The DISCO DEVELOPMENT SHELL and its Application in the COSMA System

Günter Neumann*

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

Abstract

This paper describes the DISCO DEVELOPMENT SHELL, which serves as a basic tool for the integration of natural language components in the DISCO project, and its application in the COSMA system, a Cooperative Schedule Management Agent. 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 location. 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 processing of multiagent natural language dialogue. Multiagent capabilities make it an appropriate front end for autonomous cooperative agents, exemplified by the COSMA system described in section 3. The primary task of DISCO is to serve as a kernel linguistic system in order to support distributed cooperative dialogues.

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

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

The architecture of the DISCO system and COSMA have both been realized using the DISCO DEVELOPMENT SHELL, which we are introducing in the following section. Because of the lack of space we give only a short description of the basic ideas.

2 Overview of the DISCO development shell

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 'contentless' 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.

v

It is useful to divide the system components into different types according to their specific tasks. Currently, we distinguish:[1]

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

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:

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

to direct the data flow between the individual components

to define which components should run together to realize 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 is different from the class(es) it inherits from. This supports the design of modular and robust systems that are easy to use and extend.[2]

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

[2]'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.

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 1 shows a portion of the current hierarchy.

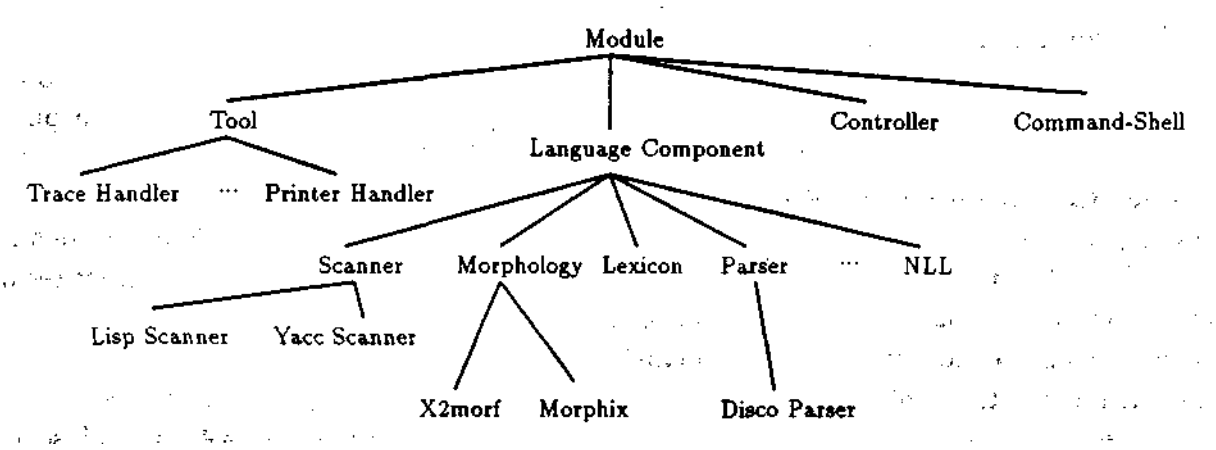

Figure 1: A portion of the current class hierarchic in the DISCO system.

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 hierarchic 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. This would mean that we 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 schema:

```
(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)
```

```
(check-and—transform controller component-(n-1) component-n)
(call-component controller component-n)
```

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 detected an unknown word X further processing would be 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 in the following example:

```
(call-component controller component-1)
(check-and-transform controller component-1 component-2)
(call-component controller component-2)
(call-component controller component-3)
```

```
(check-and-transform controller component-(n-1) component-n)
(call-component controller component-n)
```

In this example, COMPONENT-2 and COMPONENT-3 interact directly, as shown in Figure 2.

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

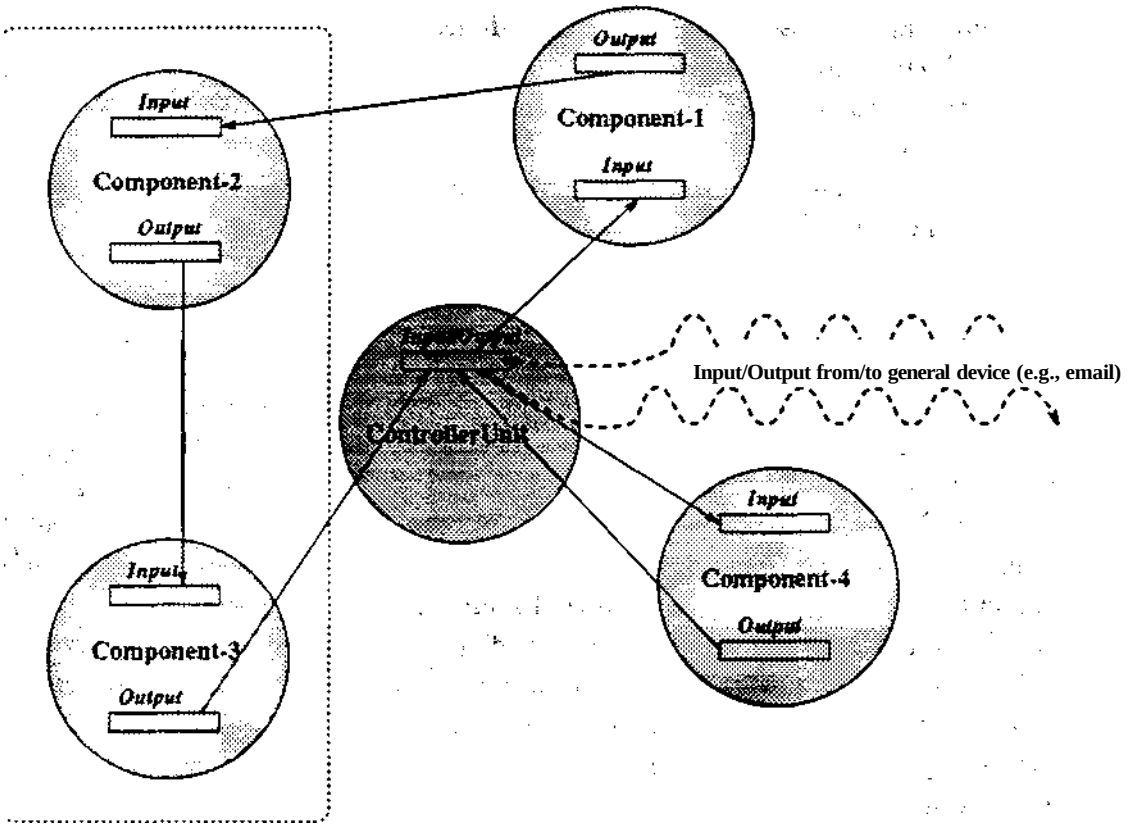

Figure 2: Flow of control between four components. In this protocol component 2 and 3 interact directly. The controller views them as being one component (indicated by the dotted lines around them).

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 appeels to be general enough to realize *negotiation-based* architectures.

Status of the DISCO kernel We will give a very brief overview of the current status of the DISCO kernel. We are using an HPSG style of linguistic description (see Netter this volume). The grammar is formalized using the type description language TDL. The basic machinery for linguistic processing is UDINE, a powerful feature structure unifier. The unifier is used in TDL and in almost all linguistic processing units (like parser, generator). The grammar, TDL and UDINE (see Netter this volume for references) constitute the basic linguistic resources. They are not part of the class hierarchy (but accessible from the command level). The basic tool we are using for tracing, debugging and editing feature structures is FEGRAMED, developed by Bernd Kiefer in the DISCO project. FEGRAMED also serves as generic printing device in the kernel machinery as well as in the COSMA system. 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.

3 Overview of the COSMA system

In this final section we will give a brief overview of the COSMA system. Today, there already exists appointment and resource scheduling tools that allow to display day, week, month, year views or schedule single or repeating events and set beeping, flashing, or pop-up reminders. 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 one planner mantains a global calendar data-base we have choosen a distributed solution. We assume that each person has its own (therefore local) calendar data-base available on hers computer where each calendar is managed by an individual plannning component. Scheduling of appointments between several participants is viewed as a cooperative negotiation dialogue between the different agents.

It is assumed that electronic mail will be used as a basic means of communication. 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 assisent access. 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. To sum up, each COSMA 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 , ;

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 Stephan 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 mantains the calendar database.

The current version (developed by the AKA-MOD group) 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 natural language system is used to analyse natural language dialogue contributions, which is the normal case for non-COSMA participants. The natural language system has been developed in the DISCO project, and hence is what we refer as DISCO. 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. Netter (this volume) and Nerbonne et al. (this volume) describe in detail the several knowledge sources and processes that are currently in use to solve this task. 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. Buseman (this volume) describes the current approach for generating natural language expressions in DISCO in more detail.

Short Example Figure 3 gives an overview of a configuration where three participants, a human (Tick) and two COSMAs (Trick, Track) are involved.

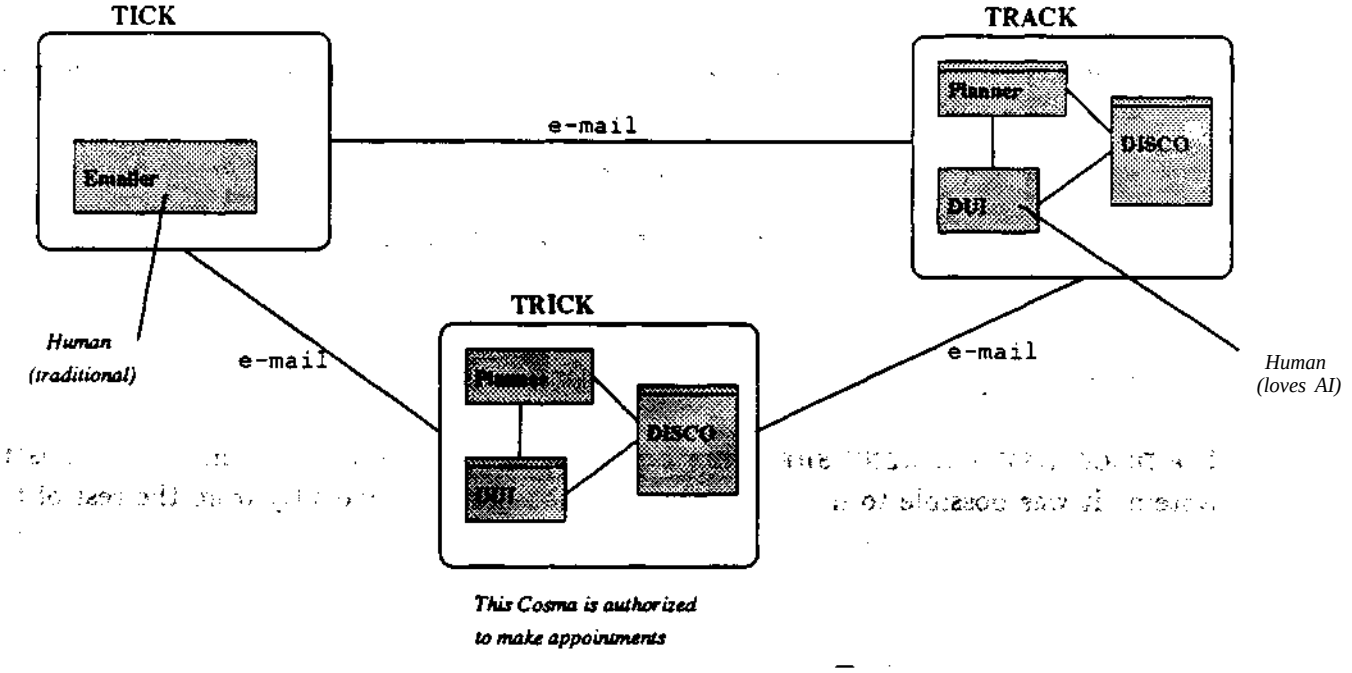

Figure 3: General Overview of the Sample Scenarios

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. (/ *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. 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.

To be able to integrate the DISCO system into this domain the following modules have been integrated or modified:

- Exchange of a Lisp-based scanner for a more powerful one implemented using the Unix tools YACC and LEX.
- Integration of a surface-based speech act system (SAR which has been placed between the parser and NLL, cf. (Hinkelman and Spackman, 1992)).
- Application interface for mapping NLl expressions into an internal representation language for the planner (see Nerbonne et al. (this volume)).
- exchange of the semantic head-driven generator with a 'canned text' generator
- A set of communication interfaces to e-mail, the graphical user interface and the planner

4 Conclusion

The DISCO DEVELOPMENT SHELL has been proven very useful in setting up the COSMA system. It was possible to intergrate 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 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

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.

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