DESIGN PRINCIPLES OF THE DISCO SYSTEM

Günter Neumann*

Deutsches Forschungszentrum für Künstliche Intelligenz
Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11, Germany
neumann@dfki.uni-sb.de

To appear in: Proceedings of the TWLT 5, Twente, Netherlands, 1993.

ABSTRACT

In this paper we introduce the basic design principles of the DISCO system, a Natural Language analysis and generation system. In particular we describe the DISCO DEVELOPMENT SHELL, the basic tool for the integration of natural language components in the DISCO system, and its application in the COSMA (COoperative Schedule MAnagement Agent) system.

Following an *object oriented* architectural model we introduce a *two–step approach*, where in the first phase the architecture is developed independently of specific components to be used and of a particular flow of control. In the second phase the "frame system" is instantiated by the integration of existing components as well as by defining the particular flow of control between these components. Because of the object–oriented paradigm it is easy to augment the frame system, which increases the flexibility of the whole system with respect to new applications. The development of the COSMA system will serve as an example of this claim.

1 INTRODUCTION

Today's natural language systems are large software products. They consist of serveral mutually connected components of different kinds, each developed by different researchers often placed on different locations. The integration of these components has therefore become a software engineering and mangement problem. We will consider the project DISCO (DIalogue Systems for COoperating agents) from this perspective.

DISCO's primary goal is the processing of multiagent natural language dialogue. Multiagent capabilities make it an appropriate front end for autonomous cooperative agents, which will be exemplified by the COSMA system (COoperative Scheduling MAnagement system) described in section 4.

The project DISCO as a whole is a four-year, eight-person research effort funded by the German Ministry for Research and Technology. DISCO is completing its

*The research underlying this paper was supported by a research grant, FKZ ITW 9002 0, from the German Bundesministerium für Forschung und Technologie to the DFKI project DISCO.

fourth year. The first task of the development of the DISCO system was to provide a uniform core formalism based on unification of feature structures. The second was the construction of a modular architecture orthogonal to the representation formalism, as a platform for experimentation. Third, research and construction of dialogue components is ongoing, as is (fourth) investigation of the interface between dialogue components and multiagent systems.

We will emphasize in this paper the second phase, i.e. the description of the underlying architecture. The solutions to the other tasks will be reported briefly in the next section.

2 OVERVIEW OF THE DISCO SYSTEM

The linguistic core machinery extends a constraint-based approach of linguistic description [Pollard and Sag, 1987]. In this paradigm linguistic objects are described by a set of constraints which express mutually co-occurence restrictions of phonological, syntactic and semantic information. A fundamental aspect of these theories is that they are *declarative*, i.e. they only describe *what* constraints are necessary to describe linguistic objects not the way in *what order* the constraints involved are to be solved.

Such a uniform view has not only a lot of advantages for linguistic description but also for the design of Natural Language systems because it leads to more *compact* systems. If, for example, the different stratas (e.g., syntax, semantics) would be *represented* and *processed* in different modules than a complex internal flow of control between these modules would be necessary if the mutual co-occurence restrictions should be maintained. Using an uniform approach instead allows to process these constraints in an incremental and parallel way by the very same constraint solving algorithm.

Figure 1 shows graphically the structure of the linguistic kernel.

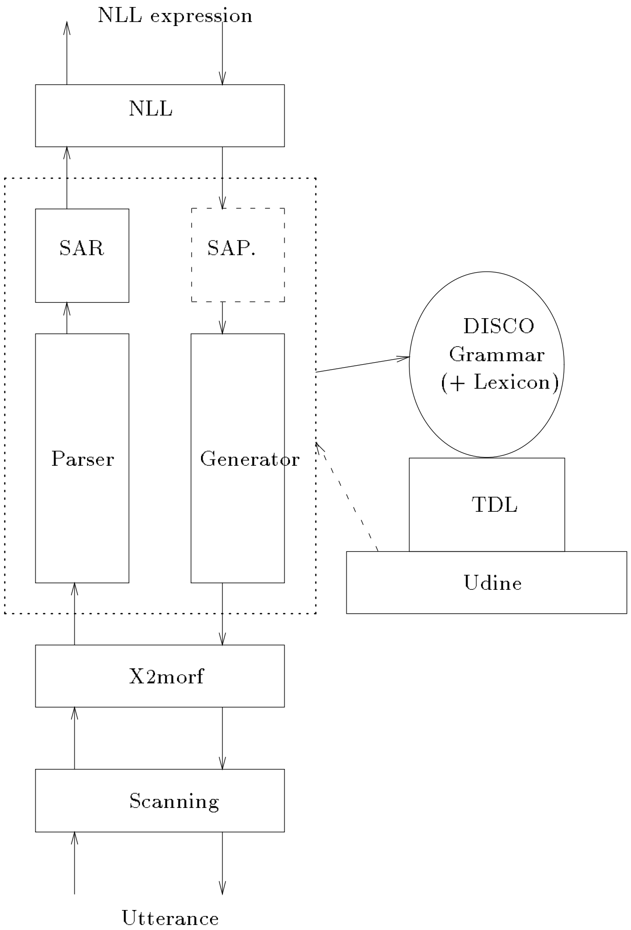

Figure 1: Overview of the DISCO kernel.

Uniform Core Formalism The linguistic knowledge is specified in the typed feature formalism TDL [Schäfer and Krieger, 1992], which incorporates the unification engine UDiNe. TDL is the exclusive formal device employed to specify grammar rules, lexical entries and all other linguistic knowledge relevant for the grammar. UDiNe serves also as the basic machinery for linguistic processing (e.g., during parsing, generation or speech act recognition).

In TDL, typed feature structures can be defined through simple or multiple inheritance relations. TDL performs full type expansion at compile-time, i.e., if in a type definition a type inherits from other types or if the value of an attribute is restricted to a type, these types are replaced by the associated feature structures with only limited simplification.[1]

Unification of feature structures in TDL and elsewhere in the system is executed by the unfier UDiNe, which is one of the most comprehensive unifiers so far implemented. It comprises full negation, including negation of co-references, and full disjunction. UDiNe provides for so-called *distributed disjunction* through which disjunctive information can be kept as local as possible in the structural specification. The main advantage of distributed disjunction is that it helps to avoid the translation of structures containing disjunctions into a disjunctive normal form, which, given the size of structures in question, could lead to a serious efficiency problem. UDiNe has a mechanism for treating feature values defined as *functional constraints*.

Grammar The DISCO grammar is a German grammar whose syntactic part was developed by and described in [Netter, 1993] and which has an integrated semantic representation described in [Nerbonne, 1992], [Kasper, 1993]. The style of the grammar follows very much the spirit of Head Driven Phrase Structure Grammar (HPSG) [Pollard and Sag, 1987], [Pollard and Sag, to appear]. The grammar is *reversible* in the sense that it is used for both parsing and generation.

Present coverage of German in the DISCO grammar[2] includes on the

- nominal level: determiners and numerals, bare plurals and mass nouns, postnominal prepositional phrases, nominal ellipsis
- adjectival level: complex and simple adjective phrases, attributive and predicative functions
- verb level: V-initial, V-second and V-final, modal verbs and construction of verb complexes
- clausal level: free insertion of adjuncts, topicalization, sentence types (Y/N and WH-interrogatives, imperatives, declaratives).

In addition, the grammar has been extended to include multiword lexemes, permitting opaque idioms to reside in the lexicon. Fig. 2 shows an example of such an entry. This example demonstrates also the uniform representation of different levels of linguistic information in HPSG.

Semantic information is represented within the grammar following the ideas of constraint-based semantics [Nerbonne, 1992]. The current status of semantic representation in the DISCO project is described in [Kasper, 1993]. It follows the practice to specify the semantic representation only partially up to a level of 'quasi-logical form' which leaves for example some contextual restrictions out of consideration. These are left to a second step of semantic interpretation. Such quasi-logical forms contain the following kind of information:

[1] There is now a new version of TDL available, which enables partial or delayed type expansion, the incremental definition of types, negation, disjunctive types and partitioning. However this version needs still to be integrated into the whole system.

[2] The following is not a complete list of the current coverage. A detailed description can be found in [Netter, 1993].

(1) $$ {}_{np}\left[\text{CAT}\left[\begin{array}{ll} \text{SYN} & \left[\text{LOCAL}\left[\begin{array}{ll}\text{HEAD} & \left[\text{MAJ}\left[\begin{array}{l}\text{N}\ +\\ \text{V}\ -\end{array}\right]\right]\\ \text{SUBCAT} & \langle\ \rangle\\ \text{LEXICAL} & +\end{array}\right]\right]\\ \text{SEM} & \left[\begin{array}{ll}\text{PRED} & \textit{time}\\ \text{HOUR} & \boxed{1}\\ \text{MIN} & \boxed{2}\end{array}\right]\\ \text{MORPH} & \left\langle \left[\begin{array}{l}\text{STEM}\ \langle \textit{cardinal}\rangle\\ \text{HEAD}\left[\text{VALUE}\ \boxed{1}\right]\end{array}\right], \left[\begin{array}{l}\text{STEM}\ \langle \textit{uhr}\rangle\\ \text{HEAD}\left[\text{NUM}\ \textit{sg}\right]\end{array}\right], \left[\begin{array}{l}\text{STEM}\ \langle \textit{cardinal}\rangle\\ \text{HEAD}\left[\text{VALUE}\ \boxed{2}\right]\end{array}\right], \right\rangle \end{array}\right]\right] $$

Figure 2: An example of a multi-word lexeme (taken from [Netter, 93]) that covers time expressions such as "14 Uhr 30" (2.30 p.m.).

- predicate-argument-structure
- thematic roles of arguments
- the modification relation
- sortal and selectional restrictions on predicate-argument structures and modification. These have proved to be very efficient means for disambiguation especially of PP-attachment ambiguities (see [Kasper, 1993]).

$\mathcal{NLL}$ Although the core formalism has a great deal of power, the DISCO system also provides a logical form module which facilitates translation into various types of back-end systems. $\mathcal{NLL}$ is a representation of standard predicate logic, with lambda abstraction and several features that support representation of natural language constructions [Laubsch and Nerbonne, 1991]. These include

- Named predicate argument positions

```
ship(agt:John th:shipment-47
     time:15-Mar)
```

- Plural terms

  Jim and John are shippers.
```
(exist ?x shipper(inst/i:?x) =
(arg1:+{Jim,John} arg2:?x))
```

- Location terms

  John is in Saarbrücken on the Saar.
```
located(th:John
      loc:reg-X{SB,on-fn(Saar)})
```

- Restricted parameters
- Generalized quantifiers
- Complex determiners

Modularity is achieved by providing several types of interfaces. $\mathcal{NLL}$ structures may be created using constructor functions, a structure parser, or a feature description language.

The goal of modifiability is realized through the implementation of $\mathcal{NLL}$ using the compiler tools, Zebu (a LISP version of YACC, [Laubsch, 1992]) and REFINE.

Processing Components In the following we describe briefly the main processing components of the linguistic kernel (cf. fig. 1).

Scanning The scanner is mainly responsible for preprocessing the input string. It is implemented in LEX and YACC. The scanner can expand abbreviations into their full forms, as for example "h" into "Uhr", "Jan." into "Januar", etc. For tokens algorithmically encoding their denotations the scanner also performs a morphological analysis by assigning a feature structure to them. Such tokens are, above all, cardinal and ordinal numbers, e.g., "12" and "12.", but also complex time and date expressions, such as "14:31:15" or "12.03.1993".

Morphology The morphological component receives as input those tokens which have not been analysed by the scanner. It produces as output a feature structure which contains as a key or index the lemma of the respective item, as well as other relevant morphosyntactic information which uniquely identifies the form. During generation, the morphological component receives as input a feature structure description of the form to be produced.

At present, the morphological information is precompiled into a morphological full form lexicon, so that runtime processing reduces to the lookup of full forms and the initialization of the lexical component with the associated feature structures. The precompilation is performed by the X2MORF system described in [Trost, 1991]. Part of this system was redesigned with the results that the feature part is now also specified in TDL, and that the morphology can be integrated into the system for a full runtime analysis.

Parsing The parser is a bidirectional bottom-up chart parser which operates on a context-free backbone implicitly contained in the grammar. The parser provides parameterized general parsing strategies, as well as giving control over the processing of individual rules. For example, it is possible to set the control strategy to a breadth-first strategy, to give priority to certain rules, or to determine in which order types of daughters, e.g., head daughters, adjunct daughters etc., as well as individual daughters in a specific rule are processed. In addition, the parser provides the facility to filter out useless tasks, i.e., tasks where a rule application can be predicted to eventually fail due to an inevitable unification failure.

Generation The surface generator currently in use is a variant of the semantic-head-driven algorithm described in [Shieber *et al.*, 1991]. It is a (syntactic) bottom-up process guided by semantic information passed top-down. Given a semantic representation expressed as a typed feature structures it generates all expressions permitted by the grammar. Instead of using a full-form lexicon as it is the case for [Shieber *et al.*, 1991] the generator yields as output a sequence of feature structure containing the lexical stems together with morphosyntacic information. These elements are then passed to the morphological component which computes the appropriate surface forms.

Speech act component The current DISCO system incorporates a first prototype of a surface speech act component developed by and described in more detail in [Hinkelman and Spackman, 1992].

In linking text to task, it is crucial to capture not only propositional content of linguistic expressions but also the attitude or intention behind them. Speech acts, or utterances construed as actions, serve this purpose. An agent is assumed to be an inference system with a planning algorithm. The planner constructs chains of actions that would, if executed, result in the achievements of the agent's goal. Plan recognition then consists in reversing this process, taking observed actions as evidence for the intentions.

In the approach followed in the DISCO system the HPSG framework is used to represent the necessary inference rules for speech act recognition (e.g. simple implications can be expressed using negation and distributed disjunction).[3] Based on this kind of representation the relationship between sentence types and conventionalized speech acts (like INFORM or REQUEST) can be expressed declaratively in the grammar. Parsing of an expression now not only yields an propositional content but also the possible types of speech acts. The current version of the speech act component offers solutions to the following subproblems:

- general linkage between utterances and intentions
- specific mechanisms linking speech acts to surface representations
- reliable, n-way speech acts.

3 THE DISCO DEVELOPMENT SHELL

The architecture of the DISCO system and COSMA have both been realized using the DISCO DEVELOPMENT SHELL, which was developed by the author and which we are now going to present in more detail.

The use of modern programming techniques in system integration is crucial to support the following desiderata:

- modularity of NLP components
- experimentation with flow of control
- incorporation of new modules
- building of subsystems and standalone applications
- accommodation of alternative modules with similar functionality

[3] The current approach only supports speech act recognition but it is planned to use the same basic approach also for the planning of speech acts (see [Hinkelman and Spackman, 1992]).

In order to perform the tasks mentioned above we have chosen a *two–step approach* to realize DISCO's architecture:

1. In a first step the architecture is described and developed independently of the components to be used and of the particular flow of control. Possible components are viewed as black boxes and the flow of control is described independently of specific components. In such an abstract view the architecture realizes only a schema called the *frame–system*.
2. Next the frame–system has to be 'instantiated' by the integration of existing modules and by defining the particular flow of control between these modules.

It is useful to divide the system components into different types according to their specific tasks. Currently, we distinguish:[4]

- tool components (e.g., graphic devices, printer, debugger, errror handler)
- natural language components (e.g., morphology, parser, generator, speech act recognition)
- control component

[4] We do not assume that this list is complete. For example, it is also possible to have knowledge sources as components of the frame system.

In order to obtain a high degree of *flexibility* and *robustness* (especially during the development phase of a system) the control unit directs and monitors the flow of information between the other components. The important tasks of the control unit are:

- to direct the data flow between the individual components
- to define which components should run together to form a 'subsystem'
- to check the data received from one component before they are sent to another one
- to manage global memory and call specific tools

There is a command level for direct communication with the kernel. The purpose of the command level is to provide commands that allow users to run subsystems, to activate or inactivate tracing of modules and to specify printing devices. Users may also specify values for global variables interactively or with configuration files for each module.

Object Oriented Design If a new component must be integrated, one would like to concentrate only on those parts that are of specific interest for these new components. Algorithms or data which are common to all components (or components of a specific type) should be defined only once and then be added automatically for each new component without side–effects to other already integrated components.

We have choosen an *object–oriented programming style* (OOP style) using the Common Lisp Object System (CLOS) in order to realize the two–step approach described above. In the object–oriented paradigm a program is viewed as a set of objects that are manipulated by actions. The state of each object and the actions that manipulate the state are defined once and for all when the object is created. The essential ingredients of object–oriented programming are *objects*, *classes* and *inheritance*. *Objects* are modules that encapsulate data and operations on that data. Every object is an instance of a specific class which determines its structure and behaviour. *Inheritance* allows new classes to specialize already defined classes. The result is a hierarchy of classes where classes inherit the behaviour (data and operations) from superclasses. The advantage for the programmer is that she need only specify to what extend the new class differes from the class(es) it inherits from. This supports the design of modular and robust systems that are easy to use and extend.[5]

CLOS The main programming language for the DISCO project is Common Lisp. Because CLOS is defined to be a *standard language extension* to Common Lisp it is easy to combine 'ordinary' Lisp code with OOP style. CLOS allows us to define an hierarchical organization of classes that models the relationship among the various kinds of objects. Furthermore, because CLOS supports multiple inheritance it is possible to define methods that are defined for particular combinations of classes. Therefore a large amount of control flow is automatically realized by CLOS. This helps us to concentrate on the individual properties of new components, which simplifies and speeds up their integration extremly. Of course, CLOS itself does not enforce modularity

[5]The reader should consult e.g., [Keene, 1988] for an excellent introduction into CLOS if more detailed information on object oriented programming is of interest.

or makes it possible to organize programs poorly; it is just a tool that helps us to achieve such modular systems.

DISCO's Class Hierachy The DISCO DEVELOPMENT SHELL consists of the *class hierarchy* and the specification of class specific *methods*. Every type of component and its specializations are defined as CLOS classes. Figure 3 shows a portion of the current hierarchy.

MODULE is the most general class. All other classes inherit its data structure and associated methods. The class LANGUAGE COMPONENT subsumes all modules of the current system which are responsible for language processing. A module that is actually used in the system is an *instance* of one of the classes.

New modules are added to the system by associating a class with them. CLOS supports dynamic extension of the class hierarchy so that new types can be added even at run–time. For example, if we wanted to add a new parser module we would either use the already existing class PARSER or define a new class, say ALTERNATIVE-PARSER. In the first case we assume that we only need the methods that have already been defined for the PARSER class. In the second case we would have to add new methods or could specialize some of the methods that ALTERNATIVE-PARSER inherits from PARSER. In principle it could also happen that the new parser shares many properties with DISCO-PARSER. In that case we would have to refine the parser subnet in order to avoid redundancy.

Protocols The flow of control between a set of components is mediated by means of *protocols*. Protocols are methods defined for the class *controller*. They specify the set of language components to be used and the input/output relation between the language components. All current protocols are defined using the same structure as shown in fig. 4.

The controller uses the generic function CALL-COMPONENT to activate an individual language component instance, specialized to the appropriate subclass. Control flow is determined by the sequence of CALL-COMPONENT invocations. Between calls, output is verified and converted to the following component's input format by calling the generic function CHECK-AND-TRANSFORM. This mechanism is very important to support *robustness* especially during the development phase of the system. Specific methods are defined for each module that indicate what to do if a module does not come up with a correct result. These methods are activated by the controller during the call of CHECK-AND-TRANSFORM. In the current version of the system further processing is then interrupted and the user is informed about the problem that occured.

For example, the output of MORPHOLOGY defines the input to PARSER and so on. By calling (CHECK-AND-TRANSFORM CONTROLLER MORPHOLOGY PARSER) the controller checks whether the morphology yielded a valid output and eventually transforms the output for the parser. If MORPHOLOGY detects an unknown word X further processing is interrupted and the user receives a message notifying him that X is unknown to the morphological component.

Some Remarks If two adjacent components have been proven to work together without problems CHECK-AND-TRANSFORM need not be called for them as it is the case for the components COMPONENT_2 and COMPONENT_3 in the example shown in fig. 5.

Input and output for the whole system is specifed using general communication

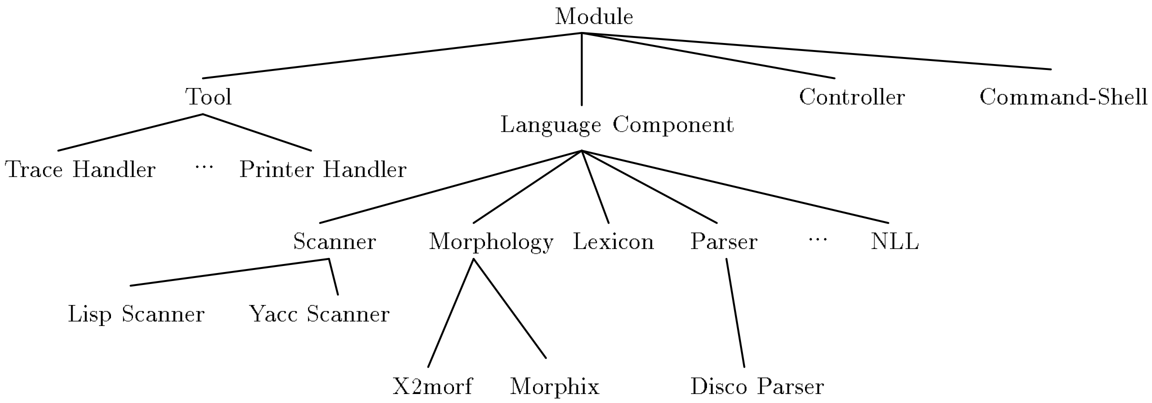

Figure 3: A portion of the current class hierarchie in the DISCO system.

(call-component controller $component_1$*)*
(check-and-transform controller $component_1$ $component_2$*)*
(call-component controller $component_2$*)*
(check-and-transform controller $component_2$ $component_3$*)*
(call-component controller $component_3$*)*
$\vdots$
(check-and-transform controller $component_{(n-1)}$ $component_n$*)*
(call-component controller $component_n$*)*

Figure 4: Schematic structure of protocols.

channels. In the normal case this is the standard terminal input/output stream of Common Lisp. In the COSMA system an e–mail interface for standard e–mail is used as the principle communication channel. Besides the general input/output device the controller also manages *working* and *long-term memory*. These memories are used to process a sequence of sentences. In this case the controller stores each analysed sentence in long–term memory.

The architecture by itself is not restricted to pipeline processing but would be used in modeling *cascade* or *blackboard* architectures as well. In the latter case the working memory can be used to realize the (possibly structured) blackboard. This has already been partially realized in the current version. In principle, the architecture appeals to be general enough to realize *negotiation*–based architectures.

Current Subsystems For grammar debugging it is possible to run serveral subsystems (called standalone applications), which are activated via the command level. For example, one might want to run the parser and generator without morphology or only the set of components necessary during analysis aso. In each case the same functionality is available as well as the same set of tools. In principle it would be possible for a user to define protocols himself e.g., to test self–written modules because the integration of modules and the definition of protocols takes place in a standardized fashion.

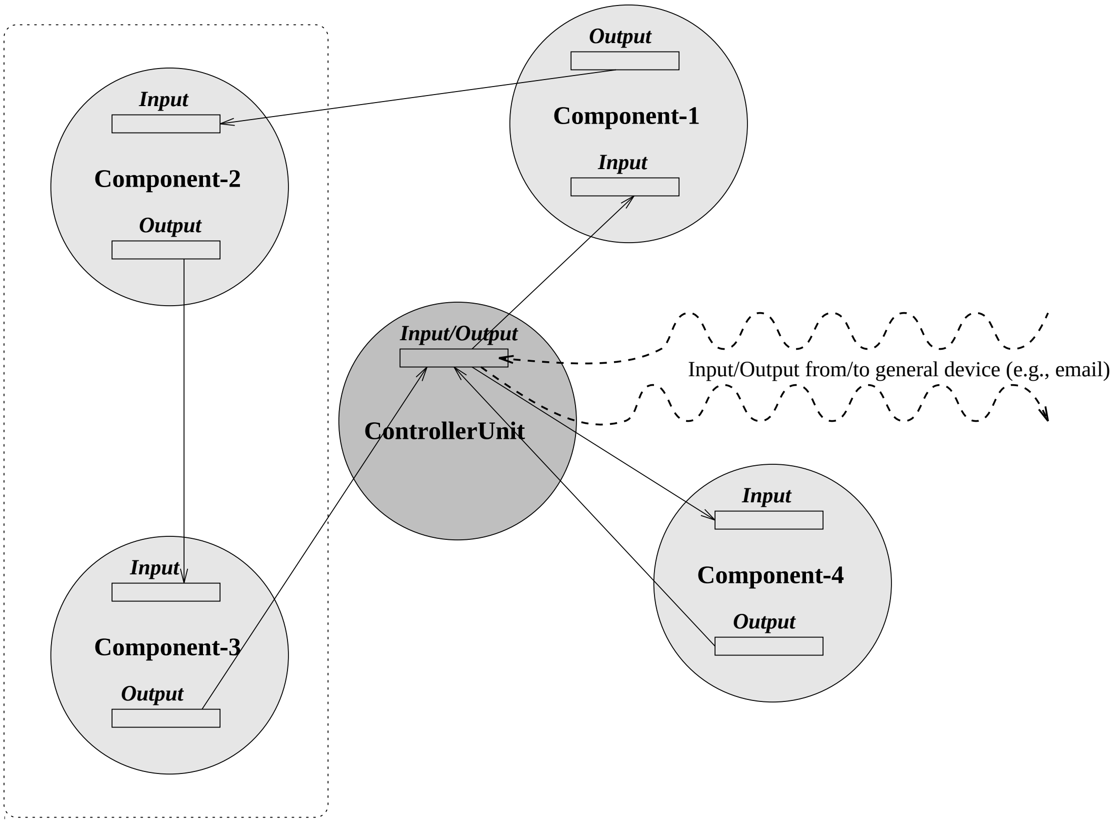

Figure 5: Flow of control between four components. In this protocol component 2 and 3 interact directly. The controller views them as being one component (indicated by the dotted lines around them).

4 OVERVIEW OF THE COSMA SYSTEM

In this final section we will give a brief overview of the COSMA system. The principle idea behind the COSMA system is to support scheduling of appointments between several human participants by means of *distributed intelligent calendar assistents*. Instead of using a centralized solution where only a global calendar database is maintained we have choosen a distributed solution. Scheduling of appointments between several participants is than viewed as a cooperative negotiation dialogue between the different agents.

We assume that each person has its own (therefore local) calendar database available on her computer where each calendar is managed by an individual planning component. Each COSMA system consists of three basic components

- An intelligent assistent that keeps and manages the calendar database
- A graphical user interface to the calendar data–base application planner
- The natural language system DISCO

It is assumed that electronic mail will be used as a basic communication channel. Information concerning the schedule of particular appointments (e.g., request to arrange a meeting, cancelation of a previously setup appointment or other information relevant in performing some negotiation) is sent around the set of relevant participants via e–mail. Using standard e–mail software has the advantage that scheduling of appointments can be done in a distributed and asychronous way.

Natural language (NL) comes into play because we allow humans to participate who have no calendar assistent available. The only restriction is that they have electronic mail available. Such a (poor) person is responsible for mantaining an old–fashioned calendar but is allowed to use *natural language* during appoinment negotiation. Consequently, each COSMA system needs to be able to process natural language, either to understand a NL dialogue contribution or to produce one.

Each user of a COSMA system has access to the calendar data–base by means of a graphical user interface. The graphical user interface — developed by Stephen Spackman who named the tool DUI — is used to display and update existing items and enter new items into the data–base. The intelligent calendar manager maintains the calendar database. The current version (developed by the AKA–MOD group of the DFKI) consists of time processing functions, a finite-state protocol for arranging appointments, and an action memory storing the protocol state and original e–mail for each arrangement in progress. The principle task of the DISCO system is to extract that information from an natural language expression that can be used by the calendar manager. DISCO is also responsible for the production of natural language text from the internal representation of scheduling information computed by the calendar manager. The produced text is sent in addition to the internal structure of scheduling expressions to the participants via e–mail. [Busemann, 1993] describes the current approach for generating natural language expressions in COSMA in more detail.

Short Example Figure 6 gives an overview of a configuration where three participants, a human (Tick) and two COSMAs (Trick, Track) are involved.

A possible appointment scheduling is as follows (abstracting away from details):

Track to Trick and Tick:
arrange(meeting, 21.10.1992,1p.m.)
Trick to Track:
accept(meeting)
Tick to Track:
Ich bin mit dem Termin einverstanden. (*I accept the appointment*).
Track to Trick and Tick:
confirm(meeting)

In words: Track wanted to arrange a meeting and sends this request to Trick and Tick (in form of an internal planning expression). Trick automatically sends an acception. Because there are no conflicting entries in his calendar data–base, Tick sends an acception using NL. Track will update its calendar while sending a confirmation to the two participants notifying them that all participants accepted the appointment.

The current version of the system is able to handle more complex dialogs, e.g., appointment scheduling initiated by a non-COSMA user, cancellation and modification of already set up appointments.

5 CONCLUSION

In this paper we have given an overview of the DISCO system and a detailed description of the DISCO DEVELOPMENT SHELL. The DISCO DEVELOPMENT SHELL has been proven very useful in setting up the COSMA system. It was possible to integrate the new modules independently from the rest of the system. Existing modules have been exchanged by new ones without the need of adding new methods. Because different researchers were

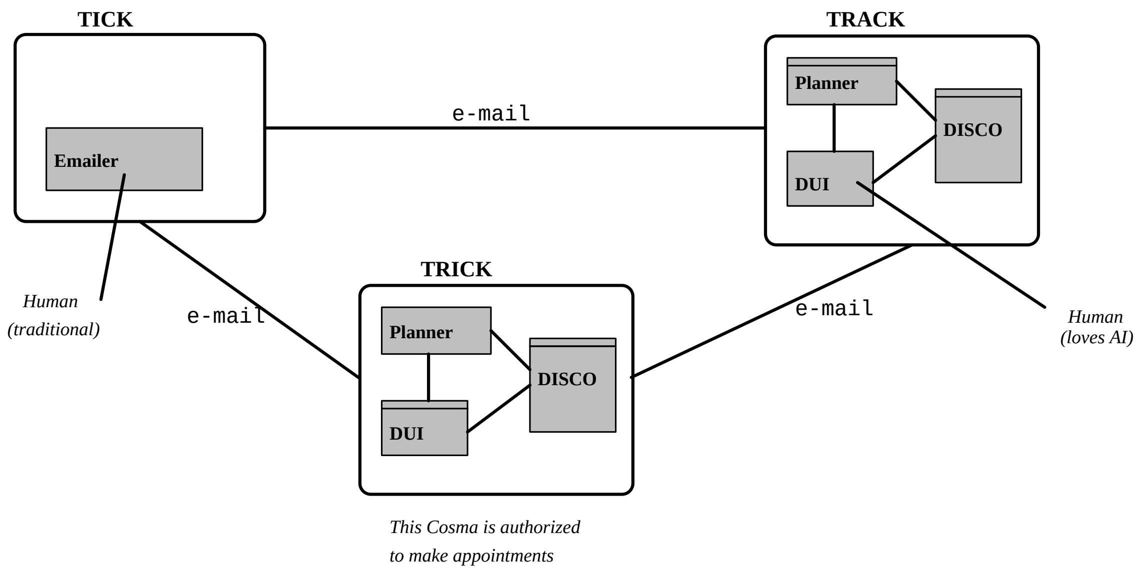

Figure 6: General Overview of the Sample Scenarios

able to run subsystems the development of the whole system could be done in a distributed way. Therefore eight very different modules have been intergrated in less than three weeks including test phases.

Based on these experiences we believe that the oject–oriented architectual model of our approach is a fruitful basis for managing large–scale projects. It makes it possible to develop the basis of a whole system in parallel to the development of the individual components. Therefore it is possible to take into account restrictions and modifications of each component as early as possible.

References

[Busemann and Harbusch, 1993] Stephan Busemann and Karin Harbusch, editors. *Proc. DFKI Workshop on Natural Language Systems: Modularity and Re-usability*, Saarbrücken, germany, 1993.

[Busemann, 1993] S. Busemann. Towards the configuration of generation systems: Some initial ideas. In Busemann and Harbusch [1993].

[Hinkelman and Spackman, 1992] Elizabeth A. Hinkelman and Stephen P. Spackman. Abductive speech act recognition, corporate agents and the cosma system. In W. J. Black, G. Sabah, and T. J. Wachtel, editors, *Abduction, Beliefs and Context: Proceedings of the second ESPRIT PLUS workshop in computational pragmatics*, 1992.

[Inc., 1990] Reasoning Systems Inc. Refine user's guide. Technical report, Palo Alto, CA, 1990.

[Kasper, 1993] Walter Kasper. Integration of syntax and semantics in feature structures. In Busemann and Harbusch [1993].

[Keene, 1988] Sonya E. Keene. *Object-Oriented Programming in Common Lisp. A Programmer's Guide to CLOS.* Addision-Wesley, 1988.

[Laubsch and Nerbonne, 1991] J. Laubsch and J. Nerbonne. An overview of nll. Technical report, Hewlett-Packard Laboraties, Palo Alto, CA, 1991.

[Laubsch, 1992] J. Laubsch. Zebu: A tool for specifying reversible lalr(1) parsers. Technical report, Hewlett-Packard Laboraties, Palo Alto, CA, 1992.

[Nerbonne, 1992] John Nerbonne. Constraint-based semantics. In Paul Dekker and Martin Stokhof, editors, *Proceedings of the 8th Amsterdam Colloquium*, pages 425–444. Institute for Logic, Language and Computation, 1992. also DFKI RR-92-18.

[Netter, 1993] Klaus Netter. Architecture and coverage of the disco grammar. In Busemann and Harbusch [1993].

[Pollard and Sag, 1987] C. Pollard and I. A. Sag. *Information Based Syntax and Semantics, Volume 1.* Center for the Study of Language and Information Stanford, 1987.

[Pollard and Sag, to appear] C. Pollard and I. M. Sag. *Information Based Syntax and Semantics, Volume 2.* Center for the Study of Language and Information Stanford, to appear.

[Schäfer and Krieger, 1992] Ulrich Schäfer and Hans-Ulrich Krieger. TDL *extra-light User's Guide: Franz Allegro Common LISP Version.* DISCO, 1992.

[Shieber *et al.*, 1991] S. M. Shieber, F. C. N. Pereira, G. van Noord, and R. C. Moore. Semantic-head-driven generation. *Computational Linguistics*, 16:30–42, 1991.

[Trost, 1991] Harald Trost. X2MORF: A morphological component based on augmented two-level morphology. Technical Report RR-91-04, Deutsches Forschungsinstitut für Künstliche Intelligenz, Saarbrücken, Germany, 1991.