Domain-adaptive Information Extraction
(Draft Version)

Günter Neumann and Giampaolo Mazzini
DFKI
Saarbrücken, Germany

February 28, 1999

1 Introduction

There is no doubt that the amount of textual information electronically available today has passed its critical mass leading to the emerging problem that the more electronic text data is available the more difficulty it is to find or extract relevant information. In order to overcome this problem new technologies for future information management systems are explored by various researchers. One new line of such research is the investigation and development of *information extraction* systems.

Information extraction (IE) is the task of identifying, collecting and normalizing information from natural language text. The information of interest is typically prespecified in form of uninstantiated frame-like structures also called *templates*. The templates are domain and task specific. The major task of an IE-system is then the identification of the relevant parts of the text which are used to fill a template's slot. In order to achieve the necessary degree of robustness and efficiency, domain knowledge and shallow algorithms are used which have been tuned —usually by hand— for the actual domain and task.

The current IE technology has already a high degree of application potential (e.g., intelligent information retrieval, linguistically based data mining, automatic term extraction for data bases, fine-grained text classification) and has shown important industrial application impact. However, the major disadvantage of the current IE technology is that the application specific knowledge has to be directly integrated with the domain-independent linguistic knowledge sources in order in achieve the necessary degree of robustness and efficiency. This means that the system customization is based on a very low-level direct linkage of linguistic and domain specific knowledge. Since defining and implementing this low-level linkage is a time-consuming task which has to be performed manually by very specialized experts, the necessary high amount of system customization leads to a low degree of flexibility and adaptability of a system to new applications and domains. But then, it will be more and more difficult to effectively use and adapt a IE-system based on current IE technology for the rapidly increasing different kinds of user requests and the dynamically increase and change of information of the Internet organism.

In this paper we will present methods and technologies which support the development of IE systems which can rapidly be configured for new applications domains and tasks, i.e., IE systems which support a fast application development cycle. The core idea is to model the linkage of domain-independent and domain knowledge on a much higher and abstract level than it is usually done in current IE systems *without* critical compromises wrt. robustness and efficiency. More precisely, the new IE model we propose is based on:

- a domain-independent shallow text processor, which maps an NL text into a set of underspecified (partial) functional descriptions;
- modeling of domain and template definitions using type hierarchies without reference to linguistic knowledge;
- modeling of abstract linguistic types on the basis of a generalization of subcategorization frames
- declarative linkage of linguistic and domain knowledge through multiple inheritance (the resulting types are called linking types);
- the construction of new applications basically through specialization of the domain knowledge hierarchy and specification of a domain lexicon.

The main advantages towards a faster development cycle are then:

- a high degree of modularity, re-usability and knowledge-sharing;
- template definition and specification of linked types can be separated;

- no deep internal knowledge of the system (especially about the relationship between NL and its domain specific meaning representation) is necessary for building a new application.

A prototype of the model described here has been implemented, and we call it SMES.[1] SMES has been fully implemented and uses broad coverage linguistic knowledge sources. Although the SMES technology has only been used up to now for handling German the underlying strategies are also applicable to other languages as well.

The rest of the paper is organized as follows. In the next section we give an overview of SMES and describe briefly the general flow of control using a simple example. In section 3 we describe the shallow text processor, and in section 4 the domain knowledge processing using a typed-feature formalism. In section 5 we describe some implementative aspects of the system.

2 Overview of the IE-model

The architecture of SMES shown in figure 1, includes four main components, namely:

1. A shallow text processor STP, which performs the analysis of a textual input through different levels: scanning, morphological analysis, phrase recognition, clause-level phrase collection, and determination of grammatical functions at the clause level. We will refer to all these processing steps collectively as *underspecified text processing*.

2. A domain model DM, which is made of different hierarchical knowledge bases, aimed at modelling *templates* to be extracted, and the relationships they have with the linguistic knowledge used by the system, which is represented here in terms of hierarchies of *phrasal types* and *predicate-argument structure types*. Such relationships are explicitely defined as *linking types*. The different hierarchies that constitute the domain model have been defined in the present implementation using $\mathcal{TDL}$[Krieger and Schäfer, 1994b] as type description language.

3. A domain lexicon DL, which associates a subset of the entries of the lexicon used in STP (basically a stem lexicon, with additional informa-

[1]It is actually a direct successor of an IE-core system for real world German text processing described in [Neumann *et al.*, 1997].

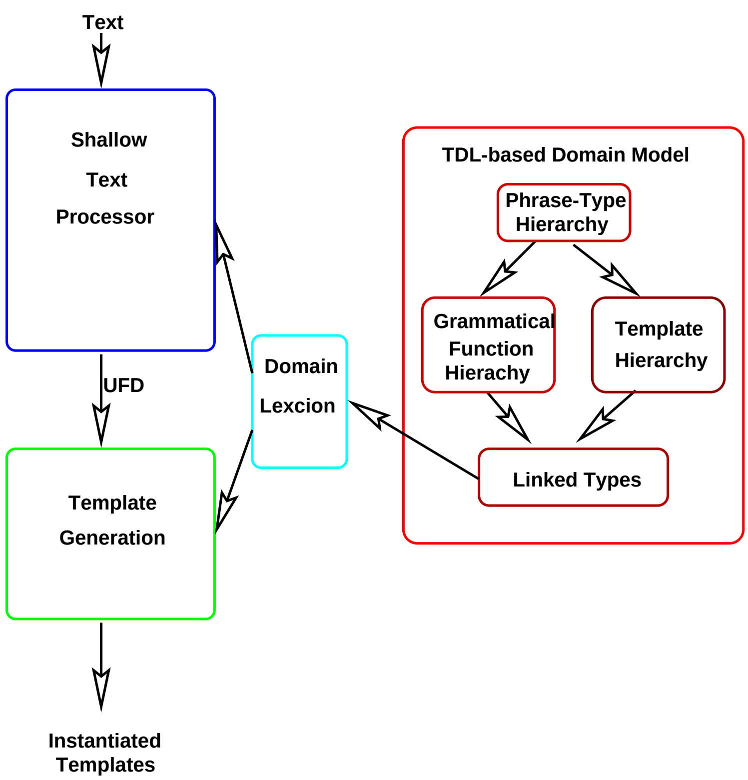

Figure 1: The architecture of SMES.

tion about predicate-argument structures for verbs), which are relevant for a particular domain, to the appropriate linking types.

4. A template generator, which uses the flat linguistic output from STP as a structural guidance for filling and merging the templates which have been retrieved by means of the domain lexicon DL.

The general flow of information through the above components is, very schematically, the following one. First of all, a natural language text is processed by the shallow text processor STP (which is made of a number of pipelined subcomponents, as described before), which returns a sequence of *underspecified (partial) functional descriptions* UFDS as output[2].

Each UFD computed by the shallow text processor is then passed to the template generator, which makes use of the knowledge represented in the domain model in order to end out with a set of filled templates. The component of SMES architecture that is resposible for bridging the domain knowledge and UFDs is, of course, the domain lexicon DL. The template generator actually performs the following three steps:

- type merging: a UFD is merged with the linking types found for its head (usually the verb), retrieved from the domain lexicon; the resulting object — still a UFD with additional information — is then mapped into an equivalent TDL object, i.e. an *unexpanded instance*. Such an object, which we label as UFD$_{TDL}$, is basically an feature structure (an attribute value matrix) provided with information originating from STP, but still to be matched against the corresponding general definition of its type in order to inherit all the information encoded there (this is what *unexpanded* vs. *expanded* means in this context). Type merging can be done in a profitable way during shallow text processing already — and this is exactly the approach we have chosen in our implementation. Thus, we describe it here only for the sake of clarity.

- template filling: The UFD$_{TDL}$ resulting from *type merging* is passed to a TDL machine implementing the domain model, in order to get (pos-

[2]An UFD is a flat dependency-based structure of a sentence, where attachments are not yet resolved (avoiding in that way the cost of massive overgeneration, due to the lack of very sophisticated lexical information at this level); rather, a kind of *upper bounds* over different possibilities for attachment and scoping of modifiers are expressed, to be further specified — whenever possible — only in the light of the domain-specific knowledge model at a later stage.

sibly) expanded by means of the tdl-internal mechanism of *instance expansion* (see the relevant TDL literature to see a detailed description of the algorithms involved), according to the actual model definition. If expansions does not fail (as in the case the unexpanded instance shows conflicting information wrt. the definition of its type the domain model, or that type does not exist at all in the model) used for governing expansion, then this automatically fills all those slots in the templates, which are accessible via the linking types associated to the instance.

- template merging: merging the information of two or more templates together; also implies reference resolution.

3 Shallow Text Processing

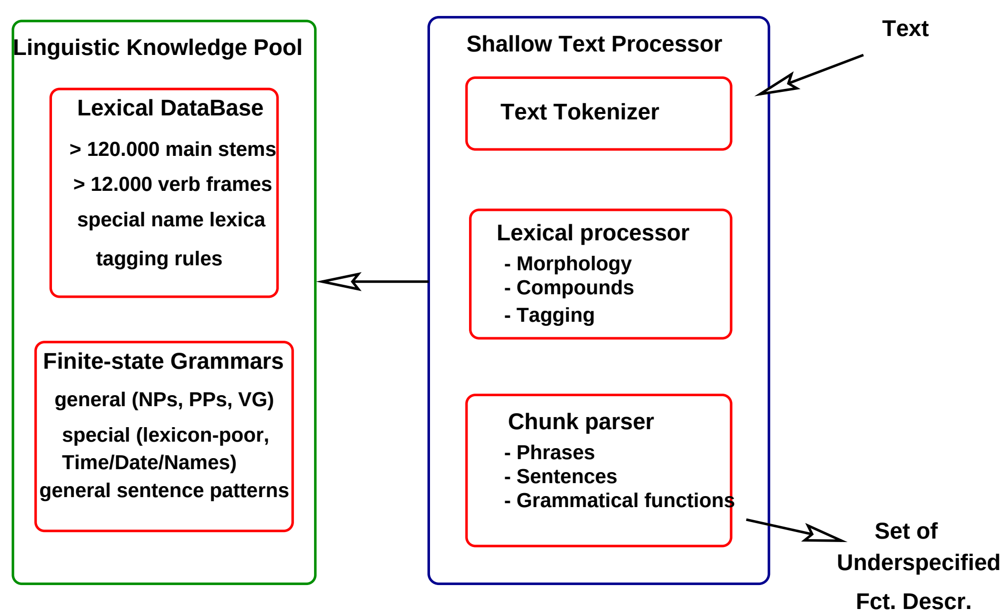

Figure 2: The Architecture of Shallow Text Processor STP

An NL-text is processed by the shallow text processor STP, which results in a sequence of *underspecified (partial) functional descriptions* UFDS. An UFD is a flat dependency-based structure of a sentence, where only upper bounds for attachment and scoping of modifiers are expressed. Besides the head/modifier relationship, grammatical functions are determined using a subcategorization lexicon. An UFD can be partial in the sense that some phrasal chunks of the sentence in question could not be inserted into the head/modifier relationship. In that case an UFD will represent the *longest matching subclause* together with a list of the non-recognized fragments. To retain the non-recognized fragments is important, because it makes possible for specific IE-templates or inference rules to have access to this information even in the case it could not linguistically be analyzed.

The architecture of STP is shown in figure 2. It consists of two major components, a) the Linguistic Knowledge Pool (LKP) and b) STP, the shallow text processor itself. Before describing the shallow text processor in more detail, we will briefly explain the notion of *underspecified text processing.*

3.1 Underspecified text processing

We assume that an UFD, and hence, the output description of every finite-state grammar (see fig. 2) represents a kind of logical form LF. Such a LF minimally consists of all word stems and their head/modifier relationships. However, it is shallow (i.e., coarse-grained) wrt. domain specific knowledge. Finite-state grammars express a direct relationship between word sequences and their possible LF, i.e., there is no intermediate structural level between a string and an LF. Since a UFD does only specify upper bounds for attachment and scoping, it is assumed that the detailed attachment and scoping information will be *computed* by the concrete application. This is in contrast to the common approach of grammatical processing, where the goal is to find all possible readings of an expression wrt. all possible worlds. In some sense such an approach is domain-independent by just enumerating all possible readings. The task of domain-specificity is then reduced to the task of "selecting the right reading" of the current specific domain. In our approach, we provide a complete but underspecified representation, by only computing a coarse-grained structure. This structure then has to be "unfolded" by the current application. In some sense this means that after shallow processing we only obtain a very general, rough meaning of an expression whose actual interpretation has to be "computed" (not selected) in the current application. This is what we mean by *underspecified text processing.*

Using underspecified representations for linguistic processing is not a new idea (consider e.g., [Gardent and Webber, 1998], [Muskens and Krahmer, 1998]). However, these approaches do not discuss in necessary detail *how* an underspecified form is actually *expanded* in case a certain application requests for a particular reading. Rather it seems to be assumed that a follow-up component is responsible for that. However, this simply means that the problem of desambiguation is just put to the next level.

What these approaches are avoiding is an explicit treatment of the relationship between linguistic expressions and their corresponding meanings in a certain application. However, this is exactly what we are going to do with our linking types.

3.2 Overview of the major processing modules of STP

Internally, a NL-text is processed through a chain of modules (in capital letters here). First the TEXT TOKENIZER maps sequences of characters into greater units, usually called TOKENS. Each token, which is identified as a potential wordform is then lexically processed by the LEXICAL PROCESSOR. Lexical processing includes morphological analysis, recognition of compounds, retrieval of lexical information (where retrieval allows fuzzy matching), and tagging, which performs word-based disambiguation. Next the CHUNK PARSER is called for the analysis of phrasal and clausal expressions. Chunk parsing is subdivided into three steps. In a first step phrasal fragments are recognized, like general nominal expressions and verb groups or specialized expressions for time, date, and named entity. The structure of potential phrasal fragments is defined using finite state transducers FST. In the second step, the dependency-based structure of the fragments of each sentence is analysed using a set of specific sentence patterns.[3] These patterns are also expresssed by means of finite state transducers FST, so that step one and three are uniformly realized by the same underlying mechanism. In the final fourth step, the grammatical functions are determined for each dependency-based structure on the basis of a large subcategorization lexicon. We will now describe the processing modules in more detail.

[3] In the current system sentence boundaries are recognized after step one and before step two, so that the set of found fragments are subdivided into subsets where each subset contains all recognized fragments belonging to one sentence. We decided to place sentence-boundary recognition after phrasal recognition because it allows us to treat proper certain ambiguity problems (like separated verb prefixes) even in the case of corrupt or incomplete input texts. Of course this might be a language specific decision. However, it is very easy to place language specific sentence-boundaries in another place of the SMES system without effecting the behaviour of the CHUNK PARSER.

3.3 Text tokenization

The TEXT TOKENIZER performs the following tasks:

- recognizing of tokens of different types using a set of regular expressions;
- expanding abbreviations using a list of known abbreviations;
- normalizing of found tokens.

Each file is firstly preprocessed by the TEXT TOKENIZER. Applying regular expressions (the text scanner is implemented in LEX, the well-known Unix tool), the text scanner identifies some text structure (e.g., paragraphs, indentations), word, number, date and time tokens (e.g, "1.3.96", "12:00 h"), and expands abbreviations. The output of the text scanner is a stream of tokens, where each word is simply represented as a string of alphabetic characters (including delimiters, e.g. "Daimler-Benz"). Number, date and time expressions are normalized and represented as attribute values structures. For example the character stream "1.3.96" is represented as (:date ((:day 1)(:mon 3)(:year 96)), and "13:15 h" as (:time ((:hour 13)(:min 15))). Abbreviations are recognized on the basis of a set of pairs of known abbreviations and their expanding counterpart. Using LEX this list as well as the list of regular expressions can easily be updated.

3.4 Lexical processing

Each token, which is identified as a potential wordform is lexically processed by the LEXICAL PROCESSOR. Lexical processing includes morphological analysis, recognition of compounds, retrieval of lexical information (where retrieval allows fuzzy matching), and tagging, which performs word-based disambiguation.

Morphological processing performs inflection, and processing of compounds. The capability of efficiently processing compounds *on-line* is crucial since compounding is a very productive process of the German language.

The morphological component called MORPHIX++ is a descendant of MORPHIX, a fast classification-based morphology component for German [Finkler and Neumann, 1988]. MORPHIX++ improves MORPHIX in that the classification-based approach has been combined with the well-known two-level approach, originally developed by [Koskenniemi, 1984]. Actually, the extensions concern

- the use of tries (see [Aho *et al.*, 1983]) as the sole storage device for all sorts of lexical information in MORPHIX++ (e.g., for lexical entries, prefix, inflectional endings). Beside the usual functionality (insertion, retrieval, deletion), a set of more complex functions is available, most notably a regular trie matcher, and

- the analysis of compound expressions which is realized by means of a recursive trie traversal. During traversal two-level rules are applied for recognizing linguistically well-formed decompositions of the word form in question.

The basic processing strategy employed by MORPHIX++ consists of trie traversal combined with the application of finite state automata. The morphology returns a list of lists where each lists corresponds to a sentence. Each morphological analysed word form (word forms are represented as strings by the tokenizer; every other token, e.g., time or date expression is represented as a list) are represented as a triple of the form $\langle stem, inflection, pos \rangle$, where $stem$ is a string or a list of strings (in the case of compounds), $inflection$ is the inflectional information, and pos is the part of speech.

MORPHIX++ has a flexible parametrizable output interface wrt. the form of the $inflection$ information, since it is used as a standalone component in several other quite distinct NLP applications at our LT-Lab. In SMES the $inflection$ information is expressed as feature-vector representation in disjunctive normal form (DNF). A feature-vector is a vector of simple feature-value pairs, where each value is atomic. Although MORPHIX++ has a rich tag set we only make use of a small subset of inflectional features (see also next paragraph), namely:

:TENSE :FORM :PERSON :GENDER :NUMBER :CASE

Note that in case a wordform misses one of the features (e.g., a noun form has no :TENSE feature) the missing feature is returned with the special value :NO. During chunk parsing and agreement checking (see next section) this value is handled as an anonymous variable. Here follows the result of the morphological component for the sentence (abbreviated where convenient):

“Der Mann sieht die Frau mit dem Fernrohr.”
(The man sees the woman with the telescope)

```
((("Der"
   ("d-det"
```

```
  ((... (:GENDER . :M) (:NUMBER . :S) (:CASE . :NOM))
   (... (:GENDER . :M) (:NUMBER . :P) (:CASE . :GEN))
   (... (:GENDER . :F) (:NUMBER . :S) (:CASE . :GEN))
   (... (:GENDER . :F) (:NUMBER . :S) (:CASE . :DAT))
   (... (:GENDER . :F) (:NUMBER . :P) (:CASE . :GEN))
   (... (:GENDER . :NT) (:NUMBER . :P) (:CASE . :GEN)))
  . :DEF))
("Mann"
 ("mann"
  ((... (:GENDER . :M) (:NUMBER . :S) (:CASE . :NOM))
   (... (:GENDER . :M) (:NUMBER . :S) (:CASE . :DAT))
   (... (:GENDER . :M) (:NUMBER . :S) (:CASE . :AKK)))
  . :N))
("sieht"
 ("seh"
  (((:TENSE . :PRES) (:FORM . :FIN) (:PERSON . 3)
    (:GENDER . :NO)  (:NUMBER . :S) (:CASE . :NO)))
  . :V))
("die"
 ("d-det"
  ((... (:GENDER . :M) (:NUMBER . :P) (:CASE . :NOM))
   (... (:GENDER . :M) (:NUMBER . :P) (:CASE . :AKK))
   (... (:GENDER . :F) (:NUMBER . :S) (:CASE . :NOM))
   (... (:GENDER . :F) (:NUMBER . :S) (:CASE . :AKK))
   (... (:GENDER . :F) (:NUMBER . :P) (:CASE . :NOM))
   (... (:GENDER . :F) (:NUMBER . :P) (:CASE . :AKK))
   (... (:GENDER . :NT) (:NUMBER . :P) (:CASE . :NOM))
   (... (:GENDER . :NT) (:NUMBER . :P) (:CASE . :AKK)))
  . :DEF))
("Frau"
 ("frau"
  ((... (:GENDER . :F) (:NUMBER . :S) (:CASE . :NOM))
   (... (:GENDER . :F) (:NUMBER . :S) (:CASE . :GEN))
   (... (:GENDER . :F) (:NUMBER . :S) (:CASE . :DAT))
   (... (:GENDER . :F) (:NUMBER . :S) (:CASE . :AKK)))
  . :N))
("mit"
 ("mit"
  ((... (:GENDER . :NO) (:NUMBER . :NO) (:CASE . :DAT)))
  . :PREP))
("dem"
 ("d-det"
  ((... (:GENDER . :M) (:NUMBER . :S) (:CASE . :DAT))
   (... (:GENDER . :NT) (:NUMBER . :S) (:CASE . :DAT)))
```

```
   . :DEF))
 ("Fernrohr"
  ("fernrohr"
   ((... (:GENDER . :NT) (:NUMBER . :S) (:CASE . :NOM))
    (... (:GENDER . :NT) (:NUMBER . :S) (:CASE . :DAT))
    (... (:GENDER . :NT) (:NUMBER . :S) (:CASE . :AKK)))
   . :N))
 ("." ("." NIL . :INTP))))
```

Dynamic feature selection The set of features as well as their possible domain is finite, so that in principle it would also be possible to express each inflectional reading as a unique identifier. However, we use the feature-vector representation because it supports the specification of very compact finite state grammars. Furthermore, the SMES system allows a dynamic change of the set of features to be considered. For example, it is possible to specify that only the features :PERSON, :GENDER and :CASE are of interest in a particular application and that all other features should be relaxed. The specification of a relevant subset of features can be done on-line, which means that the finite state grammars will run without changes. Of course, if we would have compiled out the DNF representation into unique identifiers as mentioned above, then all grammars which are effected by the new set of relevant features would have to be re-compiled. However, this would only allow a restricted kind of on-line feature relaxation.

Currently, MORPHIX++ is used for the German and Italian language. The German version has a very broad coverage (a lexicon of more then 120.000 stem entries), and an excellent speed (5000 words/sec without compound handling, 2800 words/sec with compound processing (where for each compound all lexically possible decompositions are computed).[4]

Part-of-speech disambiguation Morphological ambiguous readings are disambiguated wrt. part-of-speech using

- *case-sensitive rules*, and
- *filtering rules*

which have been determined using Brill's un-supervised tagger [Brill, 1995]. The filtering rules are also used for tagging unknown words.

[4]Measurement has been performed on a Sun 20 using an on-line lexicon of 120.000 entries.

Generally, only nouns (and proper names) are written in standard German with a capitalized initial letter (e.g., "der Wagen" *the car* vs. "wir wagen" *we venture*). Since typing errors are relatively rare in press releases (or similar documents) the application of case-sensitive rules are a reliable and straightforward tagging means for the German language. Of course, the case-sensitive rules should only be applied, if there is enough evidence that the document's author actually followed typical German spelling rules. For that reason, the case-sensitive rules are only applied, if the value of a stastically driven lookahead function TRIGGER-WR(k) exceeds some threshold y applied on the k-first tokens computed by the TEXT SCANNER. Actually, the method counts all strings s (i.e., the k-first possible wordforms) that do not follow the dot sign and all lower-case strings ls of s. Then if $s > 0$ and $ls/s > y$ then TRIGGER-WR(k) returns T (meaning that the case-sensitive rules will be applied); otherwise F is returned which means that no disambiguation on the basis of cases-sensitivity will be made.

The filtering rules are determined on the basis of unannotated corpora. Starting from untagged corpora, MORPHIX++ is used for initial tagging, where unknown words are ambiguously tagged as noun, verb, and adjective. Then, using contextual information from unambiguously analysed word forms, filter rules are determined which are of the form *change tag of word form from noun or verb to noun if the previous word is a determiner.*

First experiments using a training set of 100.000 words and a set of about 280 learned filter rules yields a tagging accuracy (including tagging of unknown words) of 91.4%.[5] The un-supervised tagger required no hand-tagged corpora and considered unknown words. We expect to increase the accuracy by improving the un-supervised tagger through the use of more linguistic information determined by MORPHIX++ especially for the case of unknowns words (for more details see [Declerck *et al.*, 1998]).

3.5 Chunk parsing

The task of chunk parsing is to construct a hierarchical structure over the words of a sentence. In contrast to deep parsing strategies, where recognition of phrases and clauses is interleaved, a chunk parser explicitly separates these steps. However in contrast to most IE approaches of shallow parsing where

[5]Brill reports a 96% accuracy using a training set of 350.000 words and 1729 rules. However, he does not handle unknown words. In [Aone and Hausman, 1996], an extended version of Brill's tagger is used for tagging Spanish texts, which includes unknown words. They report an accuracy of 92.1%.

a very direct linkage between word groups (or phrases) and their domain-specific meaning and usage is expressed, we consider chunk parsing as a domain-independent task[6]. In our system, chunk parsing is performed in three steps:

- recognition of phrases
- recognition of clauses
- recognition of grammatical functions

The first two steps are performed by finite state grammars expressed by means of finite state transducers. The last step is performed by means of a large subcategorization lexicon and subcategorization resolution mechanisms.

The chunk parser is called on the result of the lexical component, i.e., the input is a list of words, where a word has either been computed by the tokenizer (e.g., date or time expressions) or by the morphological component (which also includes identification of lexically unknown wordforms). We also assume that the text already has been broken down into sentence units.

3.6 Finite State Transducers

Phrase and clause recognition is performed through fragment extraction patterns expressed as finite state transducers (FST), which are compiled to Lisp functions using a compiler based on [Krieger, 1987]. An FST consists of a unique name, the recognition part, the output description, and a set of compiler parameters.

The recognition part An FST operates on a stream of tokens. The recognition part of an FST is used for describing regular patterns over such token streams. For supporting modularity the different possible kind of tokens are handled via *basic edges*, where a basic edge can be viewed as a predicate for a specific class of tokens. More precisely a basic edge is a tuple of the form $\langle name, test, variable \rangle$, where *name* is the name of the edge, *test* is a predicate, and *variable* holds the current token T_c , if *test* applied on T_c holds. For example the following basic edge (:morphix-cat "partikel" pre) tests whether T_c produced by MORPHIX++ is a particle, and if so binds

[6] Of course this does not exclude the use of word-based or concept-based triggers and filters which are applicable before or during chunk parsing.

the token to the variable *pre* (more precisely, each variable of a basic edge denotes a stack, so that the current token is actually pushed onto the stack).

We assume that for each component of the system for which fragment extraction patterns are to be defined, a set of basic edges exists. Furthermore, we assume that such a set of basic edges remains fixed at some point in the development of the system and thus can be re-used as pre-specified basic building blocks to a grammar writer (i.e., the set of basic edges intentionally define the alphabet of a finite state grammar).

Using basic edges the recognition part of an FST is then defined as a regular expression written in functional notation. For example the recognition part for simple nominal phrases might be defined as follows:

```
(:conc
  (:star≤n (:morphix-cat "det" det) 1)
  (:star (:morphix-cat "adj" adj))
  (:morphix-cat "n" noun))
```

Thus defined, a nominal phrase is the concatenation of one optional determiner (expressed by the loop operator :star≤n, where n starts from 0 and ends by 1), followed by zero or more adjectives followed by a noun.

Output description part The output structure of an FST is constructed by collecting together the variables of the recognition part's basic edges followed by some specific construction handlers. In order to support reusability of FST to other applications, it is important to separate the construction handlers from the FST definition. Therefore, the output description part is realized through a function called BUILD-ITEM which receives as input the edge variables and a symbol denoting the class of the FST. For example, if :np is used as a type name for nominal phrases then the output description of the above NP-recognition part is

```
(build-item :type :np :out (list det adj noun)).
```

The function BUILD-ITEM then discriminates according to the specified type and constructs the desired output to some pre-defined requests (note, that in the above case the variables DET and ADJ might have received no token. In that case their default value NIL is used as an indication of this fact). Using this mechanism it is possible to define or re-define the output structure without changing the whole FST.

Special edges There exist some special basic edges namely (:var var), (:current-pos pos) and (:seek name var). The edge (:var var) is used for simply skipping or consuming a token without any checks. The edge :current-pos is used for storing the position of the current token in the variable pos, and the edge :seek is used for calling the FST named name, where var is used as a storage for the output of name. This is similar to the :seek edge known from Augmented Transition Networks with the notably distinction that in our system recursive calls are *disallowed*. Thus :seek can also be seen as a macro expanding operator. The :seek mechanism is very useful in defining modular grammars, since it allows for a hierarchical definition of finite state grammars, from general to specific constructions (or vice versa). The following example demonstrates the use of these special edges:

```
(compile-regexp
 (:conc
   (:current-pos start)
   (:alt
    (:seek time-phase time)
    (:conc
      (:star≤n (:seek time-expr-vorfield vorfield) 1)
      (:seek mona-time time)))
   (:current-pos end))
  :name time-expr
  :output-desc
   (build-item :type time-expr :start start
              :end end :out (list vorfield time))))
```

This FST recognizes expressions like "spätestens um 14:00 h" (*by two o'clock at the latest*) with the output description ((:out (:time-rel . "spaet") (:time-prep . "um") (:minute . 0) (:hour . 14)) (:end . 4) (:start . 0) (:type . time-expr))

Agreement checking On the basis of the feature vector representation of morpho-syntactic information computed by the morphological component MORPHIX++ we have defined a very efficient method for the unification of two feature vectors. It is very easy to integrated this unifier into an FST via the definition of an edge of the form

(:morphix-unify-cat Cat AgrIn AgrOut Out) where CAT is the part-of-speech of the current morphix-token, AGRIN a feature vector (or NIL, representing the empty featurce vector) and AGROUT the feature vector resulting from unification of the feature vector of the current token and AGRIN. OUT bounds the current token. Using agreement information in this way it is possible to propagate the agreement information through several edges of one FST. In some sense this mechanism helps to define "PATR-style" finite state grammars. As an example consider the following regular expression of a simple NP for German (where the numbered box represents structure sharing between the basic elements of the expressions):

$$\mathbf{NP} \rightarrow [\mathbf{Def}_{\boxed{1}} \mid \mathbf{Indef}_{\boxed{1}}] \quad [\mathbf{Adj}_{\boxed{1}}] \quad \mathbf{Noun}_{\boxed{1}}$$

described in SMES through the following finite state transducer:

```
(compile-regexp
 '(:conc
    (:current-pos start)
    (:alt
      (:star<=n (:morphix-unify :indef  NIL agr det) 1)
      (:star<=n (:morphix-unify :def  NIL agr det) 1))
    (:star<=n (:morphix-unify :a agr agr adj) 1)
    (:morphix-unify :n  agr agr noun)
    (:current-pos end))
 :output-desc
 '(:lisp (build-item :type :np :start start :end end :agr agr
                     :det det :adj adj :noun noun))
 :name 'small-np)
```

In the alternative expression, it is checked whether the current token is either a definite or indefinite determiner. Assuming that the current token is indeed a definite determiner, its feature vector $fvec(:def)$ is unified with the empty feature vector NIL and the resulting feacture vector is bound to AGR. Since unification of a feature vector $fvec$ with NIL yields $fvec$, AGR is bound to $fvec(def)$. In the next call of :MORPHIX-UNIFY the current token is tested for adjective (expressed as :A. If so, the feature vector $fvec(:a)$ is unified with the current value bound to AGR, i.e., $fvec(:a)$ and $fvec(:def)$ are unified in the resulting $fvec$ is bound to AGR. In the next step the feature vector associated to the noun is unified with $fvec$ and if unification succeeds, the whole expression is recognized as an simple NP where the current feature vector bound to AGR defines the agreement information of the complete NP.

Control parameters In order to obtain flexible control mechanisms for the matching phase it is possible to specify whether an exact match is requested or whether an FST should already succeed when the recognition part matches a prefix of the input string (or suffix, respectively). The prefix matching mechanism is used in conjunction with the Kleene-star operator, which we note :star, and the identity edge :var, to allow for searching the whole input stream for extracting all matching expressions of an FST (e.g., extracting all NP's, or time expressions). The resulting behaviour is comparable to what [Roche and Schabes, 1995] call "the local extension of a transducer". For example the following FST extracts all genitive NPs found in the input stream and collects them in a list [7]:

```
(compile-regexp
  (:star
   (:alt
    (:seek gen-phrase x)
    (:var dummy)))
  :output-desc (build-item :type list :out x )
  :prefix T
  :suffix NIL
  :name gen-star)
```

Additionally, a boolean parameter can be used to specify whether longest or shortest matches should be prefered (the default is longest match, see also [Appelt *et al.*, 1993] where also longest subsuming phrases are preferred).

3.7 Phrase recognition

In the first phase the chunk parser tries to identify all phrases. What actually counts as a valid phrase is expressed via a set of finite state transducers FST (as described above), each responsible for the analysis of a particular class of linguistic expressions. In the current system we defined the following FSTs:

- nominal phrases NP;

[7]Note, that if the variable of the :var edge would also be x, x would be the same as the input with the exception that all matching token groups would have been substituted by the output description of the successful FST

- prepositional phrases PP;
- verb groups VG;
- proper name phrases;
- complex time & date phrases TP and DP;
- number & currency phrases.

The phrasal grammars only consider *continuous substrings* and try to recognize the *longest matching substring*. Since we do not recognize discontinous phrases at that level, recognition of verb groups is only partial as long as a verb group is splitted into a left and right part, as is common in German. For example in passive sentences the auxiliar and the corresponding main verb are splitted into two parts such that other phrases are spliced into the splitting point.

"Gestern [ist] der Linguist vom neuen Manager [entlassen worden]."
Yesterday, the linguist [has] [been fired] by the new manager.

Each Finite-state grammar is expected to represent its resulting structures uniformly as feature value structures, together with its type and the corresponding start and end position of the spanned input expression. We call these output structures *text items*. The specific information computed by a particular finite-state grammar is then stored as the value of the attribute SEM. Consider the following simple examples:

"Der Mann sieht die Frau mit dem Fernrohr."
(The man sees the woman with the telescope)

which yields (abbreviated where convenient):

```
(((:SEM (:HEAD "mann") (:QUANTIFIER "d-det"))
  (:AGR
   ((:TENSE . :NO) ... (:CASE . :NOM)))
  (:END . 2) (:START . 0) (:TYPE . :NP))
 ((:SEM (:HEAD "frau") (:QUANTIFIER "d-det"))
  (:AGR
   ((:TENSE . :NO) ... (:GENDER . :F) (:NUMBER . :S)
    (:CASE . :NOM))
   ((:TENSE . :NO) ... (:GENDER . :F) (:NUMBER . :S)
```

```
   (:CASE . :AKK)))
 (:END . 5) (:START . 3) (:TYPE . :NP))
((:SEM (:HEAD "mit")
       (:COMP (:QUANTIFIER "d-det") (:HEAD "fernrohr")))
 (:AGR
  ((:TENSE . :NO) ... (:GENDER . :NT) (:NUMBER . :S)
   (:CASE . :DAT)))
 (:END . 8) (:START . 5) (:TYPE . :PP)))
```

The resulting structures are interpreted as head/modifier structures. The PP "mit dem Fernrohr" has not been attached to one of the NPs since this cannot deterministically decided without further analysis. The type information associated with each phrasal expreession is crucial since it provides for an important interface for the definition of the phrase-type hierarchy in the domain model (see also 2 and 4.5).

3.8 Clause recognition

The main task of clause recognition is to construct a dependency relation between all recognized phrases. Basically this includes combination of corresponding left and right verb groups, identification of subclauses, and collection of all phrases which are dominated by the found verb groups. Rules for clause recognition are expressed as finite state transducers FST.

Linguistically, clause recognition is performed by means of a dependency-based grammar, i.e., the resulting parse tree is actually a dependency tree. However, in the same way as we delay PP-attachment during phrase recognition, also clause recognition only defines *upper bounds* for attachment, which is mainly defined by the head element. Since a dependency tree is potentially recursive, such trees are not totally flat, but coarse-grained wrt. attachement. We call such a dependency tree an *underspecified dependency tree*. Furthermore, all phrases which cannot be inserted into the dependency structure are collected into a separate list (and bound under a feature named REST). Retaining the non-recognized fragments in the output is important, because it makes possible for specific rules (e.g. information extraction or inference rules) to have access to that information, even though it has not been properly assigned to a specific slot in the dependency structure. Consider the following example:

"Der Mann sieht die Frau mit dem Fernrohr."

which yields the following underspecified dependency tree (abbreviated where convenient):

```
(((:PPS
   ((:SEM (:HEAD "mit")
          (:COMP (:QUANTIFIER "d-det") (:HEAD "fernrohr")))
    (:AGR
     ((:TENSE . :NO) ... (:CASE . :DAT)))
    (:END . 8) (:START . 5) (:TYPE . :PP)))
  (:NPS
   ((:SEM (:HEAD "mann") (:QUANTIFIER "d-det"))
    (:AGR
     ((:TENSE . :NO) ... (:CASE . :NOM)))
    (:END . 2) (:START . 0) (:TYPE . :NP))
   ((:SEM (:HEAD "frau") (:QUANTIFIER "d-det"))
    (:AGR
     ((:TENSE . :NO) ... (:CASE . :NOM))
     ((:TENSE . :NO) ... (:CASE . :AKK)))
    (:END . 5) (:START . 3) (:TYPE . :NP)))
  (:VERB
   (:COMPACT-MORPH
    ((:TEMPUS . :PRAES) ... (:PERSON . 3)
     (:GENUS . :AKTIV)))
   (:MORPH-INFO
    ((:TENSE . :PRES) (:FORM . :FIN) ... (:CASE . :NO)))
   (:ART . :FIN) (:STEM . "seh")
   (:FORM . "sieht") (:C-END . 3) (:C-START . 2)
   (:TYPE . :VERBCOMPLEX))
  (:END . 8) (:START . 0) (:TYPE . :VERB-NODE)))
```

In this structure, the feature :VERB collects all information of the complex verb group which is the head of the sentence. :PPS collects all prepositional phrases and :NPS is a list of all dependent nominal phrases.

3.9 Grammatical Functions

After clause recognition, a further analysis step is done by the *grammatical function recognizer* (GFR), which adds an additional level of information to an underspecified dependence tree, consisting of:

- Identification of possible *arguments* on the basis of the lexical subcategorization information available for the local head.

- Marking of the other non-head elements of the dependence tree as *adjuncts*, possibly by applying a distinctive criterion for standard and specialized adjuncts. Adjuncts — opposed to arguments, for which an attachment resolution is attempted — have to be considered underspecified wrt. attachment, even after GFR: in other words, their dependency relation to the head counts as an upper border rather than an attachment.

Currently, GFR applies at the sentential levels (no analysis of argumental structure internal to other classes of constituents, like NPs or ADJPs for example, has been provided so far). Therefore the recursion potential is restricted to subclauses (either as arguments or adjuncts), which are internally analyzed for grammatical functions in the same way the main clause is; all the other constituents maintain the same structure they had in the input structure. As an example, consider GF assignment to a sentence like the following one:

"Der Mann sieht die Frau mit dem Fernrohr."

which has been determined by GFR from the underspecified dependency tree shown in 3.8 above (abbreviated where convenient):

```
(((:SYN
    (:SUBJ
     (:RANGE (:SEM (:HEAD "mann") (:QUANTIFIER "d-det"))
      (:AGR
       ((:PERSON . 3) (:GENDER . :M)
        (:NUMBER . :S) (:CASE . :NOM)))
      (:END . 2) (:START . 0) (:TYPE . :NP)))
    (:OBJ
     (:RANGE (:SEM (:HEAD "frau") (:QUANTIFIER "d-det"))
      (:AGR
       ((:PERSON . 3) (:GENDER . :F)
        (:NUMBER . :S) (:CASE . :NOM))
       ((:PERSON . 3) (:GENDER . :F)
        (:NUMBER . :S) (:CASE . :AKK)))
      (:END . 5) (:START . 3) (:TYPE . :NP)))
    (:NP-MODS)
    (:PP-MODS
     ((:SEM (:HEAD "mit")
            (:COMP (:QUANTIFIER "d-det") (:HEAD "fernrohr")))
      (:AGR ((:PERSON . 3) (:GENDER . :NT)
```

```
        (:NUMBER . :S) (:CASE . :DAT)))
  (:END . 8) (:START . 5) (:TYPE . :PP)))
(:PROCESS
 (:COMPACT-MORPH
  ((:TEMPUS . :PRAES) ... (:GENUS . :AKTIV)))
 (:MORPH-INFO
  ((:TENSE . :PRES) ... (:NUMBER . :S) (:CASE . :NO)))
 (:ART . :FIN) (:STEM . "seh") (:FORM . "sieht")
 (:TYPE . :VERBCOMPLEX))
(:SC-FRAME ((:NP . :NOM) (:NP . :AKK)))
(:START . 0) (:END . 8)
(:TYPE . :SUBJ-OBJ))))
```

The subcategorization lexicon Most of the linguistic information actually used in grammatical function recognition — which is a kind of partial disambiguation of underspecified dependence trees — comes from a subcategorization lexicon. The lexicon we use in our implementation, which now counts more than 25500 entries for German verbs, has been obtained by automatically converting part of the information encoded in the SARDIC lexical database described in [Buchholz, 1996]. In particular, the information conveyed by the verb subcategorization lexicon we use, includes subcategorization patterns, like arity, case assigned to nominal arguments, preposition/subconjunction form for other classes of complements.

In general, a verb has several different subcategorization frames. As an example, consider the different frames associated to the main verb entry *fahr* ("to drive"):

$$\begin{array}{ll} fahr\text{:} & \{\langle np, nom\rangle\} \\ & \{\langle np, nom\rangle,\ \langle pp, dat, mit\rangle\} \\ & \{\langle np, nom\rangle,\ \langle np, acc\rangle\} \end{array}$$

Here, it is specified that *fahr* has three different subcategorization frames. For each frame, the number of subcategorized elements is given (through enumeration) and for each subcategorized element the phrasal type and its case information is given. In case of prepositional elements the preposition is also specified. Thus,a frame like $\{\langle np, nom\rangle, \langle pp, dat, mit\rangle\}$ says, that *fahr* subcategorizes for two elements, where one is a nominative NP and the other is a dative PP with preposition *mit* ("with"). For the elements, there is no ordering presupposed, i.e., frames are handled as sets. The main reason is that German is a free word order language so that the assumption of an ordered frame would suggest a certain word order (or at least a preference).

The main purpose of a (syntactic) subcategorization frame is to provide for syntactic constraints used for the determination of the grammatical functions. Other information of relevance is the state of the sentence (e.g., active vs. passive), the attachment borders of a dependence tree, and the required person and number agreement between verb and subject.

Shallow strategy Directly connected with any analysis of grammatical functions is the problem of possible ambiguity between argumental and adjunct attachment, as well as between the assignment of different possible roles to the same "modifier" (in the special sense explained above). Since all the linguistic processing in our system, thus including GFR, is supposed to be of shallow nature, it does not seem a resonable solution to simply compute *all* the attachment possibilities that the subcategorization information of the verb allows in principle, and to possibly postpone any attempt at solving or reducing the ambiguity to a later stage of the analysis. Rather, what is needed is a kind of *heuristic* resolution of the attachment ambiguity, in order for the GFR to produce unambiguous and reasonably accurate results[8].

The strategy we implemented to deal with this important requirement is a rather minimalistic one: given a set of different subcategorization frames that the lexicon associates to a verb, the structure chosen as the final ("disambiguated") solution is the one corresponding to the *maximal subcategorization frame* available in the set, which is the frame mentioning the largest number of arguments that may be succesfully applied to the input dependence tree.

The grammatical functions recognized by GFR correspond to a set of role labels, implicitely ordered according to an *obliquity hierarchy*:

- SUBJ: deep subject;
- OBJ: deep object;
- OBJ1: indirect object;
- P-OBJ: prepositional object;
- XCOMP: subcategorized subclause.

[8] Obviously, it should also be possible to set a flag to decide if the heuristic disambiguation should be applied or not — in the latter case all the results computed should be simply collected. The GFR module we have implemented in fact allows this possibility.

These label are meant to denote *deep grammatical functions*, such that, for instance, the notion of subject and object does not necessarily correspond to the surface subject and direct object in the sentence. This is precisely the case for passive sentences, whose arguments are assigned the same roles as in the corresponding active sentence.

Processing strategy On the basis of the information given so far, we will now describe the processing method underlying GFR in more detail. The input to GFR is the underspecified dependency tree of a sentence computed through clause recognition (see 3.8 for more details). The main steps are:

1. retrieve the subcategorization frames for the verbal head of the root node of the input dependency tree;
2. apply lexical rules in order to determine deep case information depending on the verb diathesis; since frames are expressed for active sentences only, a passivation rule exists which transforms NP-nominative to NP-accusative, and NP-nominative to PP-accusative with prepositions *von* and *durch*;
3. for each subcat frame SC do:
   (a) match SC with the dependent elements; if matching succeeds, then call SC a valid subcat frame; otherwise SC is discarded;
   (b) if SC is a valid subcat frame and SC_p is the current active subcat frame compute in the previous step of the loop, then if $|\text{SC}| > |\text{SC}_p|$ select SC as the current active subcat frame;
   (c) insert the domain-specific information found for the verbal head of the root (if available); this information can be retrieved from the domain lexicon using the stem entry of the head verb (for more details concerning domain-specific information and template processing see section 4);
4. the same method is recursively applied on sub clauses
5. finally return the new dependency tree marked for deep grammatical function; we call such dependency tree an *underspecified functional description*

The most complex step is 3.(a) where a subcat frame is matched with the dependent elements of a dependency tree. The main purpose of this matching step is the identification of those phrases which satify the constraints of the elements of the subcat frame. For example, in the case of nominal phrases we have to check case, and in the case of PPs case as well as the word form of the preposition must be checked. Note that in the case of a nominative NP we also have to check agreement between the finite head verb and that NP. The main problem for this step are:

- usually phrases are ambiguous wrt. case
- the position of a phrase in a sentence cannot be used for determining grammatical function because of German's free word order
- a flat dependency tree only describes upper bounds for attachment, so that "deeper" phrases might be considered for deep case assigment

Processing is directed basically by looping through the dependent elements and by checking the information of the subcat element with the morpho-syntactic information of the selected phrase. Checking will be performed by means of unification. In order to do this, each subcat element is firstly expanded into a corresponding feature vector, such that all features not mentioned in the subcat element are set to :NO. meaning that the value from the current phrase will be taken over. One exception concerns the subcat element whose case value is nominative. In that case a feature vector is built by merging the case information with that of the morpho-syntactic feature vector of the verbal head element. This is important because the nominative element will be the subject and hence agreement with the finite verb is necessary.

For example, for the sentence *Der Mann sieht die Frau.* ("The men sees the woman"), the following subcat frame is retrieved for the stem *seh* ("to see"): $\{\langle np, nom\rangle, \langle np, acc\rangle\}$. The feature vector for *sieht* "sees" is as follows:

((:TENSE . :PRES) (:FORM . :FIN) (:PERSON . 3)
(:GENDER . :NO)(:NUMBER . :S) (:CASE . :NO))

Combining the subcat frame and the head's feature vector, the following expanded subcat frame results:

{((:TENSE . :PRES) (:FORM . :FIN) (:PERSON . 3)
(:GENDER . :NO) (:NUMBER . :S) (:CASE . :NOM)),

((:TENSE . :NO) (:FORM . :NO) (:PERSON . :NO)
(:GENDER . :NO) (:NUMBER . :NO) (:CASE . :ACC))}

This expanded subcat frame is now used to identify the corresponding nominal phrases from the dependency tree which will receive subject and object roles. Since German has free worder order, we cannot use the position of a phrase in the dependency tree as criterion for identifying functional role. Furthermore, case information alone also does not help in general, because phrases are often highly ambigious especially for nominative and accusative. In order to get a small amount of control information, we actually first check for phrases which only have one case assigned. Those phrases receive directly the corresponding deep case. If no such phrases exists, we use a heuristic such that the leftmost nominative phrase which agrees with the verbal head is assigned the subject role.

Adjuncts All the NPs, PPs and Subclauses that have not been previously recognized as arguments compatible with the selected frame are collected in *adjunct lists*[9]. We distinguish two different sets of adjunct lists; the first one includes three *general adjunct lists*, namely:

- NP-MODS, for modifier NPs;
- PP-MODS for modifier PPs;
- SC-MODS for modifier Subclauses.

After grammatical function recognition, all the three above attributes are present (possibly with an empty list as value) in every dependence tree. Besides them, a second set of optional *special adjunct lists* is available. The most remarkable feature of *special adjunct lists* is that they are not defined as a fixed set, being rather triggered "on the fly" by the presence of some special information in adjunct phrases themselves: in the current implementation, this "special information" is encoded by means of the feature SUBTYPE. It is possible to define specialized finite state grammars (see 3.7), aimed at identifying restricted classes of phrases (for instance, temporal or locational expressions), which impose for that phrases to be treated as special adjuncts by means of the value assigned to SUBTYPE. In the GFR grammar, a set of

[9]They are basically list-valued attribute-value pairs. In the actual format used in STP modules, they are expressed as *associative lists* of the form `(label . (item1 item2 ...))` or the equivalent `(label item1 item2 ...)`.

correspondences between values of SUBTYPE and special adjunct lists can be declared; here is some example:

- {LOC-PP, LOC-NP, RANGE-LOC-PP} $\mapsto$ LOC-MODS
- {DATE-PP, DATE-NP} $\mapsto$ DATE-MODS

This means that, for example, if a modifier marked as LOC-NP is present in an dependence tree, then an attribute LOC-MODS is going to be created in the corresponding underspecified functional description after grammatical function recognition, whose value is a list containing that modifier; then, if another modifier belonging to the same class (say a RANGE-LOC-PP) is found, it is simply added to that list. The same mechanism applies for each class of SUBTYPE values considered by GFR.

The possibility of defining special subgrammars that interact whith GFR in the way we have just described turns out to be very useful in connection to *domain modeling* (see 4.4), in that elements which could have a special relevance for a domain — thus being likely to be linked in highly specific way to template slots — have been already identified in a clear way after GFR.[10] Being those elements already marked and separated from generic ones in an underspecified functional description UFD, such specific links can be expressed in a purely declarative way, so that template slots can be actually filled on the basis of the analysis of a portion of text, without the need of any further processing, only by means of *type expansion* (see 4.2 below).

4 Template Processing

4.1 Overview

As already stated in section 1 above, one of the main goals to be achieved in an information extraction system is the possibility of a *fast application development cycle*. In this section we are going to describe some components of our system architecture that turned out to be crucial for that goal to be achieved. They are:

- a *domain model*, which includes (for each domain being defined, with possible overlaps) a hierarchical definition of templates (see 4.4) and

[10] Depending, of course, on how subgrammars have been actually defined. For the interaction between subgrammars and domain modelling, see also 4.5 below.

possibly a number of specialized fillers expanding the general filler hierarchy (see 4.5);

- a set of relations (*links*) between linguistic and domain model types (see 4.6);
- a *linguistic model*, relying on generic entities such as *relations* (basically predicate-argument structures, see 4.3) and phrases to be used as *fillers* for argumental roles, or as modifiers.

All the above components are actually defined as *hierarchies of TDL types*[11]. Here we want to stress how the design of those modules — let us refer to them collectively as *TDL modules* — contributes to a fast application development cycle. Additionally, the interface module that maps output structures of the shallow text processor into TDL objects (instances) in a parametrizable way, should also be mentioned.

Domain model is of course a critical point, in that it is reasonable to assume that the actual starting point for the adaptation of the system to new domains and/or tasks be precisely the accurate definition of a new domain model. For that reason, the availability of strategies and tools allowing domain model definition and updating in the most modular way, including the reuse of all the useful knowledge already encoded in the system, strongly influences the concrete possibility of using the system in a fast development cycle. The object-oriented domain modeling strategies we suggest in 4.4 below, together with the use of the TDL language, is aimed at providing the user with the maximum of flexibility in domain design, on the assumption that domain modeling should be as much independent as possible both on the actual system implementation (completely transparent to the user in the ideal case), and on the separately defined linguistic knowledge.

Linking model A typical weakness of "traditional" information extraction systems, as we have already pointed out in 1, lies in the fact that linguistic and domain-specific knowledge are tightly interconnected. On the contrary, in our system we have attempted to keep the maximal degree of separation between those two dimensions, both being declaratively expressed by means of independent hierarchies of types. This kind of modular

[11] Some motivation in support of the choice of TDL as modelling language are presented in the subsequent section 4.2.

organization of the knowledge used by the system is profitable, in that it provides the user with a clear picture of what is actually present in the system's knowledge base, thus making the task of adapting the system to new domain easier; on the other hand, it is clear that, at a certain point, a level of knowledge providing a connection between linguistic (see below) and domain-oriented knowledge is required to make the system operatively usable. Therefore we provide another knowledge dimension to the system, which we label *linking model*. This additional model, like the other $\mathcal{TDL}$ modules described here, is defined in a fully declarative way, i.e. in the form of a hierarchy of *linking types* (see 4.6 for a more detailed description), which map particular linguistic types to particular relational domain entities (templates), in such a way that part of the information contained in phrases used as argumental fillers or modifiers is actually linked to specific slots declared in a template. Such a mechanism cooperates in a very efficient way with the overall organization of the information in the system: the restrictions imposed by a linking type on the type of the linguistic structure, as well as on the allowable fillers, can be located at any point in the respective hierarchies — i.e. either intermediate or terminal nodes are admitted. In this way, it is very easy to tune the system's behaviour — for instance after having checked the results of a test corpus analysis — by changing the degree of specificity of any restriction declared in a linking type.

Linguistic model In our system it is basically intended as an abstraction of the linguistic structures computed by lower-level modules. The information available after shallow text processing is now modeled in a more transparent and understandable way, where the basic entities available (fillers and relations) are described according to an object-oriented view (a multiple-inheritance network of types acting as classes governs and constrains the actual *instantiation* of structures describing parts of the analyzed text). Moreover, a clear boundary separating general entities and domain-specific ones is drawn at this level. This means, for instance, that it should be immediately clear — just by inspecting how the current linguistic model is defined — whether new entities (typically new classes of fillers which could be associated to specific syntactic patterns) need to be introduced in order either to complete the model of an existing domain, or to add a completely new domain. Of course, the definition of new entities specializing the linguistic model is not sufficient, as the associated syntactic patterns need to be captured by means of specific *subgrammars* at the shallow text processor, to be separately implemented. However, we claim that the existence of such an abstract linguistic model, whose partitions basically mirror partitions among different STP subgrammars, contributes in a significant way to the modularity of the system, thus allowing a much more consistent and fast development and maintenance of subgrammar themselves.

4.2 Use of $\mathcal{TDL}$ language

We assume that the adoption of a *typed feature structure*-based formalism like $\mathcal{TDL}$ (for a detailed description, the reader is referred to [Krieger and Schäfer, 1994a]) as the knowledge representation language used by the system provides the user with a very flexible and powerful interface; moreover, besides the modularity of the formalism itself, $\mathcal{TDL}$ environment is equipped with a powerful inferential mechanism based on *type expansion* and *unification*, which turns out to be of crucial importance for the approach to IE we are presenting.

Here is a short list of useful features of $\mathcal{TDL}$ for our application tasks:

- completely declarative user language;
- use of a single data structure, namely *typed feature structures*;
- multiple inheritance hierachies;
- unification;
- negation, disjunction, distributed disjunction;
- possibility of customizing¿ the mechanism of type expansion(for instance *open world reasoning* vs. *closed world reasoning*).

The first point, namely the *declarativity* of the language, is of course of crucial importance when a reasonably user-friendly interface to the system is required. We claim that a well-specified TFS-based language like $\mathcal{TDL}$ requires less training before being successfully used by a knowledge engineer in order to define IE tasks in new domains and subdomains, compared to the very long and specialized training needed before being able to modify — even slightly — a "traditional" IE system (see section 1 above).

The advantages of a highly expressive representation language, provided with feature structures as uniform data structure, a fully user-definable type system, multiple inheritance and complex constraints over feature structures, is quite self-evident; however, it will become even clearer from the description of our general/domain-specific modelling architecture.

The core idea behind the knowledge modelling in terms of $\mathcal{TDL}$ types is the possibility of:

1. defining a level of *general knowledge* by abstracting over the information computed by the STP. This is made by means of different *hierarchies*, which represent a model of the information computed by the previous shallow analysis at different levels: for instance, an ontology of different predicate-argument structures (see 4.3) is a way of organizing the possible output structures of the grammatical function recognition module, while an ontology of general and specialized phrasal types (see 4.5) is closely related to the information computed by the fragment recognition module (see section 3 above). The connection between these two dimensions is accomplished by the use of attributes within the predicate-argument structure types, whose values are restricted to particular phrasal types.

2. modelling of the domain specific knowledge by means of appropriate, non-linguistic types representing the entities/relations typical of that domain.

3. explicitely defining the connection between the two dimensions above, namely the ontologies of linguistic types on the one hand, and of domain types on the other. The relevant objects performing this crucial task are what we call *linking types* (see 4.6); they are hierarchically organized, and they are pointed at by the domain lexicon DL (see 2).

Finally, the possibility of controlling the basic inference mechanism of $\mathcal{TDL}$, namely the *type expansion*, through of a number of user-settable switches, contributes to the flexibility of the system and allows the user to make a more fine-grained tuning of the knowledge base for a particular domain, e.g. by changing the filtering power of the type system by setting the *open/closed-world reasoning* flag, or by limiting the possibility of recursive type expansion and so forth.

4.3 Linguistic types

As already pointed out, the most general level of knowledge used in our system is a *linguistic oriented ontology*, where the word *ontology* is used in a sense quite close to the one proposed in the *Generalized Upper Model* ([Bateman, 1992], [Bateman, 1993]). In our system, this kind of model is of course simplified, due to the fact that the amount of linguistic information we want to come into play is reduced compared to the rather complete *lexico-grammar* assumed in the Generalized Upper Model. In fact, in order to guarantee the degree of efficiency and robustness required by information extraction tasks, we chose to rely on a *shallow* linguistic analysis on the one hand, and to minimize the use of lexical information on the other.

The linguistic-oriented ontology we implemented (using $\mathcal{TDL}$ types) basically consists of two separate hierarchies:

- a hierarchy of *relations*;
- a hierarchy of *fillers*.

The second one, namely the hierarchy of fillers, is an abstract model of the different classes of functional descriptions computed by previous modules of the system (see 3.5, 3.9). In particular, it is aimed at modelling subsentential *phrases* as recognized in the shallow text processor. While the top level of this hierarchy belongs to the linguistic ontology, in the sense that partitions among different classes of phrases are established on a domain-independent basis, it is possible to *specialize* the hierarchy by expanding the general types into more specific, and therefore domain-oriented, subtypes, as we will explain in some more detail in the subsequent section 4.5.

The hierarchy of *relations*, or predicative types, on the other hand, should be viewed as an abstraction built on the basis of the different kinds of predicative lexical entries encoded in a *subcategorization lexicon*. In particular, the information conveyed by the verb subcategorization lexicon we use(see 3.9), includes subcategorization patterns, like arity, case assigned to nominal arguments, preposition/subconjunction form for other classes of complements.

But, for the time being, only the very top level of the hierarchy of relations has been already developed; every relation introduces, or inherits from one or more of its supertypes, at least one argumental slot (labelled as SUBJ, OBJ, OBJ1 and so on). The ontology is then conceived as a multiple inheritance network, which accounts for the relative obliquity of basic functional roles. Concretely, for each of those functional role, a non-terminal node is introduced in the relations ontology, aimed at introducing it as a labelled slot within the feature structure representing the relation. The terminal nodes in the hierarchy, i.e. relational feature structures typed according to classes of correlated subcategoriztion frames in the lexicon, inherit their roles from one or more intermediate nodes. The legal values for the argumental slots intoduced in intermediate or terminal relational types must belong to the hierarchy of fillers, in such a way that it is possible to define restrictions when an argumental slot is introduced, as well as to restrict the value of previously introduced slots.

4.4 The Domain Model: Templates and fillers

As the use of simple text annotation schemata based on SGML or other markup languages is only suitable for very basic information extraction tasks, (a typical example could be the extraction of company names), *structured templates* have become the most common way of representing the information extracted from texts in IE systems. A structured template looks very much like a database record: basically a set of slots, whose possible fillers range from strings of text to numbers or predefined symbols.

Yet, the increasing complexity of information extraction tasks on one hand, and the requirement of modularity and flexibility we have already stressed on the other, suggest a further advance in the view of templates and of their design criteria, by adopting an *object oriented* approach[12], which in principle allows a more compact and consistent modelling of the different, specialized templates to be used in an information extraction application, where the relationships among them are made clear and explicit.

As for the theoretical status of templates, an important point is outlined in [Hobbs, 1994]: a template should be conceived as *representational device*, such that the problem of modelling templates is just a subtopic of knowledge representation. Thus, what structured IE templates, expressed in a constrained formalism, represent, can be clearly pointed out as:

> ...essential facts about situations in a way that can mediate between texts that describe those situations and a variety of applications that involve reasoning about them.
>
> ([Hobbs, 1994], pag. 1)

In the light of this strong assumption about the nature and the theoretical status of templates as part of a knowledge representation model, ¿an

[12] The distinction between *text annotation* and *templates*, distinguished in turn in *flat templates* (roughly corresponding to what we call *structured templates*) and *object-oriented templates*, comes from [Onyshkevych,].

explicit definition of the basic entities that populate the particular world defined by a specific IE context, as well as their relation with the basic entities that can be assumed as stable across the whole set of different possible domains, becomes of crucial importance.

As already mentioned in 4.2 above, we use $\mathcal{TDL}$ types in order to model both the general and the domain-specific knowledge used by the system, in such a way that it can be easily extended or adapted to new domains or subdomains. The basic entities we assume for domain-specific modelling are *event templates*, and *fillers*. Event templates and fillers are organized in two different hierarchies. The core idea in our template modelling mechanism is that it should be possible for the user to define templates from a pure knowledge representation perspective, independently on linguistic considerations. Then, the connection between the representation model (templates and fillers) and the linguistic structures produced by the preceding shallow analysis will be stated independently, by defining a furter hierarchy of *linking types* (see. 4.6 below).

Event templates are defined as feature structures whose attributes correspond to template slots. The use of a typed language like $\mathcal{TDL}$ allows, of course, the possibility of restricting the possible filler for any attribute to a particular value.

The advantage of hierarchically organizing the design of *event templates* can be made clear by examining, for example, a particular information extraction domain whose model includes a number of different, specialized templates. Among the set of attributes that are relevant for a particular template, some could be shared with other templates, while some others are really specific to the template. If the templates were defined as standalone feature structures, there would be an obvious redundancy, as well as a high probability of inconsistencies once the model have become more complex. The hierarchical approach to template modelling allows, on the other hand, to introduce very general slots (obvious examples could be slots like `date` or `place`) in a general template type. When a more specialized template is declared as a subtype (not necessarily an immediate one) of the more general type, all of its attributes are inherited, without the need of a further declaration, so that consistency is guaranteed. Moreover, this mechanism allows to specialize the legal value of the inherited attribute by restricting the class of its value in a monotonic way (i.e. when a type inherits a slot from one of its ancestors, expressed as an attribute-value pair, it is always possible to restrict the value to a more specific, but still compatible, value — a subtype of the value declared in the ancestor).

4.5 General *vs.* domain-specific phrasal types

The hierarchy of possible fillers is conceptually split in two layers. The first one, which constitutes the top level of the filler hierarchy, is basically a hierarchy of general phrasal types. The organizing criterion for this level of the phrase ontology is a linguistic one. To be more precise, this level of the hierarchy is an abstract model of the distinctions computed by the shallow linguistic component at the level of the recognition of subsentential constituents.

At a deeper level in the hierarchy it is possible to attach more specific knowledge in terms of *specialization* of the higher (linguistic oriented) types. In order to clarify the distinction between the two levels, let us take nominal phases (NPs) as an example. A classification of referential NPs depending on their head noun, either a common noun or a proper name, is likely to hold in general, in a completely independent way wrt. the domain a text belongs to. It is also likely that the syntactic criteria that allow the shallow analysis component is able to identify proper names and to separate them from normal NP's are still valid across different domains. If this turns out to be correct, including the distinction between proper names and normal NP's as part of the more general phrase hierarchy seems to be a quite natural choice.

However, even though the proper name *vs.* common noun distiction holds for every domain, it is possible that, for a particular domain, a more fine-grained distinction among different classes of proper names is required, either because distinct subclasses of proper names identify different entities in that domain, or because only a subset of all the possible proper names is relevant for the domain. In that case, a *specialization* mechanism is used: more specific daughters of the general node `proper-noun` are defined for the domain at stake.

Once more domains have been defined through the specialization mechanism described above, it is reasonable to expect that some of the new specialized individuals are shared among different domains, while some others are really specific to one domain. This means that domains are viewed as sets of entities, that are allowed to partially overlap, without any need to explicitely define *a priori* boundaries to separate entities belonging to one domain rather than to another one. In summary, the easy parametrization of the system is guaranteed through:

- the possibility of fully reusing any part of knowledge already defined for new domains when appropriate.

- the use of specialization mechanism for defining or improving entity domains; it can either be used in order to add completely new domains, or it can be incrementally applied to any of the existing domains to make it more exhaustive or to partition it into more specific sub-domains.

4.6 Linking Types

The task of bridging the domain specific models, defined by the user in terms of updating and specialization of the hierarchies of templates and fillers, to the general linguistic knowledge is performed in our prototype by explicitely defined *linking types*. They are characterized by the presence of two different slots, called `syn` and `template` in the current implementation. The value of `syn` is required to be a structure belonging to the predicate-argument structure hierarchy; `template` values have to belong to the hierarchy of relational templates.

Three main components are relevant to the definition of a particular linking type (which counts as a *domain specific lexical type* in our system, as already mentioned). They are:

- the type restriction of `template` value;
- the type restriction of `syn` value;
- a set of identities established among paths internal to `syn` and `template` value respectively.

Among these three components, the first one is of course crucial. Since a linking type is a prototype for domain-specific lexical entries in the domain lexicon DL (see 2), this kind of restriction actually corresponds to a lexical selection of a particular template to be filled.

Depending on how the template hierarchy is organized, the type assigned to `template` can be restricted to a leaf node of the template hierarchy, or also an intermediate one. This depends on how the template hierarchy is organized: for instance, in the template hierarchy we experimentally implemented, templates to be filled strictly correspond to leaf nodes, while intermediate nodes are aimed at providing a non-redundant way of encoding pieces of information being shared among different templates. According to this modelling style, the linking types should restrict `template` to a leaf type (namely, one of the types that actually count as templates). However, since the task of template modelling is up to the user who defines the information extraction tasks to be performed, nothing prevents the choice of a different approach; for instance, one that allows some of the intermediate nodes of the template hierarchy to be selected as *final* templates, whose subtypes count as more specific "subtemplates".

As for the restriction of the `syn` value, intermediate types are more likely to be used than leaves. The use of a terminal type would actually impose a very strict constraint, such that a template is associated with *exactly one* predicate-argument structure type. While there might be particular situations in which exactly this one-to-one association is desirable, in the most frequent cases a linking type would probably impose weaker contraints on the type of predicate-argument structure. As a typical case, we can immagine that a linking type might require *at least* certain specified roles to be instantiated, irrespective of the presence of *other* roles. As the hierarchy of PAS types is designed as a multiple inheritance network, with intermediate nodes introducing different role slots one by one (see 4.3) , the use of one of them — let us say `obj` for example — as type restriction on `syn` value within a linking type would make it compatible with *any* PAS type which includes at least the `obj` role. Moreover, as $\mathcal{TDL}$ allows conjunctive type restrictions to be declared, a linking type can restrict its `syn` value to the conjunction of two — or more — different intermediate types, being therefore compatible with all the PAS types they subsume.

Finally, the third component we identified is the one that justifies the use of the label "linking types". Actually, the identity statements among different substructures are exactly the way in which *links* between (relevant portions of) linguistic information and the appropriate slots in the template structure are defined. The mechanism involved here, which leads to the the actual *filling* of a template on the basis of the output structure delivered by the preceding STP modules, is *type expansion.* Variable identities in the declaration of a $\mathcal{TDL}$ type count as actual *structure sharing* relationships after type expansion has taken place. So the outcome of type expansion applied to a newly defined linking type will be a structure where all the possible inferences resulting from the general type system have been drawn[13], and a number of structure-sharing relations have been explicitely stated.

[13]Concretely, this means that attributes are inherited from all the ancestor, and in same cases, i.e. with particular applications of *closed-world reasoning* as well as sort types, more specific subtype are inferred by type unification. For details on how exactly type expansion works, the reader is of course referred to the available literature about $\mathcal{TDL}$, in particular [Krieger and Schäfer, 1994a].

Examples showing both the local definition of some linking type, and the way they look after type exapnsion can be found in appendix A.1 below.

A linking type is a statical definition of a relation between a "syntactic" structure and a template. Just to understand how it turns out to be useful for an operative, dynamic process like filling a new template on the basis of the analysis of a sentence performed by the STP, the distinction between a *type* and an *instance* must be taken into account. What is actually passed to $\mathcal{TDL}$ module as input is an *unexpended instance*, namely a $\mathcal{TDL}$ object storing all the relevant information computed during the preceding STP as the value of `syn` attibute; moreover, as the result of looking up the domain-specific lexicon DL, the object is *typed* according to one of the existing linking types. Up to now, this object contains only linguistic information and the type name. The mechanism which makes possible for a template to be filled is the so called *instance expansion*; the unexpanded object, or *local instance*, is merged with the correspondent (expanded) type. Only if no conflicting information is found, i.e. the local instances observes all the constraints stated in the type definition, the expansion succeeds, otherwise it fails (the expanded instance is not produced at all).

As the result of the succsessful expansion of an instance, the value of `template` attribute is filled with the relevant substructures found in the "syntactic" substructure, according to what the structure sharings defined in the linking type impose. An example of the effect of expanding an instance can be found in A.1 below: an unexpanded instance is shown in figure 4, while figure 5 shows how the same instance looks like after expansion. What is very important to notice here is the resulting instantiation of `template` value.

5 Implementation and Performance

5.1 Coverage

Company names Three different blind evalutions have been performed:

- corpus: TAZ
  - expressions in corpus: 144
  - found by SMES: 87
  - recall: 60%, precision: 90%
- corpus: 50kB economy news

- trial 1 (stress on recall):
  recall 70%, precision: 82%
- trial 2 (strees on precision):
  recall 63%, precision: 91%

Nominal phrases Tagged corpus of 396 NPs (1700 words). The tag set included POS, case, gender, number. Although the corpus included complex tagged NPs, we only considered the simple NPs, i.e., NPs with no modifier attachment (because the shallow text processor STP explicitly does not perform any attachment on the phrasal level). Note further that STP additionaly determines a head/modifier structure for each NP. However, we only measured correct brackets. The result:

- recall: 78.6%, precision: 99.5%

Verb groups

- corpus: 200 sentences (with 3600 words)
- number of tagged expressions (including type of verbgroup): 451
- found: 472, where
- correct: 428, wrong: 7, not found: 16
- recall: 94.9%, precision: 90.68%

6 Implementation

currently Lisp, but C++ version under development;

7 Related work

GATE, MUC-7, etc

8 Conclusion and Future Work

A Details of domain modeling

A.1 Linking Types

```
(:Type KAMPF-DOMAIN-DAMAGING)
[SYN      : (:Type OBJ-NODE)
            [INDEX     : (:Type EVENT) []
             PROCESS   : [TEMPLATE : %1
                          STEM     : %3]
             DATE-MODS : %4
             LOC-MODS  : %2
             SUBJ      : [RANGE : [SEM : %5]]
             OBJ       : (:Type THETA-ROLE)
                         [DOMAIN : (:Type INDEX) []
                          RANGE  : (:Type PHRASE) [SEM : %6]]]
 TEMPLATE : %1 =(:Type FIGHTING-TEMPLATE)
               [PLACE    : %2 =[]
                ACTION   : %3 =[]
                DATE     : %4 =[]
                VICTIM   : (:Type SEM) []
                ATTACKER : %5 =(:Type SEM) []
                ATTACKED : %6 =(:Type SEM) []]]
```

Figure 3: Example of an expanded *linking type*

$$
\left[\begin{array}{ll}
\textsf{kampf} - \textsf{domain} - \textsf{kill} & \\
\textsc{process} & \left[\begin{array}{ll}
\textsf{verb} & \\
\textsc{subtype} & [\,] \\
\textsc{morph-info} & [\,] \\
\textsc{form} & \textsf{string} \\
\textsc{template} & \boxed{6} \left[\begin{array}{ll}
\textsf{fighting} - \textsf{template} & \\
\textsc{place} & \boxed{4}\,[\,] \\
\textsc{action} & \boxed{2}\,[\,] \\
\textsc{date} & \boxed{3}\,[\,] \\
\textsc{attacker} & \boxed{5}\,\textsf{sem} \\
\textsc{attacked} & \boxed{1}\,\textsf{sem} \\
\textsc{victim} & \boxed{1}
\end{array}\right] \\
\textsc{stem} & \boxed{2}
\end{array}\right] \\
\textsc{date-mods} & \boxed{3} \\
\textsc{loc-mods} & \boxed{4} \\
\textsc{subj} & \left[\begin{array}{ll}
\textsf{actor} & \\
\textsc{domain} & \left[\begin{array}{ll}
\textsf{configuration} & \\
\textsc{process} & \left[\begin{array}{ll}
\textsf{verb} & \\
\textsc{form} & \textsf{string} \\
\textsc{morph-info} & [\,] \\
\textsc{stem} & [\,] \\
\textsc{subtype} & [\,]
\end{array}\right]
\end{array}\right] \\
\textsc{range} & \left[\begin{array}{ll}
\textsf{np} & \\
\textsc{sem} & \boxed{5}
\end{array}\right]
\end{array}\right] \\
\textsc{obj} & \left[\begin{array}{ll}
\textsf{actee} & \\
\textsc{domain} & \left[\begin{array}{ll}
\textsf{configuration} & \\
\textsc{process} & \left[\begin{array}{ll}
\textsf{verb} & \\
\textsc{form} & \textsf{string} \\
\textsc{morph-info} & [\,] \\
\textsc{stem} & [\,] \\
\textsc{subtype} & [\,]
\end{array}\right]
\end{array}\right] \\
\textsc{range} & \left[\begin{array}{ll}
\textsf{phrase} & \\
\textsc{sem} & \boxed{1}
\end{array}\right]
\end{array}\right] \\
\textsc{template} & \boxed{6}
\end{array}\right]
$$

Figure 4: Example of an unexpanded instance

$$
\begin{bmatrix}
\textsf{kampf} - \textsf{domain} - \textsf{kill} & \\
\textit{:expanded false} & \\
\textsc{syn} & \begin{bmatrix}
\textsf{subj} - \textsf{obj} & \\
\textit{:expanded false} & \\
\textsc{subj} & \begin{bmatrix}
\textit{:expanded false} & \\
\textsc{range} & \begin{bmatrix}
\textsf{np} & \\
\textit{:expanded false} & \\
\textsc{sem} & \begin{bmatrix}
\textit{:expanded false} & \\
\textsc{head} & \langle \texttt{"mann"} \rangle \\
\textsc{quantifier} & \langle \texttt{"ein"} \rangle
\end{bmatrix}
\end{bmatrix}
\end{bmatrix} \\
\textsc{obj} & \begin{bmatrix}
\textit{:expanded false} & \\
\textsc{range} & \begin{bmatrix}
\textsf{np} & \\
\textit{:expanded false} & \\
\textsc{sem} & \begin{bmatrix}
\textit{:expanded false} & \\
\textsc{head} & \langle \texttt{"soldat"} \rangle \\
\textsc{quantifier} & \langle \texttt{"d-det"} \rangle
\end{bmatrix}
\end{bmatrix}
\end{bmatrix} \\
\textsc{np-mods} & \langle \rangle \\
\textsc{loc-mods} & \left\langle \begin{bmatrix}
\textsf{loc} - \textsf{pp} & \\
\textit{:expanded false} & \\
\textsc{sem} & \begin{bmatrix}
\textsc{place} & \texttt{"Mostar"} \\
\textsc{loc-prep} & \texttt{"in"}
\end{bmatrix}
\end{bmatrix} \right\rangle \\
\textsc{sc-mods} & \langle \rangle \\
\textsc{process} & \begin{bmatrix}
\textsf{verbcomplex} & \\
\textit{:expanded false} & \\
\textsc{morph-info} & \left\langle
\begin{bmatrix}
\textsc{tense} & \texttt{pres} \\
\textsc{person} & \texttt{2} \\
\textsc{gender} & \texttt{no} \\
\textsc{number} & \texttt{s} \\
\textsc{case} & \texttt{no}
\end{bmatrix},
\begin{bmatrix}
\textsc{tense} & \texttt{pres} \\
\textsc{person} & \texttt{3} \\
\textsc{gender} & \texttt{no} \\
\textsc{number} & \texttt{s} \\
\textsc{case} & \texttt{no}
\end{bmatrix},
\begin{bmatrix}
\textsc{tense} & \texttt{pres} \\
\textsc{person} & \texttt{2} \\
\textsc{gender} & \texttt{no} \\
\textsc{number} & \texttt{p} \\
\textsc{case} & \texttt{no}
\end{bmatrix}
\right\rangle \\
\textsc{art} & \texttt{fin} \\
\textsc{stem} & \texttt{"erschiess"}
\end{bmatrix} \\
\textsc{sql-type} & \texttt{gf-verb-node}
\end{bmatrix}
\end{bmatrix}
$$

Figure 5: The instance of figure 4 after expansion

$$
\begin{bmatrix}
\mathsf{kampf} - \mathsf{domain} - \mathsf{kill} & \\
\textsc{template} & \begin{bmatrix}
\mathsf{fighting} - \mathsf{template} & \\
\textsc{place} & \boxed{4} \left\langle \begin{bmatrix} \mathsf{loc} - \mathsf{pp} & \\ \textsc{sem} & \begin{bmatrix} \mathsf{sem} & \\ \textsc{place} & \texttt{"Mostar"} \\ \textsc{loc-prep} & \texttt{"in"} \end{bmatrix} \end{bmatrix} \right\rangle \\
\textsc{action} & \boxed{5}\,\texttt{"erschiess"} \\
\textsc{date} & \boxed{2}\,[\,] \\
\textsc{attacker} & \boxed{3} \begin{bmatrix} \mathsf{sem} & \\ \textsc{head} & \langle \texttt{"mann"} \rangle \\ \textsc{quantifier} & \langle \texttt{"ein"} \rangle \end{bmatrix} \\
\textsc{attacked} & \boxed{1} \begin{bmatrix} \mathsf{sem} & \\ \textsc{head} & \langle \texttt{"soldat"} \rangle \\ \textsc{quantifier} & \langle \texttt{"d-det"} \rangle \end{bmatrix} \\
\textsc{victim} & \boxed{1}
\end{bmatrix} \\
\textsc{syn} & \begin{bmatrix}
\mathsf{subj} - \mathsf{obj} & \\
\textsc{date-mods} & \boxed{2} \\
\textsc{index} & \mathsf{event} \\
\textsc{subj} & \begin{bmatrix} \mathsf{theta} - \mathsf{role} & \\ \textsc{domain} & \mathsf{index} \\ \textsc{range} & \begin{bmatrix} \mathsf{np} & \\ \textsc{sem} & \boxed{3} \end{bmatrix} \end{bmatrix} \\
\textsc{obj} & \begin{bmatrix} \mathsf{theta} - \mathsf{role} & \\ \textsc{domain} & \mathsf{index} \\ \textsc{range} & \begin{bmatrix} \mathsf{np} & \\ \textsc{sem} & \boxed{1} \end{bmatrix} \end{bmatrix} \\
\textsc{np-mods} & \langle\,\rangle \\
\textsc{loc-mods} & \boxed{4} \\
\textsc{sc-mods} & \langle\,\rangle \\
\textsc{process} & \begin{bmatrix}
\mathsf{verbcomplex} & \\
\textsc{subtype} & [\,] \\
\textsc{form} & \mathsf{string} \\
\textsc{morph-info} & \left\langle \begin{bmatrix} \textsc{tense} & \texttt{pres} \\ \textsc{person} & \texttt{2} \\ \textsc{gender} & \texttt{no} \\ \textsc{number} & \texttt{s} \\ \textsc{case} & \texttt{no} \end{bmatrix}, \begin{bmatrix} \textsc{tense} & \texttt{pres} \\ \textsc{person} & \texttt{3} \\ \textsc{gender} & \texttt{no} \\ \textsc{number} & \texttt{s} \\ \textsc{case} & \texttt{no} \end{bmatrix}, \begin{bmatrix} \textsc{tense} & \texttt{pres} \\ \textsc{person} & \texttt{2} \\ \textsc{gender} & \texttt{no} \\ \textsc{number} & \texttt{p} \\ \textsc{case} & \texttt{no} \end{bmatrix} \right\rangle \\
\textsc{art} & \texttt{fin} \\
\textsc{stem} & \boxed{5}
\end{bmatrix} \\
\textsc{sql-type} & \texttt{gf-verb-node}
\end{bmatrix}
\end{bmatrix}
$$

References

[Aho *et al.*, 1983] A. Aho, J. Hopcraft, and J. Ullmann. *Data structures and and algorithms.* Addison Wesley, Reading, Mass., 1983.

[Aone and Hausman, 1996] C. Aone and K. Hausman. Unsupervised learning of a rule-based Spanish part of speech tagger. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING)*, pages 53–58, Kopenhagen, Denmark, Europe, 1996.

[Appelt *et al.*, 1993] D. Appelt, J. Hobbs, J. Bear, D. Israel, and M. Tyson. Fastus: A finite state processor for information extraction from real world text. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, Chambery, France, August 1993.

[Bateman, 1992] J. A. Bateman. The theoretical status of ontologies in natural language processing. In *Text Representation and Domain Modelling*, KIT-Report 97. Technische Universität Berlin., 1992.

[Bateman, 1993] J. A. Bateman. Ontology construction and natural language. In *Proceedings of the International Workshop on Formal Ontology*, 1993.

[Brill, 1995] E. Brill. Unsupervised learning of disambiguation rules for part of speech tagging. In *Very Large Corpora Workshop*, 1995.

[Buchholz, 1996] Sabine Buchholz. Entwicklung einer lexikographischen Datenbank für die Verben des Deutschen. Master's thesis, Universität des Saarlandes, Saarbrücken, 1996.

[Declerck *et al.*, 1998] Thierry Declerck, Judith Klein, and Guenter Neumann. Evaluation of the nlp components of an information extraction system for german. In *Proceedings of the first international Conference on Language Resources and Evaluation (LREC) 1998*, pages 293–297, Granada, 1998.

[Finkler and Neumann, 1988] W. Finkler and G. Neumann. Morphix: A fast realization of a classification–based approach to morphology. In H. Trost, editor, *Proceedings der 4. Österreichischen Artificial–Intelligence Tagung, Wiener Workshop Wissensbasierte Sprachverarbeitung*, Berlin, August 1988. Springer.

[Gardent and Webber, 1998] C. Gardent and B. Webber. Describing discourse semantics. In *Proceedings of the 4th International Workshop Tree Adjoining Grammars and Related Frameworks (TAG+4)*, pages 50–53, University of Pennsylvania, PA, 1998.

[Hobbs, 1994] J. Hobbs. Principles of template design. 1994.

[Koskenniemi, 1984] K. Koskenniemi. A general computational model for word-form recognition and production. In *Proceedings of the 10th International Conference on Computational Linguistics and the 22nd Annual Meeting of the Association for Computational Linguistics (COLING)*, pages 178–181, Stanford University, California, 1984.

[Krieger and Schäfer, 1994a] H. Krieger and U. Schäfer. Tdl - a type description language for hpsg. Technical report, DFKI, 1994.

[Krieger and Schäfer, 1994b] Hans-Ulrich Krieger and Ulrich Schäfer. $\mathcal{TDL}$—a type description language for constraint-based grammars. In *Proceedings of the 15th International Conference on Computational Linguistics, COLING-94*, pages 893–899, 1994.

[Krieger, 1987] Hans-Ulrich Krieger. Nil—eine lisp-basierte natürlichsprachliche schnittstelle zu ramses. unterstützung der nmr-diagnostik von hirn- und mammatumoren. Master's thesis, RWTH Aachen, 1987.

[Muskens and Krahmer, 1998] R. Muskens and E. Krahmer. Description theory, ltags and underspecified semantics. In *Proceedings of the 4th International Workshop Tree Adjoining Grammars and Related Frameworks (TAG+4)*, pages 112–115, University of Pennsylvania, PA, 1998.

[Neumann *et al.*, 1997] G. Neumann, R. Backofen, J. Baur, M. Becker, and C. Braun. An information extraction core system for real world german text processing. In *5th International Conference of Applied Natural Language*, pages 208–215, Washington, USA, March 1997.

[Onyshkevych,] B. Onyshkevych. Issues and methodology for template design for information extraction.

[Roche and Schabes, 1995] E. Roche and Y. Schabes. Deterministic part-of-speech tagging with finite state transducers. *Comnputational Linguistics*, 21(2):227–253, 1995.