The identifiers $i_n$ are assigned to nodes based on their position in the list of child nodes, but they do not have an interpretation by themselves. Examples of Gorn addresses are shown in the graph in fig. 5.1.

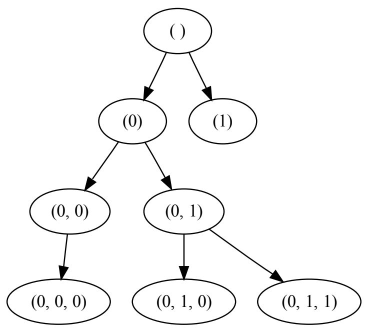


Figure 5.1: Nodes with Gorn addresses

The addresses are stored in the node data entries for DNODES and used by the various dominance constraints. Equations 5.5 and 5.6 show the definitions for transitive and direct dominance between two nodes $n, m$ based on their Gorn addresses.

$$\mathrm{dom}(n, m) = G_n \sqsubseteq G_m \wedge |G_m| - |G_n| == 1 \tag{5.5}$$
$$\mathrm{dom}^*(n, m) = G_n \sqsubseteq G_m \wedge |G_n| < |G_m| \tag{5.6}$$

$|G|$ is the length of the address $G$, with $|()| == 0$, $\sqsubseteq$ the string prefix operator and $()$ the trivial prefix of any string.

For the relations between SEMANTICS and SYNTAX nodes, we want to reuse this addressing scheme. However, it is not possible to include the frames and SYNSEM nodes into the address hierarchy and give them their own Gorn addresses, because for each of the SYNTAX nodes in a frame, this introduces a new path to reach it from the root and thus an alternative Gorn address. If a node can be represented by several different addresses, the checks for dominance as presented earlier do not work any more. Furthermore, we have to account for one SYNSEM node having references to any number of SYNTAX nodes.

Our solution to this problem is to give each SYNSEM node entry an *address-list* field that contains the Gorn addresses $G_0, \ldots G_i$ of all referenced nodes. The addresses are stored as a tuple $(P, \{S_0, \ldots, S_i\})$ of the longest prefix $P$ that is common to all addresses and the set of suffixes $S$ for the individual nodes. The original addresses can be reconstructed by concatenation, for instance $G_0 = P \oplus S_0$.

A demonstration of this storage scheme is given below. The list of three addresses in example 5.7 is compressed to the value in example 5.8.

(5.7) $(1,0,1,2), (1,0,1,3), (1,0,1,4,5)$

(5.8) $((1,0,1), \{(2), (3), (4,5)\})$

If a SYNSEM node has just a single reference, the set of suffixes only contains the empty suffix address $()$.

This form of storage has several advantages over just listing the individual addresses:

- In the general case, the compressed version can be stored more efficiently than the concatenation of addresses
- The prefix $P$ is also the address of the node that is the lowest common ancestor of all nodes. It can be used directly by the NT ^ SEMANTICS constraint.

The generalizations of eqs. 5.5 and 5.6 for references between a SYNSEM node $n$ and a SYNTAX node $m$ is given in eqs. 5.9 and 5.10, where $P_n$ and $\mathcal{S}_n$ refer to the prefix and suffix set of node $n$.

$$\mathrm{ref}(n,m) = \exists_S (S \in \mathcal{S}_n \land P_n \oplus S = G_m) \tag{5.9}$$

$$\mathrm{ref}^*(n,m) = \exists_S (S \in \mathcal{S}_n \land P_n \oplus S \sqsubseteq G_m) \tag{5.10}$$

To cover the case of FRAME nodes on the left hand side of the operator, the *address-list* field contains the Gorn addresses of all syntax nodes referenced by a role or the target, the algorithms do not need to be changed.

### *Core Sets and Non-local References*

Each node data entry also contains the *flags* entry, which is a bit field. The exact interpretation of the individual bits depends on the node type. For FRAME nodes, the preprocessing stores if any of the core sets from this frame's formal description is satisfied. The implementations of the predicates `has_coreset` and `no_coreset` check if this specific bit in the *flag* field is set when they are evaluated.

For FE nodes, the preprocessing includes a search for non-local references. Any edge with a target node that is from another graph is replaced by a special node ID[6] and for the predicates `has_external` and `no_external`, a bit in the *flags* field is set.

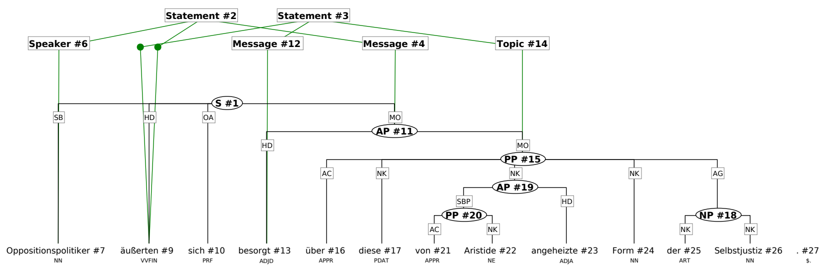
## 5.3 QUERY EVALUATION EXAMPLE

In this section, we will demonstrate the different stages of query evaluation by means of the query found in section 4.4, repeated in example 5.11. This query will be evaluated on the graph from fig. 5.2, which contains a single match.

(5.11)
```
{frame="Statement"} > #r:{role="Topic"} &
#pp:[cat="PP"] >AC [word="über"] &
#r > #pp & arity(#r, 1)
```

For better reference, all nodes in the example graph are numbered. In the following text, any numeric node variable like `#1` or `#25` refers to the corresponding node in fig. 5.2.

6 While non-local references are not considered in the queries, they still need to be kept in one form or another for the graph rendering.



*Politicians of the opposition expressed their concern about this form of vigilantism encouraged by Aristide.*

Figure 5.2: A graph containing a match of the query in example 5.11

## 5.3.1 *Query Preparation*

In the query preprocessing, all node descriptions in the original query are bound to a variable and separated from the constraints. If no such variable exists yet, a new one is created. The type inferencer (cf. section 5.1.4) determines the most specific type possible for each variable, based on the domains of the features used in the node descriptions. This results in the node descriptions shown in table 5.1.

| VARIABLE | TYPE | NODE QUERY |
|---|---|---|
| `#f` | FRAME | `{frame="Statement"}` |
| `#r` | FE | `{role="Topic"} & arity(#r, 1)`[7] |
| `#pp` | NT | `[cat="PP"]` |
| `#w` | T | `[word="über"]` |

Table 5.1: The node descriptions and variables of query example 5.11

The relation constraints, shown in example 5.12, are disambiguated using the inferred types of their operand variables, which are listed in parentheses following the original operator symbol.

```
(5.12) #f >(FRAME,SYNSEM) #r &
       #pp >(NT,DNODE)AC #w &
       #r >(SYNSEM,SYNTAX) #pp
```

The types in the constraints do not list the effective types of node variables as in table 5.1, but the types from the constraint definitions from section 4.3.2.

## 5.3.2 *Node Queries*

In the next stage, the feature indices are used to find all graph nodes that match the given feature constraints and predicates defined for a node query. When a graph has at least one matching node for each variable, a data structure is created that contains all candidates in this graph and handed over to the constraint checker. The node candidates for the sample graph are given table 5.2.

7 Predicates are handled during node candidate retrieval, cf. section 5.1.5.

| i | VARIABLE | NODES |
|---|---|---|
| 1 | #f | #2, #3 |
| 2 | #r | #14 |
| 3 | #pp | #15, #20 |
| 4 | #w | #16 |

Table 5.2: Results from the graph in fig. 5.2 for the node queries in table 5.1

### 5.3.3 *Result Set Generation*

Based on the node candidates, the relation constraint checker has to find those assignments of actual nodes to variables that satisfy all the relation constraints in the query (cf. example 5.12). With C being the node candidate retrieval function mapping variables $\text{var}_i$ to sets of nodes, there are

$$\prod_{i=1}^{j} |C(\text{var}_i)| = 2 \cdot 1 \cdot 2 \cdot 1 = 4 \tag{5.13}$$

possible assignments of nodes to variables given the results from the previous section. For each of the four assignments, all relation constraints have to be evaluated. The results of this step are summarized in table 5.3, which lists the assignment of nodes to variables and the result for each relation constraint check given this particular assignment. The last column contains the overall results, which is just the conjunction of all previous truth values in the same row. If this value is true, the assignment constitutes a match for the original query.

| | ASSIGNMENT | | | | CONSTRAINT CHECKS | | | MATCH |
|---|---|---|---|---|---|---|---|---|
| | #f | #r | #pp | #w | #f > #r | #r > #pp | #pp >AC #w | |
| 1. | #2 | #14 | #15 | #16 | F | T | T | F |
| 2. | #3 | #14 | #15 | #16 | T | T | T | T |
| 3. | #2 | #14 | #20 | #16 | F | F | F | F |
| 4. | #3 | #14 | #20 | #16 | T | F | F | F |

Table 5.3: Relation constraint check results for all possible variable assignments, based on table 5.2

The individual constraint checks are carried out using the node data entries from the corpus index. The frame membership constraint `#f > #r` uses frame numbers (cf. section 5.2.1), the relation constraints `#r > #pp` and `#pp >AC #w` are both based on Gorn addresses, employing eqs. 5.9 and 5.5 from section 5.2.1 respectively.

The final results in the last column of the table state that of all possible assignments, only no. 2 fulfills all conditions of the query in example 5.11. The PP *"über diese . . . Form der Selbstjustiz"* (#15) is referenced by the role TOPIC (#14) in the frame STATEMENT (#3). This match is reported to the user and presented with an according highlighting of the nodes.

## 5.4 EFFICIENT QUERY EVALUATION

### 5.4.1 *Constraint Checks*

On closer inspection of table 5.3, some of the problems in efficient match finding become evident. Though the sample graph has a comparably small number of possible solutions, several constraints need to be checked more than once. There are only 6 unique relation constraints to be evaluated, shown in 5.4.

| N | CONSTRAINT | OPERANDS | RESULT |
|---|---|---|---|
| 1. | `#f > #r` | `#2, #14` | F |
| 2. | | `#3, #14` | T |
| 3. | `#r > #pp` | `#14, #15` | T |
| 4. | | `#14, #20` | F |
| 5. | `#pp >AC #w` | `#15, #16` | T |
| 6. | | `#20, #16` | F |

Table 5.4: Unique relation constraints based on the nodes in table 5.2

In a naïve implementation, all constraints are evaluated for each assignment. This results in 12 checks, twice as many as needed. While in this case, the difference might be hardly noticeable, it can quickly become a problem. Even a single variable that has many candidates leads to combinatorial explosion of the possible assignments. Therefore, an efficient implementation has to make sure that constraint checks are only done once and results are cached for later checks.

The actual order in which relation constraints are evaluated is irrelevant for the final results, because it is simply a conjunction of truth values. An intelligent ordering of unique relation constraint checks, however, can lower the number of overall checks and improve the speed of result set generation.

In the example at hand, only one of then relation checks 4 and 6 has to be carried out. After either, it is obvious hat binding `#20` to `#pp` cannot possibly satisfy all constraints, independent of the outcome of other constraints involving `#20`. Any assignment containing it can be disregarded completely, which reduces the number of required unique constraint checks further from 6 to 5. This is largely a heuristic improvement and it is up to the query optimizer to find a good ordering.

### 5.4.2 *Node Ordering*

Since annotation graphs, with or without frame semantic annotation, are directed and acyclic, it is possible to order the nodes using topological sort. A topological ordering of the nodes $V_G$ in a graph $G$ is a permutation $p$ of all nodes in which any node $j$ is preceded by all nodes $i$ that it can be reached from by a directed path using the edges $E_G$ of $G$ (Weissstein[8]).

Internally, each node $i$ of an annotation graph is represented by an integer $n_i \in \mathbb{N}$, the *node ID*. In the XML corpus file, each node

8 Topological Sort: `http://mathworld.wolfram.com/TopologicalSort.html`

has a unique *textual* identifier $t_i \in \Sigma^*$, with $\Sigma$ being the set of valid characters for XML name tokens[9]. The bijective mapping function $\tau : \Sigma^* \times G \rightarrow \mathbb{N}$ assigns the integers $n_i$ based on a topological ordering, using the equivalence

$$p_x = i \Leftrightarrow n_i = x \tag{5.14}$$

that is, the element $p_o = i$ receives the number $n_i = 0$, until $n_j = m$ for node $p_m = j$.

In the original annotation graph G, edges are typed with the relation they represent, just as individual nodes carry a type. For the topological sort, this type information is not needed, instead we want to consider some derived relations when creating the topological ordering. Therefore, a new augmented graph $G'$ is created from on G, with $V_{G'} \equiv V_G$, discarding all type information. The edges $E_{G'}$ are created according to the edge insertion rule 1.

**Edge Insertion Rule 1** *For each pair of nodes* $i, j \in V_G$ *for which any of the following relation constraint holds, add a directed edge* $(i, j)$ *to* $E_{G'}$*:*

- NT > DNODE
- T . T
- SYNSEM > SYNTAX
- FRAME > SYNSEM
- NT ^ SEMANTICS

The algorithm for topological sorting is well-known and can be found in standard algorithms textbooks like Cormen et al. (2001).

Given the algorithm and a graph created by rule 1, example 5.15 contains one valid topological ordering of the nodes #15, #16, #20 and #21 from the graph in fig. 5.2. #16, #20 and #21 are reachable from #15 by (transitive) dominance and #21 is reachable from #16 by transitive precedence.

(5.15) #15, #20, #16, #21

With node IDs being assigned in topological order, it is possible to avoid checking certain relation constraints in the first place. For all the node relations that were used to create $E_{G'}$ in rule 1, no check needs to be performed if the number $n_i$ of the node $i$ on the left-hand side of the operator is greater than $n_j$ of the node $j$ on the right-hand side, since it will only fail trivially. If no other node $k$ such that $n_i < n_k$ exists, $i$ can *never* be a part of a valid assignment and does need not be considered as a candidate for variable binding at all, reducing the number of possible combinations early on.

For the example at hand however, the topological sort from example 5.15 does not offer any benefit. $n_i < n_j$ is only a necessary condition on two nodes $i, j$ for all of the relations listed above, but not a sufficient one—the dominance constraint between #20 and #16 still needs to be checked. In order to make the topological sort also catch cases like this, we add a second rule:

**Edge Insertion Rule 2** *For all nodes* $i, j, k \in G$, *if the terminal* $i$ *immediately precedes the leftmost terminal successor* $k$ *of a nonterminal* $j$, *add the edge* $(i, j)$ *to* $E_{G'}$.

9 NMTOKEN: http://www.w3.org/TR/REC-xml/#NT-Nmtoken

Together, rules 1 and 2 produce the desired topological sort in example 5.16. `#20` can be reached from `#16` via `#17` and `#19`.

With this ordering, we never need to consider `#20` as a valid candidate for `#pp`. A dominance constraint with `#20` on the left and `#16` on the right can never be true since `#16` has a lower topological order and hence a lower ID. It is also the only candidate for `#w`, therefore an assignment with `#20` cannot fulfill all relation constraints. This eliminates combinations 3 and 4 from table 5.3, further reducing the number of unique constraint checks from 5 to 4.

(5.16) `#15, #16, #20, #21`

Every relation constraint contains information about constraints on the topological ordering of its operands. If such constraints exist, they can be applied during the node candidate retrieval, since they only depend on the node ID. At the point when constraint checks are to be carried out, a lot of candidates that only result in trivial failures are already removed.

Care has to be taken when defining these topological constraints, especially in the presence of modifiers. Negation of relation constraints usually prevents their application, for the same reason why dominance constraints still need to be checked when topological constraints are fulfilled. In the case of the negated transitive dominance '`!>*`', a node $i$ on the left, $j$ on the right-hand side with $n_i > n_j$ is sufficient for the constraint to be true, but not necessary.

## *Efficiency Gains*

The improvement achieved by enforcing topological constraints greatly depends on the relation constraints used in a query. One advantage, however, is that they follow the *"You don't pay for what you don't use"*-principle—if topological constraints exist, they are applied; if not, then apart from an additional comparison operation to check for their presence, no additional work needs to be done. This comes at the price of slightly increasing the time needed for corpus preprocessing, because the augmented graph $G'$ has to be created and its nodes sorted. Since corpus preprocessing is done only once, this is not a problem, as was argued earlier.

In case they do apply, the reduction in relation constraint checks varies greatly. To get an impression of the possible savings, we took queries 1–4 from Lai and Bird (2004)[10] and executed them on the TIGER corpus twice, without and with enforcing constraints on topological ordering of node operands. The test queries are:

Q1 `[cat="S"] >* [word="sah"]`
*Find all sentences with the word "sah".*

Q2 `[cat="S"] !>* %w:[word="sah"]`
*Find all sentences that do not contain the word "sah".*

Q3 `[cat="NP"] >@r [pos="NN"]`
*Find all NPs that have a noun as their rightmost terminal successor.*

Q4
```
#vp:[cat="VP"] >* #v:[pos=/V.+/] &
#vp >* #np:[cat="NP"] & #vp >* #pp:[cat="PP"] &
#v . #np & #np >@r #nr & #nr . #pp
```
*Find all VPs that contain a verb, an NP and a PP immediately next to each other in that order.*

| | | without TS | | with TS | |
|---|---|---|---|---|---|
| QUERY | MATCHES | NODES | CHECKS | NODES | CHECKS |
| Q1 | 62 | 160 | 109 | 129 | 78 |
| Q2 | 72,328 | 72,441 | 109 | 72,441 | 109 |
| Q3 | 79,673 | 273,393 | 432,760 | 256,590 | 394,246 |
| Q4 | 41 | 168,093 | 253,375 | 44,230 | 65,407 |

Table 5.5: Effect of constraints on the topological ordering of nodes

The results are shown in table 5.5. The column under NODES lists the numbers of distinct nodes bound to a variable at least once during evaluation, the column CHECKS shows the number of relation constraints that were evaluated.

The huge effect on Q4 is largely due to the occurrence of the unconstrained node variable `#nr`. Initially, all terminals from a graph match this description, but using topological constraints, the number of candidates can be reduced significantly. While the savings observed for Q4 are much better than average, Q3 represents the amount of reduction that can generally be expected.

## 5.5 BENCHMARKS

Running a meaningful benchmark for our system is difficult, since there is no other system with the same or at least comparable set of features readily available. To provide a basis for comparison, we benchmarked both TIGERSearch 2.1.1 and the current development version of our code using queries Q1, Q3 and Q4 from the previous section.

The test were done on a Intel Core 2 Duo T7500, 2.2 GHz with 2 GiB RAM, running on Linux-x86 with kernel 2.6.28. TIGERSearch was executed with the newest available version of Java, OpenJDK 6b14[11], using the extended corpus indexing features. Our code was executed on Python 2.5.4, the modules written in C99[12] compiled with GCC 4.3.3. In order to keep the numbers comparable, we did not use the option of evaluating queries on several CPUs in parallel, which is available in our system. The resulting times are shown in table 5.6a.

To demonstrate the evaluation time when frame semantic annotation is involved we benchmarked our query module by evaluating five queries that contain different aspects of the extended query language on the SALSA corpus, the results are shown in table 5.6b.

S1 `{FE} >* #h:[word="Manager"]`
*Find all roles that contain the word "Manager".*

S2
```
#f1:{FRAME} != #f2:{FRAME} &
#f1 > #fe1:{FEE} & #f2 > #fe2:{FEE}
& #fe1 > #x & #fe2 > #x & #f1 >* #u & #f2 >* #u:[NT]
```

10 The TIGER queries given in the paper are wrong and were rewritten.

11 TIGERSearch itself comes bundled and insists on being run with an outdated Java 1.4 VM—a misguided restriction which we removed in order to allow it to benefit from the improvements found in newer JVM versions.

12 These modules are optional, but increase the speed of core algorithms.

| QUERY | ORIGINAL | NEW |
|---|---|---|
| Q1 | 0.4 | 0.3 |
| Q3 | 12.4 | 9.9 |
| Q4 | 10.8 | 5.7 |

(a) Comparison of query run times in TIGERSearch (original) and our work (new)

| QUERY | TIME | MATCHES |
|---|---|---|
| S1 | 0.06 | 33 |
| S2 | 11.2 | 7318 |
| S3 | 0.3 | 779 |
| S4 | 0.3 | 27 |
| S5 | 1.6 | 20 |

(b) Time needed for evaluating queries in frame semantic annotation

Table 5.6: Results of the query benchmarks
All times given in seconds, average of three runs

*Find words or syntactic categories which are the target of different semantic frames or which have more than one role, each role belonging to a different frame.*[13]

S3 `{frame=[Intentionally_act]} > {role=[Agent]}`
*Find all roles that inherit from* AGENT *as elements of frames that inherit from* INTENTIONALLY_ACT.

S4 `{frame="Statement"} > #r:{role="Topic"} &`
`#pp:[cat="PP"] >AC [word="über"] & #r > #pp`
`& arity(#r, 1)`
*Find all sentences where the role* TOPIC *in the frame* STATEMENT *is realized by a PP with the preposition "über".*

S5 `{FRAME} > #t:{FEE} & #t > [pos="ADJA"]`
*Find all frames which are evoked by an adjective.*

The benchmark results in table 5.6 are not final, because the query evaluation algorithm is still developed and improved, especially with regards to better parallelization. All tables show the amount of time needed to obtain the complete result set. As mentioned earlier, response times can be improved by returning the first match as soon as possible.

## 5.6 FURTHER READING

A detailed overview for the preprocessing of syntactic annotation is given in Lezius (2002a) and Mettler (2007), along with implementations for all syntactic constraints. Lezius also describes query normalization, which involves conversion into Disjunctive Normal Form, a transformation of boolean expressions such that disjunction only occurs that the top level.

In the explanation of the query evaluation process, we only covered existentially quantified variables. Universal quantification introduces some substantial changes, especially since variables bound by a universal quantifier may not appear in a graph that is a match. A description of the modifications is given in Marek et al. (2008).

13 TIGER version of example 2.3.

# 6 SUMMARY & OUTLOOK

In this chapter, we summarize the contribution developed in this thesis and discuss possible directions of future research, both for greater expressivity in query languages and improved efficiency by exploring new techniques for indexing and parallelization.

## 6.1 SUMMARY

This thesis describes the design and implementation of an extension for the TIGER query language that allows searching over frame semantic and syntactic annotation, while the original formalism only supports queries over syntax. The original motivation was that queries like the one shown in example 6.1 were not expressible until now.

(6.1) *Find all sentences where the role* TOPIC *in the frame* STATEMENT *is realized by a PP with the preposition "über".*

Some experimental studies, using the SALSA corpus, on queries combining syntactic with semantic relation constraints were carried out by Heid et al. (2004), but no further work had been done to this date, making it impossible to use queries for linguistic exploration of this corpus.

In this work, we define several new node types and relations to express queries on sentences with frame semantic annotation. Together with the query language elements of Lezius (2002a), it is possible to write TIGER queries that formalize the structures described in example 6.1.

A TIGER query is created from typed descriptions for individual nodes (words, phrase and now frames, roles and targets as well) and relation constraints between these nodes (dominance and precedence in syntactic annotation). Node descriptions for syntax nodes are surrounded by square brackets (`[]`). In order to make descriptions of semantic nodes visually distinct, we decided to enclose them in curly braces (`{}`). This makes sure that queries that mix both kinds of nodes are easily understandable simply by looking at nodes, without requiring authors to remember feature names for the different node types.

Frame semantic annotation defines a second structural layer *on top of* the syntactic structure. Frames can contain arbitrarily many roles and have exactly one target. Each role and target can reference any number of nodes from the syntax and behave in the same way, since they serve as a connection layer between the frame and its semantic material. Because of this, the types of roles (FE) and targets (FEE) inherit from a common base type SYNSEM. The important new node relations are:

- a SYNSEM is member of a frame
- a syntactic node is referenced by a SYNSEM

In order to limit the size of the extended query language, both new relations use the dominance operator >.

In contrast to syntax nodes, whose features can be freely defined in TIGER corpora, the features of the newly introduced node types for frame semantics are fixed by the annotation data format. We define the features for all new node types, some of which are taken from the formal frame descriptions distributed together with the SALSA corpus.

All new language elements can be used in conjunction with the existing elements and the new relations provide a means to connect semantic and syntactic queries. Example 6.2 shows the formalization of the query from example 6.1.

(6.2)
```
#pp:[cat="PP"] >AC [word="über"] &
#f:{frame="Statement"} & #r:{role="Topic"} &
#f > #r &
#r > #pp & arity(#r, 1)
```

This query uses the new node description syntax for frame semantic nodes when querying for STATEMENT frames and TOPIC roles (line 2), the new basic relations for membership of roles in frames (line 3) and for references from roles to syntactic material (line 4). Existing predicates like `arity` are also valid for the new node types (line 4).

### 6.1.1 *Minor Query Language Features*

Frames are not defined in isolation; so called frame-to-frame relations can be used to define a complex network of frames. The most important relation is inheritance between frames (COMMERCE_PAY is a kind of GIVING). It is important to note that these relations are defined on abstract frames, not on concrete frame instances and thus cannot be expressed using node relation operators like > for dominance. We introduce a new literal in the feature constraints, which matches features according to an externally defined hierarchy.

The annotation format used by the SALSA corpus also allows for the underspecification of frames or roles, which can be interpreted as either a conjunction or disjunction. This annotation is represented using a new basic node relation and can be used in queries, too.

### 6.1.2 *Implementation*

All new language features have been added to our implementation of the TIGER query language, which was written within the TreeAligner project (Volk et al., 2007). Our code uses several novel approaches in query evaluation, like parallel computation and new kinds of preprocessing not found in TIGERSearch by Lezius (2002a). It also contains limited support for universal quantification introduced in Marek et al. (2008), which makes it both more expressive and, in many cases, faster than the original implementation while still having a comparably small code base.

## 6.2 FUTURE WORK

The query module provides a good basis for further extension and research in exploratory corpora analysis, linguistic annotation and also more advanced topics like graph indexing and concurrent programming.

### 6.2.1 *New Kinds of Annotation*

In the scope of this work, it was not possible to properly model non-local references, i.e. relations between nodes contained in different graphs (cf. section 4.2.4). Further work should be done to extend the object model and remove the graph locality restriction, without sacrificing efficiency.

With generalized non-local references, it is also possible to annotate and query anaphoric relations. Moreover, graphs (sentences) themselves should be first-class members of the object model, to allow annotation on top of graphs, like rhetorical structure. Graphs as elements in the query language can also be used to address the problems of no-longer implied node locality in the presence of relation constraints introducing non-local references, cf. section 4.2.4. Several unified object models already exist (often only in the form of annotation storage formats, cf. sections 2.1 and 5.1.1), whose applicability needs to be evaluated.

#### *Multilinguality*

Currently, only syntactic nodes can be aligned in parallel corpora created with the TreeAligner. Padó (2007) projected frame semantic annotation by using parallel corpora, based on automatic phrase alignment. Further work should include alignment of frame semantic structures as well as syntactic ones. An extension of the alignment constraints from Mettler (2007) should also cover these alignments.

### 6.2.2 *Quality Assurance*

Our implementation already comes with two query test suites, cf. section A.4. However, to ensure correct and fast evaluation of all possible queries, we need to extend our test suites and make sure to cover all language features. Availability of a large body of tests also helps further development, since other interested researchers can adopt our code, modify it and use our test cases to ensure that no regressions have been introduced.

### 6.2.3 *Efficiency & Scalability*

As shown in section 5.5, our implementation is already faster than TIGERSearch in some cases. In other cases, TIGERSearch needs considerably less time for query evaluation. This is explained by the SQL database we use as the corpus index. While using a database made the implementation of the corpus index much easier, a full SQL engine has many features our system does not need and also introduces an additional programming language.

#### *Custom Indices*

Some of our benchmarks have shown that in simple queries, as much as 75% of the time needed is spent in database code which is out of our control. Because of our very specific usage pattern (no updates, no aggregate queries, no function calls), we expect that a custom-written index will yield considerable gains in query evaluation speed.

### *Scalability*

The time needed for evaluation of queries on corpora of the size of TIGER or SALSA is already acceptable, ranging from a few milliseconds to almost one minute. However, if we want to be able to handle corpora 10× or 100× as large, we have to improve efficiency. Query evaluation is linear with regard to corpus size, we need real algorithmic improvements over the current state and not just marginally faster implementations of current algorithms.

### *Better Concurrency*

The usage of parallel processing in our implementation is still in a very early state. While for complex queries, parallelism does lead to a decrease of query evaluation time, for simple queries, its overhead is still higher than its savings.

In future work, we would like to explore new kinds of parallelism, both Python-specific (e.g. stackless PyPy[1]) and general (e.g. CUDA[2]).

### *Advanced Indexing Techniques*

The corpus index is mostly based on features, structural information is represented only by topographically sorted nodes (cf. section 5.4.2). In order to create better indices, we want to investigate the usefulness of graph indexing techniques like Yan et al. (2004) developed in the bioinformatics research community. DDDquery (cf. section 2.2.4), for example, was implemented within the "Research Network Linguistics – Bioinformatics"[3] at HU Berlin. Computational Linguistics and Bioinformatics often face similar problems and we are convinced that both sides can benefit from more cooperation and exchange of ideas.

1 `http://codespeak.net/pypy/dist/pypy/doc/stackless.html`
2 `http://www.nvidia.de/object/cuda_home_de.html`
3 `http://www.linguistik.hu-berlin.de/institut/professuren/korpuslinguistik/forschung/forschungsverbund_ling_bioinf`

# A GUIDE TO THE IMPLEMENTATION A

In this appendix, we will give a brief overview of the source code which this thesis is based on. We will explain which classes and modules are involved in the different stages of query evaluation and give technical background information on how to start using the query evaluation module.

## A.1 INTRODUCTION

As was already mentioned in section 1.4.2, our implementation of the TIGER corpus query language has originally been written searching parallel treebanks in the TreeAligner and still is distributed as a part of it. The original work is described in Mettler (2007). Since then, the code has been redesigned and rewritten from scratch, with a special focus on extensibility and rigorous testing.

The code is written in Python[1], a general-purpose high-level programming language. Among its key strengths are the strong, dynamic type system, support for multiple programming paradigms and the large standard library. Python is supported on all major desktop computing platforms and Python code can be ported between different systems with little or no effort.

### *Dependencies*

The minimum Python version needed is 2.5, it should be possible to run the code unchanged on any later 2.x version. The external dependencies for the query module are:

- **PyParsing**
  A library that allows the definition of recursive-descent parsers directly in Python code
  URL: `http://pyparsing.wikispaces.com/`

- **setuptools**
  An extension for Python's integrated build system with better support for package data files
  URL: `http://peak.telecommunity.com/DevCenter/setuptools`

Some dependencies are optional, since they speed up certain parts of the evaluation but do not add any functionality:

- **multiprocessing**
  A library with better support for parallel computing in Python
  URL: `http://code.google.com/p/python-multiprocessing/`

- **lxml**
  A library implementing the standard *ElementTree* interface[2] for handling XML documents, but with a validating parser
  URL: `http://codespeak.net/lxml/`

1 `http://www.python.org`
2 `http://docs.python.org/library/xml.etree.elementtree.html`

- **pysqlite2**
  Updated and improved versions of the `sqlite3` module included in Python 2.5, a module for accessing the embedded relational database SQLite
  URL: `http://pysqlite.org/`

*Graphical User Interfaces*

The only graphical user interface for the query evaluator so far is the TreeAligner, which only supports queries on parallel treebanks. This user interface is written using PyGTK2[3]. A web interface for corpus query evaluation is currently in development and available as a prototype.

*Source Code Access*

The version of the code described in this chapter is available as a download[4]. Barring fixes for critical bugs and compatibility updates, the code in this archive will always reflect the state of the TIGER implementation as described in this thesis.

## A.2 PACKAGE LAYOUT

The complete source code is stored under the directory `STA`, which is a Python package[5]. This package contains six major subpackages:

- *Alignment*: `STA.align`
  Code for handling parallel corpora with syntactic alignments
- *TreeAligner application*: `STA.app`
  Graphical user interface classes
- *Graph rendering*: `STA.drawing`
  A class for device-independent rendering of TIGER graphs
- *Frame database*: `STA.framenet`
  Loading of FrameNet distribution files
- *TIGER*: `STA.tiger`
  TIGER corpus parser, corpus index creation and access, annotation graph data structure and query evaluation
- *Utilities*: `STA.utils`
  Mixed utility functions and classes.

Some statistics on the code in the package `STA.tiger`, which is the only one we are concerned with in this appendix, are shown in table A.1.

### A.2.1 *API Documentation*

The parts of the code whose interfaces are already stabilized have inline documentation, but many core algorithms are still subject to frequent changes and therefore only have source code comments. Stand-

3 `http://www.pygtk.org/`
4 `http://diotavelli.net/files/msc/code.tar.gz`
5 Packages are simply file system directories with Python modules.

| TYPE | N | % |
|---|---|---|
| Code | 4,239 | 64.12 |
| Documentation | 1,769 | 26.76 |
| Comments | 157 | 2.37 |
| Empty | 446 | 6.75 |
| Total | 6,611 | 100.00 |

(a) Total lines of code

| TYPE | NUMBER |
|---|---|
| Modules | 28 |
| Classes | 158 |
| Methods | 514 |
| Functions | 90 |

(b) Structural elements

Table A.1: Code statistics

alone versions of the documentation in different formats can be created with Sphinx[6]. When it is installed, the command

```
$ python setup.py build_sphinx
```

can be used to create HTML files. The output will be written to the directory `build/sphinx/html`.

## A.3 QUERY EVALUATOR ARCHITECTURE

The architecture of the query evaluator implementation closely follows the one outlined in section 5.1.2:

1. Query parsing
2. Query analysis
3. Node candidate retrieval
4. Relation constraint checking, result generation

In the following sections, we will explain which modules and classes each step uses and which data structures are exchanged between the different classes. All code is part of the `STA.tiger.query` package, accessible in the directory `STA/tiger/query`.

Using the query evaluator outside the context of the TreeAligner is trivial and example code for opening corpora, with or without frame semantic annotation, is given in `tools/treebanks.py`.

### A.3.1 *The High Level Interface*

MODULES:

- `STA.tiger.query.evaluator`

`TigerQueryEvaluator` is a façade class that encapsulates the process of evaluating a query. Together with the result builder, it provides the only external interface of the query module. A query string can be evaluated using the method `TigerQueryEvaluator.`**`evaluate`**.

The most important functionality of the this class is to provide the *evaluator context*, which is the same for all queries on the same corpus. The evaluator context bundles access to the corpus index and metadata and also is used to decouple the query analysis from evaluation. This technique is called Inversion of Control (IoC).

6 http://sphinx.pocoo.org/

### A.3.2 *The Query Parser*

MODULES:

- STA.tiger.query.ast
- STA.tiger.query.parser

The parser for the TIGER query language is defined in Python code, without having to rely on external syntax definitions and parser generators. Instead, the grammar and parse actions are written directly in the code by combining various basic parser building blocks. Each parsing rule has an associated parsing action, which is used to build the abstract syntax tree.

Each element of the query language is represented by an AST node class in the module STA.tiger.query.ast. The classes representing AST nodes are organized in a hierarchy with the common abstract base class _Node[7]. AST nodes are created directly in grammar rules:

```
def regex_literal():
    regex = pyparsing.QuotedString("/")
    return regex.setParseAction(
       lambda s, l, token: ast.RegexLiteral(token[0]))
```

In this rule, a string with / as the quote symbol is defined which is supposed to be put into a RegexLiteral AST node. A complete syntax tree for the query in example A.1 is shown in listing A.1.

(A.1) {role="Topic"} > [cat="PP"]

The root of a query AST is always a TigerQueryExpression node. This node always has a single child, which can be any toplevel element of a TIGER query. Usually, this is a conjunction, but in example A.1, the only toplevel element of the query is a dominance relation constraint, represented by the DominanceOperator on l. 2 of listing A.1. The operator AST node has two children, its left and right operands. In this case, both are node descriptions, which will be replaced by node variables during the preprocessing. The arguments on ll. 11–13 are the operator modifiers, which are simply initialized to their default values in this case.

As already mentioned in section 4.2.2, the different brace types surrounding node descriptions are represented as upper type boundaries in NodeDescription AST elements. Curly braces for frame semantics elements are mapped to NodeType.SEMANTICS (l. 6), square brackets for syntax nodes to NodeType.SYNTAX (l. 10).

```
1  TigerQueryExpression(
     DominanceOperator(
       NodeDescription(
         FeatureConstraint(
           'role', StringLiteral(u'Topic')),
6        NodeType.SEMANTICS),
       NodeDescription(
         FeatureConstraint(
           'cat', StringLiteral(u'PP')),
         NodeType.SYNTAX),
```

7 In Python, using leading underscores is a convention to signal that a class or method is "private" and not for external use.

```
    op_range=(1, 1),
    negated=False,
    label=None))
```

Listing A.1: Abstract Syntax Tree of the query in example A.1

### A.3.3 *Query Analysis*

MODULES:

- `STA.tiger.query.ast_utils`
- `STA.tiger.query.ast_visitor`
- `STA.tiger.query.factory`
- `STA.tiger.query.nodes`

The main class for analyzing query ASTs and converting them to query objects is `STA.tiger.query.factory.`**`QueryFactory`**. Its method `from_ast` takes a complete parse tree like the one from listing A.1 and extracts all node descriptions from it (cf. section 5.3.1). For each node description, an instance of `STA.tiger.query.nodes.`**`NodeQuery`**, which bundles together the variable, description and the type of a node is created.

The enumeration class `STA.tiger.graph.`**`NodeType`** contains the hierarchy of valid node types as shown in fig. 4.4 on page 26. Initially, the type of a variable is set to the upper boundary from the AST nodes, the effective type is determined by the query factory. To determine the type of a single node description, it uses the class `NodeTypeInferencer` from the same module, which combines feature domains and type constraints into a single node type, or fails with an error if that is not possible.

For each abstract predicate and relation operator AST node like `DominanceOperator` in listing A.1 one line 2, the concrete implementations are instantiated using the factory classes

- `STA.tiger.query.predicates.`**`PredicateFactory`**
- `STA.tiger.query.constraints.`**`ConstraintFactory`**

Each factory class contains a mapping from AST node classes or predicate names to concrete implementation classes, which contain the code for query evaluation. This additional layer of indirection decouples representation and preprocessing of queries from the actual evaluation in later stages. This technique limits the extent of changes incurred by adding new features. In the case of new predicates, the query language grammar and the AST nodes do not have to be changed at all. New predicates can be registered in the predicate factory and will be instantiated by the query factory.

The query factory returns a result builder instance, using the evaluator context method `create_result_builder`. The concrete implementation depends on the feasibility of parallel computing and the presence of relation constraints or top-level disjunctions.

### A.3.4 *Node Candidate Searching*

MODULES:

- `STA.tiger.query.nodesearcher`

- `STA.tiger.query.predicates`

The evaluator context is also used to retrieve the instance of the node searcher class `STA.tiger.query.nodesearcher.`**`NodeSearcher`** for a corpus index. All feature constraints of a node query are compiled into a single SQL query and evaluated on the corpus index. The node searcher also handles non-string literals like regular expressions or types and finds the matching features for those constraints.

The implementations of all predicates are contained in the module `STA.tiger.query.predicates`. Each one contributes a small snippet to the final SQL query statement, which usually defines an additional condition on the node data entry (cf. section 5.1.1).

The core class of the node candidate retrieval joins the result sets all node queries, `STA.tiger.query.nodesearcher.`**`GraphIterator`**. This class the single node candidates, groups them by graphs and applies any topological constraints on the node IDs (cf. 5.4.2). The algorithm is conceptually simple but very important for the time required for evaluating a query. Because of that, it has been reimplemented in a C extension module. This extension module is optional, but makes some queries up to 30% faster[8].

### A.3.5 *Relation Constraint Checking*

MODULES:

- `STA.tiger.query.constraints`
- `STA.tiger.query.result`

The module `STA.tiger.query.result` contains several different implementations of classes for result set generation:

- **`SimpleBuilder`**
  Used for queries without any constraints, simply enumerates all possible variable assignments.

- **`ResultBuilder`**
  The default result builder.

- **`ParallelResultBuilder`**
  The result builder used for parallel query evaluation, which splits the corpus in $n$ parts and runs $n$ worker processes.

- **`DisjunctionResultBuilder`**
  The builder used when toplevel disjunctions are present, which are evaluated as separate queries.

- **`ParallelDisjuctionResultBuilder`**
  In the presence of toplevel disjunctions, the worker processes are not run on corpus index parts, instead each disjunct is run on the whole corpus in its own worker process.

All builders that need to perform relation constraint checks use the class `STA.tiger.query.result.`**`ConstraintChecker`**. This class contains the core algorithm for efficient constraint checking as sketched in section 5.4.1, implemented in the following methods:

8 When using a single processor for evaluation, the gains for parallel processing are a little lower.

- `ConstraintChecker.`**`_filter_nodes`**:
  Evaluates all constraints on the node candidates, successively filters out nodes which cannot satisfy all constraints and stores the node pairs which satisfy the constraints.

- `ConstraintChecker.`**`extract`**:
  Based on the remaining node candidates and the pairs of valid combinations, creates all valid variable assignments.

The result builder also provides the *query context*. In contrast to the evaluator context, whose life cycle is bound to the corpus itself, a query context exists only as long as the result builder. Its role is comparable, since it provides access to the corpus index using IoC.

The classes for evaluating the different relation constraints are defined in the module `STA.tiger.query.constraints`. They all derive from the common base class `Constraint` and define a number of properties:

- *Predicates*
  Each relation constraint can introduce more predicates for its operands. For example, labeled dominance adds a predicate for the label, which is stored in the node data entry. The predicates are used by the node searcher.

- *Topological Constraint*
  Defines a constraint (cf. 5.4.2) on the topological order of the the two operand nodes $l, r$; one of $(n_l < n_r, n_r < n_l, \textit{undef})$. Topological constraints are applied by the node searcher.

- *Data Fields*
  Lists the names of the fields from the node data entries needed for evaluation, used for minimizing the amount of data read from the database.

- *Match Limits*
  Some constraints can have only a single match if one of the operands is fixed. In immediate dominance for instance, for one right operand (child node) only one node, if at all, can make the constraint true, because a node only has one parent. The other direction does not hold, since one parent node can have any number of immediate children.

| E | EXAMPLE QUERY |
|---|---|
| $1:X$ | `[NT] > [DNODE]` |
| $1:1$ | `[T] . [T]` |
| $X:1$ | `[NT] >@l [T]` |
| $X:X$ | `[NT] >* [DNODE]` |

Table A.2: Match limits between operands of different constraints

Table A.2 shows the four different types of match limits along with example relation constraints. A match limit expression is of the form $a : b$, with $a, b \in \{1, X\}$. If by replacing either $a$ or $b$ (but not both) by 1, the match limit expression of a constraint $R$ becomes $1 : 1$, a match limit can be applied.

If $a$ was replaced, then as soon as there is a pair of nodes $(i, j)$ that satisfies $R$, no other pair $(i, k)$ can be a solution and $i$ does not have to be considered for further tests, vice versa if $b$ was replaced.

In the case of the expression being $1 : 1$ *without* replacement, it is even possible to disregard both nodes for further tests, but support for this is not yet implemented in the relation constraint checker.

The builder `STA.tiger.query.result.`**`ParallelResultBuilder`** uses the `multiprocessing` module to evaluate queries on several graphs in parallel. Due to technical restrictions in the standard Python implementation, only one thread can execute Python code at the same time[9]. `multiprocessing` supports parallelization by creating several Python processes, but provides an interface similar to Python threads. The result builder in this case creates as many child Python processes as available processors, each of them working on a part of the corpus. Since individual graphs are independent of each other, the parallel result builder just takes the output of each child process and returns the concatenation of all result sets.

## A.4 QUALITY ASSURANCE

While developing the query evaluator, we have put specific emphasis on the availability of extensive test suites to ensure high code quality and simplify maintenance. The distribution contains two types of automated test suites that cover the full feature set of the implementation.

### A.4.1 *Unit Tests*

The low-level unit tests are white-box tests which are used for testing each component (at a sensible granularity) in isolation. The tests use *nose*[10], an enhanced unit testing framework, and pmock[11], a mock object library. The unit tests can be executed with

```
$ python setup.py test --quiet
...
...
...
Ran 506 tests in 5.665s

OK
```

Unit tests ensure that all code units function in isolation, according to their intent and documentation. The ability to test classes on their own requires loose coupling between the individual parts of the code, which further facilitates good design.

### A.4.2 *Integration Tests*

The high-level integration tests are end-to-end black-box tests that have no knowledge about implementation details or which code units are

9 Global Interpreter Lock, cf. the CPython documentation.
10 `http://somethingaboutorange.com/mrl/projects/nose/`
11 `http://pmock.sourceforge.net`

involved in running the tests. The integration tests for the query module open a corpus, evaluate a query and check if the result sets contains the expected number of graphs and matches.

For the integration tests to run, the SALSA and TIGER corpora must be present. The paths to the corpus files must be set in `corpora.cfg`, which can be created using the template `misc/corpora.cfg.in`. If the paths are correct, the test suites can be executed using the `query_tool` script, which also automatically creates corpus indices:

```
$ ./query_tool test all
Running test suite 'semantics'
...

Running test suite 'basic'
...

Tests: 38
Passed: 38
Failed: 0

Time: 45.46s
```

The advantage of integration tests is that problems in the interaction between different parts of the code can be found, or problems which only occur under very rare circumstances. The disadvantage is that in case of a failure, the part of code that actually caused the failure is unknown and has to be determined by the means of debugging, tracing or simply reading the code.

When a bug is found during integration testing, usually a unit test is written that triggers the bug before fixing it in the code. This not only simplifies testing the bug fix, but also makes sure that no future changes introduce any regressions for this particular fix.

# B QUICK REFERENCE

## Node Types for Frame Semantics and their Relations

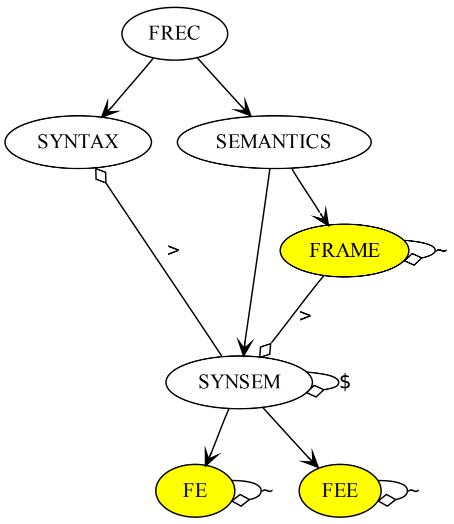


Figure B.1: Node types and relations.
Leaf nodes are filled, node relation are shown as edges labeled with the operator symbol and a diamond arrow pointing at the right operand.

## Features

| TYPE | FEATURE | DESCRIPTION | Hierarchy |
|---|---|---|---|
| SEMANTICS | *flag* | Linguistic phenomena | |
| | *semtype* | Ontological type | ✓ |
| FRAME | *frame* | Frame name | ✓ |
| FE | *role* | Role name | ✓ |
| | *coretype* | Importance of role | |
| FEE | *lemma* | Lemma of lexical unit | |
| | *head* | Head of the lemma phrase | |
| PART | *part* | Word part | |

## Relation Constraints

| DEFINITION | DESCRIPTION | EXAMPLE |
|---|---|---|
| FRAME > SYNSEM | Element in frame | `{frame="Cure"} > {role="Healer"}` |
| SYNSEM $ SYNSEM | Elements in same frame | `{role="Goods"} $ {role="Buyer"}` |
| SYNSEM > SYNTAX | Syntactic material | `{role="Context"} > [cat="S"]` |
| FRAME >* SYNTAX | Syntactic material | `{frame="Telling"} >* [word="Radio"]` |
| NT ^ SEMANTICS | First common ancestor | `[cat="NP"] ^ {FRAME}` |
| FRAME ~ FRAME | Underspecified frames | `{frame="Statement"} ~ {FRAME}` |
| FE ~ FE | Underspecified roles | `{role="Speaker"} ~ {role="Medium"}` |
| FEE ~ FEE | Underspecified targets | `{FEE} ~ {FEE}` |
| PART < T | Part of | `[PART] < [pos!="NN"]` |

## Predicates

| PREDICATE | DESCRIPTION |
|---|---|
| `spec`(SEMANTICS) | Node is not underspecified |
| `uspec`(SEMANTICS) | Node is part of an underspecification block |
| `has_external`(FE) | Non-local references on role |
| `no_external`(FE) | No non-local references on role |
| `has_coreset`(FRAME) | At least one core set of frame complete |
| `no_coreset`(FRAME) | Frame has no complete core sets |

# BIBLIOGRAPHY

Baker, Collin F., Charles J. Fillmore and John B. Lowe (1998). The Berkeley FrameNet Project. In *Proceedings of COLING-ACL 1998*, pp. 86–90. (Cited on page 2.)

Baumann, Stefan, Caren Brinckmann, Silvia Hansen-Schirra, Geert-Jan M. Kruijff, Ivana Kruijff-Korbayová, Stella Neumann and Elke Teich (2004). Multi-dimensional Annotation of Linguistics Corpora for Investigating Information Structure. In A. Meyers, ed., *Proceedings of the HLT-NAACL 2004 Workshop: Frontiers in Corpus Annotation*, pp. 39–46. Boston, MA, USA. (Cited on page 7.)

Bird, Steven, Yi Chen, Susan B. Davidson, Haejoong Lee and Yifeng Zheng (2005). Extending XPath to Support Linguistic Queries. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, pp. 35–46. Long Beach, California. (Cited on pages 11, 13, and 46.)

Bird, Steven and Haejoong Lee (2006). Designing and Evaluating an XPath Dialect for Linguistic Queries. In *Proceedings of the 22nd International Conference on Data Engineering*. (Cited on pages 13 and 46.)

Blackburn, Patrick, Claire Gardent and Wilfried Meyer-Viol (1993). Talking about Trees. In *Proceedings of the 6th Conference of the European Chapter of the Association for Computational Linguistics*, pp. 21–29. Utrecht, The Netherlands. (Cited on page 13.)

Brants, Sabine, Stefanie Dipper, Silvia Hansen, Wolfgang Lezius and George Smith (2002). The TIGER Treebank. In *Proceedings of the First Workshop on Treebanks and Linguistic Theories, TLT 2002*. Sozopol, Bulgaria. (Cited on pages 3 and 13.)

Brants, Thorsten, Roland Hendriks, Sabine Kramp, Brigitte Krenn, Cordula Preis, Wojciech Skut and Hans Uszkoreit (1997). Das NEGRA-Annotationsschema. Technical report, Universität des Saarlandes, Saarbrücken, Germany. (Cited on page 15.)

Brants, Thorsten, Wojciech Skut and Hans Uszkoreit (1999). Syntactic Annotation of a German Newspaper Corpus. In Anne Abeillé, ed., *ATALA sur le Corpus Annotés pour la Syntaxe Treebanks*, pp. 69–76. Paris, France. (Cited on page 15.)

Burchardt, Aljoscha, Katrin Erk, Anette Frank, Andrea Kowalski, Sebastian Padó and Manfred Pinkal (2006a). The SALSA Corpus: A German Corpus Resource for Lexical Semantics. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation, LREC 2006*. Genoa, Italy. (Cited on pages 2, 3, 7, 21, and 83.)

Burchardt, Aljoscha, Katrin Erk, Annette Frank, Andrea Kowalski and Sebastian Padó (2006b). SALTO – A Versatile Multi-Level Annotation Tool. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation, LREC 2006*. Genoa, Italy. (Cited on page 3.)

Carletta, Jean, Jonathan Kilgour, Tim J. O'Donnell, Stefan Evert and Holger Voorman (2003). The NITE object model library for handling structured linguistic annotation on multimodal data sets. In *Proceedings of the EACL Workshop on Language Technology and the Semantic Web (NLPXML-2003)*. Budapest, Hungary. (Cited on pages 8 and 45.)

Charniak, Eugene (1997). Statistical Parsing with a Context-Free Grammar and Word Statistics. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI '96)*, pp. 598–603. (Cited on page 7.)

Chiarcos, Christian, Julia Ritz and Manfred Stede (2008). Investigating non-canonical constructions in context: Efficient corpus annotation and retrieval. In Angelika Storrer, Alexander Geyken, Alexander Siebert and Kay-Michael Würzner, eds., *Text Resources and Lexical Knowledge, Companion Volume*, pp. 1–8. Berlin, Germany: Mouton de Gruyter. (Cited on pages 9 and 10.)

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest and Clifford Stein (2001). *Introduction to Algorithms*, chapter 22.4: Topological Sort, pp. 549–552. MIT Press and McGraw-Hill, second edition. (Cited on page 57.)

Dipper, Stefanie (2005). XML-based Stand-off Representation and Exploitation of Multi-level Linguistic Annotation. In *Proceedings of Berliner XML-Tage (BXML) 2005*, pp. 39–50. Berlin, Germany. (Cited on pages 10 and 45.)

Dipper, Stefanie, Michael Götze, Uwe Küssner and Manfred Stede (2007). Representing and Querying Standoff XML. In Georg Rehm, Andreas Witt and Lothar Lemnitzer, eds., *Proceedings of the GLDV-Frühjahrstagung*. Tübingen, Germany. (Cited on pages 10 and 11.)

Dipper, Stefanie, Erhard Hinrichs, Thomas Schmidt, Andreas Wagner and Andreas Witt (2006). Sustainability of Linguistic Resources. In Erhard Hinrichs, Nancy Ide, Martha Palmer and James Pustejovsky, eds., *Proceedings of the LREC 2006 Satellite Workshop on Merging and Layering Linguistic Information*. Genoa, Italy. (Cited on page 7.)

Erk, Katrin, Andrea Kowalski, Sebastian Padó and Manfred Pinkal (2003). Towards a Resource for Lexical Semantics: A Large German Corpus with Extensive Semantic Annotation. In *Proceedings of the ACL 2003*, pp. 537–544. (Cited on page 3.)

Erk, Katrin and Sebastian Padó (2004). A powerful and versatile XML format for representing role-semantic annotation. In *Proceedings of the Fourth International Conference on Language Resources and Evaluation, LREC 2004*. Lisbon, Portugal. (Cited on pages 4, 6, and 21.)

Evert, Stefan and Holger Voormann (2003). NQL – A Query Language for Multi-Modal Language Data. Technical report, IMS, University of Stuttgart, Stuttgart, Germany. (Cited on page 8.)

Faulstich, Lukas C. and Ulf Leser (2005). Implementing Linguistic Query Languages Using LoToS. (Cited on page 11.)

Fillmore, Charles J. (1976). Frame Semantics and the Nature of Language. In *Annals of the New York Academy of Sciences: Conference on the Origin and Development of Language and Speech*, volume 280, pp. 20–32. (Cited on page 1.)

Fillmore, Charles J. (1985). Frames and the semantics of understanding. *Quaderni di Semantica*, IV(2):pp. 222–254. (Cited on page 1.)

Gorn, Saul (1967). Explicit Definitions and Linguistic Dominoes. In John F. Hart and Satoru Takasu, eds., *Proceedings of the Systems and Computer Science Conference at University of Western Ontario*, pp. 77–115. Toronto, Canada: University of Toronto Press. (Cited on page 51.)

Götze, Michael and Stefanie Dipper (2006). ANNIS: Complex Multilevel Annotations in a Linguistic Database. In *Proceedings of the 5th Workshop on NLP and XML (NLPXML-2006)*, pp. 61–64. Trento, Italy. (Cited on page 10.)

Heid, Ulrich, Holger Voormann, Jan-Torsten Milde, Ulrike Gut, Katrin Erk and Sebastian Padó (2004). Querying both time-aligned and hierarchical corpora with NXT search. In *Proceedings of the Fourth International Conference on Language Resources and Evaluation, LREC 2004*. Lisbon, Portugal. (Cited on pages 8, 12, and 61.)

Ide, Nancy, Patrice Bonhomme and Laurent Romary (2000). XCES: An XML-based encoding standard for linguistic corpora. In *Proceedings of the Second International Language Resources and Evaluation Conference. Paris: European Language Resources Association*. (Cited on page 45.)

Ide, Nancy and Keith Suderman (2007). GrAF: A Graph-based Format for Linguistic Annotations. In *Proceedings of the Linguistic Annotation Workshop*, pp. 1–8. Prague, Czech Republic: Association for Computational Linguistics. (Cited on page 45.)

Koehn, Philipp (2005). Europarl: A Parallel Corpus for Statistical Machine Translation. In *MT Summit 2005*. (Cited on page 7.)

König, Esther and Wolfgang Lezius (2003). The TIGER Language – A Description Language for Syntax Graphs, Formal Definition. Technical report, IMS, University of Stuttgart, Stuttgart, Germany. (Cited on page 19.)

Lai, Catherine and Steven Bird (2004). Querying and Updating Treebanks: A Critical Survey and Requirements Analysis. In *Proceedings of the Australasian Language Technology Workshop*. (Cited on pages 10, 19, and 58.)

Lezius, Wolfgang (2002a). *Ein Suchwerkzeug für syntaktisch annotierte Textkorpora*. Ph.D. thesis, IMS, University of Stuttgart, Stuttgart, Germany. (Cited on pages v, 3, 5, 13, 19, 31, 60, 61, and 62.)

Lezius, Wolfgang (2002b). TIGERSearch – Ein Suchwerkzeug für Baumbanken. In *Proceedings of KONVENS 2002*. Saarbrücken, Germany. (Cited on pages 3 and 13.)

Lüdeling, Anke, Thorwald Poschenrieder and Lukas C. Faulstich (2004). DeutschDiachronDigital – Ein diachrones Korpus des Deutschen. In *Jahrbuch für Computerphilologie*, pp. 119–136. Paderborn, Germany: Mentis. (Cited on page 10.)

Marcus, Mitchell, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz and Britta Schasberger (1994). The Penn Treebank: Annotating Predicate Argument Structure. In *HLT '94: Proceedings of the workshop on Human Language Technology*, pp. 114–119. Morristown, NJ, USA: ACL. (Cited on page 7.)

Marek, Torsten, Joakim Lundborg and Martin Volk (2008). Extending the TIGER Query Language with Universal Quantification. In Angelika Storrer, Alexander Geyken, Alexander Siebert and Kay-Michael Würzner, eds., *Text Resources and Lexical Knowledge*, pp. 3–14. Berlin, Germany: Mouton de Gruyter. (Cited on pages 19, 60, and 62.)

Mettler, Maël B. (2007). *Parallel Treebank Search - The Implementation of the Stockholm TreeAligner Search*. C-uppsats, Stockholm University, Stockholm, Sweden. (Cited on pages 5, 60, 63, and 65.)

Müller, Christoph (2005). Simplified MMAXQL: An Intuitive Query Language for Corpora with Annotations on Multiple Levels. In Claire Gardent and Bertrand Gaiffe, eds., *Proceedings of the Ninth Workshop on the Semantics and Pragmatics of Dialogue (SEMDIAL)*, pp. 151–154. Nancy, France. (Cited on page 9.)

Müller, Christoph (2006). Representing and accessing multi-level annotation in MMAX2. In *Proceedings of the 5th Workshop on NLP and XML (NLPXML-2006): Multi-Dimensional Markup in Natural Language Processing*, pp. 73–76. Trento, Italy. (Cited on pages 9 and 45.)

Ohara, Kyoko Hirose, Seiko Fujii, Hiroaki Saito, Shun Ishizaki, Toshio Ohori and Ryoko Suzuki (2003). The Japanese FrameNet Project: A Preliminary Report. In *Proceedings of Pacific Association for Computational Linguistics (PACLING'03)*, pp. 249–254. Halifax, Canada. (Cited on page 3.)

Padó, Sebastian (2007). *Cross-Lingual Annotation Projection Models for Role-Semantic Information*. Ph.D. thesis, Saarland University, Saarbrücken, Germany. (Cited on page 63.)

Palmer, Martha, Daniel Gildea and Paul Kingsbury (2005). The Proposition Bank: An Annotated Corpus of Semantic Roles. *Computational Linguistics*, 31(1):pp. 71–106. (Cited on pages 2 and 7.)

Rehm, Georg, Richard Eckart, Christian Chiarcos and Johannes Dellert (2008a). Ontology-Based XQuery'ing of XML-Encoded Language Resources on Multiple Annotation Layers. In *Proceedings of the Sixth International Conference on Language Resources and Evaluation, LREC 2008*. Marrakech, Morocco. (Cited on page 11.)

Rehm, Georg, Oliver Schonefeld, Andreas Witt, Christian Chiarcos and Timm Lehmberg (2008b). SPLICR: A Sustainability Platform for Linguistic Corpora and Resources. In Angelika Storrer, Alexander Geyken, Alexander Siebert and Kay-Michael Würzner, eds., *Text Resources and Lexical Knowledge, Companion Volume*, pp. 85–96. Berlin, Germany: Mouton de Gruyter. (Cited on page 11.)

Rogers, James and K. Vijay-Shanker (1992). Reasoning with Descriptions of Trees. In *Proceedings of the Annual Meetings of the ACL*. Newark, DE, USA. (Cited on page 13.)

Rohde, Douglas L. T. (2005). *TGrep2 User Manual*. MIT, Cambridge, MA, USA. Available from http://tedlab.mit.edu/~dr/Tgrep2/. (Cited on page 13.)

Ruppenhofer, Josef, Michael Ellsworth, Miriam R. L. Petruck, Christopher R. Johnson and Jan Scheffczyk (2006). FrameNet II: Extended Theory and Practice. (Cited on pages 2, 21, 30, 33, and 35.)

Samuelsson, Yvonne and Martin Volk (2006). Phrase Alignment in Parallel Treebanks. In Jan Hajic and Joakim Nivre, eds., *Proceedings of the Fifth Workshop on Treebanks and Linguistic Theories, TLT 2006*, pp. 91–102. Prague, Czech Republic. (Cited on page 7.)

Schmidt, Thomas (2001). The transcription system EXMARaLDA: An application of the annotation graph formalism as the Basis of a Database of Multilingual Spoken Discourse. In *Proceedings of the IRCS Workshop On Linguistic Databases, 11-13 December 2001*, pp. 219–227. Philadelphia, PA, USA: Institute for Research in Cognitive Science, University of Pennsylvania. (Cited on page 45.)

Stede, Manfred (2004). The Potsdam Commentary Corpus. In *Proceedings of the ACL-04 Workshop on Discourse Annotation*. Barcelona, Spain. (Cited on page 7.)

Subirats, Carlos and Hirokai Sato (2004). Spanish FrameNet and FrameSQL. In *Proceedings of the 4th International Conference on Language Resources and Evaluation (LREC). Workshop on Building Lexical Resources from Semantically Annotated Corpora*. Lisbon, Portugal. (Cited on pages 3 and 5.)

Vitt, Thorsten (2005). *DDDquery – Anfragen and komplexe Korpora*. Diploma thesis, Humboldt-Universität zu Berlin, Berlin, Germany. (Cited on page 11.)

Volk, Martin, Joakim Lundborg and Maël B. Mettler (2007). A Search Tool for Parallel Treebanks. In *Proceedings of The Linguistic Annotation Workshop (LAW) at ACL*. Prague, Czech Republic. (Cited on pages 5 and 62.)

Weissstein, Eric W. (2009). Topological Sort. From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/TopologicalSort.html`. (Cited on page 56.)

Yan, Xiafeng, Philip S. Yu and Jiawei Han (2004). Graph Indexing: A Frequent Structure-based Approach. In *Proceedings of SIGMOD 2004*. (Cited on page 64.)

## COLOPHON

### *Fonts and Layout*

This thesis was typeset with LaTeX $2_\varepsilon$ using Hermann Zapf's *Palatino* and *Euler* type faces. The Type 1 PostScript fonts *URW Palladio L* and *FPL* were used. The listings are typeset in *Bera Mono*, originally developed by Bitstream, Inc. as "Bitstream Vera". Type 1 PostScript fonts were made available by Malte Rosenau and Ulrich Dirr.

The layout is based on the `classicthesis` style © André Miede, which is available for from CTAN as "`classicthesis`" and licensed under the GNU GPL v2.

### *Figures*

The logo "Neue Eule v2.10" on the titlepage is © Joachim Blum, available from Saarland University.

Figures 4.1, 4.3–4.5, 5.1 and B.1 were created with the program `dot` from the Graphviz software package for graph visualization. Graphviz was written by Emden Gansner et al. at AT&T Research.

Figures 3.1, 3.2, 4.2, 4.6, 4.7 and 5.2 are based on sentences from v1.0 of the SALSA corpus created by the SALSA II project at Saarland University and described in Burchardt et al. (2006a). The figures were created with the latest development version of the TreeAligner, © 2005–2009 Joakim Lundborg and Torsten Marek.

*Final Web Version* (rev. 82) as of March 23, 2009 at 21:31.