Language Engineering in Dialogue Systems

Hugo ter Doest, Mark Moll, René Bos,
Stan van de Burgt and Anton Nijholt

Memoranda Informatica 96-2
January 1996

ISSN 0924-3755

University of Twente
Department of Computer Science
P.O. Box 217
7500 AE Enschede
The Netherlands

Order-adress: University of Twente
TO/INF library
The Memoranda Informatica Secretary
P.O. Box 217
7500 AE Enschede
The Netherlands
Tel.: 053-4894021

Language Engineering in Dialogue Systems

Memoranda Informatica 96-2

Hugo ter Doest†, Mark Moll†, René Bos†,
Stan van de Burgt‡ and Anton Nijholt†

†: Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
e-mail: {terdoest,moll,bosd,stan,anijholt}@cs.utwente.nl

‡: KPN Research
P.O. box 421, Leidschendam, The Netherlands
e-mail: S.P.vandeBurgt@research.kpn.com

January 1996

Abstract

The analysis of natural language in the context of keyboard-driven dialogue systems is the central issue addressed in this paper. A module that corrects typing errors, performs domain-specific morpholical analysis is developed. A parser for typed unification grammars is designed and implemented in C++; for description of the lexicon and the grammer a specialised specification language is developed. It is argued that typed unification grammars and especially the newly developed specification language are convenient formalisms for describing natural language use in dialogue systems. Finally we present a dialogue manager that is based on a finite state automaton; transitions in the automaton depend upon availability of information in utterances of the user. In order to keep track of the history of the dialogue, a context stack is constructed in the course of the dialogue; for each utterance a new context (possibly an increment of an old context) is pushed on the stack. Given an utterance, the manager may decide to pop a context from the stack that was pushed on the stack earlier in the dialogue. The manager is implemented in Prolog.

Research on these issues is carried out in the context of the SCHISMA project, a research project in language engineering; participants in SCHISMA are KPN Research (the R&D department of Royal PTT Nederland) and the University of Twente.

The aims of the SCHISMA project are twofold: both the accumulation of knowledge in the field of computational linguistics and the development of a natural language interfaced theatre information and booking system is envisaged. The SCHISMA project serves as a testbed for the development of the various language analysis modules necessary for dialogue systems.

1 Introduction

A dialogue system is a system that allows users to communicate with an information system by means of a dialogue using a natural language. Such systems always provide their users information about a particular restricted domain, information that is stored in a data or knowledge base, like for instance a travel information system. Within our Parlevink language-engineering project we are currently developing a natural language interfaced theater information and booking system.

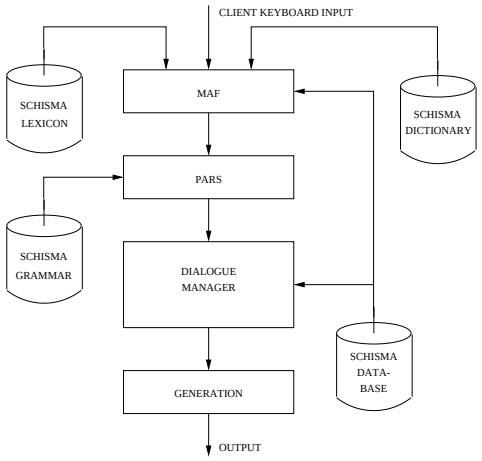


Figure 1: **Global architecture of Schisma.**

A dialogue system as meant here has several aspects: extracting the relevant information from the user input, dialogue management, user- and dialogue modeling, information retrieval from the database, and language generation. A global architecture of the SCHISMA dialogue system is shown in figure 1. It is the architecture of a first prototype of the dialogue system we are currently developing. The actual status of a dialogue does not dynamically influence the process of parsing, nor the preprocessing of the input in the module MAF handling Morphological Analysis and Fault correction. The system processes user input typed in on a keyboard. The MAF module accounts for typing errors in the input, detects word boundaries, use of punctuation and tags the distinguished lexical units with their syntactic type (i.e. category). Details on the MAF module are presented in section 3.

The parser is that part of the dialogue system that should identify the relevant semantic information communicated by the user. It outputs one (ideally) or several (in case of syntactic ambiguities) analyses of the input from the MAF module. The dialogue manager then selects the most likely analysis given the status of the current dialogue. From experiments with a semi-automatic system, in which selected persons set up a dialogue with a Wizard, we gained insight in how users express their wants, information, and answers in Dutch when they think they are communicating with a machine. Analyses of the corpus of dialogues resulting from these experiments, known as Wizard of Oz experiments, not only give answers to questions like "what words or phrases do users use" but they also point out that users make typing errors, sometimes react very unpredictable, and often express themselves by means of ungrammatical but pragmatically well understandable (given the context of the dialogue) sentences. In dialogues in which people ask information about theater performances in a number of theaters and/or want to book seats for a particular performance, time, date and location phrases very often occur. These domain-dependent characteristics of the SCHISMA dialogues influence the morphological and parse modules that have to concentrate on recognition of these and other domain-specific phrases or words.

Other experiments with existing (commercially available) natural language interfaces (Komen 1995) show that systems in which input analyses are restricted to the search for particular (domain-dependent) phrases or patterns are often missing relevant information provided by the user. In conclusion, input analyses should do more than this and should try to extract as much syntactic/semantic structure from the input, using general linguistic/grammatical knowledge about the natural language as well as domain specific information concerning the most relevant concepts and their relations. On the other hand we are aware of the fact that a formal syntactical/semantical specification of the user language cannot be complete: users will subject the analyzer with input not covered by the formal language, although completely understandable for a human being. Hence, the quest for a robust grammar and parser. In section 4 we will discuss shortly the basic parsing technique and consider robustness.

The quest for a good, i.e. a practically useful, grammar and parsing system is a language engineering exercise consisting of several stages of development out of which, on the basis of experiments with earlier versions, finally and hopefully a satisfying system results. For the development of syntactic/semantic typed unification grammars we have defined and implemented a practical specification language for typed unification grammars based on

a context-free phrase-structure grammar.

The parser we developed for the system uses typed feature structures; the algorithm applied is head-corner parsing, a specialization of left-corner parsing. For the development of typed unification grammars we have defined and implemented a practical specification language for typed unification grammars based on a context-free phrase-structure grammar.

In sections 6 to 7 our parser and the formal language to specify types, lexicon and grammar is presented. We describe how a syntactic/semantic specification of a fragment of a natural language is compiled and linked with the parser resulting in a parser for the language that translates input sequences into their semantic representation as specified by the syntactic/semantic specification grammar. Unification of typed feature structures is the main and the most costly operation in our parsing technique. For a good performance of the PARS module an efficient implementation of this operation is a prerequisite. Section 6 treats unification of typed feature structures and section 7 presents the specification language for defining feature types, lexicon entries and grammar rules.

The dialogue manager is the module responsible for understanding the user's intentions in the context of the dialogue and for continuing the dialogue in a way that satisfies the user. The dialogue manager accepts from the parser the constituent structure and the semantics associated with it. The SCHISMA dialogue manager is an information state-based automaton; transitions are triggered by availability of information in the user utterances as provided by the parser; in addition a list is used for storing dialogue contexts; a dialogue context is an information state incrementally constructed from user utterances. In section 8 the dialogue manager is treated in detail. Also the connection to the parser is sketched. Finally, in section 9 we come to conclusions and present plans for the near future.

The formalisms and techniques presented in this paper are in principal applicable and useful for (the specification of) dialogue systems for natural language interfaces in general. The examples used in this paper come from the SCHISMA application, that functions as the main test environment for gaining insight in the practical value of the efforts and products reported and presented in this paper.

Before turning to our project however, in the next section we will look at some dialogue systems similar to SCHISMA.

2 Related Projects

In the current section we compare the SCHISMA project to other projects aiming at the development of natural language interfaces to databases: the ATIS projects, Verbmobil, Trains and the Philips Automatic Train Timetable Information System. The Verbmobil project by DFKI in Saarbrücken and the Trains project at the University of Rochester are treated more in depth at the end of this section.

The ATIS, Air Travel Information System, project (Hemphill et al. 1990) aims at the development of a spoken natural language interface to an air travel relational database. In the course of the ATIS project in four sub-projects, ATIS0 to ATIS3 corpora were collected. In the first project, ATIS0, a Wizard of Oz environment was used; in the later projects, ATIS1 to ATIS3, the fully-automated data collection systems were, in fact, working ATIS prototypes.

At the research department of Philips in Aachen (Germany), a spoken natural language interface to the time table of the Deutsche Bundesbahn (German railway corporation) is developed and implemented (Aust and Oerder 1995). The domain of the system is simpler than that of SCHISMA and the system is less powerful than the one we aim at. Users must conform themselves to fairly simple sentences that express their wishes unambiguously and answer questions of the system in an explicit manner. Currently a corpus is collected and a new design and implementation of the system is being worked on.

The Verbmobil project, executed by DFKI in Saarbrücken, is concerned with the development of a dialogue/translation system that assists Japanese and German business men in scheduling appointments (Alexandersson et al. 1995). The most apparent distinction between the SCHISMA and Verbmobil systems is that the Verbmobil system serves as a (translating) intermediary between two parties (business men negotiating appointments), whereas in SCHISMA one party is represented by the natural language interface to a database and the other party is the user that requests information and/or makes reservations. Another difference is the use of two languages in the Verbmobil system (Japanese and English); consequently the Verbmobil project involves machine translation (Japanese to English and vice versa). In SCHISMA communication is in Dutch.

The Trains project (Allen et al.1994) bears more similarity to SCHISMA than the Verbmobil system,

as it implements a natural language interface to a database. The Trains system implements an assistant that helps the user to manage a railway transportation system. In addition to knowledge about locations of trains, boxes and train stations, the assistant knows about products to be transported, places where they are kept and where they are produced. Comparing the SCHISMA domain to the Trains domain reveals that the Trains domain is more restricted than that of SCHISMA. The SCHISMA system allows the user to talk about and query several aspects of performances that are in the database; these include questions about players, artists, contents of performances. Dialogues in the Trains system are restricted to products (that can be produced, stored and transported), trains and railways.

We will consider the latter two systems more into detail and review their similarities and differences to SCHISMA. We have chosen the Trains and Verbmobil projects for further elaboration, because they are in an advanced status of development and implementation and therefore suitable for comparison.

2.1 Verbmobil

There are a number of fundamental differences between the system we have in mind in the SCHISMA project and the Verbmobil system (Weber and Hauenstein 1994; Maier and McGlashan 1994). The first and most important is that the Verbmobil system serves as an (translating) intermediary between two parties (business men negotiating appointments), whereas in SCHISMA one party is represented by the natural language interface to the database and the other party is the user that requests information and/or makes reservations.

Another difference is the use of two languages in the Verbmobil system (Japanese and English); consequently the Verbmobil project involves machine translation (Japanese to English and vice versa). In SCHISMA the user/system language is Dutch. Another difference comes from the current status of the SCHISMA project: we are working on a prototype system that implements a written interface to the database as opposed to the spoken Verbmobil system which comprises both speech recognition and speech generation.

Looking more in detail at the Verbmobil architecture, we see that the parser provides the acoustic module (speech recognition) with expectations with respect to the next utterance to be analysed. In the SCHISMA system this is not possible at the moment. The communication between the modules of the system is one-way only (from preprocessor to parser to dialogue manager). In our opinion two-way communication helps to decrease the amount of candidate recognitions (words for the speech recognizing preprocessor and structures for the parser) supplied to the next processing level; thus passing expectations to lower-level modules shifts recognition decisions to lower levels and does not add to the functionality of the system.

Finally, in the SCHISMA system preprocessing is not restricted to the recognition of words (or tokens) but extends to recognition of words (and phrases) carrying important domain-specific information. In the SCHISMA domain phrases that carry important information include date and time phrases, number names, phrases representing database information (titles of performances, names of artists, etc.). The Verbmobil system is not equipped with such functionality. Probably the Verbmobil domain does not require this type of preprocessing, as there is no database (querying and updating) involved.

2.2 Trains

The Trains system (Allen et al. 1994) is more similar to the SCHISMA system than the Verbmobil system, as it implements a natural language interface to a database. The Trains system implements an assistant that helps the user to manage a railway transportation system. In addition to knowledge about locations of trains, boxes and train stations, the assistant knows about products to be transported, places where they are kept and where they are produced.

Comparing the SCHISMA domain to the Trains domain reveals that the Trains domain is more restricted than that of SCHISMA. The SCHISMA system allows the user to talk about and query several aspects of performances that are in the database; these include questions about players, artists, contents of performances. Dialogues in the Trains system are restricted to products (that can be produced, stored and transported), trains and railways.

Both systems claim to be capable of managing mixed-initiative dialogues; this means that the initiative may be taken (or given) by both the user and the system. Clearly the possibility of a shift in initiative increases the complexity of the dialogue modeling problem.

Striking is the *pipeline approach* taken in both projects; the identified modules in the architecture

(typically preprocessor, parser and dialogue manager) only have one-way bottom-up communication; modules are not provided with expectations of any kind by higher level modules.

Text generation in the Trains system is similar to that in SCHISMA. Both systems use simple template driven sentence generation. The dialogue manager selects a template sentence and provides information for filling the slots to the generation module.

Having reviewed some related dialogue system projects we will now treat the SCHISMA system in full detail.

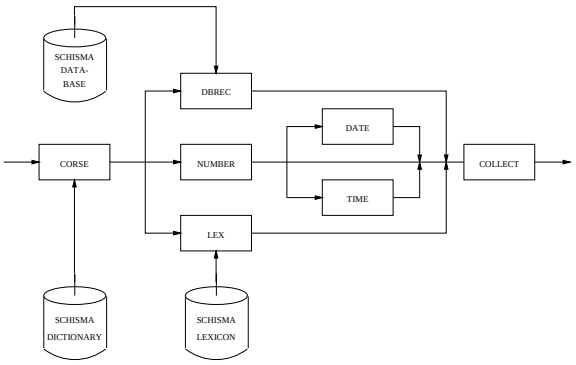


Figure 2: **Architecture of the MAF module.**

3 The Preprocessor MAF

As we postponed the development of a spoken interface to the SCHISMA system, we must concentrate here on the analysis of keyboard input; that is, the input of the MAF module is the character string typed in by the user. The MAF module is best seen as the preprocessor of the SCHISMA system. It handles typing errors and detects certain types of phrases (proper names that occur in the database, date and time phrases, number names, etc.). The latter task of MAF is especially important, since it extracts information crucial for the continuation of the dialogue from the input string.

Output of the MAF module is a *word graph*. We define a word graph here as a directed graph having as its nodes the positions in the input string identified as (possible) word boundaries. Nodes are numbered starting with 0 for the leftmost boundary; that is the position left to the first input character. A pair $(index_1, index_2)$ is an edge of the graph if $index_1$ and $index_2$ are word boundaries, $index_1 < index_2$ and the words enclosed between $index_1$ and $index_2$ are identified as one text unit. This implies that edges may provide text units to the parser that, in fact, contain more than word.

In addition, the MAF module labels the edges of the graph with a value that indicates the quality of the recognition (and, possibly, correction) performed.

3.1 Output of MAF

Of course word graphs are only used on a conceptual level in developing the preprocessor. The word graph is communicated among the distinguished submodules of MAF by means of items of the form $(s_{orig}, s_{recog}, cat, info, i_1, i_2, p)$. The first element s_{orig} denotes the string as it appeared in the input string typed by the user. s_{recog} is the string that CORSE corrects, if necessary, s_{orig} to. For the same s_{orig} multiple tuples may be output by CORSE. The syntactic category/type is given by the fourth element of the tuple. Items communicated between the submodules of MAF may have an unspecified value for this element; this holds especially for the items output by CORSE. Every item in the final output by MAF has its type specified, however. The grammar/parser is provided with the appropriate semantics for the types output by MAF. i_1 and i_2 are indices that specify the boundaries in the input string at hand. The last element of each item tuple is used for storing the probability that the recognition (if any) performed for this string is the right one.

The architecture of the MAF module as depicted in figure 2 should now be understood as follows: the error correcting module CORSE outputs a word graph that is provided to the tagging modules DBREC, NUMBER, DATE and TIME which scan the graph for phrases that are special in the SCHISMA domain. For details on the tagging modules and the phrases they recognise we refer to section 3.3. In addition, the word graph is provided to the LEX module. For performing the error correction CORSE has access to a large dictionary (typically 200,000 words). The tagging modules look for phrases in the input string that contain particularly important information for the dialogue; especially the detection of proper names referring to database items, phrases indicating date and time information and number names is aimed at here; for detecting proper names referring to the database the DBREC module needs access to the SCHISMA database. LEX searches the word graph for words that appear in the domain-specific lexicon and determines the appropriate feature type(s)

in order to capture their meaning content.

3.2 Segmentation and Error Correction

Clearly the analysis of typed input is somewhat simpler than spoken language recognition. However a typed interface introduces the challenge of handling typing errors the user makes and detecting word boundaries. In detecting and correcting typing errors we have to use knowledge of what character sequences are allowed in Dutch. Roughly there are two approaches to this from the engineering point of view: the *integrated* approach and the *preprocessor* approach. In the integrated approach recognition of tokens (lexical items, number names, etc. (see discussion below)) is done simultaneously with the error correction; this can be done by recording all word components in a trie structure and then operate on it by means of a cost function. See (Oflazer 1995) for details. The preprocessor approach requires the introduction of generic knowledge of what character sequences definitely may and what may not occur. It makes use of what trigrams of characters and what triphones (trigrams of phonemes) are viable in the Dutch language (given a dictionary of words that may occur). Using these trigrams substrings of the input string are compared to words in the dictionary. We refer to (Vosse 1994) for details on this error correction method. For reasons of compositionality we have chosen the latter approach: for the MAF module to be kept dividable in submodules, this option offers the best possibility to partition MAF in a number of submodules that have clear input/output specifications and thus can be developed and implemented separately.

In general, the CORSE module has to account for the following kinds of typing errors:

- insertion of characters,
- deletion of characters,
- substitution of character by other ones,
- exchange of characters; like in `auhtor`.

Clearly special attention must be given to typing errors concerning word boundaries and the detection of word boundaries themselves. Related to this is that punctuation symbols are handled as one character words.

3.3 Tagging Modules

Clearly, the choice for an error correcting preprocessor is a design decision that has some consequences for the architecture of the MAF component. Most important implication is that the components following the preprocessor, whatever they are have clear input/output relations and may perform their task in sequential order as well as parallel (in contrast to integrated).

The modules described below are in fact specialised taggers; each of them looks for a special type of phrases; if they find the type of phrase they are looking for the phrase is tagged and output to the postprocessor COLLECT. In general the output of the taggers is as described in section 3.1. The capitalised literals below correspond to module names in the architecture of MAF as given in figure 2.

- NUMBER; recognition of number names; `40`, `veertig` (fourty), etc.,
- DATE; recognition of date phrases; for instance `morgen` (tomorrow), `vandaag` (today), `12␣februari` (the 12th of February),
- TIME; recognition of time phrases; examples are: `vanavond` (tonight), `om␣acht␣uur` (at eight o'clock),
- DBREC; recognition of the use of proper names occurring in the SCHISMA database; examples: `Youp␣van␣het␣Hek` (proper name of an actor), `Hond␣op␣'t␣IJs` (title of his performance).

The TIME, DATE and DBREC modules are of special importance in the SCHISMA domain, since they detect phrases on which queries and updates to the SCHISMA database are based. The recognition of number names is considered a task necessary in any dialogue system.

To illustrate how the CORSE and tagging modules work, in figure 3 the word graph of the string `Ikw␣il␣de␣veertiende␣naar␣Verde` (On the 14th I would like to go to Verdi) is given. Spaces are represented by the ␣ character. It can be seen that the CORSE module corrected strings `w␣il` and `Verde` to `wil` (would like) and `Verdi` (the componist), respectively. Also the strings `de␣veertiende` (the 14th), `veertiende` (14th) and `Verde` have been tagged by the DATE, NUMBER and DBREC modules respectively.

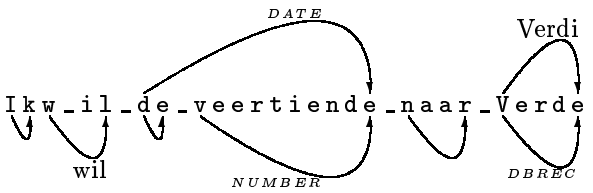


Figure 3: **A word graph of an input string**

3.4 The LEX and COLLECT Modules

The submodule LEX searches the lexicon for words that are provided by the CORSE module through the word graph. The lexicon is a rather small list of words that carry important domain semantics; for each of the words a feature type is supplied in the lexicon. We refer to section 7 for a discussion on typical feature types used for representing meaning in SCHISMA. Before the lexicon actually is searched, a morphological analysis is performed that accounts for inflection of verbs, adjectives, nouns, etc. Strings for which the lexicon does not contain an entry are assigned the (most general) type `word`, a prototype feature type for words not occurring in the lexicon. We refer to section 6 for details on typed feature structures. In addition LEX assigns feature types to punctuation symbols in the word graph (remember that the error correcting module provides these symbols as words in the word graph).

The COLLECT submodule accepts the items as generated by the taggers, the error correcting module and LEX, collects them and surrounds the set by tokens to indicate where the word graph starts/ends.

In figure 4 an example is given of a possible dialogue with the system. Referring to the linenumbers in front of the utterances in the dialogue we will show how the distinct submodules and especially the tagging modules act on the user input.

2 `Verdi` is recognised by the DBREC module as a database item; this both restricts the genre (classical music) and the performances the user may aim at (opera's by Verdi); the DATE module detects the phrase `on the 14th next month` as indicating a date;

7,8 the CORSE module corrects the ill-formed string `Verde` to `Verdi`; it therefore uses the SCHISMA lexicon which contains all strings occurring in the database; the string `opera` is recognised by LEX as a word carrying important domain semantics; in addition the question mark (and other punctuation symbols) are tagged as such by the CORSE module;

```
 1  S How may I help you ?
 2  C Are there any tickets left for
 3    the Verdi opera on the 14th
 4    next month?
 5  S Yes, there are,
 6    but only first rank.
 7  C oh that's ok ! What is the Verde
 8    opera about ?
 9  S [description of the plot of
10    the opera]
11  C I'd like two tickets
12  S I will make reservations for two
13    tickets for the Verdi opera on
14    the 14th of October. It's  $40,
15    tickets are $20 each.
16  C ok, thanks
17  S You're welcome, goodbye
```

Figure 4: **Example of a SCHISMA dialogue;** `C` refers to client input and `S` is system output.

11 the NUMBER module tags the string `two` as a number;

16 LEX tags the string `ok` and `thanks` with a feature type indicating the end of the dialogue.

3.5 Stage of Development

Currently modules DBREC, CORSE and LEX are available. DBREC and LEX have been developed by the SCHISMA partners, and CORSE has been adapted from the source code based on (Vosse 1994). The other modules DATE, TIME and NUMBER are still under construction.

Future research concerning MAF will follow the integrated approach as discussed in section 3.2. Expertise accumulated in working on the separate modules of MAF, will then be incorporated in the new design.

4 Parsing

The input of the parser is the output of the morphological/fault-detecting and -correcting module discussed in the previous section: a word graph, a compact structure representing a set of readings of the (corrected) input from the user,

where a reading is a path through the word graph. The parser analyses each of these readings independently of other readings.

Basically the parser is a head-corner chart parser for typed unification grammars. Like all chart parsers it starts with a set of basic items trying to construct completed items covering more and more adjacent words in the input until it has find a complete parse, covering all words. Head-corner parsing can best be seen as a generalisation of left-corner parsing. Both parsing techniques use top-down prediction (unlike pure bottom up chart parsers), but while in left-corner parsing, the input is processed strictly from left to right, in head-corner parsing the parser starts looking for a possible head of the sequence of input words. The set of possible heads of a sequence of words is completely specified by the context-free rules. One of the nonterminals (if any) in the right-hand side of each rule is specified as the head of the rule. It is up to the grammar writer to assign heads to rules. The basic idea is that the head of a rule derives (generates) the most informative (semantically relevant) words of the part of the sentence covered by this rule and that the parser should start looking for these words. An earlier version of this chart parser was presented in (Sikkel and Op den Akker 1993) and we refer to this paper for the technical details.

4.1 Robustness

Parsing in the context of a natural language dialogue system should be robust. Robustness here is a property of a typed feature unification grammar in combination with its parser. Robustness means, as far as natural language processing is concerned, filtering the relevant structural information from the reading. We are not interested in a linguistically sound and complete grammar and parser for Dutch. The syntactic structure of a reading is only of interest as far as it reflects its semantic meaning. (Pragmatic meaning is the communicative function of the user utterance in the context of a specific dialogue.) Although some parts of a grammar will be useful for dialogue systems for different specific domains and applications, parts of it have to be developed specifically for a particular domain. Hence, a context-free unification grammar is developed on the basis of an analysis of the corpus of dialogues resulting from Wizard of Oz experiments.

For the most important (domain-dependent) phrases and sentence-structures that occur in the corpus, we have developed a context-free unification grammar in conjunction with a lexicon of words with typed feature structures. Words contained in the lexicon, nouns in particular, have feature types assigned to them that are designed to allow disambiguation by feature unification failure. In developing the grammar we have striven at assigning one analysis to a reading if there is only one meaning. This is difficult to accomplish especially if the reading contains words that have obtained the default category `word` by the module MAF. This happens if the word could not be matched properly with a word in the domain-specific lexicon or with other domain-specific words. Since the semantic information that goes with these unknown words misses (bottom or $\perp$, see the next section) disambiguation can not be performed by means of unification failure. As a consequence the number of candidate parses will increase substantially for sentences containing unknown words.

The grammar for the SCHISMA prototype we are currently developing contains about 100 context-free rules.

Each rule is accompanied by a set of feature constraints. These constraints serve two purposes. They specify how to build the semantic feature structure of the left-hand side category compositionally using the feature structures from the right-hand side categories. Also they restrict the applicability of the context-free rule using failure of unification performed on the feature structures of the categories that occur in the rule. Unification is performed during parsing: items on the chart consist of categories with typed feature structures.

The parser outputs for each parse of its input a set of completed items. A completed item is a feature structure and it covers the whole or a part (i.e. a list of subsequent words) of the reading. The part it covers is indicated by the indices of the item. In case of an ambiguous sentence or phrase, it may output several completed items covering the whole reading. It is up to the dialogue manager using the status of the current dialogue to select the most likely parse or partial parse out of the semantic feature structures. The selection of the best parse will also be based on heuristic rules using the values assigned by the preprocessor indicating plausibility (of recognition) and information content of the words.

5 Unification

The basic operation on feature structures is unification. New feature structures are created by uni-

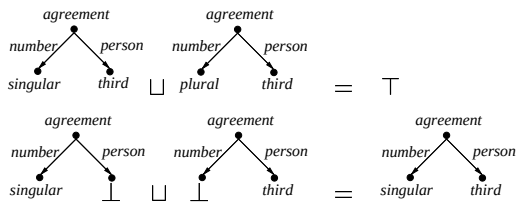


Figure 5: **Basic unifications.** In the first case unification fails, i.e. the feature structures contain conflicting information. The second case is self-evident.

n	parses	HCP	TOM
4	5	100	542
5	14	249	1,827
6	42	662	6,268
7	132	1,897	22,187
8	429	5,799	80,685

Table 1: **An untyped version of the unification algorithm compared with Tomabechi's algorithm.**

fying two existing ones. In figure 5 two basic examples show what unification means. In these feature graphs *agreement*, *singular*, *plural* and *third* are names of types, and *number* and *person* are names of features.

The unification of two feature structures fails if:

- the least upper bound of the two root nodes is $\top$, or
- the unification of the feature values of two features with the same name fails.

Unification is a costly operation in unification-based parse systems, because it involves a lot of copying of feature structures. In many implementations of parsing systems it takes more than 80% of the total parse time. Several algorithms have been devised to do unification efficiently (Tomabechi 1991; Wroblewski 1987; Sikkel 1993). The efficiency of unification can be increased by minimizing the amount of copying in cases that unification fails, while on the other hand the overhead costs to do this should be as small as possible. Up to now Tomabechi's algorithm seems to be the fastest. With this algorithm the copying of (partial) feature structures is postponed until it has been established that unification can succeed. But Tomabechi already suggests in a footnote that the algorithm can be improved by sharing substructures. This idea has been worked out into an algorithm (Veldhuijzen van Zanten and Op den Akker 1994). The copy algorithm has been implemented in a predecessor of the current parser and has proven to be very effective in experiments. Table 1 shows the results of one of these experiments. The algorithms were tested with the 'sentences' $Jan^n, n = 4 \ldots 8$, using the following grammar: $S \rightarrow S\ S \mid Jan$ (so the sentences are extremely ambiguous). The first column stands for the sentence length, the second column shows the number of parses and the third and fourth column show the number of nodes created during unification for Veldhuijzen van Zanten's and Tomabechi's unification algorithm.

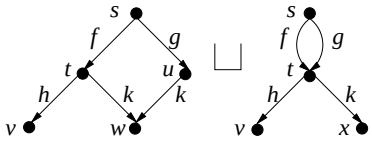
By introducing the types, the overhead increases slightly; for every two nodes that are to be unified the least upper bound has to be looked up in a table. But still, we expect an improvement in the performance.

Before the algorithm is described in more detail, it is necessary to define the general properties of a node in a feature structure. These properties ('members' in the object-oriented programming terminology, or 'fields' in a traditional record implementation) can be divided in two kinds: (1) properties that describe the structure of a feature structure and (2) bookkeeping properties, that are used to store intermediate results. For the first kind only

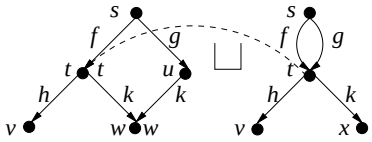
- a type id, that uniquely defines the feature names and the appropriate values for the corresponding features, and
- a set of features, where each feature consists of a name and a value (i.e. an instance of a certain type)

are needed. To handle the bookkeeping we need the following properties:

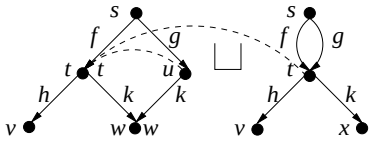
- a forward pointer: a pointer to another node, of which the unification algorithm has established that unification with this node is possible,
- an auxiliary type id: the type id of the corresponding node in the unifact (the result of unification),
- auxiliary features: features of the node that is unified with the current node, that do not occur in the set of features of the current node,
- a unifact pointer: a pointer to the unifact that is constructed by the copy algorithm,



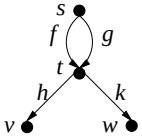
(a) The initial feature structures.



(b) The feature values of f have been unified.



(c) The feature values of g have been unified.



(d) The resulting unifact.

Figure 6: **An example unification.** A type id on the right of a node stands for the auxiliary type of that node. The dashed arrows indicate forward links. If a node has a forward link to another node, the feature structures starting in these nodes are unifiable. It is assumed that $x \sqsubseteq w$ and $u \sqsubseteq t$.

- a forward mark and a unifact mark: markers containing a generation number indicating whether the forward and unifact pointer can be used in the current unification process.

Unification is executed as a two-step operation: first, it is checked whether unification is possible, that is, the two feature structures to be unified contain no conflicting information. Second, the unifact is constructed using the bookkeeping information left by the first step.

Though the algorithm is implemented with object-oriented techniques in C++, the algorithm is displayed in conventional pseudo-Pascal code to enhance the readability for those not familiar with these techniques. Step one, the check if unification is possible, is shown in figure 7. Some auxiliary pro-

```
proc unify(tfs1, tfs2)
  nextGeneration();
  if unifiable(tfs1, tfs2) then
    return copyUnifact(tfs1)
  else
    return ⊤
  fi
end.

proc unifiable(tfs1, tfs2)
  tfs1 := dereference(tfs1);
  tfs2 := dereference(tfs2);
  if (tfs1 = tfs2) then return true fi;
  tfs1→auxType := lub(tfs1, tfs2);
  if (tfs1→auxType = ⊤) then
    return false
  fi;
  stillUnifies := true;
  while stillUnifies do
        foreach f ∈ tfs2→features
          if (f ∈ tfs1→features) then
            stillUnifies :=
            unifiable(tfs1→f, tfs2→f)
          else
            add feature tfs2→f to
            tfs1→auxFeatures
          fi
        od
  od;
  if (stillUnifies = true) then
    forward(tfs2, tfs1);
    return true
  else
    return false
  fi
end.
```

Figure 7: **The unification algorithm.** An improved version of Tomabechi's quasi-destructive unification algorithm.

cedures for the unification algorithm are displayed in figure 8. Finally, figure 9 shows how step two, the creation of the unifact, is implemented.

The example in figure 6 shows how the unification algorithm works. First the generation counter is increased to make any old intermediate results obsolete. The procedure *unifiable* is then called with the two s nodes as arguments. Now subsequent calls are made to *unifiable* for each feature value pair of the two s nodes. First the feature structures starting in the two t nodes are unified. They differ only in the feature value for the k feature. It is assumed that $x \sqsubseteq w$, so that the two nodes are unifiable. The auxiliary type will then be w. Now the two t nodes are unifiable and a forward link from one t node to the other one can be made (see figure 6b). Now the feature values for the g feature can be unified. Because of the forward link of the previous step, the feature values of the f and g feature of the left feature structure are now unified. So for unification to succeed we have to assume that $u \sqsubseteq t$. Under this assumption a forward link from the u node to the left t node can be made and the initial call to *unifiable* returns **true** (see figure 6c). The final step is then a call to *copyUnifact* to create the unifact from the intermediate results (see figure 6d). Note that this unification is non-destructive; both operands remain intact.

The procedure *lub* (called from *unifiable*) determines the least upper bound of two types. This least upper bound can be looked up in the type lattice as explained in section 6. If two types have $\top$ as least upper bound, they are not unifiable and it is not necessary to look at the feature values of the types.

The procedure *copyUnifact* (figure 9) only creates a new node if it is not possible to share that node with an existing feature structure. A new node is created by *createTFS*, which makes a node of the right type and initializes the features with appropriate values. The variable *needToCopy* is used to check whether a new node has to be created. Only if one of the following two situations occurs it is necessary to make a new node:

- the unifact has more features than the feature structure from which it constructed, that is, the number of auxiliary features is greater than 0,
- the unifact differs from the feature structure from which it constructed in at least one feature value.

Otherwise the node will be shared with the current

```
proc nextGeneration()
  currentGeneration :=
  currentGeneration + 1
end.

proc dereference(tfs_1)
  if tfs→forwardMark = currentGeneration
     ∧ tfs→forward ≠ nil then
    return tfs→forward
  else
    return tfs
  fi
end.

proc forward(tfs_1, tfs_2)
  tfs_1→forward := tfs_2;
  tfs_1→forwardMark := currentGeneration;
end.
```

Figure 8: **Auxiliary procedures for the unification algorithm**

node of the typed feature structure from which the unifact is constructed.

6 Typed Feature Structures

6.1 Types

Types can be used to categorize linguistic and domain entities. In addition to that the relations between entities can be defined using an inheritance hierarchy. For types we follow the definition of Carpenter (1992). Types can be ordered using the subsumption relation. We write $s \sqsubseteq t$ for two types s and t if s *subsumes* t, that is, s is more general than t. In that case s is called a *supertype* of t, or inversely, t is a *subtype* of s. With the subsumption relation the set of types form a lattice (see figure 10).

The type that subsumes all other types ("the most general type") is called bottom and is denoted by $\bot$. The most general subtype for a pair of types s and t is called the *least upper bound* and is written as $s \sqcup t$. For instance, in figure 10 we have $s \sqcup t = x$ and $v \sqcup w = \top$. In the latter case the two types contain conflicting information and are hence inconsistent.

There are two ways to specify a type lattice. The first way is to express each new type in terms of its subtypes. This can be seen as a set-theoretical approach: each type is a set of possible values and a new type can be constructed by taking the union

```
proc copyUnifact(tfs)
  tfs := dereference(tfs);
  if (tfs→unifactMark = currentGeneration)
    then
    return tfs→unifact
  fi;
  needToCopy := (#tfs→auxFeatures > 0);
  i := 0;
  foreach f ∈ tfs→(features ∪ auxFeatures)
    do
    copies[i] := copyUnifact(f);
    needToCopy := needToCopy ∨
    (copies[i] ≠ f);
    i := i + 1
  od
  if needToCopy then
    if tfs→unifact = nil then
      tfs→unifact :=
      createTFS(tfs→auxType)
    fi;
    for j := 0 ... i ⊥ 1 do
        add feature copies[j]
        to tfs→unifact
    od;
    tfs→unifactMark := currentGeneration;
    return tfs→unifact
  else
    return tfs
  fi
end.
```

Figure 9: **The copy algorithm.** This procedure generates the unifact after a successful call to *unifiable*(tfs_1, tfs_2).

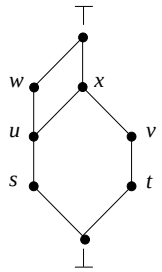


Figure 10: **A type lattice.** The lines represent the subsumption relation. More specific types are placed higher in the lattice. The top element '⊤' is used to denote inconsistency between types.

of other (possibly infinite) sets. For instance, the type `fruit` could be defined as

$$\texttt{fruit} \;:=\; \texttt{apples} \cup \texttt{bananas},$$

where `apples` is a set of all apples and `bananas` is a set of all bananas. The bottom element $\perp$ is then the set of all entities within the domain and the top element $\top$ is the empty set $\emptyset$.

The other way to specify a type lattice is to express each type in terms of its supertypes. In this context the term 'inheritance' is often used; a type inherits information from its supertypes. The disadvantage of specifying types this way is that inconsistencies in the lattice are easily introduced. If a type is specified to have two supertypes that contain conflicting information, that type would be inconsistent. With the set-theoretical approach this cannot happen. However, from the grammar writer's point of view it is often easier to first introduce general concepts and later differentiate them into more specific types than to start with the most specific types and generalize over them to construct new types. Hence, in the specification language described in section 7 the second approach is followed.

6.2 Feature Structures

Feature structures provide a convenient way to keep track of complex relations. During parsing constraints can be checked with feature structures, and after parsing the meaning of the language utterance can (hopefully) be extracted from them. The structure of our feature structures is similar to the more traditional form of feature structures as used in the PATR-II system (Shieber 1986) and those defined by Rounds and Kasper (1986).

Typed feature structures are defined as rooted DAGs (directed acyclic graphs), with labeled edges and nodes. More formally, we can define a typed feature structure *tfs* as a 2-tuple $\langle t, features \rangle$, where $t \in$ *Types*, the set of all types, and *features* is a (possibly empty) set of features. A feature is defined as a feature name / feature value pair. A feature value is again a typed feature structure. At first glance the labels on the nodes seem to be the only difference with the traditional feature structures, but there is more to typing than that. Every type has a fixed set of features. Such a feature value type can be seen as the appropriate value for a particular feature. It should be equal to the greatest lower bound (the most specific supertype) of all the possible values for that feature. So a typed feature structure is actually an instantiation of a type. Types are used

as a sort of templates. By typing feature structures we restrict the number of 'allowed' (or appropriate) feature structures. Putting these restrictions on feature structures should fasten the parsing process; at an earlier stage it can be decided if a certain parse should fail.

Another advantage of typing feature structures is that it is no longer necessary to make a distinction between nodes *with* features ('complex nodes'), nodes *without* features ('constant nodes') and nodes with type $\bot$ ('variable nodes') as is often done with traditional feature structures. In a consistent definition of the type lattice the least upper bound of a complex and a constant node should always be $\top$ (unless that constant node represents an abstract, underspecified piece of information), so that two such nodes can never be unified.

7 The Specification Language

To specify a language it is necessary to have a metalanguage to specify that language. Almost always the usage of a specification language is limited to only one grammar formalism. This is not necessarily a drawback, as such a specification language can be better tailored towards the peculiarities of the formalism. For example, ALE (Carpenter and Penn 1994) is a very powerful (type) specification language for the domain of unification-based grammar formalisms. But apart from expressiveness of the specification language, the ease with which the intended information about a language can be encoded is also important. An example of a language that combines expressiveness with ease of use is the Core Language Engine (Alshawi 1992). Unfortunately the Core Language Engine (CLE) does not support typing. Within our project a type specification language has been developed that can be positioned somewhere between ALE and CLE. This specification language can be used to specify a type lattice, a lexicon and a unification grammar for a head-corner parser. The notation is loosely based on CLE, though far less extensive. For instance, the usage of lambda calculus is not supported.

7.1 Specification of Types

A type specification consists of four parts: a type id for the type to be specified, a list of supertypes, a list of constraints and a formula expressing the semantics for the new type. Figure 11 shows how a type lattice can be specified. For each type `<constraints>` should be replaced with PATR-II-like path equations. Path equations can have the following two forms:

$$\begin{aligned} \langle f_1\ f_2 \ldots f_n \rangle &= \langle g_1\ g_2 \ldots g_n \rangle \\ \langle f_1\ f_2 \ldots f_n \rangle &:= \mathit{node} \end{aligned}$$

The first form says that two paths (i.e., sequences of features) in a feature structure should be joined. With the second form the type of a node in a feature structure can be specified. The right-hand side should be equal to a type identifier or a constant (a string or a number). During the parsing of the specification a minimal satisfying feature structure is constructed for each path equation. So all the nodes in a feature structure have type $\bot$, unless specified otherwise. Subsequent path equations are unified to generate a new feature structure satisfying both constraints. Finally the resulting feature structure for all the constraints is unified with constraints inherited from the supertypes.

`<QLF>` should be replaced with the semantics in a quasi-logical formula. QLF is basically equal to first-order predicate calculus, but is extended with some extra operators to express the mood of an utterance and to express some basic set properties. A more detailed description of QLF can be found in (Moll 1995). The idea is that the constraints are only necessary *during* parsing and the semantics are passed on to be used *after* parsing. In the following example a possible quasi-logical form for a phrase is given:

de opera voorstelling op 4 januari
(the opera performance on 4 January)
`EXISTS X (opera(X) AND date(X,4-1-95))`

The `opera` predicate comes from the QLF part of the opera type and the `date` predicate is generated by parsing the time phrase. Another grammar rule combines these predicates and binds the variable `X`.

A type inherits information from its supertypes in the following way: the constraints for the type are *unified* with the constraints of the supertypes, and the quasi-logical formula for the type is *concatenated* with a list of quasi-logical formulas for the supertypes. The QLF expressions are not evaluated, but are just translated to internal representations.

7.2 Specification of Words

Lexical entries can be specified in the same way as types. This is not surprising, since words can also be seen as types. There is, however, one restriction: a word cannot be used as supertype in the specification of other types (including words). Ambiguous

```
TYPE(performance; bottom; <constraints>; <QLF>)
TYPE(play; performance; <constraints>; <QLF>)
TYPE(concert; performance; <constraints>; <QLF>)
TYPE(musical; play, concert; <constraints>; <QLF>)
TYPE(ballet; concert; <constraints>; <QLF>)
```

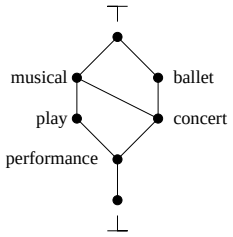


Figure 11: **Specification of a type lattice**

words can be specified by simply defining multiple entries for the same lexeme:

```
LEX("flies"; verb; <constraints>; <QLF>)
LEX("flies"; noun; <constraints>; <QLF>)
```

The type identifier that is given to a word is the type identifier of the first type of the list of supertypes. In the previous example the words only had one supertype, so there is a feature structure for "flies" with a `verb` type identifier and a feature structure with a `noun` type identifier. In the lexicon every word is associated with a list of feature structures, one for each meaning.

7.3 Specification of Grammar Rules

Grammar rules are internally also represented as typed feature structures. They have a special rule type identifier. It is not necessary to represent rules as feature structures, but such structures happen to be a very practical representation mechanism for grammar rules.

In general, a grammar rule specification looks like the one in figure 12. The asterisks mark the head in the grammar rule. Next to the specification the resulting typed feature structure is shown. Note that grammar symbols are in fact types.

The next example shows how typing can make some grammar rules superfluous.

```
TYPE(perfphrase; nounphrase; ; )
RULE(nounphrase --> *perfphrase*;
  <nounphrase kind> = <perfphrase kind>,
  <nounphrase sem> = <perfphrase sem>; )
```

The asterisks mark the head in the grammar rule. Both the type and rule specify that a performance phrase is a kind of noun phrase. So with the type specification the rule becomes superfluous.

8 The Dialogue Manager

In general, the dialogue manager's task is to give appropriate reactions to the user's utterances. What an 'appropriate' reaction is, cannot be determined easily. In the view we have on SCHISMA, an appropriate answer is one that does not have to be 'natural' (i.e. the answer does not have to be one that a human consultant would give), but it should help the dialogue to continue in a way that the client can feel comfortable with it.

The dialogue manager consists of a finite state automaton and a context list. These will be presented in section 8.2.

8.1 The Input of the Dialogue Manager

The SCHISMA dialogue manager receives its input from a syntactic and semantic parser. That is why the input can consist of a 'stripped' sentence. It expects an input containing only the following elements:

- database attributes
- information about the client
- whether the utterance is a question. If so, we distinguish wh-questions (containing words like 'who', 'when', etc.) and yes/no-questions, in which no such word is present.
- whether the client shows the intention to make a reservation.

The current system does not support sentence components like corrections of previous utterances or deviations from the domain subject. These may be present in later versions of the system.

8.2 The Finite State Automaton

The SCHISMA dialogue manager is based on a finite state automaton (FSA). An FSA is a quadruple $\langle S, Q, q_0, t \rangle$, in which S is the input alphabet, Q is a set of states, q_0 is a specific element of Q, called the start state, and t is the transition function $Q \times S \rightarrow Q$. For any sequence of elements of S (such a sequence is called a string), the FSA gives a sequence of transitions $\{(q_0, q_1), (q_1, q_2), \ldots, (q_{n-1}, q_n)\}$, in

```
RULE(s --> np *vp*;
    <np agr> = <vp agr>,
    <vp subject> = <np sem>,
    <s sem> = <vp sem>; )
```

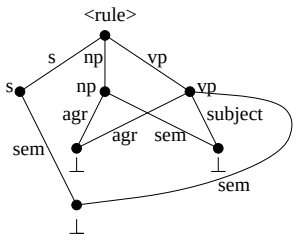


Figure 12: **Specification of a rule**

which $q_{i+1} = t(q_i, v_i)$ and v_i is the ith character in the string.

The states of the FSA denote the kind of information that is known to the system at a point in dialogue. There are three types of information:

Database items These are attributes of the entities in the database that contains the information about the performances. Some examples of these are: names of performances, names of artists and prices of performances. For each database entity, the state contains a marker that has one of the values 'unknown', 'constrained' and 'known'. The value 'known' is valid, if precisely one entity instantiation of the entity matches the information that is available in the dialogue. It is 'constrained' if some attribute values of it have been mentioned, but not enough is known to determine precisely one instantiation. If nothing has been said about the entity, it is 'unknown'. This also holds for all entities at the beginning of the dialogue.

Information about clients This is supplied by the client. Some examples are the client's name, the number of people for whom a reservation has to be made, and the reduction pass that the client has got (if any). This can only be 'known' or 'unknown'; the latter is the case at the beginning of the dialogue.

A value that denotes whether or not the client has shown that (s)he wants to make a reservation. This is used to determine the initiative of the dialogue. If the client has not shown his intention to make a reservation, the system is passive; it only answers the client's questions. When the client has shown his intention to make a reservation, the system behaves active; it asks questions itself.

Apart from this, the states contain a marker that denotes whether the client or the system is the next speaker (it is assumed that they speak alternately). The states in which this marker is 'user', the client is the first speaker; in the others it is 'system'. The states of the automaton are not defined explicitly. This is because the total number of states is far too large. Furthermore, for a lot of states, the way the system should react to an utterance is the same. Therefore the system's utterances are defined for a group of states and client utterances that share some property.

There are two types of transitions in the dialogue FSA. These are user transitions and system transitions. User transitions are caused by an utterance from the client. Such a transition is from a user state to a system state. During a user utterance, the new state is determined by updating the old state. Beginning in the old state, the items that are given in the utterance are marked as 'known' or 'constrained'. Database items are marked as 'constrained' (unless they were already known; in this case, they remain unchanged), user items are marked as 'known'.

System transitions are caused by consulting the database. The database items that the client has mentioned in his/her last utterance and in the utterance before them are used as lookup keys in the database. After the search, the items that appear to be uniquely defined by the information are marked as 'known'. The following dialogue shows the state changes (`C` = client, `S` = system):

`S This is the theatre information system. What can I do for you?`	`(all items unknown)`
`C I would like to go to a ballet performance. Are there any in your theatre?`	`(genre is known, other items are unknown)`
`S There is one ballet performance: 'The swan lake' by the National Dutch Ballet.`	`(genre, performance, performing group are known, others unknown)`

The utterance that the system produces after a system transition, is determined by the following ele-

ments:

- the items that were found to be 'known' during the system transition,
- the items that the client has asked during his utterance (if any),
- whether or not the client has shown his interest to make a reservation.

The first rule for determining the system's utterance is 'tell the client what he has asked for'. Therefore, if the client has asked a question, the system should answer it. If the question was a wh-question, it is straightforward what elements the answer should contain. If it is a yes/no-question, this is not so clear. But it is a well-known fact that yes/no-questions should very often be interpreted as wh-questions. The following example dialogues show this:

```
C Are there any plays by Shakespeare in your
  theatre?
S Yes, there are.

C Are there any plays by Shakespeare in your
  theatre?
S We present the following plays by
  Shakespeare:  'The taming of the shrew'
  on October the 18th and 'King Lear' on
  February the 5th.
```

In the first dialogue, the system's response is a direct response to the client. However, to the client it would seem strange and not very cooperative. In the second, the system does not give a direct answer to the question, but it will appear more natural to the client. That is why the dialogue manager treats yes/no-questions as wh-questions. If the utterance contains an item name without a value, this item is assumed to be asked by the client. For example, in the above dialogues, the client mentioned the word 'play' without a name of a play; the system behaves as if the client asked for the names of plays, where the author name 'Shakespeare' is a constraint.

Most other yes/no-questions can be treated as wh-questions as well, but with some more problems. For example, the question 'Is "Hamlet" a play by Shakespeare?' can be treated as 'Who wrote "Hamlet"?', and thus be answered by '"Hamlet" was written by William Shakespeare.' This answer is not natural, but it is cooperative and we therefore accept it.

The SCHISMA FSA differs from some other dialogue management systems, in that its transitions are based on the information contained in the utterance, whereas a lot of other dialogue systems base their transitions on the type of speech act of the utterance. Some of these systems are Verbmobil (Alexandersson et al. 1995) and the air travel information system that is developed at LIMSI-CNRS (Bennacef et al. 1995). There are some dialogue systems that are also based on FSA's and in which the transitions are defined by the contents of the utterances. These are often relatively small systems, however (the FSA's have ten states or less, e.g. (Naito et al. 1995). In the Danish Dialogue project, dialogues are modeled by means of the Dialogue Description Language (Baekgaard 1995). The main flow of a dialogue is described by flowcharts (directed graphs, that are closely related to FSA's), that give the sequence of actions to be performed. A disadvantage of this method is the complexity of the resulting graphs.

8.3 The Interface between the Parser and the FSA

In this section we will describe how the parser can be interfaced with the FSA. This is not a trivial problem, since the output of the parser differs substantially from the expected input for the FSA. The result that is returned by the head-corner parser is a list of recognized parsing items. For each recognized constituent there is an item indicating the positions of the constituent. A typed feature structure is associated with each item. From this feature structure the meaning of the item can be extracted. The FSA, on the other hand, is not concerned with all those individual items. Instead, a description of the complete utterance is combined with the state of the automaton and the context to represent the situation of the dialogue.

The FSA can construct a state by just taking into account the parsing items for whole sentences. The QLF expression for these items should contain enough information to construct the input for the dialogue manager. This can (and should) be enforced by the grammar writer, who should write the rules in such a way that the QLF expressions for other items percolates up to the top level. Also, variable binding should be specified in the grammar. For instance, in a grammar rule where the subject is combined with a verb, the free variable in the QLF expression for the verb can be instantiated with the QLF expression of the subject.

The QLF expression of the complete utterance can be transformed into a list of information items, which can be handled by the dialogue manager. To do this, most of the syntactic information and the information about types in the feature structure

can be removed; only semantic information is necessary. This information is internally stored in the "qlf" feature of a feature structure and can then be transferred to the dialogue manager in a simple way.

8.4 The Context List

A finite state automaton does not offer good possibilities for storing information from a previous part of the dialogue. The complete dialogue history is stored in the state. That is why an FSA cannot be used to model dialogues completely. An additional mechanism is needed to do that. In the current version of the dialogue manager, we use a context list for this purpose. This context list bears some relation to the preference stack that is given by Fanty et al. (1995). However, it has an extension that makes it more powerful than the preference stack.

The context list is an ordered list of contexts. The word context here means: a set of valid (client and database) item values, and a state. The contexts in the list are ordered in such a way, that the most recent context is always at the front. If the client says an utterance that does not match the current context, the context list is searched for a context that does match the utterance. This search begins at the front of the list; it ends when a matching context is found. This context is moved to the front of the list. A new context is created by combining this context and the contents of the utterance. This new context is put at the front of the context list and becomes the current context. In the following example dialogue, the construction of the list given (only the items of the contexts are shown, the states of the automaton are not). The initial value of the list is the empty context, denoted by an empty box. It contains no values and matches every utterance. See figure 13.

The change in the context stack after the fourth utterance can be explained as follows. First, the old context list is searched for a matching context. The first context already matches the utterance, so it remains the first context in the list. Then, a new context is created by combining the new information with the information in the context. This means that the only play name that is allowed is 'The taming of the shrew'. However, no date is mentioned in the utterance, so both dates in the old context remain valid, although only one of them is correct for 'The taming of the shrew'.

We are currently considering a more advanced context selecting mechanism, which is based on the focusing method of Rosé et al. (1995) and Lambert (1993). It is based on the use of a tree, in which the order of the branches defines the saliency of a context.

9 Concluding Remarks

For a practically useful dialogue system full handling of users' natural language input is most important. This is especially the case if the dialogue system does not force the user to provide his/her information in a predefined ("menu driven") order but leaves it to the client to control the dialogue and allows him/her to use free natural language. To find a good formal specification of the language and a robust analyser and parser is a complicated task to be performed before such a dialogue system is obtained. The making of such a system is a process consisting of several design steps.

In this paper we have reported on some of the first steps we have taken towards the design of dialogue systems. First we presented an outline of the preprocessor of our system. We showed the importance of early recognition of phrases carrying important domain-specific semantics. Also we sketched how the preprocessor for our purposes can be designed.

We presented a newly developed parser for typed feature structures grammar and a useful specification language for types, lexical entries and grammar rules. We consider the development and the implementation of our specification language for typed unification grammars to be a major step in this design. It provides a very convenient tool for the specification of the syntactic and semantic aspects of fragments of natural languages used in dialogue systems.

Finally we discussed the dialogue manager of the SCHISMA system. We argued that information state-based dialogue management is suitable for the SCHISMA domain. We store dialogue contexts implied by subsequent utterances by means of a so called context list.

9.1 Future Work

In the near future we will integrate the MAF module in the Wizard of Oz environment, hereby confronting the Wizard with the output of MAF instead of with the user input directly. Experiments with this extended environment will give more insight in the practical value (and shortcomings) of our results. Moreover we will work on an implementation of MAF based on an integrated approach of

the users input instead of the "pipe line approach" presented in this paper. Although it will have the same functionality as the one presented here we expect it to be more efficient.

We will also write a second, more complete, version of our grammar and lexicon for the Schisma application using all facilities provided by the presented specification language for typed unification grammars. The next step is then to incorporate the generated parser into the Wizard of Oz environment as well. The Wizard will then be offered the output of the parsing module: a semantic feature structure that should provide enough relevant information to select an appropriate action for continuation of the dialogue. This also offers the opportunity to verify experimentally (using a "real" grammar) our claims about the efficiency of our unification algorithm for typed feature structures. Moreover, these experiments will also be used for the specification of heuristic rules to be used by the dialogue manager in the process of selecting "the best analyses" of user input given the current status of the dialogue.

Acknowledgements

We would like to thank Gert Veldhuijzen van Zanten for making the first major improvements to Tomabechi's unification algorithm. Also we would like to thank the members of the SCHISMA project. Especially we are grateful to Rieks op den Akker for commenting on draft versions of this paper. We are indebted to Hans Kloosterman of KPN Research for fruitful discussions on the preprocessor MAF of the system.

Bibliography

Alexandersson, J., Maier, E., and Reithinger, N. (1995). A robust and efficient three-layered dialogue component for a speech-to-speech translation system. In *Proceedings of the 33rd Annual Meeting of the ACL*. Boston, MA.

Allen, J. F., Schubert, L. K., and George Ferguson, e. a. (1994). The TRAINS project: A case study in building a conversational agent. Technical Report Trains Technical Note 94-3, The University of Rochester, Rochester, New York, USA.

Alshawi, H., editor (1992). *The Core Language Engine*. Cambridge, MA: The MIT Press.

Aust, H., and Oerder, M. (1995). Dialogue control in automatic inquiry systems. In Andernach, J., van de Burgt, S., and van der Hoeven, G., editors, *Corpus-based Approaches to Dialogue Modelling*, no. 9 in Twente Workshop on Language Technology, 45–49. Enschede, The Netherlands.

Baekgaard, A. (1995). A platform for spoken dialogue systems. In *ESCA Workshop on Spoken Dialogue Systems*. Vigsø, Denmark.

Bennacef, S. K., Néel, F., and Maynard, H. B. (1995). An oral dialogue model based on speech act categorization. In *ESCA Workshop on Spoken Dialogue Systems*. Vigsø, Denmark.

Carpenter, B. (1992). *The Logic of Typed Feature Sructures*. Cambridge University Press.

Carpenter, B., and Penn, G. (1994). Ale 2.0 user's guide. Technical report, Carnegie Mellon University Laboratory for Computational Linguistics, Pittsburgh, PA.

Fanty, M., Sutton, S., Novick, D., and Cole, R. (1995). Automated appointment scheduling. In *ESCA Workshop on Spoken Dialogue Systems*. Vigsø, Denmark.

Hemphill, Godfrey, and Doddington (1990). The ATIS spoken language systems pilot corpus. In *Proceedings DARPA Speech and Natural Language Workshop*. DARPA.

Komen, E. (1995). Evaluation of Natural Language™ for the SCHISMA domain. Memoranda Informatica 95-14, University of Twente, Enschede, The Netherlands.

Lambert, L. (1993). *Recognizing Complex Discourse Acts: A Tripartite Plan-Based Model of Dialogue*. Phd dissertation. tech. rep. 93-19, Department of Computer and Information Sciences, University of Delaware.

Maier, E., and McGlashan, S. (1994). Semantic and dialogue processing in the VERBMOBIL spoken dialogue translation system. Technical Report Report 51, DFKI GmbH and FB Computerlinguistik, Universität des Saarlandes, Saarbrücken, Germany.

Moll, M. (1995). Head-corner parsing using typed feature structures. Master's thesis, Department of Computer Science, University of Twente.

Naito, M., Kuroiwa, S., Takeda, K., Yamamoto, S., and Yato, F. (1995). A real-time speech dialogue system for a voice activated telephone extension service. In *ESCA Workshop on Spoken Dialogue Systems*. Vigsø, Denmark.

Oflazer, K. (1995). Error-tolerant finite state recognition. In *Proceedings of the Fourth International Workshop on Parsing Technologies*, 196–207.

Rosé, C., Di Eugenio, B., Levin, L., and Van Ess-Dykema, C. (1995). Discourse processing of dialogues with multiple threads. In *Proceedings of the 33rd Annual Meeting of the ACL*. Boston, MA.

Rounds, W. C., and Kasper, R. T. (1986). A complete logical calculus for record structures representing linguistic information. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science*, 39–43. Cambridge, MA.

Shieber, S. M. (1986). *An Introduction to Unification-Based Approaches to Grammar.* Stanford, CA: Center for the Study of Language and Information, Stanford University.

Sikkel, K. (1993). *Parsing Schemata.* PhD thesis, Department of Computer Science, University of Twente, Enschede, The Netherlands.

Sikkel, K., and Op den Akker, R. (1993). Predictive head-corner chart parsing. In *Proceedings of the Third International Workshop on Parsing Technologies*, 267–275. Tilburg (The Netherlands), Durbuy (Belgium).

Tomabechi, H. (1991). Quasi-destructive graph unification. In *Proceedings of the 29th Annual Meeting of the ACL.* Berkeley, CA.

Veldhuijzen van Zanten, G., and Op den Akker, R. (1994). Developing natural language interfaces: a test case. In Boves, L., and Nijholt, A., editors, *Twente Workshop on Language Technology 8: Speech and Language Engineering*, 121–135. Enschede, The Netherlands.

Vosse, T. G. (1994). *The Word Connection.* PhD dissertation, Rijksuniversiteit Leiden. Neslia Paniculata.

Weber, H., and Hauenstein, A. (1994). An investigation of tightly coupled time synchronous speech language interfaces using a unification grammar. Technical Report Report 9, IMMD VIII - Kuenstliche Intelligenz, Universität Erlangen-Nürnberg and FB Informatik, AB Nats, Universität Hamburg, Hamburg, Germany.

Wroblewski, D. (1987). Nondestructive graph unification. In *Proceedings of the Sixth National Conference on Artificial Intelligence.*