Dialogue Control
in the Alparon System

A.C. Andeweg

August 1999

TU Delft

Alparon

Delft University of Technology,
Faculty of Information Technology and Systems,
Knowledge Based Systems group,
Alparon group

Abstract

This report describes the dialogue control component of the Alparon system. The Alparon system is an automatic speech processing system for information retrieval and transaction services. Work within the Alparon group is focussed on dialogue management. The dialogue control component is part of the dialogue manager and is responsible for structuring the dialogue and generation of the system responses.

Information retrieval dialogues are task-oriented, the dialogues are goal directed. A plan-based approach is used to control the dialogue. A goal structure is developed which should control the dialogue.

Dialogue control is implemented using two processing modules and a blackboard. The dialogue updating module updates the goal structure after every dialogue turn of the user. The response generation module generates a response based on the modified goal structure. And the control board stores all the information concerning dialogue control. These components together form the framework for dialogue control. This framework is an engine, where the exact operation of the engine can be specified by separately defined rulesets. These rulesets define the actual strategies for dialogue control.

Preface

This document is the report of my graduation work. This work is done for the Alparon group. The Alparon group is a research group at the section Knowledge Based Systems, headed by professor Koppelaar, of the Information Technology and Systems faculty at the Delft University of Technology.

The graduation committee consists of the following persons:

- prof. dr. H. Koppelaar
- dr. drs. L.J.M. Rothkrantz
  Supervisor
- prof. dr. ir. E.J.H. Kerckhoffs

I like to thank Leon Rothkrantz and especially Robert van Vark (my day to day mentor) for their support and supervision of my graduation work.

August 1999
Niels Andeweg

Contents

Chapter 1

Introduction

Dialogue management is the key factor in automated speech processing systems for information retrieval and transaction services for it can overcome the short-comings of a one-shot approach. In a one-shot approach the user specifies his requirements to the system in one turn. Then, the system tries to understand those requirements and give an appropriate response, also in one turn. If this approach fails, the user has to try again. There is no dialogue or interaction, just an action followed by a reaction.

To behave in a more user-friendly fashion the system has to be able to cope with misunderstandings, errors in speech recognition, user requirements spread over more than one dialogue turn, dialogue phenomena, etc. To cope with these, the system engages in a dialogue with the user. Therefore dialogue management is an essential element for a user-friendly automated speech processing system.

The aim of the Alparon group is the development of such an automated speech processing system. Therefore, the Alparon group is focussed on dialogue management. Other components of the automated speech processing system are only incorporated to make a complete system, like speech recognition, parser, speech generation, etc. This report describes the design of the dialogue control components of the Alparon system. The goal is to design a dialogue management system that uses strategies found in human-human information retrieval dialogues. So the user will be able to use the system in a natural way.

The Alparon system up until this moment is placed in the public transport information domain. The first version of the system will be restricted to train timetable information. The system is to supply the user with a travelplan for a certain train connection in the Netherlands. First the system must determine what travelplan the user wants, then it must look up that information and present it to the user.

Before the beginning of this work an overall model of the Alparon system was already designed with a dialogue manager at the core. The dialogue manager consisted of several modules. A modular design was chosen to create a flexible, extensible system. Some of the modules were already developed; others are currently being developed.

The dialogue manager should be able to interpret the user's utterance in the context of a dialogue. Thereafter the dialogue manager must generate a response to the user's utterance. Information on the current state of the dialogue and domain knowledge can be used to perform these tasks.

This report is concerned with the second task of dialogue management. A mechanism to control the dialogue should be designed and implemented. This mechanism is to reason out a system response after the user's utterance is interpreted. These responses are the means by which the system can control the dialogue. An approach surpassing a finite-state automaton approach should be applied.

In the original design of the Alparon system this would mean the design and implementation of the response generation module and the control board. This module would generate the system's response using and modifying the specific information for dialogue control on the control board. But this design is by no means final and it could well be that other, new modules will be part of the dialogue control mechanism.

This report can roughly be divided into three parts. The first part will mainly have a descriptive character. Chapter 2 will be concerned with general characteristics of information retrieval dialogues. The next chapter will describe automatic speech processing systems and the place of dialogue management in it. Then the Alparon system is described. The second part is concerned with the dialogue control mechanism, the goal of this work. The ideas behind the chosen approach are first explained in chapter 5. The next chapter gives an overview of the concepts used in the approach, followed by a chapter describing the design of the necessary modules for this mechanism. The last part is the development of a train timetable information system using the mechanism for dialogue control described in the previous part. The conclusions and suggestions for future work conclude this part.

Chapter 2

Information retrieval systems

This chapter provides some general characteristics of information retrieval systems.

An information retrieval system has a database containing information which has to be made available to the users, as the user wants certain information that is present in the database. It could be said that the user has some informational needs. We assume the desired information is indeed present in the database. After the informational needs are specified by the user, the information can be obtained from the database and be communicated to the user.

The system supplies information concerning a certain domain. The system is aware what databases are at its disposal and how to retrieve information from them. So the system knows what is necessary to specify informational needs correctly, i.e. what information the user should supply for the system to make a successful database query (a query which yields the information the user wants).

In more advanced information retrieval systems this does not have to be true. Intelligent agents will try to gather or deduce the information requested from a series of information sources available at that time.

The user himself is usually not fully aware of these things. He has some informational needs and tries to use the system to fulfill those needs. He will probably not know the exact limitations of the information retrieval system and how to specify his informational needs correctly[1]. When the system does not succeed in determining the informational needs of the user or the user's needs pass the limitations of the system, the system can not be of service. The system has to inform the user of this fact and end the session. Out-of-domain and out-of-vocabulary detection is needed.

The Alparon system will use natural language for the interaction between the user and the system. The user will engage in a dialogue with the system in order to gather the information he needs. So while it will be harder to get the user to specify his informational needs correctly (there is more freedom of expression in natural language, especially speech), the system can use more than one turn to accomplish its tasks.

[1] When spoken language is used it will be even harder to specify the informational needs correctly.

The first part of the dialogue will focus on the determination of the user's informational needs. There are several obstacles that can occur in this phase.

- Incompleteness.
  The user will not always supply all the necessary information for the system to determine his precise informational needs. The user does not specify his needs completely. In many cases he does not know what information the system exactly needs. E.g. the user does not specify on what day he wants to travel.

- Freedom of expression.
  The user can use more than one expression to specify a certain informational need. Several expressions can be used to refer to the same concept. E.g. 'a quarter past four' and 'in one hour' can mean the same, namely 16:15.

- Under-specification.
  The user refers to a certain concept, but uses an expression not specific enough to unambiguously determine the informational needs. E.g. 'Amsterdam' can refer to more than one station in the city Amsterdam.

The use of spoken natural language also produces obstacles.

- Misunderstandings.
  The user's utterance will not always be correctly understood by the system. The system has not correctly 'heard' the user - the literal recognition is incorrect[2] - or the semantics of an utterance has not been determined correctly by the system.

Finally there are some obstacles specifically related to having a dialogue. An overview of these obstacles - of which anaphoric references, ellipses, indirectness are some examples - can be found in [Churcher et al., 1997].

After the user's informational needs are determined by the system, the database is used to collect the requested information. Next this information has to be communicated to the user. This information should be divided into pieces, the user should be able to comprehend in one turn, and be presented in several turns.

During or after the presentation the user can modify his informational needs. These new informational needs should again be determined unambiguously. The database should be consulted for these new needs, the result be divided and presented to the user.

An automatic dialogue system will have to deal with all these aspects in order to be able to engage in a natural dialogue and be of service to the user.

[2] due to e.g. the speech recognition or typing mistakes

Chapter 3

Automatic Speech Processing

The Alparon system, being developed by the Alparon group, is an automated speech processing (ASP) system. Before the Alparon system is described in more detail in the next chapter, this chapter gives a general overview of ASP systems.

An ASP system is an automated system that uses human speech for interaction with the user. Furthermore the actions the system performs are dependent on the contents of the speech input. So the system not only records the speech but it also has to interpret the speech. The system has to record speech, it has to extract the words and finally it has to give a meaning to these words. This is more complex than voice mail systems where speech is only recorded or dictation systems where only the words are extracted from speech and no meaning is given to these words[1]. Speech does not have to be the only modality for interaction in an ASP system. This overview however only focuses on the aspects of speech interaction for such systems.

The steps encountered in an ASP system that uses speech for both input and output, are usually a follows.

- Speech, spoken by the user, is recorded by a microphone.
- A speech recognition component transforms the recorded sound signal into words.
- The output of the speech recognizer is interpreted by a natural language component to give a meaning to the words.
- The system performs some actions depending on the meaning of the speech input.
- The system formulates a response depending on the result of these actions.

[1] In more complex dictation systems, semantics can be used to make the speech recognition perform better. The vocabulary of dictation systems is also much larger than the vocabulary of ASP systems, which make the speech recognition play a bigger role.

- Finally the system response is transformed to speech and played through a speaker for the user.

So a general model for an ASP system contains the components as depicted in figure 3.1. Not all ASP systems can be captured by this model, but the functionality of these components will usually be present somewhere in an ASP system. The components are discussed in more detail below.

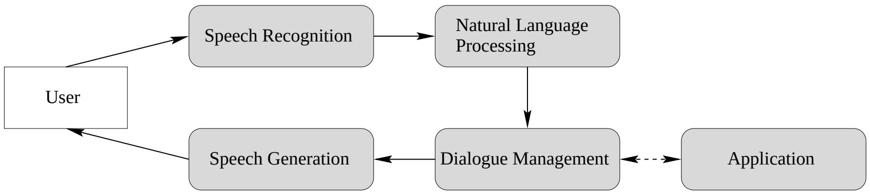

Figure 3.1: General model for an ASP system

3.1 Speech recognition

Speech recognition is the process of transforming a sound signal into text. The literal words uttered are to be recognized. Speech recognition is often classified in several dimensions.

- Discrete versus continuous speech.
  All the words have to be clearly separated from each other for discrete speech recognition. Continuous speech recognition does not place this restriction on speech. A more natural, flowing stream of words can be recognized.

- Size of the vocabulary.
  How many words can the speech recognizer recognize?

- Speaker dependency.
  Does the recognizer only properly recognize one speaker or is the recognizer speaker independent?

- Quality of sound.
  How much noise can the recognizer cope with and how much band-width does it need for the sound signal?

Speech recognition is usually done in several stages. First, speech must be detected. When is the user speaking to the system and when is he silent? Secondly, the sound signal must be segmented into pieces that can be separately recognized. Then speech recognition has to be applied to those segments. Finally, what has been recognized, must be combined to text. The unit of recognition are often phonemes, especially for recognizers with not to small a vocabulary. Phonemes are the basic sound units in natural language. After the phonemes are recognized, they should be combined into words.

The recognition itself is based on statistically trained models, usually Hidden Markov Models (HMM's)[2]. Therefore the result of the recognition is not deterministic but stochastic. The output of the speech recognizer is usually not singular, but provides an n-best list of possible interpretations (n a certain number) with their probabilities (for continuous speech recognizer it produces sentences and for discrete recognizers isolated words).

Non-textual information on speech (intonation, speech rate, intensity, etc.) are normally not captured by the speech recognizers. Prosody, as these information elements are collectively called, provides humans with clues for the interpreting of speech but would add a whole new dimension to speech recognition.

3.2 Natural language component

The task of the natural language component is to add meaning to the recognized words and is sometimes referred to as meaning extraction or semantic interpretation. It transforms the text the speech recognizer has recognized, to a logical structure with a defined meaning within the ASP system. This is still a straightforward processing step of the speech input. The logical structure is context independent.

The level of understanding added by the natural language component, can differ from one ASP system to another. Simple word spotting techniques just look for certain keywords in the recognized text, the rest is ignored. Template matching tries to match the recognized text on templates with predefined, possible parameterized, meanings. Only the parts of the text that match templates are used, the rest is also ignored. Full parsing techniques try to give meaning to the whole text, recognized by the speech recognizer. Robust parsing techniques have to be applied in order to correct failures and misunderstandings of the speech recognition process that are still present in the current state of the art speech recognizing techniques.[3]

The output of natural language components thus ranges from a stream of keywords with a defined meaning (in fact not more than a filtering of the recognized text), to logical structures representing the meaning of the sentences. Again, the output can consist of more than one possibilities. Because it is context-independent, all reasonable, possible interpretations of the meaning that might be valid should be included in the output.

The natural language processing is not always situated in one distinct component in ASP systems. The various steps and functions of meaning extracting can be placed within several components. For most ASP system, the natural language processing does not reach beyond annotated sentences. Further meaning extraction and providing a higher level of abstraction to the speech input is usually integrated in the dialogue manager which is responsible for acting on the user's input.

[Allen, 1995] gives a comprehensive treatise on issues related to natural language understanding.

[2] both for the recognition of phonemes out of sequences of sound fragments and the recognition of words out of phoneme sequences.

[3] Robust parsing techniques are also needed to deal with grammatically incorrect sentences and restarts, common in spoken language.

3.3 Dialogue manager

The dialogue manager usually performs the following tasks.

- Adding context.
  So far the user utterance is interpreted in a context-independent way. The dialogue manager can analyze the utterance in a context. The natural language component determines the meaning of the user's utterance in a context-free way. But placed within a context (of the entire dialogue and application) the meaning can be different or more specific [4]. This is actually part of the meaning extracting but has nothing to do with natural language processing and is usually a part of dialogue management.

- Problem solving.
  Dependent on the meaning of the user's utterance the system should perform some actions. In this way the dialogue manager interfaces with the actual application behind the ASP system.

- Response generation.
  After actions are performed the result of those actions should be communicated to the user. When the ASP system does not have enough information to perform a certain action, the interaction with the user should continue. In other words, the system has to decide on a system response, i.e. what should the system say to the user? [Andeweg, 1999] gives an overview of technique used to control the dialogue.

The kind of interaction the system is supposed to handle with the user using speech, strongly determines the complexity of the ASP system and thus the complexity of the dialogue manager. Some ASP systems use a one-shot approach. The user speaks to the system. The system tries to understand the user's utterance as good as possible and performs certain actions according to this. Then it communicates the results of those actions to the user. When it cannot decide on an certain action, it informs the user of this fact and the whole process starts all over again. Other system are capable of engaging in a dialogue with the user and several turns can be taken by both the system and the user, before and after other actions are performed by the system.

In the latter kind of systems the dialogue manager will be more complex. Adding context using the history of the dialogue, deciding when and which actions should be performed and generating the system responses is more complicated. Whereas in a simple one-shot system the dialogue manager is quite straightforward. There is no dialogue history to complicate the context. It just has to choose some of the predefined actions and report the results to the user. In such systems there is not really any dialogue and dialogue management degenerates to a 'simple' component, translating the meaning of the speech input to actions and the result of these actions to responses.

The initiative in the dialogue is often used to classify dialogue managers. In system-driven systems the ASP system has the initiative. It asks questions and the user supplies the system with answers. In a user-driven system the roles are

[4] Using the history and domain knowledge the dialogue manager also has to choose between the possible interpretations of the user's meaning when the natural language component gives more than one possible meaning.

switched. The system is passive and it only answers the user. Mixed-initiative is the hardest form of dialogue management. Both the user and the system can have the initiative. The initiative can be predefined according to a certain phase in the dialogue or the initiative can be negotiated.

Mixed-initiative systems are most often wanted. The user can initially be given the initiative. The system then takes over the initiative when the dialogue fails in some way.

3.4 Speech generation

Finally the system response should be communicated to the user. The system response can already consist of text or still be in some logical structure with a defined meaning. In the latter case, this logical structure first has to be transcribed to text (in a sense an inverted form of natural language processing). The text should then be transformed to speech. Currently two techniques are used to accomplish this.

- Concatenation techniques.
  Pre-recorded phrases are concatenated to form the speech belonging to the system response. In fact, the system response does not necessarily have to be literal text for this technique, but in most case it usually is.

- Speech synthesis.
  Speech is generated from the literal text using a model for human speech generation. The output is no longer restricted to pre-recorded phrases but any text can be spoken. At the moment however this technique gives less human-like speech than the first technique.

Chapter 4

The Alparon system

This chapter describes the ASP system that is currently being developed by the Alparon group. The first section will give a general description of the Alparon system. The second section will describe the modules the dialogue manager consists of in more detail. This chapter as a whole describes the environment in which the dialogue control components for the Alparon system are developed.

4.1 Overview

[Vark, 1997a] gives a good overview of the Alparon system. This section will give a short summary of that overview.

4.1.1 Background

The goal of the Alparon research group is the development of an automated speech processing system for complex information retrieval and transaction services. The research is focussed on dialogue management and related issues.

Currently the Alparon project is focussed on the domain of public transport information services. The Alparon system should provide this information service by telephone. The interaction with the user is only by speech.

The aim is to provide this service through a natural language interface. The user should be able to talk to the system in a natural way. The system has to be able to engage in a dialogue with the user. A mixed-initiative system looks ideal for this.

An intensive study has been conducted into this kind of dialogues between humans. A large corpus was recorded at OVR[1] call centers, see [Rothkrantz et al., 1996]. Based on this corpus a coding scheme has been developed, see also section 4.1.3. Based on this coding scheme and the strategies found in human-human dialogues, dialogue management of the Alparon system is being developed.

The first version of the Alparon system should be able to give the user travelplans for train connections in a mixed-initiative system contacted by telephone.

[1] OVR is the Dutch provider of public transport information by telephone.

4.1.2 Model of Alparon system

This subsection describes the overall model of the Alparon system. As stated before, focus within the Alparon group is on dialogue management. The subsection 4.1.4 give a more detailed model of the dialogue manager, the core of the system.

The system consists of several parts which will be described in turn. Differences with the model presented in the previous chapter will be noted. See figure 4.1 for an overview of the Alparon model.

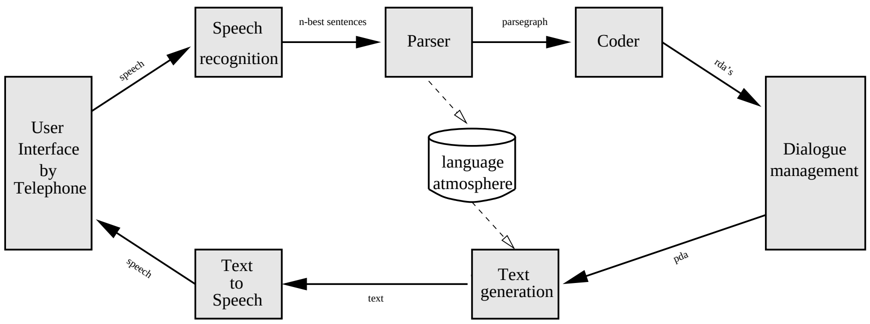

Figure 4.1: Overall model

User interface by telephone

The interface between the user and the system will take place by telephone, just as the current human-operated service for public transport information.

Speech recognition

The speech recognizer has a continuous speech signal as input. Speech is detected by this component and transcribed to text. Output of the speech recognizer are the n-best sentences, i.e. the n sentences with the highest probability to have been uttered by the user. The n-best sentences are written as word graphs.

Parser

Here the Alparon system differs from most other ASP systems. The natural language processing is placed in two components, the parser and the coder.

The parser checks the sentences it receives for grammatical correctness. The output consists of n-best grammatically correct sentences in a parse graph.

Besides the parsing of the sentences, the parser stores characteristics of the language the user uses in the language atmosphere database. These characteristics include for example the expressions and particular words used. These characteristics can be used in the generation of the text output of the system. Slight alteration can be made to the text of the system response to make the user feel more comfortable with the system.

Coder

Here the Alparon system differs most from most other ASP systems. The coder is placed outside the dialogue manager. Most ASP systems use annotated sentences as the input of the dialogue manager. The coding of those sentences to a higher level of meaning is then performed by the dialogue manager. In the Alparon coder a higher level of meaning is added to the input, outside the dialogue manager. The parsed sentences are transcribed to dialogue acts. This dialogue coding scheme is explained in some detail in the next subsection.

This transcription is only based on the linguistic information present in the sentences. The dialogue manager will possibly have to reformulate the dialogue acts a little using dialogue history and context. The natural language processing is completely outside the dialogue manager. Only the last step of meaning extraction (using context and history) takes place inside the dialogue manager.

The output is a series of possible interpretations of the user's utterance in dialogue acts, the recognized dialogue acts[2].

Dialogue management

The dialogue manager controls the dialogue. It controls the interaction between the ASP system and the user. It also interfaces with the background application, i.e. the databases at its disposal. The dialogue manager is the core of the Alparon system. In one of the following subsections the model of the dialogue manager is described.

The output of the dialogue manager consists of proposed dialogue acts. Using the same coding scheme as is used for the input, the system response is relayed to the text generation component.

Text generation

The system response is stated in the form of dialogue acts. This component transforms the proposed dialogue acts to text. It should use the language atmosphere to adapt the text to the user's use of language. This last feature however will not be developed until later.

Speech generation

The last step is to transform the text of the system response to speech and send that to the user through the same telephone interface as is used for input.

4.1.3 Coding scheme

The coding scheme is described in [Vark, 1997c]. In this section, the basics of this coding scheme will be presented. This coding scheme is devised to handle natural language on a more abstract level in the dialogue management of the Alparon system.

The analysis of a corpus of over 5000 human-human dialogues in the public transport information domain has resulted in a coding scheme which minimizes the differences in representation found in natural language. However

[2] An interpretation can consist of more than one dialogue act. So the output consists of several groups of dialogue acts.

it still captures the basic information elements found in information retrieval dialogues, see [Vark et al., 1996]. This coding scheme was devised for the purpose of analysis. The coding scheme used in the Alparon system is slightly different but based on this scheme. This modified coding scheme is called the Dialogue Management Coding Scheme (DiMaCS).

The coding scheme is similar to the coding scheme introduced by [Carletta et al., 1995]. Dialogues can be considered to consist of several conversational games. These games usually consist of initiations followed by responses. Examples of initiations are questions and statements, examples of responses are answers and acknowledgments. The individual elements of these games are called moves. The moves can be coded using dialogue acts.

The utterances will be transformed to dialogue acts by the speech recognizer, the parser and the coder[3]. The dialogue acts specify what kind of moves they are, i.e. what is the function of (a part of) the utterances. Codes, subcodes, arguments and subarguments specify what kind of information is to be conveyed by the move. So dialogue acts have a hierarchical structure.

The schematic overview of first version of the DiMaCS can be found in [Vark, 1997c]. This version is focussed on information concerning train connections and is sufficient for the current version of the Alparon system.

4.1.4 Dialogue manager model of Alparon system

The dialogue manager is the core of the Alparon system. This subsystem takes care of managing the dialogue between the user and the ASP system. Here the user's utterances are interpreted in context and the context is updated accordingly. The dialogue manager furthermore has to decide what response the system should make to the user and what actions it should undertake to fulfill the user's wishes as good as possible.

A model for the dialogue manager has already been designed. This model is depicted in figure 4.2. It consists of a task coordinator, a blackboard system, a discourse manager and a database manager. The blackboard system and discourse manager consists of several blackboards and modules. This modular approach was chosen in order to be able to flexibly add or remove modules and test different implementations of the boards and modules.

The task coordinator is the central part of the dialogue manager. It connects all the other components together. The components of the dialogue manager communicate with each other using messages. All components are connected with the portmapper and send their messages using the portmapper. The portmapper has to route these messages and map module names to instances of the modules. The portmapper can also fulfill some auxiliary functions. Load balancing, visualization and analysis of information streams within the dialogue manager, are examples of this. See [Vark, 1997a] and [Vark, 1997d] for a more detailed description of the task coordinator.

The blackboard system makes up the memory of the dialogue manager. Here, all information on the dialogue is stored. The blackboards are self-organizing

[3] There is not a one-to-one correspondence between utterances or sentences and dialogue acts. One turn of the user can consist of more than one sentence and each sentence can be transformed in more than one dialogue act. It is also possible that several sentences are together transformed to one dialogue act.

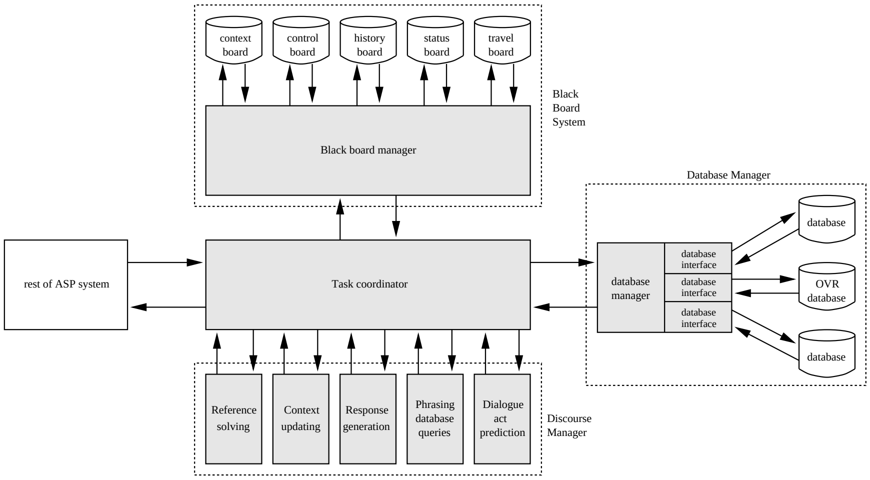

Figure 4.2: Dialogue manager, detailed lay out

boards. They are capable of manipulating a certain type of information. The history of their contents should be remembered and can prove useful for repair operations in future versions.

The discourse manager consists of modules that perform the actual intelligent processing of the dialogue manager. Many different tasks can be distinguished of which some can be performed simultaneously. They use the blackboard system to keep track of information on the ongoing dialogue. They send and retrieve information to and from the blackboards. These modules will be invoked by the dialogue manager for the processing of the user's utterances and response generation.

The database manager provides a standard interface for the databases in the background. Queries to the databases are posed through the database manager. The result of those queries will also be transformed to a standard form which the dialogue manager understands. The databases are the external information sources.

Not all the components of the dialogue manager are currently developed and implemented. The purpose of this work is to design and implement that part of the dialogue manager responsible for dialogue control. After the user has spoken to the system, the system has to reply to the user. What the system should say and what other actions the system should undertake, are for dialogue control to decide. In terms of this originally designed model, dialogue control will be implemented by the response generation module and the control board. But as stated in the introduction, it could well be that other, new modules will be part of the dialogue control mechanism.

The next section will describe the modules and blackboards other than the ones for dialogue control, in detail. The responsibilities of these modules that are important for dialogue control, will get special attention. This serves as a description of the environment in which the dialogue control components should function. Though not all these components have been developed themselves, the development of the dialogue control part will build on this description.

4.2 In detail

In this section the modules and blackboards of the Alparon system are described in more detail. This serves as a background for the development of the dialogue control components. It will give the reader some impression on how the Alparon operates as well. This specification of the modules and blackboards is not in all cases already determined and stated in previous work of the Alparon group, but it is still mentioned here to make up a complete environment for dialogue control. It usually consists of additional specifications to the already existing ones.

It starts with the modules of the discourse manager and next the black boards of the blackboard system are described. The central portmapper and database manager are not further described. The general information given in the previous section is suffices. (See also [Vark, 1997a] and [Vark, 1997b].)

4.2.1 Discourse manager

The functionalities described here, have in some cases not been described in previous reports on the Alparon system. These new functionalities are necessary, or are at least useful, for dialogue control.

Disambiguation

When the user speaks to the system, his speech has been interpreted by the speech recognizer, the parser and the coder, before it reaches the dialogue manager. The output of the coder is the input of the dialogue manager. These components transcribe speech into concepts, resulting in a list of dialogue acts. These dialogue acts describe what kind of utterance the user made and what information is conveyed in it. The elements of the resulting list are interpretations of the user's turn and each interpretation consists of one or more dialogue acts. So the input of the dialogue manager is a list of lists of dialogue acts.

The input can consist of more than one interpretation of the user's utterance because the state of the art in speech recognition is not perfect. The speech recognizer produces several possibilities of what the user could have said, with respective probabilities. The first task of the dialogue manager is to select one of these interpretation as the most probable one. This does not necessarily have to be the interpretation with the highest probability. Given the context of the domain and the ongoing dialogue, another possibility may actually be more likely. This is one of the tasks of the disambiguation module.

Furthermore the coder can only determine the semantic value of isolated utterances. The precise meaning behind an utterance (or pragmatics) can not always be determined based on isolated utterances, e.g. is the user asking a question or is he checking some information. Without domain knowledge and dialogue history such distinctions can not always be made based on one utterance only.

Another task of the disambiguation module is to reformulate the dialogue acts when necessary, given the previous turns of the ongoing dialogue and domain knowledge. A dialogue act can be changed into another kind of dialogue act. For example, a statement can be changed into a partial acknowledgment. Another way to reformulate the dialogue acts is to alter the information contained in it (i.e. the codes, subcodes, arguments or subarguments). An example of this occurs when the disambiguation module can solve a reference.

The input of the disambiguation module is the input of the dialogue manager, the recognized dialogue acts or rather a series of possibly recognized dialogue acts ordered by likeliness. The output are those same dialogue acts but possibly reformulated and reordered according to what the disambiguation module thinks is the likeliness of the possible interpretations (called the disambiguated dialogue acts). The most likely interpretation will be used for further processing by the dialogue manager.

Situations can occur when the disambiguation module is not able to make a choice between dialogue acts that only differ in some value of their arguments (i.e. the disambiguation module can not say one is more likely to be what the user said than the other). The information in the dialogue act is related to the domain of the application and the disambiguation module does not have any knowledge to prefer one over the other. An example is dialogue acts containing domain values 'Baarn' and 'Maarn' (the name of two cities that are very similar phonetically). This is the result of the speech recognizer not recognizing one word uniquely but rather giving several possible words that all have about the same probability of being uttered.

In such situations there are two possibilities:

- The disambiguation module just picks one of the alternatives to be more likely than the other(s). The rest of the dialogue management subsystem should then check for this situation and possibly make a more grounded choice or verify the arbitrary choice of the disambiguation module (e.g. using explicit verification to the user). Only the chosen alternative will be used for further processing by the dialogue manager.
- The disambiguation module gives the same likeliness to both (or more) dialogue acts. Both (contradicting) dialogue acts will be used for further processing by the dialogue manager. This situation should then be reported somewhere (the control board), because a choice must eventually be made by the dialogue manager.

The first solution looks simpler but as a consequence the dialogue manager should explicitly look for this kind of situation later on in the turn processing.

The second solution looks more promising. There is no reason to consider one dialogue act more likely than others. So why make an arbitrary choice between them and come back to the same matter later on. The decision should actually depend on the strategy for dialogue control and domain knowledge. Only after this strategy is applied a meaningful choice can be made between the alternatives.

The dialogue control is based on the latter approach to such situations.

Another aspect of disambiguation is the handling of references. The user uses an expression that is more general than the concept (in the domain of the application) he wants to convey. With a simpler expression the user refers implicitly to the a specific concept. This more general expression has to be further specified in order to unambiguous determine the informational needs of the user. E.g the user mentions *Rotterdam* as the place of departure. This is encoded in the dialogue act as `city(Rotterdam)`. In order to fully determine what travelplan the user want, a departure station has to be determined as there is more than one station in Rotterdam.

The user can also refer to something that has been said earlier in the dialogue. Using the history of the dialogue, stored on the blackboard system, the disambiguation module can reformulate the dialogue acts (with some confidence) so they contain more specific information in the domain of the application. E.g. earlier in the dialogue `station(Rotterdam central station)` is mentioned. When later on in the dialogue `city(Rotterdam)` is mentioned, the same station is probably meant.

Another way to solve a reference is by using domain knowledge. The system then 'knows' what is usually meant by some expression. Evidently in a city X with only one station the user always means `station(X)` when he uses `city(X)`[4]

When not enough knowledge is present to solve a reference, the transforming of the more general concept to a more specific one should occur later on in the processing of the dialogue turns. Some strategy in the dialogue control should be applied to solve such a reference (e.g. by asking the user).

The disambiguation module can use the reference handling module for solving references. When a reference is solved the dialogue act(s) will be reformulated on the informational level. But the unsolved reference should also be preserved with the solution. This can help with later references or be used in the exact phrasing of the system response.

Context updating

The task of the context updating is twofold.

First of all the context updating module is responsible for storing the input of the dialogue manager. The history board is used for this storage. The context updating module stores the list van possible interpretations of the user's utterance on the history board. This is the list of list of dialogue acts, the recognized dialogue acts. The outer list contains different possible interpretations of the user's utterance as elements and each such element is a list of dialogue acts which is the encoding of one possible interpretation.

Furthermore the result of the disambiguation module is stored alongside this input of the dialogue manager, namely the same input but reordered and reformulated as the disambiguation module saw fit. The interpretation of the user's utterance the disambiguation module thinks the most likely, is the first interpretation on the reordered list. This interpretation will be marked on the history board and this interpretation is used for further processing by the dialogue manager. In case the disambiguation module could not choose the most likely of several dialogue acts that only differ for some domain value (like two dialogue acts that only differ for the name of the station e.g `station(Baarn)` and `station(Maarn)`), both interpretations should be marked. When the choice is finally made, the chosen dialogue act (or actually interpretation) should then

[4]The user actually just uses X, the coder transforms this to `city(X)`. The dialogue manager works with dialogue acts that encode the user's utterances. So the expressions used are encoded. A more sophisticated coder could solve this kind of references.

be marked as the only one.

The second task of the context updating module is to store the informational contents of the interpretation which is selected for further processing[5]. This information is stored on the information board. The contents of the information board is used to determine the informational needs of the user. The context updating module extracts the information from the dialogue acts and puts it on the information board. When this information conflicts with information that is already present, the new information takes precedence over the old information. This situation is however reported to the control board in order for the rest of the dialogue manager to be aware of the situation and possibly take actions to validate the decision.

When the disambiguation module has resolved a reference, this solved reference is stored with the original reference on the information board. See also subsection 4.2.2.

When more than one interpretation is marked for further processing, the information common in all those interpretations is stored on the information board. Furthermore the context updating module tells the control board that a choice has still to be made between the different values.

Reference handling

The reference handling module is an auxiliary module to be used by other modules. It can be used to solve references.

The dialogue history, i.e. all that has been stored about the dialogue so far, can be used for this task. References to concepts that are mentioned earlier in the dialogue can be solved by this approach. Besides that, the reference handler can use domain knowledge to solve references that are commonly used.

When it is not possible to solve a reference entirely, the reference handling module can produce a list of possibilities. The module that invoked the reference handler can then decide what to do with that information.

Another task of the reference handling modules is to generate references. For a natural response the dialogue manager may decide to use references in its utterances. This module can give references to concepts for this use.

Query

When the dialogue manager thinks the user's informational needs are sufficiently determined the information sources should be consulted, i.e. the databases have to be queried. The task of the query module is to generate the queries and send them to the databases (actually to the database manager which interfaces to the databases). The result of those queries have to be stored on the presentation board partitioned in pieces suitable for presentation. The information necessary to generate the queries can be collected from the information board.

Another possible task of the query module is to verify certain information. The user has mentioned some concepts and the query module checks whether these concepts are valid in the domain of the application. For example, the user mentions a certain station for departure. The query module can then

[5] The dialogue acts contain informational elements in their structure, see subsection 4.1.3.

check whether this station name is valid in the train travel application, i.e. is the name known in the databases. The query is not being used to fulfill the informational needs of the user, but to help the dialogue manager.

There are two possible ways for the query module to work.

- The query module monitors the information board. When enough information is present to pose a query, the module generates the query and sends it to the database manager. The query module has to know what information is necessary to generate all the kinds of queries it is supposed to generate. After the result is stored on the presentation board, this fact has to be made known to the rest of the dialogue manager subsystem. This approach could be called information-driven, whenever the module can generate a query it immediately does so.

  Intelligent agents can be used for this approach. They constantly monitor the information about the ongoing dialogue and try to extract meaningful information from various information sources.

- The query module waits for a request from another module to generate a certain query. It then tries to generate the requested query. It assumes the necessary information is present. The query is sent and the result is placed on the presentation board as usual.

  The module that makes the request has to know what information has to be present to make the kind of query it requests. This approach can be called request-driven.

For the development of the dialogue control modules the second approach will be assumed, see also [Oldervoll, 1998].

Dialogue act generation

The dialogue control modules generate the system response. This response has to be transformed into dialogue acts. The dialogue act generation takes care of this transformation. It takes the system response and generates the dialogue acts to perform this response, the proposed dialogue acts. These dialogue acts are the output of the dialogue manager. They are fed to the text generator outside the dialogue manager. These proposed dialogue acts are also stored on the history board by this module.

4.2.2 Blackboard system

In this section the information, history and presentation board are described in some detail. Also the status board is described. The exact function of this board is not precisely known at this moment.

The same as for the discourse modules applies here. Not all the blackboards are already developed or have all the functionality described here.

Information board

The first part of the dialogue will be concerned with the determination of the user's informational needs. The ASP system supplies a certain kind (or several kinds) of information, e.g travelplans for train connections. To specify the exact information the user wants, some key information elements must be known (e.g. the departure and arrival station and a time for either departure or arrival). These key information elements have to be determined in the first part of the dialogue. For all the kinds of information the ASP system can supply, some key information elements have to be known.

This information can be stored in slots. So in order to generate a particular query, certain slots have to be filled with certain values. The slots are filled with values that have a meaning in the domain of the application. Therefore they are called domain values in this report. To determine the informational needs of the user, certain slots have to be filled with specific enough domain values, e.g. `city(Amsterdam)` is not specific enough for departure place but `station(Amsterdam cs)` is. For every kind of informational needs a certain combination of slot values have to be known, e.g. for price information on train travels no arrival of departure time is necessary.

The information board is used to specify the user's informational needs. The domain values mentioned in the dialogue are stored in the appropriate slots of the information board. This can be domain values mentioned by the user in his utterances, but also domain values the dialogue manager has derived or assumed.

The information board consist of a number of slots in which values can be stored. There are no restrictions on what kind of values can be stored in the slots. The slots can contain values which are not sufficient for the specification of the informational needs, but instead contain more general information. In order to specify a certain kind of informational need, restrictions can be placed on the contents of the slots. The informational need is not fully specified before certain slots have certain (kinds of) values. For example, the slot `departure place` has the value `city(Delft)`, but in order to fully determine the informational need for train travel plans, the slot has to be filled with a station like `station(Delft Zuid)`.

For every slot the following is stored with the value

- in which turn the slot got its current value;
- which module placed the value;
- what was the ground for the value (was it a default value, did the user introduce this value etc.);
- is the value verified;
- which references have been used for the value.

When a slot gets a new value, the old values are remembered, resulting in a history for every slot. When this new value conflicts with previous values, the previous values are no longer considered valid. New values can also be more specific values than the previous ones and not in conflict with the old values. The old values can then be considered references in some cases and they stay valid. Old values considered no longer valid, can become valid again when another new value is placed in the slot which does not conflict with the old ones.

Not only definite values are stored in the slots. When the user asks for a certain value of a slot, this slot can then be filled with a special value, e.g. 'asked'.

This slot should then not be essential for the specification of the informational needs of the user, but the fact that the user specifically asked for this slot can help adjust the presentation of the information to the user.

Another possibility is to store negative values, values that are explicitly denied by the user for instance[6]. The use of negative knowledge in dialogue management is for example described in [Vromans et al., 1999].

History board

On the history board, the history of the dialogue is kept up to date. All the recognized dialogue acts, disambiguated dialogue acts and proposed dialogue acts are stored here. The dialogue acts selected for further processing, i.e. the dialogue acts the disambiguation module thinks are the most likely interpretations of the user's utterances, are marked as such.

All the information on the history board can be used to predict future dialogue acts from the user in order to aid the speech recognizer in performing better. Another use of the history board is to aid in recovering from errors during the dialogue, e.g. when the selected dialogue acts are not the correct interpretation of the user's turn, one of the other interpretations is the right one, a misunderstanding in other words. The dialogue manager can then use the information of the history board to reconstruct the dialogue according to the right interpretation.

These latter functions can quite complicate the dialogue and should be used with caution. The current version of the Alparon system will only use the history board for the storage of all the dialogue acts, recognized, disambiguated and proposed.

Presentation board

This blackboard contains the results of the database queries, i.e. the information to fulfill the user's informational needs. These results are partitioned in pieces that the user can reasonably comprehend per turn of the dialogue, so all the information is not fed to the user in one huge dialogue turn.

Only the results of database queries designated for the user are stored here. If there are other queries to the databases for the use of the dialogue manager itself, these results will not be placed on the presentation board.

This module originally was called travel board. But this name is related to the train travel information application. The name presentation board seems more suitable for information retrieval dialogues in general.

Status board

The state of the dialogue manager will be recorded on this blackboard. The state can be divided in two parts:

- The state of the rest of the world, everything beyond the system boundaries, for example the state of external databases, the number of users waiting to be helped, etc.

[6] In the case of new value conflicting with older ones, the old values are not explicitly denied.

- The state of the current ongoing dialogue. In which phase is the dialogue? What is the state of various slots? What is the turn number of the current turn? Is there a travelplan present on the presentation board?

It is not precisely clear what function this blackboard plays at this moment, probably some place to keep track of general characteristics and information regarding the dialogue. How the information on the status board is used is not yet designed.

Chapter 5

Dialogue control

Dialogue control is responsible for how the dialogue evolves as far as the system can influence the ongoing of the dialogue. It has to decide whether and what actions should be undertaken by the system. It has to generate a response to the user's turn. In other words, it manages the dialogue.

The disambiguation and context updating modules are mostly involved in interpreting the user's turn. They perform a kind of preprocessing of the user's dialogue acts as they select the most probable interpretation of the user's utterance and update the dialogue context with the information of the most probable interpretation. See section 4.2.1 and 4.2.1 for more detailed information on those modules.

This chapter explains how the dialogue manager performs dialogue control. Section 5.1 describes the possibility to distinguish two kinds of information in an utterance, information concerning the application domain and information concerning the dialogue. The general approach chosen to control the dialogue in the Alparon system is explained in section 5.2. Section 5.3 then describes how dialogue control is organized in the Alparon system using a framework of two modules and a blackboard.

5.1 Two kinds of information

An utterance in a dialogue contains information. The information in utterances can be divided in two different kinds.

The first kind of information is information regarding the application domain, henceforth called domain information. This is all the factual information in the utterance. It concerns concepts in the application domain. The exchange of this kind of information is the purpose of the dialogue. For example this information helps to specify the informational needs of the user.

The second kind of information is information regarding the dialogue. This can be called dialogue information. This information tells what kind of utterance the user made and what function the utterance has in the dialogue.

Before the user's turns are processed by the dialogue manager, the user's utterances are transformed into dialogue acts (see section 4.1.2 and section 4.1.3). The distinction between the two kinds of information can also be found in the dialogue acts. The dialogue information is specified in the top levels of the dialogue acts, while the domain information can be found at the lower levels of the dialogue acts.

For example in figure 5.1, the user asks a question, that is dialogue information. In contrast the city Amsterdam is a concept in the application domain of train travel with the role of departure place. Also departure time is a concept in the application domain.

> *What time does the train to Amsterdam leave?*
> `Que(Time(DepTime(asked)), Location(DepPlace(city(Amsterdam))))`

Figure 5.1: Example

The more sophisticated dialogue systems deal with both kinds of information. When the user makes a certain move, he has a purpose in mind as the user is supposed to handles according to a plan. The system has to react to that purpose. Without the use of dialogue information it is very hard or impossible to react to the user's plan. In order to engage in a natural dialogue this can hardly be neglected. See [Cole et al., 1997] section 6.3 on purpose and plans in dialogue.

The domain-information of an utterance is extracted from the dialogue acts by the context updating module and placed on the information board. The disambiguation module and context updating module are mainly concerned with domain-information. To adequately structure and control the dialogue the modules for dialogue control will have to use dialogue-information as well as the domain-information.

5.2 Dialogue control

In order to control a specific kind of dialogue adequately it can be very useful to have an idea of how such dialogues are usually structured. When you have a model for the dialogue, you can then use it to structure and control such dialogues.

Information retrieval dialogues are task-oriented as the dialogues are structured according to the task, the dialogue structure resembles the task structure. The dialogues are goal-directed (see [Vark, 1997a] and [Cohen, 1994]).

To control these goal-directed dialogues, a plan based approach can be used, see [Churcher et al., 1997] and [Cole et al., 1997] for some general information on plan based approaches. The dialogue is controlled by a goal structure. The goals are to achieve the obvious tasks in an information retrieval dialogue. So the structure of the task of the dialogue resembles the structure of the dialogue and this structure can be controlled using goals.

In the Alparon system goals on a different level of abstraction can be distinguished. As there are tasks which can be decomposed into subtasks, complex goals can be decomposed into subgoals. The high level goals are more abstract and correspond to high level tasks in the dialogue, whereas the lower level goals are more concrete and correspond to subtasks. The goals are structured in a hierarchical way, with complex goals at the top and in the middle and atomic goals at the bottom.

Complex goals are goals that can be decomposed into subgoals. The atomic goals can not be decomposed any further. The complex goals can be decomposed into multiple complex goals, though on a lower level of abstraction. Ultimately, decomposition will result in atomic goals.

The purpose of the high level goals is to structure the dialogue. In order to achieve a high level goal it must first be decomposed to lower level goals. Only of atomic goals is it possible to determine whether they are achieved or not. Only when all the subgoals of a complex goal are achieved, the complex goal can be considered achieved as well.

The goals discussed thus far are all goals that are related to the task structure of the dialogue. These goals, especially on a high level of abstraction, will be the same for most dialogues. The way high level goals are decomposed into lower level goals follows from the task and does hardly depend on a particular dialogue. On lower levels the decomposition may depend a little on the actual dialogue, but not much. These goals will be called task goals in this report.

But besides these goals, there is another kind of goals, which is related to the dialogue. These goals deal with discourse phenomena. Questions should be answered, confirmations should be given, misunderstanding should be solved, etc. To handle such situations another kind of goals is needed: the dialogue goals. These goals arise from the dialogue. In order to achieve these goals, the dialogue manager has to deal with some discourse phenomena (see [Churcher et al., 1997] for a discussion of discourse phenomena).

`Determine informational needs` is a task goal.
`Verify slot(DepPlace)` is a dialogue goal.

Figure 5.2: Example

So there are goals that are related to the task of the dialogue, the task goals, and there are goals to deal with discourse phenomena, the dialogue goals. The same distinction between the goals can be seen in [Jokinen, 1996]. There they speak of task goals and communicative goals.

During the dialogue a (data)structure keeps track of the goals that are still to be achieved. This structure will be called the goalstack and will be described in greater detail in section 6.2.

This goalstack is not static as it will evolve during the dialogue. How it will evolve depends on how the dialogue develops. The complex task goals will be decomposed into subgoals according to the task structure and the decomposition does hardly depend on the ongoing dialogue. However dialogue goals can appear anytime during the dialogue and must then be incorporated into the goalstack.

The goals have only a descriptive function. They tell what should be achieved during the dialogue but not how it should be achieved.

The goals that still have to be achieved are the 'agenda' of the dialogue manager. In order for the dialogue to succeed these goals have to be achieved.

Every turn the dialogue manager will check whether there is enough information present to perform certain operations. These operations can help to achieve certain goals. For example in a train timetable information system a station of departure has to be determined. When the system knows the departure place is a certain city and it also has some domain knowledge telling there is only one station in that city, an operation can be performed that transforms the city information into station information. So the system can make deductions based on already present information and modify the state of the dialogue in such a way that certain goals can now be considered achieved. Another possibility is to resort to default behaviour. In some situations default modifications can be made to the state of the dialogue. E.g. the day of travel in an train timetable information system can be set to the current day by default. When the user does not explicitly states otherwise the current day is the day of travel.

When the dialogue system can not achieve goals without further interaction with the user, communication with the user is necessary concerning those goals. The user must be tempted to give such information that the goals can be achieved, either directly or indirectly by the execution of operations.

The system can influence how the dialogue evolves by the responses it generates. Those responses consist of one or more communicative actions and are the means by which the dialogue is controlled by the system. The actions are geared towards the goals the system wants to achieve, both the task goals and the dialogue goals. The communicative actions are always related to goals. An action can not exist without some goal it is related to. The same action can however be related to more than one goal.[1]

Control of the dialogue is determined by the strategy. The strategy determines how the goals should be manipulated. Thus it determines what should happen in the dialogue. But besides that, it has to define how those goals should be achieved. What operation should be performed? Which communicative actions should be part of the response?

As a result of the dialogue control the contents of the blackboards can be changed. E.g. when the system makes a decision that affect the information board, the information board is updated accordingly.

5.3 Framework

In this section the framework for dialogue control in the Alparon system is described. One of the aims of the Alparon project is to make an ASP system for complex information retrieval dialogues. It serves as a research project. It should be possible to experiment with different approaches to all aspects of dialogue management. This is one of the reasons for the modular design of the overall system.

As far as dialogue control concerns it must be possible to experiment with different strategies for dialogue control. And while the first version is placed in the train timetable information domain, it should not be the sole domain to which the Alparon system is applicable. The system should be adaptable to other domains.

The dialogue control part is therefore developed as a generic framework. The framework operates on sets of rules which define the exact strategy for dialogue

[1] The term action and communicative action are used interchangeably in this document.

control. By changing these sets of rules, the strategy for dialogue control - and therefore the dialogue itself - can be changed. The dialogue control part of the Alparon system is an engine, where the exact operation of the engine can be specified by rules.

The framework is not application or domain specific. It should be possible to use it for information retrieval dialogues in various domains. The rules defining the strategy contain application specific information.

The strategy is defined by rulesets which operate on the goals and communicative actions. It also defines whether and how the user's turn influences the goals by rules which operate on the dialogue acts of the user's turn. All these rules together implement a certain specific strategy for dialogue control in a certain application domain.

In order to let the Alparon system behave in a different manner, only the rulesets for the dialogue control have to be changed. The same framework can be used for different kinds of dialogues (system-driven, user-driven or mixed-initiative).

This approach is somewhat similar to the approach of the Philips train timetable system of the German railways, see [Aust and Oerder, 1995]. A application independent interpreter is used to control the dialogue. This interpreter operates on a declarative dialogue description which is application dependent. For another application only the declarative dialogue description has to be changed. This system is however system-driven.

There are two processing steps in dialogue control. First the consequences of the user's turn have to be processed and after that the system's response has to be generated.

This can be seen in the organization of the framework. The dialogue control part is implemented using two modules and a blackboard. The blackboard contains all the information necessary for dialogue control that last more than one turn in the dialogue. One module is responsible for the processing of the consequences of the user's turn and the other module generates an appropriate response. The first module is called the dialogue updating module and the second is called the response generation module.

This chapter closes with a description of these modules and blackboard. In further chapters the modules and blackboard will be dealt with in more detail.

5.3.1 Dialogue updating

This module updates the goalstack of the dialogue. The effects of the user's turn on the dialogue are being determined. The user's turn can affect the dialogue in several ways.

In the first place the user can provide some domain information in his turn. The context updating module extracts that information and places it on the information board. The state of the dialogue, being the combined state of the blackboards, is then modified. As a consequence some goals can be immediately considered achieved.

It is also possible that because of the change in the state, some operation can now be performed in order to achieve a goal. These operations use domain knowledge to make some deduction. This deduction could not be made earlier because insufficient information was present. E.g. the user supplies some information to define his informational needs, but this information is not specific enough for the application on hand. However there is an operation which can be used to transform or elaborate this information to information that can be used by the application. Such operation can for example be justified by statistical considerations.See figure 5.3 for an example.

> In a train travel information dialogue the user mentions `city(Delft)`. This is information understood by the system. But in order to specify the informational needs of the user not cities but station are needed. There are two stations in Delft, but statistical analysis shows that when people speak of *Delft* (=`city(Delft)`) in this context they usually mean `station(Delft)` and not `station(Delft Zuid)`. So an operation can be performed to change `city(Delft)` to `station(Delft)`.

Figure 5.3: Example of operations

As a result of such an operation goals may be achieved and it is allowed that as part of the operation new goals are added to the goalstack.

Another possible way for domain information to affect the dialogue is that the information in the user's turn is in conflict with previous information. The dialogue updating module must deal with such a situation. To handle such a situation new goals can be added to the goalstack. E.g. a goal for verification of the choice the dialogue updating module makes in a certain case.

It can also be that the previous modules could not make a choice between certain domain values (the speech understanding process gave more than one possible representation of the user's turn). The dialogue updating module must then choose some course of action to deal with such a situation. As part of a solution the goalstack may be modified or extended.

Next to the effects of domain information, the user's turn can affect the dialogue because of the dialogue information it contains. The user expects the system to react to that information. If the user wants to steer the dialogue in a certain way, it is not appropriate to ignore that. So the dialogue updating module must modify the goal structure to accommodate the user as far as the strategy allows. This occurs of course only in mixed-initiative or user-driven dialogues, but it is the aim of the Alparon group to create a mixed-initiative dialogue manager.

After the effects of the user's turn on the dialogue are processed, the modified goalstack must be processed. There may be operations which can be performed to achieve newly added goals. On the other hand, all atomic goals can be achieved and complex goals must be decomposed into subgoals until there are atomic goals on the goalstack as further processing can only be done with atomic goals. These new goals must be checked for operations and achievement as well.

So modification of the goalstack can lead to further modifications of the goalstack. Goals are being created on the fly and the new goals should be checked for decomposition (complex goals) or achievement and operations (atomic goals).

N.B. It can be argued that the processing of the effects of the user's turn on the dialogue level should be done by the context updating module. So the goals that are a result of the kind of utterance the user has made, should be added to the control board by the context updating module. This would be another task next to the modification of the goal structure as a result of the information in the user's turn. The dialogue updating module should then only be concerned with the cleaning-up of the goal structure and putting it in a proper form (performing operations, make decisions, etc.).

There is something to say for this approach. However the outline in this chapter places all the processing of goals in one module. In order to process the user's turn, one has to take some decisions that are part of the strategy for dialogue control. When you want to change the strategy, it is easier when that strategy is localized in one module. So in this outline there are only a couple of modules which do intelligent processing according to a chosen strategy. The rest of the modules are mostly pre- and post-processing modules that do not change when the dialogue strategy is changed.

5.3.2 Response generation

The dialogue updating module modifies the goal structure as a result of the user's turn. So after the dialogue updating module has processed the user's turn, the goal structure reflects what still has to be done. There are no more operations possible to achieve goals. It is not possible to achieve goals without communication with the user. The only way to proceed is to generate a response for the user. This is the task of the response generation module.

On the basis of the goals and the strategy, communicative actions have to be formed. The strategy tells what kind of communicative actions can be performed in order to achieve some goal. So for all the goals that are still to be achieved, there are some communicative actions available.

To generate a response the response generation has to pick and possibly combine some of the communicative actions. The other communicative actions can be used in a next turn if they are still available.

5.3.3 Control board

The control board is part of the blackboard system of the dialogue manager. All information specific for a particular dialogue that has to last more than one turn, is stored on this blackboard system. The goalstack and the communicative actions are such information.

The control board contains all the information for the dialogue control. So the goalstack and communicative actions with their relations are on this board.

Also when other modules detect problems, like conflicts on the information board or an undetermined choice, this information or a notice of such a situation, is placed on the control board. The other modules for dialogue control can extract such information from the control board in order to deal with it.

The response the response generation module has generated is also stored on this board. The modules responsible for processing the response further, can get the response from this board.

Chapter 6

Concepts

In the previous chapter several concepts used for the building of the dialogue control mechanism were mentioned. This chapter will look into these concepts and their features in greater detail.

Section 6.1 describes the individual goals and section 6.2 describes the goal-stack in which they are placed. The next section is involved with the operations which can be performed for the achievement of goals. Then in section 6.4 the communicative actions the system can perform are dealt with. After that there is a section concerning reaction rules. They are used to define how the dialogue system should react to specific utterances from the user. Finally the last section gives a summary of the elements of the strategy.

6.1 Goals

Goals are an important concept in the dialogue control mechanism of the Alparon dialogue manager. The goals state what should be achieved during the dialogue. They are used to structure the dialogue. For the dialogue to succeed all the goals have to be achieved. The goals are the plan of the dialogue manager.

This section describes the features and types of goals that are to be expected in an information retrieval dialogue. Some of them are generic and should be further specified for a particular application in a domain.

By ordering the goals in a hierarchical structure, they can be used to structure the dialogue according to the task. You can distinguish two kinds of goals, the complex goals and the atomic goals. The sole purpose of the complex goals is to structure the atomic goals. They are related to the tasks and subtasks of an information retrieval dialogue.

The complex goals can be decomposed into one or more goals on a lower level of abstraction (the subgoals) until atomic goals are reached. On these atomic goals it is possible to do further processing.

6.1.1 Complex goals

Complex goals are used to order the atomic goals. Their structure reflects the task structure of an information retrieval dialogue. The complex goals can only be decomposed into subgoals on a lower level of abstraction[1]. They only have an identity and no parameters. They can be decomposed into other complex goals or atomic goals. It is also possible to decompose a complex goal into a mixture of other complex goals and atomic goals. There are no restrictions. The dialogue starts with one overall goal.

$C_n \to C_i, \ldots, C_j$ or
$C_n \to A_i, \ldots, A_j$ or
$C_n \to C_i, \ldots, C_j, A_k, \ldots, A_l$
C_n = complex goal, A_n = atomic goal

The decomposition actually also specifies an order in the goals it decomposes to. A complex goal decomposes into a list of sets of other goals, both complex and atomic. The list gives an order in which the (sets of) goals should be achieved. All the goals in the elements of the list (sets of goals) have no specific order in which they should be achieved.

Complex goals are achieved when all the subgoals are achieved. So in order to achieve a particular complex goal, it is necessary the decompose it into atomic goals. This however does not have to be done in one turn. When a complex goal is decomposed into other complex goals (on a lower level of abstraction), those new complex goals do not have to be further decomposed immediately in all situations. The decomposition of the complex goals into subgoals will be done when necessary.

How the complex goals are decomposed into subgoals usually does not depend on the ongoing dialogue. It is related to the decomposition of tasks into subtasks in an information retrieval dialogue. However in some cases the dialogue can influence the decomposition. E.g. the user only wants to know the departure time in a train timetable information system. It is not necessary to decompose the `present information` goal into a complete presentation of the whole travelplan. The presentation of the departure time is enough.

The strategy has to define how the complex goals should be decomposed into lower level goals (still complex, atomic or mixed). How the main overall goal of an information retrieval dialogue is decomposed, depends on the application domain and the chosen strategy.

This is a summary of the features of complex goals:

- They define the overall structure of the dialogue dependent on the application domain.
- They have no parameters, only an identity.
- They can only be decomposed into subgoals, no other operation can be performed on them.
- The decomposition usually does not depend on the dialogue only in certain situations.
- They are achieved when all their subgoals are achieved.

[1] In a more complex system several complex goals could be combined to a complex goal on a higher level of abstraction and then be decomposed in an other way. The Alparon system's first version will not allow that.

6.1.2 Atomic goals

The atomic goals cannot be decomposed any further. They are the elementary building blocks of the dialogue structure.

One way they appear, is as a result of the decomposition of complex goals. These goals correspond with subtasks that cannot be further decomposed in other subtasks, i.e. elementary tasks. For these goals it is possible to determine whether they are achieved or not. The only way to determine whether a complex goal is achieved is to decompose it into subgoals and to determine whether they are achieved.

The atomic goals mentioned above are related to the tasks of information retrieval. Another way for atomic goals to appear, is when specific situation in the dialogue arise. Shortcomings of speech recognition or understanding component and discourse phenomena make it necessary to adopt additional goals. These goals are related to participating in a dialogue. Examples of such situations are the need for verification, confirmation, acknowledgment, solving misunderstandings, etc.

A more functional distinction between atomic goals is the distinction between informative goals and state goals. Informative goals are achieved when the user is informed about something. State goals are achieved when the dialogue system comes in a certain state. The informative and state goals are further discussed separately.

Informative goals

Informative goals are achieved when some communication has taken place, i.e. some information has to be communicated to the user. When this information is conveyed, the goal is achieved. E.g. the user asks for a specific piece of information. An informative goal to present that information can then be incorporated in the goalstack to accommodate the user.

When the system has made an utterance containing the information, this is not enough to consider the informative goal achieved. Communication has only completely taken place when the utterance is heard (and understood) by the user. In the Alparon mechanism for dialogue control an informative goal will be considered achieved when the system has made an utterance containing the information and when the user has reacted to that utterance by making an utterance himself. To simplify the matter any reaction, i.e. any dialogue act, from the user will be taken for ample justification to consider the goal achieved. The user does not have to react specificly to the information.

In order to achieve informative goals, communicative actions have to be created. These actions can then be incorporated into the system response. The strategy defines what actions can be created for a given informative goal. There can be more than one possible communicative action for one informative goal. Each of these actions will suffice in informing the user of some piece of information.

Informative goals can be generic. The goal is then to communicate some sort of information. Parameters will define the exact information to be communicated. E.g. the goal `present slot` with parameter `departure station` in a train timetable information dialogue.

So the features of informative goals are as follows:

- They are atomic goals.
- They are achieved when an accompanying communicative action is performed (incorporated in the response) and when the user has reacted to it.
- For any informative goal the strategy defines a set of communicative actions which inform the user of some information.
- In some cases they have a parameter defining the exact contents of the communication.

State goals

State goals are achieved when the dialogue comes into a certain state. These goals are achieved as a result of the user's turn or by some decision or derivation of the dialogue manager.

The strategy defines for what state of the dialogue the state goals are achieved, the state of the dialogue being the combined state of the blackboard system, especially the information board.

New domain information in the user's turn is stored on the information board. The presence of this new information can be enough to consider a goal achieved. E.g. the goal `determine slot(departure place)` in a train timetable information system, can be considered achieved when the slot `departure place` on the information board contains a specific station.

Another way the state of the dialogue can change, is by operations. The dialogue manager can perform operations that change the state of the dialogue. These operations are part of the strategy. They are always related to a specific goal and are designed to make such a change that this goal can be considered achieved. Those operations should be based on domain knowledge or statistical analysis of corpora. A further discussion on operations follows in section 6.3.

When the system cannot achieve any goals by looking at the state or performing some operations, the user must be tempted to give suitable information. This information can then be used directly to achieve some state goals or they provide enough information for the dialogue manager to perform some operation in order to achieve a goal. Without interaction with the user those goals cannot be achieved.

The strategy specifies a set of communicative actions that are designed to tempt the user to supply necessary information. This can be more than one action per state goal. There are usually multiple ways to tempt the user. Section 6.4 deals with the communicative actions themselves.

The goals can be generic, e.g. a goal to determine a certain slot on the information board. A parameter specifying the slot completes it.

In short, the features of state goals are as follows:

- They are atomic.
- They are achieved when the dialogue is in a certain state.
- Operation for the benefit of these goals can be performed to come to such a state.

- They can have a set of communicative actions associated with them designed to tempt the user in giving the right information.
- In some cases they have a parameter defining the exact nature of the goal.

6.1.3 Goal types

This section gives an overview of an initial set of goal types for information retrieval dialogues. For every goal the name, the parameters if applicable and a description is given. These are the goals that are to be expected in the current version of the Alparon system. With these goals it must be possible to control the dialogue. Chapter 8 shows how these goals are used in a train timetable information system. The actual name of the goals used in chapter 8 are given in parentheses. These are not the only goals that could be used, but these goals will probably appear in some form.

Complex goals

- `any name`
  parameters: none
  They only have an identification. They are used to structure the other goals.

Atomic goals

- `Greet and introduce (Agreet)`
  parameters: none
  The system has to greet the user and introduce itself at the beginning of the dialogue.
- `Close (Aclose)`
  parameters: none
  At the end of the dialogue, the system has to say goodbye to the user and end the conversation.
- `Present part (ApresentPart(...))`
  parameters: number of the presentation part to be presented
  The information to be presented to the user is placed on the presentation board in pieces suitable for presentation. The system presents them one by one. This goal states that a particular part has to be presented.
- `Present information`
  parameters: the information to be presented
  It may be necessary to present specific information the user has requested. This goal is achieved when that information is presented to the user.[2]
- `Determine slot (Adetermine(...))`
  parameters: name of the slot on the information board
  In order to determine the informational needs, slots on the information board have to be filled with suitable domain values.

[2]How to specify the information in this goal is not yet clear. Maybe by specifying a slot whose content is the information or by specifying a part of a travel plan or by specifying a part of the previous system response.

- `Verify slot (Averify(...))`
  parameters: name of the slot on the information board
  The contents of a slot on the information board has to be verified. The slot is filled with a value, but the system is not sure of the contents.

- `Determine choice (Achoice(...))`
  parameters: name of slot and possible values for that slot
  The system can not make a decision between two or more values for a slot. The user is probably needed to make the choice.

- `Query database (Aquery)`
  parameters: none
  The system has enough information to query the database. The database has to be queried through the query module.

- `Ask further needs (AfurtherNeeds)`
  parameters: none
  After the user has received the information the system has looked up for him, the system has to inquire whether there are further informational needs. Maybe the user has some further needs or the system did not give the right information the first time.

- `Reconfirm (Areconfirm(...))`
  parameters: the information to be reconfirmed
  The user can ask for a reconfirmation of some information. This goal states that such a reconfirmation should be given.

6.2 Goalstack

The goals are structured by means of a goalstack. This is a dynamic structure which changes during the dialogue. The goals on the goalstack are the goals that still have to be achieved. When a goal is achieved it is removed from the goalstack.[3]

The goalstack consists of levels. The bottom level is level zero, the next level one, etc. On each level there can be one or more goals. The goals higher on the stack should generally be achieved first as they have a higher priority. The goals that stand next to each other on the same level, have the same priority. It is possible that goals with a lower priority are achieved before goals with a higher priority. This is however not the intention of the dialogue manager. The dialogue manager will try to achieve the goals with a higher priority first. But as a consequence of the user's utterances, lower priority goals can be achieved earlier. As soon as a goal is achieved it is removed from the stack, wherever it is on the stack.

When new goals are formed by decomposition (see subsection 6.1.1), they can be placed anywhere on the stack, but usually at the top. The dialogue updating module maintains the goalstack. After the dialogue updating module is finished with the stack there should be atomic goals at the top. Complex goals are popped off the stack and decomposed into subgoals. The subgoals are then

[3]Strictly speaking the goalstack is not a stack. The term stack is used because it generally reflects the use of it.

pushed on the stack (on the same level or at consecutive levels depending on the rules for decomposition). When the stack is empty the dialogue should be finished. So the goalstack usually contains atomic goals at the top and complex goals at the bottom. The dynamic behaviour of the goalstack is explained in chapter 7.

6.3 Operations

The operations are part of the strategy for dialogue control. Operations alter the state of the dialogue. They alter the state in such a way to help the goals to be achieved, see also section 5.2.

To achieve state goals, operations may be executed to change the state of the dialogue. The operations try to change the state in such a manner that goals can be considered achieved. The operations are based on domain knowledge and operate on the current state of the dialogue. The dialogue manager always tries to achieve the goals by means of operations, i.e. without further help from the user. When the dialogue manager does not succeed, the communicative actions can be used.

The operations are related to goals. Every operation belongs to a certain goal. The operation tries to change the state of the dialogue in such a manner that the goal it belongs to, is achieved.

Operations are not always applicable. Only when the state of the dialogue fulfills certain conditions, operations are executed. Separate conditions are specified for every individual operation. The operations should be based on domain knowledge and are usually only applicable in certain situations. The conditions specify those situations. The rest of this section describes how the operations are specified.

First of all, the goal the operation is related to, must be specified. Each operation belongs to a particular goal. The operation is performed to change the state of the dialogue in such a way that that goal is achieved. Next, the conditions for executing must be specified. The state of the dialogue is the combined state of the blackboards. Most operations can only be executed when certain information is already present. Therefore conditions of the information board are useful conditions for operations. Other useful conditions are possibly conditions of the goalstack or control board. Here is a list of conditions that might prove necessary.

- A particular slot on the information board has a particular value, or is greater or smaller than a particular value.
- A particular slot on the information board is 'asked for'. The slot has no value but the user stated that he wanted to know that information.
- A particular slot on the information board has no value.
- A particular goal is part of the goalstack.
- True. This conditions is always met. This operation it belongs to, is always executed.

When the conditions are met, the operation can be executed. The operations themselves should be able to modify the state of the dialogue. In order to do so, changes to the information board and the goalstack must be possible operations. It is also possible to let another module make some changes to the dialogue state. These modules have to be asked to do so. A initial set of types of operations is as follows.

- Place a certain value in a particular slot on the information board.
  The particular slot on the information board has to be specified and the value which should be put there.
- Add a goal to the goalstack.
  The goal that is to be placed on the goalstack, has to be specified[4].
- Send a certain message to another module.
  The module the message should be sent to, the message name and the body should be specified. When this operation is executed the module can wait for an reply if necessary. The reply is only used to indicate that the other module has done its job.

So to define an operation you have to specify

- the goal it belongs to,
- the conditions for which the operations are to be executed and
- the operations that are to be executed.

With this way of specifying operations it is possible to define some obvious operations in an information retrieval dialogue.

Default value

Goals of the type `Determine slot` can sometimes be achieved by filling them with a default value. This can be used for slots that normally have a particular value unless the user states otherwise. A justification for a default value can be a statistical analysis showing the same value for a certain slot in most situations. E.g. as long as the user does not state a date in a train timetable information dialogue, the user usually wants to travel on the same day as the dialogue takes place.

The use of a default value means that the system will not engage in a sub-dialogue about the value of that slot. Due to the default value, the dialogue will not falter when the user has not provided information for that slot. In case the user does bring it up later in the dialogue, the default value can be replaced.

This kind of operation should be used when there is no information for the slot. When the slot contains some information that may not be specific enough, the default value will not be used. So only when nothing is known about the slot in the dialogue, this kind of operation can be executed. This also means that when the user denies the default value after verification but does not give another value, this operation should not be performed again.

[4] In the first version such a goal will be placed on top of the stack. In future versions maybe an optional level can be specified. The goal is then placed on that level on the goalstack or a level relatively to the top or another goal.

The fact that the contents of a certain slot on the information board is a default value should be made known to the information board, so they will not use the value as something the user has mentioned.

Send message

For some goals it is not possible to achieve them by dialogue. The change in the state of the dialogue which achieves this goal, can only be brought about by a certain module. A message must then be send to that module telling it to perform some specific action.

E.g. the goal `query database` can not be achieved by something the user says. This goal can only be achieved when a query is posed to some database and the result is processed in manageable pieces and placed on the presentation board. In order to achieve this goal a message should be sent to the query module. This operation should always be executed for this goal, so there are no conditions (or actually only a `true` condition).

Substitution

In some situations the slots on the information board are filled, but the information in it is not precise or specific enough. The user mentioned this information, but in order to achieve a goal, the slot has to be filled with more specific information. The dialogue manager could not determine what specific information is meant by the user, there is more than one possibility, e.g. given by the reference module. An operation can then be executed which substitutes the existing information in that slot for a more specific value. The decision to choose between the possible values that are more specific is then part of the strategy.

6.4 Actions

The goals state what should be achieved during the dialogue and thus what should be done. They do not state how this should be done. There are several ways to achieve the goals, as shown in previous sections.

State goals are achieved when the state of the dialogue fulfills certain conditions. The state of the dialogue can be changed either by operations performed by the dialogue updating module or by processing the user's turn in general. When it is not possible to achieve the state goal by operations, some interaction with the user is needed to change the state in such a way that the goal can be considered achieved. The user must be tempted to supply the needed information which will put the dialogue in the desired state (see subsection 6.1.2, page 36).

Informative goals are achieved by informing the user of some piece of information. When the user gives any reaction, the informative goal can be considered achieved (see subsection 6.1.2, page 35).

So in order to achieve goals, the dialogue system often has to communicate with the user, either the user must be tempted to give the right information or the user must be informed of some piece of information. Communicative actions are the units of communication with the user as they cannot be divided into smaller parts. Several communicative actions however can combined into one system response.

When the dialogue updating module has finished, the top of the goalstack is filled with atomic goals for which communication with the user is needed[5]. The task of the response generation module is to create communicative actions for these goals and combine them in a system response.

Each atomic goal has an associated list of communicative actions. All those actions are aimed at achieving the goal, but possibly in different ways. The actions can be quite specific and only targeted at a specific goal. They may also be of a more general nature, applicable to more than one specific goal.

The same action can thus be presented in more than one list of actions (related to different goals). E.g. a general encouragement can be used to tempt the user in giving relevant information. The system assumes the user knows approximately what information is needed by the system. This action can be used for trying to achieve various goals.

The list specifies the order in which the actions are tried when trying to achieve the goal. The actions at the end of the list should be used before actions at the beginning.

In a mixed-initiative dialogue the first action in each list should be directly targeted to achieve the related goal. It should be unique for that goal. This action belongs to a directive approach where the system has the initiative. Subsequent actions should be of a more general nature. They are no longer targeted only to the related goal but can also be used to try to achieve other goals. They belong to a more open approach of the dialogue, where the user has the initiative. When the actions are tried from end to beginning, the initiative in the dialogue will initially be given to the user. If the user does not supply the desired information the system utterances becomes more specific and the system takes over the initiative.

The list of actions related to an atomic goal will be generated only once. When the response generation module sees atomic goals without a list of actions on the goalstack, it generates the related list of actions.

The actions on a list will only be used one time. So if it is intended to use a certain action more than one time, the same action must be put in the action list several times.

The action lists determine for a great deal the directiveness of the dialogue. What action list is generated for a specific goal is defined as part of the strategy. The strategy specifies for every possible goal what action should be part of the action list for that goal and in which order they should appear. The priority of the actions is the same as the priority of the related goal.

All the action lists together form the agenda of the response generation module. The agenda is located on the control board. The agenda is rooted in the goalstack (every list of actions is linked to a certain goal). The control board deletes the list of actions when the related goal is removed, because it has been achieved. This is signaled by the dialogue updating module.

For the generation of the system response, the response generating module

[5]The only other possibility is an empty goalstack. The dialogue should then be closed and the communication with the user should be finished.

first selects one action. The response generation module looks at the actions at the end of all the action lists. Only those actions are eligible for selection for the system response. The same actions can also appear somewhere else in action lists, but that is not the reason they are eligible for selection. The action with the highest priority is selected for the system response in any case. It can then possibly be combined with other actions.

Rules in the strategy define which communicative actions can be combined in a system response. If several actions can be combined with the action of the highest priority, the action with the highest priority among those is selected.

After the response is made to the user, the communicative actions that were part of that response are removed from the action lists. When a certain action, that was part of the system response, appears in several lists, it is removed one time in each list (the appearance closest to the end). This is done to make it possible to have the same action appear more than once in an action list when the same communicative action should be executed more than once trying to achieve a goal.

If the action that is to be removed, is not located at the end of the action list, all the actions located between that action and the end of the action list should also be removed. This can happen when the action appears in more than one list and is located at the end of at least one of those lists. The actions that appear after the selected action should be used before this action not after. So if this action is nevertheless part of the system response, those actions that appear after it should not be part of future system responses and therefore be removed as well.

The system response can be targeted to the achievement of more than one goal. The response can contain an action that appears in more than one action list, each list related to its own goal. The response can consist of more than one communicative actions, each at the end of an action list.

When the action list related to a certain goal is empty and the goal is not yet achieved, there is a problem. The system has then tried to achieve the goal in several manners[6]. However somehow the system failed to achieve the goal. In such circumstances the dialogue should be aborted and the user, when possible, redirected to another service.

An overview of the initial set of communicative action for an information retrieval dialogue will end the discussion on communicative actions. In chapter 8 these actions will be used. The exact form of the actions used in that chapter is place between parentheses.

This is only an example of how the actions can be defined and what kind of action are to be expected. In a complete system the action will be specified by the action rules. The dialogue act generation must be able to transform those action to dialogue acts. So the action rules and the dialogue act generation module must be defined in conjunction.

- **`Present part (PresentPart(...))`**
  parameters: serial number of the part to be presented
  The information to be presented to the user is placed on the presentation

[6] It is possibly the same manner tried more than once (the same communicative action appearing more than once in the initial action list of that goal).

board by the query module in manageable pieces suitable for easy presentation. These pieces can be referenced by their position in the presentation (see [Oldervoll, 1998]).

- `Present information`
  parameters: the information to be presented
  Some specific piece of information is presented to the user.[7]

- `Greet and introduce (Greetings)`
  parameters: none
  Greet the user and introduce the system.

- `Goodbye (Bye)`
  parameters: none
  Say goodbye to the user.

- `Verify slot implicitly (VerifyImplicit(...))`
  parameters: the name of the slot to verify
  The value of a slot has to be verified. Implicitly verify this slot. This action should thus be combined with another action if possible otherwise it would be an explicit verification.

- `Verify slot explicitly (VerifyExplicit(...))`
  parameters: the name of the slot to verify
  The value of a slot has to be verified. Explicitly ask for a verification of that value.

- `Ask slot (Ask(...))`
  parameters: the slot to ask for
  A slot does not have a value yet. Ask the user for the value of that slot. Instead of a slot the parameter can also be a more general indication. The dialogue act generation has to know though, how to transform that kind of communicative action to dialogue acts.

- `Ask choice (Choice(...))`
  parameters: the slot for which their are several possibilities and the possibilities to choose from
  Ask the user for a choice between several possible values for a certain slot.

- `Reconfirm (Reconfirm(...))`
  parameters: the information to be reconfirmed
  Reconfirm the information the user wants reconfirmed. The dialogue act generation should be able to determine with what dialogue acts that can be accomplished.

- `Ask further needs (AskFurtherNeeds)`
  parameters: none
  Ask the user whether he wants more information.

[7] How to specify the information in this action is not yet clear. Maybe by specifying a slot whose contents is the information or by specifying a part of a travelplan or by specifying a part of the previous system response.

- `Please wait`
  parameters: none
  Ask the user to wait one moment while the system is busy. Maybe this is necessary when the querying of the database takes a lot of time. It can be used to control the initiative.

- `Encouragement (Encouragement)`
  parameters: none
  A general encouragement.

6.5 Reaction rules

The reaction rules describe how the system should react to certain utterances from the user. These rules make it possible to alter the standard structure of the dialogue defined by the complex goals. With these rules a mechanism is created with which the system can react to user requests. This section describe what these reaction rules look like.

The reaction rules state what operations should be executed in what circumstances. The conditions for execution are now not only related to the state of the dialogue as contained by the information board, but are also related to the user's utterance. So the conditions for certain operations are met by the user's dialogue acts in a turn. The same conditions as for the operations can be used for the reaction rules, but only in combination with conditions to the dialogue acts of the user.

In order to determine whether the system should react to specific dialogue acts of the user, it will also be useful to be able to pose conditions on which goals were incorporated with their actions in the last system response, for the user can be reacting on the last system response. So next to the conditions already encountered in the discussion of the operations in section 6.3, there are two new conditions which can be used for the reaction rules.

- The dialogue acts of the user's utterance contains a certain dialogue act.

- A particular goal has an action in its actionlist which action was incorporated in the last system response.

Since the reaction rules are used to define how the system should react to certain utterances from the user, a condition on the dialogue acts of the user's turn should always be part of the conditions of a reaction rule.

The operation to be performed when the conditions are met, are the same as the operations defined in section 6.3. With those operations it is possible to add goals to the goalstack in order to change the standard goal structure as prescribe by the complex goals and their decomposition rules.

6.6 Strategy

By using the framework as explained in section 5.3, an application independent engine is created to control the dialogue. The exact operation of the engine is specified by certain sets of rules. These rules together form the strategy for dialogue control of a particular system working within this framework for an ASP system.

The various rulesets the strategy consists of, are mentioned throughout the text. As the conclusion of this chapter a complete overview of the rulesets is given.

Reaction rules
These rules specify how the system should react to some user utterances. A description is given in section 6.5.

Decomposition rules
These rules specify how the complex goals should be decomposed into subgoals. See subsection 6.1.1.

Achievement rules
These rules state when atomic goals are achieved. See subsection 6.1.2.

Operation rules
What operations can be executed in order to achieve certain goals and under what conditions, are stated in these rules. See section 6.3.

Action rules
These rules specify the actionlist for every possible goal. See section 6.4.

Combination rules
These rule specify which actions can be combined in a system response. See section 6.4.

Chapter 7

Dialogue control modules

This chapter describes the development and implementation of the actual modules and blackboards for dialogue control. Dialogue control in the Alparon system is placed in two modules and one blackboard. The dialogue updating module updates the goal structure and the response generation module generates the system response using the updated goal structure. The goal structure and everything connected to it, is placed on the control board together with some other information for dialogue control. The design and implementation of these modules and blackboard will be described in detail in this chapter. First, this chapter will start with a description of what actually happens before and after dialogue control in the dialogue manager, because some situations may occur which should be dealt with by dialogue control later on. The description is given from an implementation point of view.

7.1 Besides dialogue control

After every user's turn several modules are invoked one after the other in a fixed order by the dialogue manager. The user's turn is passed through the speech recognizer, parser and coder before it reaches the dialogue manager. The input of the dialogue manager is a list of possible interpretations of the user's turn with their probabilities. Each interpretation consists of a list of dialogue acts which represent the user's utterance. So the input is a list of lists of dialogue acts, the recognized dialogue acts (rda's). The output of the dialogue manager is a list of dialogue acts which represent the system's response to the user, the proposed dialogue acts (pda's).

To process the user's turn and generate a system response the following modules are invoked by the dialogue manager one after the other.

1. Disambiguation module
   arguments: list of lists of dialogue acts (rda's)
   result: reordered list of lists of reformulated dialogue acts (the disambiguated dialogue acts, dda's)
   This module reorders and reformulates the interpretations of the user's turn given the probabilities of the input interpretations, the dialogue context and domain knowledge. When the disambiguation module can not choose between several interpretations that only differ for some domain

values, all these interpretations will be considered equally likely and placed next to each other in the reordering, they could all be selected for further processing. The reformulation can alter both the move and information codes of the dialogue acts. See also section 4.2.1 for a more detailed description of the disambiguation module.

2. Context updating module
   arguments: original list of lists of dialogue acts (rda's) with the reordered list of lists of reformulated dialogue acts (dda's)
   result: most likely list of reformulated dialogue acts, the selected dialogue acts (sda's)
   The interpretation of the user's turn the disambiguation module thinks most likely, is selected for further processing. Both the original list of interpretations and the reordered list of reformulated interpretations are stored on the history board for future use. The context updating module then brings the context up-to-date with the selected interpretation. The information contained in the selected interpretation is stored on the information board. New values for slots on the information board that conflict with already existing values for these slots, will nevertheless be put in the slots, but the control board will be notified of this conflict. Dialogue control should deal with this.

   Are more than one interpretations selected for further processing, the common information of all these interpretation will be put on the information board and the different domain values for a certain slot will be supplied to the control board as a choice still to be made. This also should be dealt with by dialogue control. See also subsection 4.2.1 for a more detailed description of the context updating module.

3. Dialogue Updating module
   arguments: selected list of reformulated dialogue acts (sda's)
   result: *empty*
   This is the first part of the dialogue control. The goal structure of the dialogue will be updated. What still has to be done during the dialogue will be determined here. See section 7.2.

4. Response generation module
   arguments: *empty*
   result: *empty*
   This is the second part of the dialogue control. The communicative actions will be determined and a system response formulated. How the things have to be done will be determined here. See section 7.3.

5. Dialogue act generation
   arguments: *empty*
   result: the proposed dialogue acts (pda's)
   The communicative actions of the system response are transformed to dialogue acts. See also subsection 4.2.1.

The next sections will deal with the modules and blackboard for dialogue control.

7.2 Dialogue updating module

In this section the tasks of the dialogue updating module will be described first, followed by a discussion of the design issues. Next the interface the module presents to the rest of the dialogue management subsystem will be presented, i.e. the messages it can receive and send. Finally some comments will be made regarding the implementation of this module.

7.2.1 Tasks

When the dialogue updating module is invoked, the disambiguation module and the context updating module have already been invoked and have done their work. The context updating module possibly reported conflicts for values of slots on the information board and choices for values of slots to the control board. The domain information of the dialogue acts representing the user's turn, have already been extracted from the dialogue acts and placed on the information board. So the domain values in the dialogue acts are hardly of any interest to the dialogue updating module, that information can be obtained from the information board[1].

The dialogue is controlled by a goal structure, the goalstack, as described in chapter 5. The main task of the dialogue updating module is to update the goalstack. The goalstack contains information on the dialogue that persists more than one turn of the dialogue and is therefore placed on the blackboard system, namely on the control board. The dialogue updating module thus changes the state of the dialogue[2]. For this modification of the dialogue state, the dialogue updating module uses the information on the blackboard system and the representation of the user's turn consisting of the selected dialogue acts. So when the dialogue updating module is asked to 'update the dialogue':

- dialogue acts have been selected and possibly reformulated to represent the user's turn,
- the contents of these dialogue acts have been placed on the information board,
- these dialogue acts are passed the the dialogue updating module in the message body,
- conflicts for slots on the information board may have been reported to the control board and
- choices between domain values for certain slots that still have to be made, may have been reported to the control board.

The main task of the dialogue updating module can be decomposed into several subtasks.

[1] In case of a choice in domain values still to be made, the domain values are placed on the control board as possible values for a slot and not yet on the information board.

[2] Next to changes to the goalstack, it can also make changes to the information board as a side-effect.

Reaction

The initial goal and the decomposition rules define the standard structure of the dialogue. The user will probably not conform to this standard structure but deviate a little. He expects the system to react to his utterances. For example when the user asks for a clarification, he will want a clarification and not the predefined continuation of the dialogue. So when the user deviates from the standard dialogue structure, the system has to react adequately to this. The dialogue updating module receives the selected dialogue acts representing the user's utterance, so it can determine whether it should react to the user's turn. How to react on certain dialogue acts, what consequences this has for the goalstack, is one of the subtasks of the dialogue updating module.

Conflicts

The context updating module could have established conflicts on the information board for certain slots. A domain value, in the selected interpretation of the user's turn, which should be put in a certain slot, conflicted with already present information in that slot. The new domain value is placed in the slot, thus rendering the old, conflicting value not longer valid, but this fact is reported to the control board. Another subtask of the dialogue updating module is to deal with such situations. To do nothing about it, is one way to deal with it. Another possibility could be to verify the new value with the user.

Choices

Another situation the context module could encounter, is a still open choice between conflicting domain values for a certain slot. The disambiguation module could not determine one interpretation more likely than another, for they only differ in some domain value[3]. Those possible values for a certain slot are not put on the information board, but instead reported to the control board. To apply a strategy to make a choice between these values is another subtask of the dialogue updating.

Achievement and operations

The first three subtasks were directly concerned with the user's turn. The subtask described here is concerned with the goals on the goalstack. The goalstack may be modified as a result of the first subtasks and the whole state of the dialogue is modified according to the user's turn. All the atomic goals on the goalstack should be checked for achievement. Informative goals are considered achieved when a communicative actions related to that goal, was part of a previous system response. State goals are achieved when the state of the dialogue meets certain conditions. As a result of the previous system response and the modifications of the dialogue state by the user's turn, some goals on the goalstack can be considered achieved and should therefore be removed from the goalstack.

[3]For example the situation where the dialogue acts contain the information element `station(IJlst)` and `station(Elst)`, see subsection 4.2.1.

The changes to the dialogue state can also have the effect that some operations can now be executed. The execution of these operation can help with the achievement of the atomic goals on the goalstack.

The checking for achievement and operations that can be executed, is a very important subtask of the dialogue updating module, for the dialogue can only be considered successfully finished when all the goals are achieved. The system has then completed its job.

Proper form

The last subtask of the dialogue updating module is to ensure the goalstack is in a proper form when the module finishes. When the dialogue updating module is finished, the response generation module is invoked to generate a system response. The response consists of communicative actions that belong to certain atomic goals which still have to be achieved. So it is important that the goalstack contains atomic goals for the response generation module to work on. The last task of the dialogue updating module is to ensure there are indeed atomic goals on the goalstack (actually even on top of the goalstack) which are not already achieved (otherwise they would not be there, achieved goals are removed from the goalstack).

7.2.2 Design issues

This subsection covers the design of the dialogue updating module. This design is a first version for the dialogue updating module. The design should be extendible, so future versions can be made by adding and refining functionalities. The design is discussed according to the subtasks described in the previous section.

The actual behaviour of the dialogue control is defined for a certain application by various rulesets. These rulesets define how various aspects of dialogue control should be executed, the strategy for dialogue control. The rulesets are stored in files. They are read by the modules for dialogue control when they are started and stay fixed thereafter. When another ruleset has to be used, the module that uses that ruleset should be restarted.

Reaction

How the system has to react to certain dialogue acts in the user's turn can be defined with reaction rules. Section 6.5 describes what the reaction rules should be able to specify.

A reaction rule consist of a condition part and an operation part. The condition part defines the conditions which should be met if the operations of the operation part are to be executed. All the conditions of the condition part should be met before the operations may be executed. When there are alternate conditions for the operation part, a new reaction rule should be specified. In other words all the conditions are taken in conjunction (And-ed) and not disjunction (Or-ed).

The syntax of a reaction rule is

condition_1 & ... & *condition_m* : *operation_1*, ... , *operation_n*

Conditions

There are several types of conditions that can be used as described in section 6.3 and 6.5. The syntax and semantics of these types of conditions are as follows:

- The dialogue-act condition:

  `DA` *dialogue_act_expression*

  The two letters at the beginning of the dialogue-act condition (`DA`) signify that this condition is a dialogue-act condition. The dialogue act expression specifies a dialogue act (or part of a dialogue act) that must be present in the dialogue acts that represent the user's turn. The dialogue act expression may contain an asterisk (*), which matches with everything.

  E.g. When the representation of the user's turn contain the dialogue act `Sta(place(city(Delft)))`, the conditions `DA Sta(place(*))`, `DA Sta(place(city(Delft)))` and `DA Sta` are all met. The condition `DA Sta(place(station(*)))` is not met.

- The goal condition:

  `Goal` *goal_expression* `exist`

  The words `Goal` and `exist` signify that this condition is a goal condition. The goal expression specifies a goal that must be part of the goalstack for this condition to be met. Again the goal expression may contain an asterisk, which matches with everything. When more than one goal matches the goal_expression, the first goal encountered will be used to define for what the asterisk in the operations should be replaced.

  E.g. When the goal `Averify(place)` exist on the goalstack, the conditions `Goal Averify(place) exist` and `Goal Averify(*) exist` are both met.

- The goal-in-response condition:

  `Goal` *goal_expression* `in response`

  The words `Goal` and `in response` signify that this condition is a goal-in-response condition. The goal expression specifies a goal that must be part of the goalstack and that has an action in the actionlist related to this goal that was part of the previous system response. So only when the goal exists on the goalstack and a communicative action related to this goal was part of the previous system response, this condition is met. This condition may be necessary to relate the user's turn to the previous system response. The goal expression may contain an asterisk, which matches with everything.

- The slot-value condition:

  slotname `is` *value_expression*

  The word `is` signifies that this is a slot-value condition. The slotname specifies a certain slot of the information board which should contain at least the value expression in order to be met. In other words, this condition is met when a specified slot on the information board contains a specified value expression or the value expression is at least part of the actual value of the slot. The value expression can contain an asterisk which matches everything.

E.g. When the slot `place` contains the value `city(Delft)`, the conditions `place is city(Delft)`, `place is city` and `place is city(*)` are all met. The condition `place is city(Rotterdam)` is obviously not met.

- The slot-attribute condition:

  slotname `has` *attribute_expression*
  slotname `hasnot` *attribute_expression*

  The words `has` or `hasnot` signify that this condition is a slot-attribute condition. The slotname specifies a certain slot on the information board which should have or should not have the attribute as specified by the attribute expression as attribute. What attributes a slot should possibly have is not determined by the dialogue control modules. But the attributes `empty`, `asked`, `verified`, `user`, `system` and `default` can be used with the current version of the information board.

  E.g. When the slot `place` has `user` and `verified` as attributes, the conditions `place has user` and `place hasnot empty` are both met.

- The true condition:

  `true`

  This condition with only the word `true`, is always met. The operations combined with this condition will be executed every turn of the dialogue.

The conditions of a reaction rule should probably always contain a dialogue act condition, for the reaction rules define how to react on certain dialogue acts of the user's turn.

Operations

There are three kind of operations that can be executed and therefore be part of the operation part of the reaction rule. They are described in section 6.3.

- The put operation:

  `put(`*slotname*, *value_expression*, *ground*`)`

  This operation places the value *value_expression* in the slot *slotname* on the information board with *ground* as ground for this value. This ground should state the reason for placing the value in the slot (examples are `default` or `system`). This ground will then be an attribute of the slot for the current value. The *value_expression* may contain an asterisk. Whatever matched the last asterisk in the conditions is inserted for the asterisk of the *value_expression*.

- The goal operation:

  `goal(`*goal_expression*`)`

The goal *goal_expression* will be placed on top of the goalstack. The *goal_expression* may contain an asterisk, whatever matched the last asterisk of the conditions is inserted for the asterisk of the *goal_expression*.

- The send operation:

  `send(`*modulename*, *messagename*, *messagebody_expression*`)` or
  `send(`*modulename*, *messagename*, *messagebody_expression*, *replymessagename*`)`

  This operation sends a message to the module *modulename*. The message name is *messagename* and the body of the message consists of one line which is *messagebody_expression*. The *messagebody_expression* can contain an asterisk, whatever matched the last asterisk of the conditions is inserted for the asterisk of the *messagebody_expression*.

  After the message is sent the dialogue updating module can do two things: it can wait for a reply message or it can just go on without waiting for a reply. When the dialogue updating module should wait for reply, the name of the reply message can be specified by *replymessagename*. The dialogue updating stops running until it receives a message with that name.

The asterisk (`*`) can be used in some places both in the conditions and operations to form generic rules. The asterisk matches on everything for the conditions. Whatever matches the last asterisk of the conditions is inserted for all the asterisks of the operations. If the conditions do not have an asterisk there should not be any asterisk in the operation. The asterisk is then just removed from the operations.

The dialogue updating module checks whether there are some reaction rules for which the conditions are met, every time the dialogue updating module is invoked to process the user's turn and executes the accompanying operations. The reaction rules will be tried in the order in which they appear in the file with the reaction rules.

Conflicts

The dialogue updating module has to check whether the context updating encountered some conflicts while extracting domain information out of the dialogue acts of the user's turn. In order to do so the module must send a message to the control board to collect the possible conflicts. The control board replies with the name of the slots for which the control board reported conflicts. The new conflicting values are already put on the information board by the context updating module.

This version for dialogue control will just add a goal for verification of the slot to the goalstack in those case where the previous value of the slot has `user` or `system` as ground. A ground of `user` means the user introduced the old value and a ground of `system` means the system introduced the old value with reasonable motivation.[4] Such old values can not be easily discarded. So the new value is accepted but will be verified with the user in these cases. A goal to

[4]Default values are also introduced by the system but should have `default` as ground not `system`.

verify the slot provides at least some feedback to the user. The user then knows another value is understood by the system and can take action to correct the system if necessary. Another possibility is to ignore the conflict and just accept the new value without further ado. This is done if the old value does not have the afore mentioned grounds.

When the dialogue updating module collects the conflicts from the control board, they are automaticly removed from the control board, so the conflicts are handled in the same turn of the dialogue as they arise. Off course the consequences of handling conflicts are spread over more than one turn, but now the goals to verify slots are part of the overall goal structure and are treated just as the other goals.

Choices

When there are still choices between domain values for slots to be made, the dialogue updating module can collect these choices from the control board. The control board specifies a choice as a slotname followed by the possible values for the slot with that name. The dialogue updating module handles this kind of situation by adding a goal for making a choice between the possible values to the goalstack. So it just transforms the notification for a choice to a goal for making that choice. This goal can then be treated just as the rest of the goals.

The same as with the conflicts, the notification of a choice situation will be removed from the control board as soon as it is collected. The dialogue updating module will handle such a situation in the same turn as it appears.

Achievement and operations

The checking for achievement and operations could be considered two separate subtasks. But the rules for goal achievement and the rules defining under what circumstances, what operations are to be executed, are both linked to particular atomic goals. So both tasks are executed simultaneously. All the atomic goals on the goalstack are checked one by one for achievement and operations.

Each achievement rule is linked to a particular atomic goal. It defines the conditions for which the goal is considered achieved. It consists of alternate sets of conditions. The goal is considered achieved when for at least one of the alternate sets of conditions, all conditions of that set are met. The same kind of conditions could be used as for the reaction subtask described above, except for the dialogue act condition. But it only makes sense to use the `slot-value`, the `slot-attribute` and the `true` condition. One other condition will be added for the achievement rules that is not applicable for the reaction rules, namely the communicated condition. This condition is met when a communicative action linked to the goal was incorporated in the last system response. This condition can be used for the achievement of informative goals.

- The communicated condition:

  ```
  communicated
  ```

  Just the word `communicated` is used to designate this condition.

This condition resembles the goal-in-response condition. But in the goal-in-response condition the goal is specified in the condition itself and in the communicated condition the goal is only implicitly specified (as the goal to which this achievement rule is linked).[5]

The syntax of an achievement rule is

goal:
condition_11 & ... & *condition_1n*
...
condition_m1 & ... & *condition_mn*

In the current implementation, there should only exist one achievement rule per goal in the achievement ruleset.

Instead of linked to a particular goal, it is also possible to link an achievement rule to a generic goal. The *goal* then contains an asterisk in round brackets (`(*)`). When the dialogue updating module cannot find an achievement rule for a particular goal, it will check whether one of the achievement rules for generic goals can be applied for this particular goal. If a generic goal can be matched against the particular goal - with the asterisk matching everything - the achievement rule it is linked to, will be used for that particular goal.

Every operation rule is also linked to a particular goal. Linkage to a generic rule as above is not yet implemented, but can easily be implemented when the need arises. First of all the goal is specified. Then the following lines are just like the reaction rules and consists of two parts. The first part defines the conditions which must be met before the operations in the second part are to be executed. More than one pair of conditions and operations can be linked to a particular goal in a operation rule. The syntax of the operation rule is

goal:
condition_11 & ... & *condition_1n* : *operation_11*, ... , *operation_1r*
...
condition_m1 & ... & *condition_mn* : *operation_s1*, ... , *operation_sr*

The conditions described above can be used, but only the `slot-value`, `slot-attribute` and `true` condition. Conditions on the goalstack are not used for operation rules, just the conditions on the information board. The operation should benefit particular goals whatever other goals there are at a certain moment. When the need arise the operation rules mechanism can be easily extended to be able to handle conditions on the goalstack. The operation are the same as for the reaction rules, the `put`, `send` and `goal` operation. When all the conditions before a set of operations are met, the operations are executed[6].

When a set of operations is executed, those operations will not be executed again during the dialogue as a consequence of the same conditions being met for the same goal. Those conditions can be met for a number of turns, or even for the rest of the turns in the dialogue. The purpose of the operation rules is not to execute the same operations every turn in the dialogue. So after a set of conditions is met and the operations are executed, this will not be repeated[7].

[5] As a matter of fact the communicated condition can be replaced with the goal-in-response condition.

[6] And of course, the goal they are linked to exists on the goalstack, otherwise the dialogue updating module will not even consider the operation rule.

[7] Some of these operation can appear elsewhere in the operation rules. Then it is possible that the same operation is executed again, but this time because another set of conditions is met. Or one can say it is a different operation with the same result.

Generic operation rules are also allowed, as with the reaction rules. The asterisk should be matched in at least one of the conditions and is replaced in the operations with the last match of the asterisk in the conditions. Now the same operation will not be executed with the same replacement of the asterisk. Are the conditions met again but with another match for the asterisk, then the operations are executed again with this new replacement for the asterisk.

As with the achievement rules, there should be one operation rule per goal in the ruleset.

When performing this subtask, the dialogue updating module first requests all atomic goals on the goalstack from the control board. Then, for each of these goals in turn, the dialogue updating module will first check the achievement rules for this goal. If the goal can be considered achieved it is removed from the goalstack. In order to do so the dialogue updating module sends a message to the control board. The control board then removes the goal from the goalstack. Secondly, the operation rules will be checked for this goal, if it is not already achieved. When a set of conditions is met and a set of operations is executed, the same goal is checked for achievement afterwards. This happens for each of the atomic goals one after the other.

The goals being added to the goalstack because some operation is executed, are not checked for achievement and operations during the same turn as they appear. The dialogue updating module collects all the atomic goals from the goalstack in one batch and then starts checking them for achievement and operations.

Proper form

To perform this task the dialogue updating module has to check whether there are atomic goals on top of the goalstack. The goals on the top of the goalstack have the highest priority. The dialogue manager should attempt to achieve these first. But in order to make a response that will tempt the user to make utterances that change the state of the dialogue in such a way these goals can be considered achieved, the goals should be atomic. This means the dialogue updating module should ensure there are atomic goals on top of the goalstack[8].

This subtask is completed when there are atomic goals on top of the goalstack (possible with complex goals[8]). The dialogue updating module can check this by sending a message to the control board. When there are no atomic goals on top of the goalstack, the top level complex goals should be collected, they are popped off the goalstack (i.e. the are removed). All the goals of the top level will be collected. Then the complex goals should be decomposed in subgoals and those subgoals placed on the goalstack. The complex goals are decomposed in the order they are received from the control board.

For the decomposition of the complex goals decomposition rules are defined. A decomposition rules defines what subgoals a complex goal should be decom-

[8] Another possibility is to ensure there are only atomic goals on top of the goalstack and no complex goals. But when there are some atomic goals on top of the goalstack, a system response can be made. So the current version will implement this subtask by ensuring there is at least one atomic goal on top of the stack. Besides it is not clear why one wants complex and atomic goals on the same level in the goalstack. It will always be possible to structure the goals in such a way that this can be avoided.

posed into. A complex goal can be decomposed into several levels each with several goals. The goals within a level are placed on the same level of the goalstack. The level will be put on the goalstack in the order they are defined in in the decomposition rule. So the last level to add will become the level with the highest priority. The goals or level that should be achieved first should be placed at the end of the decomposition rule. There should be one decomposition rule per complex goal[9].

Because the complex goals are decomposed one after the other, the levels of the decomposition rule of the first complex goal will be put first on the goalstack, then the levels of the next complex goal will be put on top of that and so on. So when there is more than one complex goal on a level the subgoals are placed on successive levels. This seems to have the same effect as putting the complex goals on successive levels in the first place. However the complex goals are all decomposed in one turn and any atomic goal in that decomposition will be tested for achievement, operations and incorporation in the system response. So there is a difference.

The syntax of decomposition rules is

> *complexgoal*: [[*subgoal11*, ... , *subgoal1n*], ... , [*subgoalm1*, ... , *subgoalmn*]]

The levels are place between square brackets separated with commas, the levels themselves consist of goals between square brackets and separated with commas.

Because there is only one decomposition rule for each particular goal the decomposition of the complex goals is fixed for the current version. There is one exception. In some situations one does not know beforehand in how many subgoals a complex goal should be decomposed, all the subgoals being the same except for an index. E.g. the `present travel plan` goal can be decomposed in several `present part(`*`some number`*`)` subgoals. But because the number of parts the travel plan consists of, will differ from one dialogue to another, the number of subgoals can not be determined beforehand.

When the subgoals contain a goal with contains an asterisk in parentheses (`(*)`) the complex goal is decomposed in a different manner. The subgoal with the asterisk should be an only goal in its level. The other levels with subgoals are placed in the usual manner on the goalstack. The level with the subgoal with an asterisk will be expanded to several levels. Each of these levels will contain one subgoal. That subgoal will be the same as the subgoal with an asterisk but the asterisk will be replaced by descending numbers from some maximum to 1. This maximum should be collected from somewhere[10]. See figure 7.1 for an example.

After the complex goals are popped off the top of the goalstack, they are all decomposed one after the other. Then the previous subtask will be executed again. The atomic goals (if any) will be checked for achievement and operations. Finally the dialogue updating module will check whether there are atomic goals

[9] In future version more than one decomposition rule may be allowed. The dialogue updating module must then decide which rule to use.

[10] That number can be collected from the information board where it is put in the slot with the name of the complex goal. This is only so in the first version of the dialogue updating module. In future version the status board will be a good place to store this kind of information.

When the following decomposition rule exists:
`present: [[subgoal1], [part(*)], [subgoal2, subgoal3]]`
and slot `present` contains: 2
the complex goal `present` will be decomposed in 4 levels:

```
[subgoal1]
[part(2)]
[part(1)]
[subgoal2, subgoal3]
```

Figure 7.1: Decomposition example

on top of the goalstack. If there are not, the whole process starts all over. So only one level is decomposed each time and when this does not put atomic goals on top of the goalstack (after the subtask achievement and operations is executed) the level that is then on top of the goalstack, is decomposed.

Order of execution

The main task of the dialogue updating module can be achieved by subsequently executing the subtasks. Only the order in which they should be executed must be determined. The subtasks that deal with consequences of the user's turn should be executed first. The first three subtasks - concerning reaction, conflicts and choices - are those subtasks. After that the goals can be checked for achievement and operations. And finally the goalstack has to be put in proper form.

Goals that are placed on the goalstack at a later time are placed higher on the goalstack and have a higher priority automaticly. So the order in which the first three subtasks are executed is important, because often the result of these subtasks will be new goals on the goalstack. It seems convenient to first determine unresolved choices, then to solve conflicts and only after that react the user's utterance. In order to react in a meaningful way it is necessary there are no conflicts and choices left, otherwise the exact manner to react will not be clear. It will probably not occur often that all the three first subtasks result in new goals. Usually only one of them or none will supply new goals.

So the subtasks are executed in the order in which they are described here.

7.2.3 Interface

The rest of the dialogue management system can interface with the dialogue updating module by sending messages. Using this method all the communication within the dialogue manager is handled. [Vark, 1997d] gives a description of this message sending mechanism. In this subsection the messages that can be send to the dialogue updating module will be given. Only the messages that are necessary for dialogue control issues - i.e. normal operation of the module - are described. Some auxiliary messages used for development and testing are not described here but are documented in the implementation.

The dialogue updating module can be asked to update the dialogue by sending it the `turn` message:

Message name: `turn`
Message body: dialogue acts
Reply message name: `turn`
Reply message body: none

The message arguments are the selected dialogue acts representing the user's turn.

During the execution of its tasks, the dialogue updating module will send some messages to other modules. In most cases the dialogue updating module expects a reply message. An overview of the messages the module sends for the benefit of its tasks, can be found in table 7.1[11].

Table 7.1: Messages sent to other modules, with replies indicated

Module	Message name	Reply name
control board (CB)	popComplexGoals	ComplexGoals
	getConflicts	Conflicts
	getChoices	Choices
	getAtomicGoals	AtomicGoals
	isAtomicOnTop	AtomicOnTop
	isEmpty	Empty
	isOnlyComplexOnTop	OnlyComplexOnTop
	isCommunicated	Communicated
	existGoal	Goal
	inResponse	Goal
	addLevel	
	removeGoal	
information board (IB)	get	Slot
	getAttr	SlotAttr
	getPrevious	PreviousAttr
	put	

7.2.4 Implementation

The dialogue updating module is implemented in Java 1.2. Java is an object-oriented, portable programming language. Good reference material on Java is [Flanagan, 1997] and the Java documentation supplied with JDK^{TM} 1.2 of Sun Microsystems.

The implementation uses the framework for messaging modules described in [Oldervoll, 1998]. The Alparon library used for the implementation of the dialogue updating module is a slightly updated version of the one described in [Oldervoll, 1998].

The names of the files with the rulesets are hard encoded, the program for the dialogue updating module must be rewritten in order to use other filenames.

[11] During execution the dialogue updating module can send any message as a result of executing a send operation. These messages are not part of the process of updating the dialogue but are a result of it and are not listed in the table.

Table 7.2 lists the various filenames for the rulesets. The files should be in the directory from which the dialogue updating module is started.

Table 7.2: Filenames of the rulesets

Ruleset	Filename
reaction rules	reaction.rules
achievement rules	achievement.rules
operation rules	operation.rules
decomposition rules	decomposition.rules

The source code of the dialogue updating module can be found in appendix C. The source code is documented using javadoc comments. This documentation can be translated into html-pages by the javadoc tool.

7.3 Response generation module

In this section the response generation module will be described. First the tasks of the response generation module will be described. Then the design issues will be dealt with, followed by a description of the interface of this module. At the end of this section, some words will be said about the implementation of this module.

7.3.1 Tasks

When the response generation module is invoked, the dialogue updating module has already done its job. The goalstack is updated so it reflects the current dialogue state. The consequences of the user's turn are processed both on an informational level as on a dialogue level. To progress in the dialogue, a system response must be generated. It is not possible to achieve goals without interaction with the user. To generate a system response is the main task of the response generation module.

The main task of the response generation module can be decomposed into two subtasks.

Updating the actionlists

As is discussed in section 6.4 about actions, certain communicative actions can be performed in order to achieve atomic goals. Since the dialogue updating module tried to achieve atomic goals without further interaction with the user, the atomic goals that are still present on the goalstack can only be achieved throuhg interaction with the user. The interaction with the user takes place by communicative actions. These units of communication can be combined to form a system response.

The communicative actions are targeted towards the achievement of goals. Every atomic goal should be linked to an actionlist which contains actions, which will help the achievement of the atomic goal they are linked to. There can be more than one action linked to an atomic goal because more than one attempt can be made to to achieve a certain goal through communication.

The first subtask of the response generation module is to ensure the actionlists, linked to the atomic goals of the goalstack, are present and up-to-date for all the atomic goals on the goalstack. Existing actionlists must be updated with respect to the last system response and new actionlists must be generated for new atomic goals on the goalstack.

Actual response generation

When all the atomic goals have up-to-date actionlists, the response generation module must generate a system response. The system response is generated by selecting and combining communicative actions that appear in the actionlists of the atomic goals. So this subtask takes care of the actual response generation.

7.3.2 Design issues

In this subsection the design issues for the task of the response generation module are discussed. The design is discussed according to the subtasks.

The actual behaviour of the response generation module is defined by some rulesets. These rulesets are stored in files and read when the module is started. They stay active as long as the module runs. In order to use another ruleset the module must be restarted.

Updating the actionlists

What actionlist should be linked to a particular goal is defined in the action rules. The action rules give for every possible atomic goal a list with communicative actions that can be used to help to achieve that atomic goal. The action rule can be generic, i.e. be instantiated for a particular atomic goal. The syntax of an action rule is as follows

atomicgoal: [*action1*, ... , *actionn*]

When a new actionlist has to be generated for a particular atomic goal, the action rules for which the *atomicgoal* matches that goal, is used. When the action rule contains a *atomicgoal* with an asterisk between parentheses, `(*)`, the asterisk matches anything. The match for the asterisk will replace the asterisks in the actions. The response generation module will first look for a non-generic action rule (an action rule without the asterisk), but when there is none available, the generic action rules will be checked. For every possible atomic action in the dialogue, there must be an action rule[12].

E.g. When an actionlist has to be generated for the atomic goal `Agoal(arg)` and there is no action rule for `Agoal(arg)` but there is the action rule `Agoal(*): [keepquiet, sayloud(*), say(*)]` the actionlist `[keepquiet, sayloud(arg), say(arg)]` will be generated for that goal.

When an atomic goal appears on the goalstack, the next time the response generation module is activated, an actionlist will be added to the atomic goal. So when the dialogue updating module adds a new atomic goal to the

[12] Except for those atomic goals for which there is an operation rule with a true condition that will change the state of the dialogue in such a way that the goal will always be achieved when the operations are executed. In other words such an atomic goal will be achieved without any interaction with the user.

goalstack, the response generation module immediately adds an actionlist to the goal before a system response is generated. Thus all atomic goals will have an actionlist when the actual response generation takes place.

When an action is incorporated in a system response, that action should be removed once from each actionlist that contains it at least once, before a new response is generated. Along with that action, all the actions that appear after that action in an actionlist should be removed from that actionlist, see section 6.4. This has to be done for all the actions that constitute the last system response.

To perform the subtask of updating the actionlists the response generation module first charges the control board to remove the actions of the last system response from all the actionlists that already exist. The system responses are stored on the control board so the control board knows what actions should be removed. Then the response generation module asks the control board for all the atomic goals on the goalstack without an actionlist. These are goals that have just been placed on the goalstack by the dialogue updating module. The action rules are used to generate an actionlist for each of these goals. These actionlists are then placed on the control board, linked to their atomic goals.

Actual response generation

The system response consists of one or more communicative actions from the actionlists of the atomic goals on the goalstack. The system response can be targeted to the achievement of more than one goal. The system response could contain an action that is part of more than one actionlist. Some actions can be used to help the achievement of more than one goal and so occur in more than one actionlist. These are usually general communicative actions that just tempt the user to give more information. Besides that, the system response can contain more than one action from various actionlists. The system response will help the achievement of all the goals which have one of the actions of the system response in their actionlists.

The communicative actions at the end of the actionlists are the ones eligible for the system response (see section 6.4). So in order to generate the system response, the response generation module must request all the actions that are at the end of the actionlists, from the control board. The control board returns all those actions together with the priority of the goals they belong to, i.e. a list of response candidates. The goals will have a priority that is the same as the level they are on in the goalstack. The same action can occur at the end of more than one actionlist. Such an action will be returned as many times as it occurs at the end of an actionlist, each time with the priority of the goal the actionlist belongs to.

The goals placed higher on the goalstack have a higher priority, the dialogue manager should try to achieve them first. The goals on top of the goalstack have the highest priority. So at least one of the actions belonging to a goal on the top of the goalstack should be selected for the system response[13]. Of all the actions

[13]There are always atomic goals on top of the goalstack, the dialogue updating module ensures that. The response generation module ensures there is always an actionlist linked to the atomic goals.

with the highest priority, the action which occurs the most times among the candidates with the highest priority, will be selected as the basis action for the system response.

Then the response generation module will look for another candidate that can be combined with the basis action. The current version of the Alparon system will at the most put two communicative actions together in one system response. It is not useful to put too many different communicative actions in one system response. The user could get confused. Two actions at the most is perhaps a little conservative, but will do for the current version of the Alparon system.

The combination rules define which actions can be combined with other actions. A combination rule begins with an action and is followed by a list of other actions it can be combined with. The syntax is as follows:

basisaction: [*action1*, ... , *actionn*]

The actions in the combination rules can be generic, when they contain an asterisk in parentheses, `(*)`. Again the asterisk will match everything for both the basis action as the actions it can be combined with. The combination rules will be used to look for another action between the candidates with which the basis action can be combined. When there is more than one such action, the action with the highest priority is chosen to be combined with the basis action. If there is no suitable candidate available, the system response consists of only the basis action.

Finally the response generation module places the system response on the control board. There the dialogue act generation module can collect the response and transform it to dialogue acts.

Order of execution

The main task of the response generation module is to generate a response. In order to do so, the 'updating actionlists' subtask must be executed first. Then the actual response generation should take place.

7.3.3 Interface

Like the discussion of the dialogue updating module, this subsection gives an overview of the messages the response generation module can handle. The messages the response generation module sends and the messages it expects as replies, are given at the end of this subsection. Only the message for the actual tasks of the module will be described, not the messages for testing and development.

The only task the response generation module is supposed to perform, is the generation of a system response. The response generation module can be asked to perform this task by sending it the `turn` message:

Message name: `turn`
Message body: none
Reply message name: `turn`
Reply message body: none

This time the message should not have any arguments. All the information the module needs can be found on the blackboard system, notably the control board. The selected interpretation of the user's turn is not necessary for this module as all the useful information from it has been extracted by earlier modules.

Table 7.3 gives an overview of the messages the response generation module may possibly send to other modules when it is doing its work.

Table 7.3: Messages sent to other modules, with replies indicated

Module	Message name	Reply name
control board (CB)	putActionList	
	putResponse	
	removeResponse	
	getAtomicWithoutActionList	AtomicWithoutActionList
	getCandidates	Candidates

7.3.4 Implementation

This module is also implemented in Java with the Alparon library for messaging modules, just like the dialogue updating module. The source code can be found in appendix D. The source code is supplied with javadoc comments. Documentation can be generated using the javadoc tool.

The names of the files that contain the action and the combination rules are hard encoded in the response generation module. The filenames used are show in table 7.4 and should be in the same directory as in which the response generation module is started.

Table 7.4: Filenames of the rulesets

Ruleset	Filename
action rules	action.rules
combination rules	combination.rules

7.4 Control board

This section describes the last component for dialogue control, the control board. On this board all information related to dialogue control will be stored, i.e. information regarding the dialogue that persists between turns and that is used for dialogue control issues. First, the task of the control board will be described. Then the design issues are discussed. An overview of the messages this blackboard can handle, follows the design issues and finally some comments regarding the implementation are given.

7.4.1 Tasks

This is not a processing module. The control board is a blackboard. It is used to store information and to intelligently handle the stored information. All the information for the dialogue control is kept on this blackboard. There are five kinds of information used for dialogue control. To keep track of these kinds of information, are the five tasks of the control board. Some of these information structures are linked to each other while others are separated from the rest. The latter structures could be placed on other separate blackboards, but are placed for now with everything else concerning dialogue control on this one board.

The control board has to update the information structures placed on it as well as to allow the modules certain inspections of the structures, e.g. adding goals and answering whether the goalstack is empty.

Conflicts

During the context updating by the context updating module, conflicts could arise. When the context updating module wants to place a new value in a slot of the information board that does not agree with an existing value in that slot, a conflict arises. The context updating places the new value in the slot, but also reports this to the control board. The control board has to store this information, so one of the dialogue control modules can deal with such an situation. This is the first task of the control board.

Choices

Another situation that could occur during context updating, is a situation with unresolved choices. The disambiguation module could not prefer one interpretation of the user's turn over some others, while they only differ in some domain values. The moves are the same in the dialogue acts of the interpretations. The dialogue acts only differ for some argument or subargument. See section 4.2.1, page 17 for a more detailed description of this situation. The context updating module does not place one of the domain values - between which a choice should still be made - in a slot, but reports an unresolved choice to the control board. To deal with such a situation is a second task of the control board.

Goalstack

The central concept in dialogue control is the goalstack. The goalstack describes what still has to be done during the dialogue. This structure is very dynamic. New goals are placed on the goalstack and existing goals are removed every turn. The control board manages the goalstack. Other information structures are linked to this goalstack: the agenda and the system response. See section 6.2 for a detailed overview of the goalstack.

Agenda

The agenda consists of all the actionlists of the atomic goals. These actionlists are linked to specific atomic goals. The control board also manages the actionlists. Several functions must be provided for actionlists: they are added to new atomic goals, they are modified, they are inspected, etc.

Response

The system response is stored on the control board. The dialogue act generation module can retrieve the response from the control board. The system response consists of actions that are also present in the actionlists. When a response is passed to the user, the actions it consists of can be removed from the actionlists. The last task of the control board consists of managing all the aspects of the system response.

7.4.2 Design issues

Here the design issues for the tasks of the control board are discussed is some detail.

Conflicts

The conflict is specified by the slotname for which there is a conflict. To store the existence of a conflict, the control board only has to store the name of that slot. The conflicting values can be looked up on the information board. The control board is just used to signal a conflict exists. When the control board receives the `putConflict` message the slotname for which there is a conflict, is specified in the body of the message. This slotname is then stored on the control board. More than one conflict is allowed to be reported to the control board. All the slotnames will be stored.

When the control board receives the `getConflicts` message, the control board replies with a `Conflicts` message putting all the slotnames for which there are conflicts in the message body. The conflicts will immediately thereafter be removed from the control board.

Choices

A choice is specified by the slotname and several possible values for that slot. The slotname and the possible values for that slot are stored on the control board. The control board is not only used to just signal the existence of a choice that still has to be made, but the choice is fully specified on the control board. More than one choice is allowed to be put on the control board. When the control board receives the `putChoice` message, the slotname and the possible values for that slot are specified in the body of that message.

The control board can be asked for all the choices that still have to be made by sending it the `getChoices` message. The control board will reply with a `Choices` message with all the choices specified in the message body. The choices will be removed from the control board as soon as the choices have been requested by some module.

Goalstack

The goalstack is a more elaborated structure than the previous structures to store conflicts and choices. How the goalstack should be organized is discussed in section 6.2. The storage of the agenda and response are linked to this structure but are described in separate subsections.

As stated in section 6.1 on goals, several kinds of goals are distinguished. The goals can be divided in complex goals and atomic goals. The atomic goals can be further divided in state goals or informative goals. Especially the distinction between complex and atomic goals is of essential importance for dialogue control in the Alparon system. They perform a different function in the framework for dialogue control. The distinction between state and informative goals is not so important for the framework. Whether an atomic goal is a state goal or an informative goal, can be defined by the achievement rules.

Goals are identified by their name. To distinguish complex goals from atomic goals, all complex goals should begin with a capital `C` and all atomic goals with a capital `A`. The control board can then easily determine whether a particular goal is an atomic or a complex goal.

The goalstack consists of levels. The levels are numbered starting with number 0. New levels are placed on existing levels and have a number 1 higher than the level directly below them. When levels are removed the numbering is automaticly adjusted. The numbering of the levels will be used as a priority indication for the goals on the levels. Every level consist of at least one goal, both complex goals and atomic goals are allowed on the same level. The goals are not ordered within a level, they are all ranked equally.

A new level will be added to the goalstack when the control board receives the `addLevel` message. The body of that message must contain the goals that should be put on the new level. This new level will be put on top of the existing levels[14].

Goals are removed from the goalstack by sending the `removeGoal` message. The body should contain one goal. This goal is then removed from the goalstack. When this goal is the last goal on a level, that level is also removed from the goalstack and the level numbering is adjusted.

Another way to remove goals is by popping the complex goals from the top of the goalstack. When there are no atomic goals on top of the goalstack, the dialogue updating module will decompose the complex goals on the top level. When the control board receives the `popComplexGoals` message the complex goals are removed from the top level and returned in the body of the reply message `ComplexGoals`.

These are the only ways to manipulate the goalstack. But next to manipulation, it must be possible to inspect the goalstack for certain aspects. This is necessary for certain dialogue control tasks.

Modules must be able to determine whether there is an atomic goal on top of the goalstack. The message `isAtomicOnTop`, sent to the control board, provides the answer in a `AtomicOnTop` message. To check whether the goalstack is empty the message `isEmpty` can be used (the reply message is `Empty`). The message `isOnlyComplexOnTop` with reply message `OnlyComplexOnTop`, tells whether there exists only complex goals on top of the stack.

It should also be possible to check whether a certain goal does exist on the goalstack. The `existGoal` message requests this check from the control board. A goal should be specified. This goal is allowed to be in generic form, it then

[14] Theoretically, it is allowed to place the same goal more than once on the goalstack. It is, however, not yet clear why one would want to do that.

contains an asterisk between parentheses, `('*')`. When the specified goal exists or when there are goals on the goalstack that are instantiation of the generic goal (with the asterisk replaced by anything), the control board sends a reply message `Goal` with the first goal that matches the specified goal in the body.

The message `getAtomicGoals` will ask the control board to return all the atomic goals on the goalstack. The atomic goals are placed in the body of the reply message `AtomicGoals`. They are however not removed from the goalstack as is the case when popping the complex goals off the top. All the atomic goals on all the levels are returned without any indication of priority.

Agenda

A list with communicative actions can be attached to each atomic goal. This actionlist is used for response generation. An actionlist for an atomic goal will be generated once and linked to the atomic goal. The control board manages the actionlists and links them to atomic goals on the goalstack. Actionlists cannot be removed. They are only removed automaticly when the goals they are linked to are removed.

When the control board receives a `putActionList`, it places the actions specified in the body on the control board. This actionlist will be linked to the goal that is also specified in the message body. The actions in the actionlist are ordered. They are placed in the actionlist in the same order as they appear in the message body.

It must be possible to get all the atomic goals from the goalstack that do not have an actionlist attached to them. The `getAtomicWithoutActionList` message can be used to ask the control board for all these goals. The control board will reply with the `AtomicWithoutActionList` message with those goals in the body.

Another action that must be able to be performed on the actionlists is to collect all the actions that are at the end of the actionlists. This is used by the response generation module. Not only the actions should be returned, but the priority for each action as well. This priority is the same as the priority of the atomic goals they are attached to (which is the same as the level number the goal is in). The control board will reply the `getCandidates` message with a `Candidates` message with these actions and their priorities.

Response

The system response of the dialogue manager is also stored on the control board. It consists of one or two communicative actions. These actions appear at the end of some of the actionlists. The control board just has to store the response, no manipulation has to be performed on it. Only one response can be stored on the control board.

To put a response on the control board, one sends a `putResponse` message to the control board with the communicative actions in the body. To collect the response, one sends the `getResponse` message. The control board will then reply with a `Response` message with the actions in the body.

The removal of a response is a little bit more complicated. A response should be removed because it is passed to the user. The actions that were part of the response must then be removed from the actionlists as well. So when the control board receives the `removeResponse` message the response is removed from the control board and the actions in the response are removed from the actionlists. When such actions occurs more than once in an actionlist only the last occurrence is removed. When there are other actions placed after the action that is to be removed, all those actions will also be removed from that particular actionlist.

Finally, it should be possible to check whether an action belonging to the actionlist of a certain goal is part of the current system response. Two messages can be used to request this information from the control board. The first is the `isCommunicated` message. The body of that message should specify a goal. The control board sends a reply message `Communicated` back with the answer in its body. The message `inResponse` requests the same information, but the goal specified in the body can be generic (i.e. with `('*')`). The reply message will now be a `Goal` message with in the body the first encountered goal that matches the specified, possibly generic, goal and has an action in its actionlist that is part of the current response.

7.4.3 Interface

This subsection gives an overview of the messages the control board can handle. The exact meanings of these messages are mostly explained in the previous subsection about design issues. When there are multiple arguments to a message, each argument should be put on a new line. The only messages the control board sends are the reply messages.

Message name: `clear`
Message body: none
Reply message name: none

Message name: `putConflict`
Message body: slotname
Reply message name: none

Message name: `getConflicts`
Message body: none
Reply message name: `Conflicts`
Reply message body: slotnames

Message name: `putChoice`
Message body: slotname and values
Reply message name: none

Message name: `getChoices`
Message body: none
Reply message name: `Choices`
Reply message body: slotnames and values

Message name: `addLevel`
Message body: goals
Reply message name: none

Message name: `removeGoal`
Message body: goal
Reply message name: none

Message name: `popComplexGoals`
Message body: none
Reply message name: `ComplexGoals`
Reply message body: complex goals

Message name: `getAtomicGoals`
Message body: none
Reply message name: `AtomicGoals`
Reply message body: atomic goals

Message name: `isAtomicOnTop`
Message body: none
Reply message name: `AtomicOnTop`
Reply message body: `true` or `false`

Message name: `isEmpty`
Message body: none
Reply message name: `Empty`
Reply message body: `true` or `false`

Message name: `isOnlyComplexOnTop`
Message body: none
Reply message name: `OnlyComplexOnTop`
Reply message body: `true` or `false`

Message name: `existGoal`
Message body: goal
Reply message name: `Goal`
Reply message body: goal

Message name: `putActionList`
Message body: atomic goal and actions
Reply message name: none

Message name: `getAtomicWithoutActionList`
Message body: none
Reply message name: `AtomicWithoutActionList`
Reply message body: atomic goals

Message name: `getCandidates`
Message body: none
Reply message name: `Candidates`
Reply message body: actions

Message name: `putResponse`

Message body: actions
Reply message name: none

Message name: `getResponse`
Message body: none
Reply message name: `Response`
Reply message body: actions

Message name: `removeResponse`
Message body: none
Reply message name: none

Message name: `isCommunicated`
Message body: goal
Reply message name: `Communicated`
Reply message body: `true` or `false`

Message name: `inResponse`
Message body: goal
Reply message name: `Goal`
Reply message body: goal

The `clear` message can be used to wipe out the entire blackboard. All conflicts and choices, the goalstack, agenda and system response are removed from the control board. This should only be used at the beginning of a dialogue.

Each specification of a choice, in the body of the `Choice` message, begins with the slotname of the slot followed by the number of possible values on the next line. Then the possible values are specified on successive lines. The various choices are placed one after the other in the message body.

7.4.4 Implementation

This blackboard is also implemented in Java using the Alparon library for messaging modules, just like the two modules for dialogue control. The documentation can be generated by the javadoc tool.

The data structures are manipulated on the board, but no log or history information is recorded. To reset the control board to a previous state, these features should be implemented or the message log of the portmapper can be used to reconstruct a previous state from scratch.

The source code of this blackboard can be found in appendix E.

Chapter 8

Train timetable information system

The last chapters described the design of a framework for dialogue control for the Alparon system. However, designing dialogue control modules does not guarantee their suitability as a framework for automatic speech processing systems for information retrieval. The framework itself is not a complete subsystem for dialogue control.

This chapter provides a validation for the use of this framework in dialogue control. In order to get a working subsystem the various rulesets must be made and used by the framework. The performance of the dialogue control strongly depends on the quality of the rulesets. The framework leaves a lot of space for these rulesets. It does not force any restrictions on the rulesets. When the framework is used with inadequate rulesets, the dialogue control will fail.

This chapter describes the development and testing of dialogue control for a train timetable information system. The system will be small and simple. It is not the aim to make a complete system for train timetable information. The system should only demonstrate that with the correct rulesets, adequate dialogue control can be achieved by the framework. Various phenomena and features should be handled correctly by the dialogue control modules.

First, the setup of the test environment will be discussed. Next, the train timetable information system will be described. Then, a demonstration will be given of how the dialogue control handles certain phenomena and features.

8.1 Setup

The Alparon system in not yet fully developed. Besides the components for dialogue control, not all the other components for dialogue management and the components for the total system, are ready to be used. So it is not possible yet to run tests with a complete system. This chapter will describe testing of the dialogue control components only.

Fortunately there is already an implementation present for the central part of the dialogue manager, the portmapper. The components for dialogue control - the dialogue updating module, the response generation module and the control board - will be started together with the portmapper as the test system for dialogue control. Two other components have been developed in order to simulate the rest of the dialogue management system.

The information on the information board is essential for dialogue control. The value and attributes of slots are important indicators for the progression of the dialogue. Especially for the achievement of goals, this kind of information is important. So in order to test dialogue control, an information board or a simulation for an information board must be present. A simple information board is developed with only the functionalities necessary for dialogue control. Appendix A gives an overview of the implementation of this blackboard.

To simulate the rest of the dialogue manager, a messaging module is developed which can send and receive arbitrary messages. Every message it receives, is displayed on standard output. To send an arbitrary message, the message one wants to send, must be given through standard input and the message is then sent. This module, called the send and receive module, is described in appendix B.

This last module is used to manipulate the blackboards (especially the information board) as the rest of the dialogue manager, the modules besides the modules for dialogue control, would do in a complete dialogue system. It is also used to invoke the modules for dialogue control to process a user's turn and generate a system response. In a complete dialogue management system the central portmapper will invoke the necessary modules to process the dialogue turns, one after the other. This feature of the portmapper is not used during the testing of the dialogue control components, instead the send and receive module is used to invoke the dialogue control modules with the right arguments as the portmapper would have done.

The user's turn must be given as an argument when the dialogue updating module is invoked to process the user's turn. A simplified form of dialogue acts is used during the testing. As the dialogue control components not really do any processing with the dialogue acts themselves, this should not be a problem. They only have to match the dialogue acts to conditions of reaction rules. So in order to be able to use the real dialogue acts at a later stage, the dialogue acts in the conditions of the reaction rules must be defined like the real dialogue acts. For simplicity and illustrativeness a simplified form of the dialogue acts is used here to represent the user's turn.

8.2 Train timetable information system

The application used to test the dialogue control components of the Alparon system is a small train timetable information system. The user should be able to consult this system in order to get a travel plan for a train connection. The system should be able to engage in a mixed-initiative dialogue with the user for this purpose. The user interacts with the system using speech by telephone[1].

In order for the system to look up a travel plan, the system needs to know the departure and arrival stations, an approximate arrival or departure time and a date. After a greeting phase, the exact informational needs of the user must be determined (the stations, time and date). Then the information must

[1]This is not a part of the test system, but this feature can have consequences for the dialogue control. E.g. misunderstanding and errors in the interpretation of the user's turn, must be dealt with.

be collected from databases and presented to the user. The presentation of the travelplan should be split up in manageable pieces and presented in several turns. Then the user must be asked whether he wants another travelplan. If the user wants another travelplan, the dialogue continues with a new determination of the user's informational needs. When the user eventually does not want more travelplans, the dialogue should be closed in a polite fashion.

The user should be allowed the initiative in the first part of the dialogue, i.e. during the period the systems tries to determine the actual informational needs. The user should get a change to specify his informational needs in his own words and order. Only when the user does not supply the necessary information, the system is allowed to take the initiative and asks the user directly for the needed information. In the presentation phase, the system may take the initiative and presents the parts of the travelplan in several turns. The user can only interrupt the presentation to request repetitions, etc.

8.3 Dialogue control

This section will describe the actual testing of the dialogue control mechanism of the Alparon system.

An information retrieval dialogue can be divided into several phases. [Rothkrantz et al., 1996] gives a analysis of over 5.000 human-to-human dialogues on public transport information. Several main phases are distinguished in such dialogues. Dialogues start with a greeting phase. Then a phase follows during which the informational needs are determined[2], the user usually has the initiative in this phase. After this phase, the requested information is looked up. Next, the information will be presented to the user[3], giving the system the initiative in most cases. The dialogues are ended with a goodbye phase. The phase in which the informational needs are determined and the phase in which the information is presented, can be interrupted by another phase in which the one who does not have the initiative, takes over the initiative: a subquery.

In the system of this chapter five phases will be recognized. The greeting phase, a phase in which the informational needs of the user are determined (also called inform phase in this chapter), a presentation phase, a phase in which the user is asked whether he wants another travelplan and a goodbye phase in which the dialogue is closed. The subquery phase is not a separate phase, but will be part of a main phase and not identified as such. As it is, none of the phases will be identified as such. The phases found in human-human dialogues are actually only used to define a structure for the dialogue. The system will not try to determine in what phase the dialogue is in at a given moment. But the dialogue is structured according to these phases (as will become apparent in the rest of this chapter).

The phases in a dialogue are not always neatly delimited by turns in the dialogue. For example the first turn of the user will often be part of the greeting phase and the inform phase. In one turn the user greets and introduces himself and begins specifying his informational needs. The Alparon system will behave more strictly, every turn of the system will belong to a certain phase.

[2]called query phase in [Rothkrantz et al., 1996]
[3]called the information phase in [Rothkrantz et al., 1996]

The discussion of the test system will start with the individual phases. Per phase the rules for goals and actions will be discussed and the working of the dialogue control demonstrated. At the end some issues concerning the whole dialogue are discussed.

8.3.1 Greeting

Greeting takes place at the beginning of the dialogue. The system answers the telephone call of the user. The system should make the first turn in the dialogue. It will suffice to greet the user and introduce the system in this first turn. By not doing anything else in this turn, the initiative is placed with the user. Usually the user will return the greeting, often introduce himself and start to specify his informational needs. From the viewpoint of the system this phase will only last one turn.

The complex goal `Cgreet` in the test system, is used to indicate that greeting has to take place. Since the system should only communicate some information to the user - in only one turn - during the greeting, this complex goal can be decomposed to one informative goal. The following decomposition rule defines this behaviour[4]

```
G1: Cgreet: [[Agreet]]
```

The numbering, G1:, is not part of the rule. It is used in this chapter to refer to specific rules.

Informative goals are achieved when an action in the actionlist attached to this goal is used in the system response. The following achievement rule defines that:

```
G2: Agreet:
    communicated
```

The action that can be used to achieve the greeting, is defined as an atomic action `Greetings`. The dialogue act generation module should be able to transform this action to the appropriate dialogue acts. This is however not part of the dialogue control. Dialogue control ends when this action is placed in the system response. This is the action rule for the goal `Agreet`:

```
G3: Agreet: [Greetings]
```

This action should not appear with other communicative actions in the system response, so no combination rules for this action are defined. There are no situations in this phase of the dialogue, when the system must react to the user's turn. This phase will only last the first turn of the system, before the user has made a turn. So there are also no reaction rules defined for this phase[5].

With these three rules the greeting of the system is defined and works as intended. When the `Cgreet` is placed on the goalstack and the dialogue

[4] Note that complex goals always start with a capital C and atomic goals with a capital A.

[5] Reaction rules are not defined and active for only some phase (the system does not even determine the phase). They are active during the whole dialogue. They can however be defined with a situation in mind that usually occurs in a certain phase of the dialogue. Therefore the reaction rules will be discussed in the section that discusses the phase where a reaction rule might apply.

updating and response generation modules have been invoked the system response is `Greetings`.

This is a very simple phase in the dialogue and there is not much to the rules that have to be defined to get the desired behaviour. The next subsection however will deal with the inform phase, the phase in which the user's informational needs must be determined, and is much more complex.

8.3.2 Determination of the user's informational needs

The first part of the dialogue will be concerned with the determination of the user's informational needs. After the greeting phase, information must be collected to determine where and when the user wants to travel. The complex goal `CdetermineNeed` will be used to indicate this situation.

Goals

In order to determine the informational needs, the system must determine a departure and arrival station, a departure or arrival time and a date. So the decomposition rule is as follows:

```
I1: CdetermineNeed: [[Adetermine(Date), Adetermine(Time)],
    [Adetermine(DepPlace), Adetermine(ArrPlace)]]
```

The informational needs of the user's are specified on the information board in slots. The new goals are all atomic state goals of the form `Adetermine(*)`, where the asterisk is replaced by a slotname of the information board. They are all atomic goals that will appear at the same time. However, the goals to determine the departure and arrival stations are placed one level higher on the goalstack then the goal for the determination of the travel date and time. As a result the system will first try to determine the departure and arrival station before the time and date. The goals for time and date are also atomic goals and therefore active as soon as they appear on the goalstack. So their actions could also be part of the system response.

Achievement

These goals can be considered achieved when the slots on the information board contain the right information. The system knows on what date the user wants to travel if the slot `Date` has a value, i.e. it is not empty. The context updating should ensure only meaningful values are placed in this slot. The time is known if the slot `Time` is filled with a value indicating a certain departure or arrival time. For the slots `DepPlace` and `ArrPlace` there are more restrictions. The slots should be filled with a value indicating a station, as a value indicating a city will not be specific enough, those values should be mentioned by the user or be verified or the system should be fairly sure about the value (indicated by ground `system`). So it will not suffice when the system places a default value in those slots, for example. These definitions are stated in these achievement rules:

```
I2: Adetermine(Date):
    Date hasnot empty
```

```
I3: Adetermine(Time):
    Time is ArrTime(*)
    Time is DepTime(*)
I4: Adetermine(DepPlace):
    DepPlace is station(*) & DepPlace has user
    DepPlace is station(*) & DepPlace has verified
    DepPlace is station(*) & DepPlace has system
I5: Adetermine(ArrPlace):
    ArrPlace is station(*) & ArrPlace has user
    ArrPlace is station(*) & ArrPlace has verified
    ArrPlace is station(*) & ArrPlace has system
```

These goals are achieved when the dialogue is in a certain state. The state changes when the user's turns are processed, i.e. new information is placed on the information board.

Operations

Another way for the state of the dialogue to change is by execution of operations. The dialogue updating module can execute a certain operation when the specified conditions for that operation are met. These operations can be used to define deductions the system is allowed to make. E.g. when the slot `DepPlace` contains a value `city(Leiden)`, the value `station(Leiden)` can be placed in that slot, because the user usually means station Leiden when he speaks of just Leiden (there is another station in Leiden, namely station Leiden Lammeschans). For good measure a goal to verify the slot can be added to the goalstack.

Operations where a value of the form `city(*)` will be replaced by a value `station(*)` are used in the test system. If the city has only one station, that station is placed in the slot. When the city has more than one station, but usually the main station is meant when people just speak of the city, the main station is put in the slot and a goal for verification is added. When the city has more than one major station, a goal will be added to the goalstack to choose between the stations in that city.[6] This results in the following operation rules:

```
I6: Adetermine(DepPlace):
    DepPlace is city(Delft) : put(DepPlace, station(Delft),
    toverify), goal(Averify(DepPlace))
    DepPlace is city(Leiden) : put(DepPlace, station(Leiden),
    toverify), goal(Averify(DepPlace))
    DepPlace is city(Rotterdam) : goal(Achoice(DepPlace,
    station(Rotterdam cs), station(Rotterdam Hofplein),
    station(Rotterdam Blaak)))
    DepPlace is city(Den Haag) : goal(Achoice(DepPlace,
    station(Den Haag hs), station(Den Haag cs)))
    DepPlace is city(Baarn) : put(DepPlace, station(Baarn), system)
    DepPlace is city(Maarn) : put(DepPlace, station(Maarn), system)
I7: Adetermine(ArrPlace):
    ArrPlace is city(Delft) : put(ArrPlace, station(Delft),
```

[6] This is domain knowledge the designer of the rules must collect and apply.

```
    toverify), goal(Averify(ArrPlace))
    ArrPlace is city(Leiden) : put(ArrPlace, station(Leiden),
    toverify), goal(Averify(ArrPlace))
    ArrPlace is city(Rotterdam) : goal(Achoice(ArrPlace,
    station(Rotterdam cs), station(Rotterdam Hofplein),
    station(Rotterdam Blaak)))
    ArrPlace is city(Den Haag) : goal(Achoice(ArrPlace,
    station(Den Haag hs), station(Den Haag cs)))
    ArrPlace is city(Baarn) : put(ArrPlace, station(Baarn), system)
    ArrPlace is city(Maarn) : put(ArrPlace, station(Maarn), system)
I8: Adetermine(Time):
    Time is Time(*) : goal(Achoice(Time, DepTime(*), ArrTime(*)))
```

The last operation rule for the `Adetermine(Time)` goal is quite similar. Another use of operation rules can be seen in this operation rule:

```
I9: Adetermine(Date):
    Date has empty : put(Date, today, default)
```

This operation rule puts a default value in the `Date` slot on the information board.

Actions

The dialogue updating module can only modify the state of the dialogue in certain specific situations which matches the conditions of an operation. Often the system must tempt the user to give the necessary information to change the state. For the state goals, action rules must be defined containing actions that tempt the user in the right way. The initiative however should initially be with the user in this phase of the dialogue. So the actions must be passive[7]. Only when those actions fail to retrieve the right information from the user, more active[8] actions should be used. These actions can first be targeted to multiple goals, but should eventually only be applicable for the goal they belong to. In this manner the initiative is initially placed with the user. When the user fails to give the right information, the system will take over the initiative and ask more specific questions each time. This is the mixed-initiative behaviour the system should have.

These action rules are defined for the state goals:

```
I10: Adetermine(Date): [Ask(Date), Ask(When)]
I11: Adetermine(Time): [Ask(DepTime), Ask(Time), Ask(When)]
I12: Adetermine(DepPlace): [Ask(DepPlace), Ask(Place), Encourage]
I13: Adetermine(ArrPlace): [Ask(ArrPlace), Ask(Place), Encourage]
```

The operations can result in the appearance of `Averify` and `Achoice` goals. Interaction with the user is necessary for these goals. Therefore action rules have to be defined for these goals as well.

[7] 'Passive' is used here to indicate the action should not take over the initiative, the system just utters a general phrase and waits.

[8] 'Active' is used here to indicate the system should take over the initiative and ask (direct) questions.

```
I14: Averify(*): [VerifyExplicit(*), VerifyImplicit(*)]
I15: Achoice(*): [Choice(*)]
```

These rules are generic rules for all `Averify` and `Achoice` goals. The dialogue act generation module should make the appropriate dialogue acts for these actions (the actions are instantiated then).

Achievement

For the `Averify` and `Achoice` goals achievement rules still have to be defined. These goals can appear as a result of some operations.

```
I16: Averify(*):
     * has verified
I17: Achoice(*):
     communicated
```

The `Averify(*)` goal is achieved when the slot, an instantiation of the asterisk, is verified. It will have the attribute `verified` then. The `Achoice(*)` goal is achieved when the system has communicated with the user about this goal.

Combination

An implicit verification is only implicit when it is used together with another communicative action. Therefore the following combination rule is defined:

```
I18: VerifyImplicit(*): [Ask(*)]
```

When a `VerifyImplicit(*)` action can not be combined with another action, it actually becomes an explicit verification. The dialogue act generation module takes care of that.

Reaction

The initiative in this phase is usually with the user. There are no situations defined yet where the system must take some action according to the user's turn. So no specific reaction rules are defined at this stage.

Example dialogues

Table 8.1 gives an example of a subdialogue in this phase. The goalstack has the goal `CdetermineNeed` on top of the goalstack (CB) when the user turn U1 ("*I want to know from Baarn station to Den Haag.*") is received by the dialogue manager. The processing of the disambiguation and context updating module is simulated by placing the values `station(Baarn)` and `city(Den Haag)` in the slots `DepPlace` and `ArrPlace` on the information board (IB). Then the dialogue updating module is invoked, indicated by the three dots. There are no atomic goals on top, so no achievement and operations are checked. However, the dialogue updating module must put the goalstack in proper form. This is done by the complex goal `CdetermineNeed` is decomposed according to the decomposition rule I1. For the new atomic goals, achievement and operations

Table 8.1: Example subdialogue 1

CB:	`[CdetermineNeed]`
U1:	*I want to know from Baarn station to Den Haag.*
IB:	DepPlace: `station(Baarn)`
	ArrPlace: `city(Den Haag)`
...	
IB:	Date: `today`
CB:	`[Adetermine(Time)]`
	`[Adetermine(ArrPlace)]`
	`[Achoice(ArrPlace, Den Haag cs, Den Haag hs)]`
	Response: `[Choice(ArrPlace, Den Haag cs, Den Haag hs)]`
S1:	*Which station in Den Haag do you want to go to, central station or Holland Spoor?*
U2:	*I want to go to Den Haag central station.*
IB:	ArrPlace: `city(Den Haag), station(Den Haag cs)`
	...
CB:	`[Adetermine(Time)]`
	Response: [Ask(When)]
S2:	*When do you want to travel?*
U3:	*And I want to arrive at half past nine.*
IB:	Time: `ArrTime(9:30)`
...	
CB:	empty
	...

are checked. For the `Adetermine(Date)` goal the default value is placed in slot `Date` (I9) and the goal is achieved (I2). The goal `Adetermine(DepPlace)` is achieved (I4) and for the goal `Adetermine(ArrPlace)` an operation is executed placing a new goal on the goalstack, `[Achoice(ArrPlace, Den Haag cs, Den Haag hs)]` (I7). After that the response generation module is invoked. It places actionlists at the new atomic goal according to the action rules above (I11, I13, I15). These actionlists are not shown in the table. Finally it generates a response, `[Choice(ArrPlace, Den Haag cs, Den Haag hs)]`. The action belonging to the `Achoice(ArrPlace, Den Haag cs, Den Haag hs)` goal has the highest priority and can not be combined with another action. So the system response is S1 ("*Which station in Den Haag do you want to go to, central station or Holland Spoor?*").

The user's next turn is U2 ("*I want to go to Den Haag central station.*"). This is simulated by placing `station(Den Haag)` in the `ArrPlace` slot. The information board for that slot is shown at IB. When the dialogue updating module processes this turn, the `Achoice` and `Adetermine(ArrPlace)` goals are considered achieved (I17, I5) and removed from the goalstack (with their actionlists). For the last goal, `Adetermine(Time)`, the dialogue updating module cannot do anything. The response generation picks the last action of the actionlist of the `Adetermine(Time)` goal for the response. This should be transformed by the rest of the dialogue management subsystem to something like S2 ("*When do you want to travel?*").

The user answers with U3 ("*And I want to arrive at half past nine.*"). To simulate this the value `ArrTime(9:30)` is placed in the `Time` slot. When the dialogue updating module is invoked the last atomic goal of the subdialogue is achieved (I3) and removed from the goalstack. The dialogue manager now has all the necessary information to pose a meaningful query to the databases. The inform phase is over.

Table 8.2: Example subdialogue 2

CB:	`[CdetermineNeed]`
U1:	*I want to know from Baarn/Maarn to Leiden.*
IB:	ArrPlace: `city(Leiden)`
CB:	Choice: (DepPlace, city(Baarn), city(Maarn))
...	
CB:	`[CdetermineNeed]`
	`[Achoice(DepPlace, city(Baarn), city(Maarn))]`
	Response: `[Choice(DepPlace, city(Baarn), city(Maarn))]`
S1:	*Do you want to travel from Baarn or Maarn?*
U2:	*I want to travel from Baarn.*
IB:	DepPlace: `city(Baarn)`
	...
IB:	Date: `today`
	DepPlace: `city(Baarn), station(Baarn)`
	ArrPlace: `city(Leiden), station(Leiden)`
CB:	`[Adetermine(Time)]`
	`[Adetermine(ArrPlace)]`
	`[Averify(ArrPlace)]`
	Response: [VerifyImplicit(ArrPlace), Ask(When)]
S2:	*At what time do you want to travel to Leiden station?*
U3:	*And I want to depart at half past nine.*
IB:	Time: `DepTime(9:30)`
	ArrPlace: `city(Leiden), station(Leiden)(verified)`
...	
CB:	empty
	...

Another example subdialogue can be found in table 8.2. In this example the first utterance of the user (U1: "*I want to know from Baarn/Maarn to Leiden.*") is not completely understood. The user has either said Baarn or Maarn. To deal with this situation, a goal to choose between these values is placed on the goalstack (`Achoice(DepPlace, city(Baarn), city(Maarn))`.

In the second turn the system replaces `city(Leiden)` with `station(Leiden)`, but places a goal for verification on the goalstack. An implicit verification is used in system turn S2 ("*At what time do you want to travel to Leiden station?*"). When the user does not object to `station(Leiden)`, the context updating module should place a `verified` attribute at slot `ArrPlace` and the goal `Adetermine(ArrPlace)` will be achieved.

The subdialogue in table 8.3 shows how several actions of the same actionlist are used to tempt the user to give the necessary information. According to action rule I12 several actions are attached to a `Adetermine(DepPlace)` goal. In S1 ("*Yes?*") the system only use an encouragement, the initiative is with the user. But the user forgets or does not know he has to specify a departure place. Then the system takes over the initiative and asks for the missing information in turn S2 ("*From where do you want to travel?*").

Table 8.3: Example subdialogue 3

CB:	`[CdetermineNeed]`
U1:	*I want to go to Den Haag cs tomorrow.*
IB:	ArrPlace: `station(Den Haag cs)`
	Date: `tomorrow`
...	
CB:	`[Adetermine(Time)]`
	`[Adetermine(DepPlace)]`
	Response: `[Encourage]`
S1:	*Yes?*
U2:	*At what time do I leave then?*
...	
CB:	`[Adetermine(Time)]`
	`[Adetermine(DepPlace)]`
	Response: [Ask(Place)]
S2:	*From where do you want to travel?*
U3:	*From Rotterdam Hofplein*
IB:	DepPlace: `station(Rotterdam Hofplein)`
...	
CB:	`[Adetermine(Time)]`
	Response: [Ask(When)]
S3:	*When do you want to travel?*
U4:	*I want to arrive at 9:30.*
IB:	Time: `ArrTime(9:30)`
...	
CB:	empty
	...

8.3.3 Query

After the user's informational needs are determined, the databases should be consulted. This is stated with the goal `Cquery`. This complex goal is decomposed to the atomic goal `Aquery` by defining the decomposition rule:

```
Q1: Cquery: [[Aquery]]
```

In order to achieve this goal the message `query` must be sent to the query module. The query module will then construct a query using the information present on the information board. This query is posed to the databases through the database management module. The result - a travelplan[9] - is then segmented

[9]The result can also be more than one travelplan. Another task of the query module is then to select one of these travelplans for presentation. The modules for dialogue control do

in pieces for presentation and placed on the presentation board. The query module replies with the message `queryDone` when a presentation is ready on the presentation board. This is defined by the operation rule

```
Q2: Aquery:
    true : send(SRMODULE, query, empty body, queryDone)
```

The condition is `true`. This mean this operation should always be executed when this goal is present.

Every goal must have a achievement rule, otherwise it can not be removed from the goalstack. The achievement rule for the `Aquery` is as follows:

```
Q3: Aquery:
    query is done
```

Actually this goal is achieved when the query is posed and the result stored on the presentation board. To check this situation, the dialogue updating module looks in slot `query` on the information board. When this slot contains the value `done` the goal is considered achieved. So in order for this to work properly the query module must place the value `done` in the `query` slot on the information board. This is not an elegant solution. It would be more logical to stored this kind of information (the 'query is done'-information) on the status board. However this board is not yet designed and for convenience this situation is solved as described above. A more elegant solution should be applied for this kind of achievement rules in the future.

8.3.4 Presentation

During the presentation phase the travelplan must be communicated to the user. This is indicated by the goal `Cpresent`. On the presentation board a travelplan has already been prepared for presentation. This travelplan is divided into manageable pieces. The number of pieces is not the same for every travelplan. Long travelplans are divided into more pieces than shorter travelplans. The division of a travelplan into pieces is the task of the query module. The modules for dialogue control just have to present the pieces one at a time. The `Cpresent` goal can be decomposed into several `ApresentPart(*)` goals, where the asterisk is replaced with the numbers 1 to the number of pieces. The dialogue updating module has to retrieve the number of pieces somewhere. Subsection 7.2.2 on page 58 states how to achieve this kind of conditional decomposition, using the decomposition rule

```
P1: Cpresent: [[ApresentPart(*)]]
```

The dialogue manager collects the value of the slot `Cpresent` from the information board[10]. Then the `Cpresent` goal is decomposed in several levels each containing a `ApresentPart(*)` goals, with the asterisk replaced by 1 to the collected value.

These `ApresentPart(*)` goals are informative goals. They are achieved when an accompanying action is used in the system response. This can be defined by the generic achievement rule:

not have to concern themselves with this.

[10] The query module should place this number in this slot. In future versions this kind of information should be placed on the status board.

```
P2: ApresentPart(*):
    communicated
```

The actions belonging to such goals can also be quite simple

```
P3: ApresentPart(*): [PresentPart(*)]
```

The dialogue act generation module has to collect the exact information for the dialogue acts from the presentation board. The pieces can be retrieved from the presentation board by their number. The pieces should, of course, not be combined in the system response. The travelplan is divided in such pieces that each piece is suitable for presentation in one turn. There are no combination rules for these actions.

During the presentation, the system has the initiative. The user can however take over the initiative for a subquery, e.g. when he wants something reconfirmed. Acknowledgments can be ignored by the system. The following reaction rule is defined to handle requests for reconfirmation

```
P4: DA Rec(*) : goal(Areconfirm(*))
```

When the user's turn contains a `Rec(*)` dialogue act, he wants * to be reconfirmed, where * is probably something said during the previous system response. The atomic goal `Areconfirm(*)` is added to the goalstack, with the asterisk replaced according to the matching of the condition (`DA Rec(*)`). This is an atomic goal and has the following action rule

```
P5: Areconfirm(*): [Reconfirm(*)]
```

The dialogue act generation module must retrieve the information the user wants reconfirmed from the previous system response or the complete travelplan on the presentation board. The dialogue control module does not collect that information, but passes the specification on while replacing the asterisk by the specification).

The `Areconfirm(*)` goals are achieved when an action belonging to a particular goal is incorporated in the system response, defined by the achievement rule:

```
P6: Areconfirm(*):
    communicated
```

Table 8.4 gives a straightforward example of this phase. The travelplan is divided into three part, signified by the value `3` in slot `Cpresent` on the information board. In the first turn, the complex goal is decomposed into three atomic goals of the form `ApresentPart(*)`. The action `PresentPart(1)` is the system response in that turn. The dialogue act generation module should collect this part from the presentation board and transform it to dialogue acts. The system's turn should be something like S1 ("*Departure at 7:53 in Baarn to Utrecht.*").

The user answers with a vague sound ("*Uhum.*") that should be recognized as a acknowledgment. This acknowledgment is ignored by the dialogue control. The `[ApresentPart(1)]` goal is achieved and the `PresentPart(2)` action is

Table 8.4: Example subdialogue 4

CB:	`[Cpresent]`
IB:	Cpresent: `3`
...	
CB:	`[ApresentPart(3)]`
	`[ApresentPart(2)]`
	`[ApresentPart(1)]`
	Response: `[PresentPart(1)]`
S1:	*Departure at 7:53 in Baarn to Utrecht.*
U2:	*Uhum.*
...	
CB:	`[ApresentPart(3)]`
	`[ApresentPart(2)]`
	Response: `[PresentPart(2)]`
S2:	*Departure at 8:32 to Den Haag cs.*
U3:	*8:32 to Den Haag cs.*
...	
CB:	`[ApresentPart(3)]`
	Response: `[PresentPart(3)]`
S3:	*Arrival at 9:13 in Den Haag.*
U4:	*Arrival at 9:13.*
...	
CB:	empty
	...

placed in the system response, resulting in S2 ("*Departure at 8:32 to Den Haag cs.*").

Again the user gives an acknowledgment (partial repetition: "*8:32 to Den Haag cs.*") and the next part of the travelplan is communicated to the user in turn 3, S3 ("*Arrival at 9:13 in Den Haag.*"). And after the user partially repeats, all the goals for presentation are achieved.

The example subdialogue in table 8.5 shows the reaction to a request for reconfirmation. In turn 2 the user asks for the departure time again, "*When do I depart, did you say?*". This is coded in the dialogue act `Rec(Time(DepTime))`. The conditions of reaction rule P4 are met and the goal `[Areconfirm(Time(DepTime))]` is added to the goalstack. A communicative action belonging to this goal is part of the system response in turn 2 (`Reconfirm(Time(DepTime))`), resulting in the system's turn "*Departure at 7:53.*" Thereafter the presentation continues with no further problems.

8.3.5 Further needs

After the system has presented the travelplan to the user, the system should ask whether the user wants another travelplan. This is indicated by the goal `CfurtherNeeds`. This goal decomposes to an atomic goal of the same name:

```
F1: CfurtherNeeds: [[AfurtherNeeds]]
```

Table 8.5: Example subdialogue 5

CB:	`[Cpresent]`

IB:
...
CB:
 	
S1:
	Cpresent: 2
	
	`[ApresentPart(2)]`
	`[ApresentPart(1)]`
	Response: `[PresentPart(1)]`
	Departure at 7:53 in Baarn to Utrecht.

U2:
 	
...
CB:
 	
S2:
	When do I depart, did you say?
	Dialogue act: `Rec(Time(DepTime))`
	
	`[ApresentPart(2)]`
	`[Areconfirm(Time(DepTime))]`
	Response: `[Reconfirm(Time(DepTime))]`
	Departure at 7:53.

U3: ... CB: S3:	*OK* `[ApresentPart(2)]` Response: `[PresentPart(2)]` *Arrival at 8:23 in Utrecht.*
U4: ... CB:	*Arrival at 8:23.* empty ...

This atomic goal states that the user should be asked whether he has further needs. This in an informative goal, i.e. it has be achieved by asking the user the question. The achievement rule is the same as for all informative goals

```
F2: AfurtherNeeds:
    communicated
```

The `AskFurtherNeeds` action belongs to this goal. The dialogue act generation module will transform this action to the right question (in dialogue act form). This is defined by

```
F3: AfurtherNeeds: [AskFurtherNeeds]
```

The `CfurtherNeeds` goal is achieved and removed when the system asks the accompanying question. But the system should take some action dependent on the user's response. When the user does not want further travelplans, he will acknowledge negatively. Nothing special should happen then. The dialogue can continue in its normal way. When the user however answer positively, the dialogue should start over beginning at the inform phase. This can be specified by reaction rules.

```
F4: Goal AfurtherNeeds in response & DA Ack(pos) : send(IB, clear,
    dummy), goal(Ctravelplan)
```

When the system asks the question, the goal `AfurtherNeeds` is incorporated in the system response. If the user answers positively (indicated by the dialogue act `Ack(pos)`) then the dialogue should go back to the inform phase. The information specifying the previous informational needs of the user should be removed. Therefore a `clear` message is sent to the information board[11].

This is a simple way to deal with this situation. E.g. when the user wants an earlier connection, he has to specify his informational needs again completely (departure and arrival station and time) because the previous specification is completely removed from the information board. How the system deals with an answer to the `AskFurtherNeeds` question, can be refined by refining the reaction rules.

Table 8.6: Example subdialogue 6

CB:	`[CfurtherNeeds]`
...	
CB:	`[AfurtherNeeds]`
	Response: `[AskFurtherNeeds]`
S1:	*Do you want another travelplan?*
U2:	*Yes, please*
	Dialogue act: `Ack(pos)`
...	
CB:	`[Ctravelplan]`
	...

In table 8.6, an example of this phase is given. In turn 2, the user answers with a positive acknowledgment. The result of reaction rule `F4` is the placement of the goal `Ctravelplan` on the goalstack, meaning the dialogue should go back the inform phase. Before a system response will be generated, this complex goal will be decomposed, etc.

8.3.6 Closing

The last part of the dialogue will be the closing. The system has to say goodbye to the user. The user will probably return the goodbye and end the connection. The goodbye phase is indicated by the `Cclose` goal. The decomposition rule is

```
C1: Cclose: [[Aclose]]
```

The `Bye` action is used for the atomic `Aclose` goal, indicated by the action rule

```
C2: Aclose: [Bye]
```

This is also an informative goal. The goal is achieved when the system response contained an action belonging to the actionlist of this goal. The usual achievement goal is defined,

[11] The message does not have any argument, just a name. So the `dummy` argument has no meaning and is only used to put the operation in the right format.

```
C3: Aclose:
    communicated
```

The `Cclose` goal should appear as the last goal on the goalstack. The last system response will be a goodbye. The user can answer that goodbye, but there are no further goals on the goalstack and the dialogue is ended after the user's turn.

8.3.7 Whole system

In the previous subsections all the phases of the train timetable information dialogue have been discussed. This subsection will deal with the whole system.

When the user makes a connection to the train timetable information system, the dialogue act `Sys(Start_Conversation)` is sent to the dialogue manager. The dialogue updating module can use this signal to initialize before the dialogue starts and generate the first turn of the dialogue. Before a new dialogue starts the state of the dialogue should be empty. The information board and the control board should be cleared. Also, the goalstack should contain some goals. So at the beginning the main goal `Ctop` is placed on the empty goalstack. This can be defined by a reaction rule:

```
W1: DA Sys(Start_Conversation) : send(CB, clear, dummy),
    send(IB, clear, dummy), goal(Ctop)
```

When the dialogue updating module receives the dialogue act `Sys(Start_Conversation)` the message `clear` is sent to the information and control board. Then the goal `Ctop` is placed on the goalstack.

The structure of the dialogue can be defined by how the main goal `Ctop` is decomposed to other goals. The main goal of the dialogue is to help the user. This is indicated by the `ChelpUser` goal, the main goal for this system. This goal can be accomplished by first greeting the user, then supplying him with travelplans and finally ending the dialogue. These actions should happen one after the other. The goals `Cgreet`, `Ctravelplan` and `Cclose` are used to define those stages.

In order to supply the user with travelplans, the system must take three steps: determine the informational needs, look up the travelplan that fulfills those needs and present the travelplan. Following this, the system should ask whether the user want another travelplan. So the complex goal `Ctravelplan` can be decomposed into the goals `CdetermineNeed`, `Cquery`, `Cpresent` and `CfurtherNeeds`. The following decomposition rules define this:

```
W2: Ctop: [[ChelpUser]]
W3: ChelpUser: [[Cclose], [Ctravelplan], [Cgreet]]
W4: Ctravelplan: [[CfurtherNeeds], [Cpresent], [Cquery],
    [CdetermineNeed]]
```

For the end of the dialogue there is another reaction rule. The dialogue manager receives a `Sys(End_Conversation)` dialogue act, when the user disconnects. The information and control board will be cleared then.

```
W5: DA Sys(End_Conversation) : send(CB, clear, dummy),
    send(IB, clear, dummy)
```

Together with the rules that have been defined in the previous subsections, these form the complete system.

In some cases a complex goals is decomposed in just one atomic goal. This has a different effect than only using the atomic goal. If only the atomic goal is used, that goal becomes 'active' from the moment it appears, i.e. achievement and operations are checked and the actions of this goal are eligible for selection for the system response. By first placing a complex goal on the goalstack, this can be prevented. Only when this goal appears on top of the goalstack, the goal is decomposed in its atomic goal and that atomic goal becomes 'active'. This last scenario is usually intended so most of the times a complex goal should be used that decomposes to one atomic goal.

How certain rules should be defined is somewhat dependent on the other modules of the dialogue manager (other than the modules for dialogue control). E.g. does the context updating module place restrictions on the values it puts in certain slot. The achievement rule for the goal Adetermine(Date) is only correct when the context updating module places only valid date values in the `Date` slot. So in order to define an adequate set of rules, good knowledge of the dialogue mechanism and the working of the other modules is necessary.

Of course, the rulesets only work as desired for a particular application when the rest of the modules are also customized for the same application. Which slots will be used? What value are allowed for certain slots? What is the meaning of the values of the slots? How should they be used in database queries? These are all examples of questions that must be answered. And adequate rulesets for dialogue must be designed for a complete system.

The example dialogue in table 8.7 and 8.8 is the end of this chapter. In the next chapter the conclusions are summarized.

Table 8.7: Example dialogue

U0:	The user connects with the system.
	Dialogue act: `Sys(Start_Conversation)`
...	
CB:	`[Cclose]`
	`[Ctravelplan]`
	`[Agreet]`
	Response: `[Greetings]`
S0:	*Hello, train timetable information service.*
U1:	*Hello. I want to travel from Baarn/Maarn to Den Haag.*
CB:	Choice: (DepPlace, city(Baarn), city(Maarn))
IB:	ArrPlace: `city(Den Haag)`
...	
CB:	`[Cclose]`
	`[Ctravelplan]`
	`[Achoice(DepPlace, city(Baarn), city(Maarn))]`
	Response: [Choice(DepPlace, city(Baarn), city(Maarn))]
S1:	*Do you want to travel from Baarn or Maarn?*
U2:	*I want to travel from Baarn.*
IB:	ArrPlace: `city(Baarn)`
...	
CB:	`[Cclose]`
	`[CfurtherNeeds]`
	`[Cpresent]`
	`[Cquery]`
	`[Adetermine(Time)]`
	`[Adetermine(ArrPlace)]`
	`[Achoice(ArrPlace, station(Den Haag hs), station(Den Haag cs))]`
	Response: `[Choice(ArrPlace, station(Den Haag hs), station(Den Haag cs))]`
S2:	*Which station in Den Haag do you want to go to, central station or Holland Spoor?*
U3:	*I want to go to Den Haag central station.*
IB:	ArrPlace: `city(Den Haag), station(Den Haag cs)`
...	
CB:	`[Cclose]`
	`[CfurtherNeeds]`
	`[Cpresent]`
	`[Cquery]`
	`[Adetermine(Time)]`
	Response: `[Ask(When)]`
S3:	*When do you want to travel*
U4:	*...???*
...	
CB:	`[Cclose]`
	`[CfurtherNeeds]`
	`[Cpresent]`
	`[Cquery]`
	`[Adetermine(Time)]`
	Response: `[Ask(Time)]`
S4:	*At what time do you want to travel.*

Table 8.8: Example dialogue (continued)

U5:	*I want to arrive at half past nine.*
IB:	Time: `ArrTime(9:30)`
...	
CB:	`[Cclose]`
	`[CfurtherNeeds]`
	`[ApresentPart(3)]`
	`[ApresentPart(2)]`
	`[ApresentPart(1)]`
	Response: `[PresentPart(1)]`
S5:	*OK. Departure at 7:35 in Baarn to Utrecht.*
U6:	*When do I depart again?*
	Dialogue act: `Rec(Time(DepTime))`
...	
CB:	`[Cclose]`
	`[CfurtherNeeds]`
	`[ApresentPart(3)]`
	`[ApresentPart(2)]`
	`[Areconfirm(Time(DepTime))]`
	Response: `[Reconfirm(Time(DepTime))]`
S6:	*Departure at 7:35.*
U7:	*OK*
...	
CB:	`[Cclose]`
	`[CfurtherNeeds]`
	`[ApresentPart(3)]`
	`[ApresentPart(2)]`
	Response: `[PresentPart(2)]`
S7:	*Departure at 8:32 to Den Haag cs in Utrecht.*
U8:	*8:32 to Den Haag.*
...	
CB:	`[Cclose]`
	`[CfurtherNeeds]`
	`[ApresentPart(3)]`
	Response: `[PresentPart(3)]`
S8:	*Arrival at 9:13.*
U9:	*9:13 in Den Haag*
...	
CB:	`[Cclose]`
	`[AfurtherNeeds]`
	Response: `[AskFurtherNeeds]`
S10:	*Do you want another travelplan?*
U11:	*No thanks.*
...	
CB:	`[Aclose]`
	Response: `[Bye]`
S11:	*Have a nice day.*
U12:	*Goodbye* and user disconnects
	Dialogue act: `Sys(End_Conversation)`
...	
CB:	empty
IB:	empty

Chapter 9

Conclusions

This is the final chapter of this report. In this chapter a summary of the dialogue control mechanism is given and several conclusions are drawn as far as possible, given the current state of the Alparon system. At the end some suggestions are given for future work.

9.1 Framework for dialogue control

The purpose of the work described in this report was to design and implement the components for dialogue control in the Alparon System. In the Alparon system, an ASP system for information retrieval and transaction services, these components have to generate appropriate system responses. Other parts of the Alparon system take care of the interpretation of the user's turn and the transformation of the system response to dialogue acts and speech. The system responses are the means by which the Alparon system can control the dialogue, hence the name dialogue control components.

In the original design of the Alparon system, the task of generating system responses was placed with the response generation module and the control board. All the processing necessary to generate the system response should be done by the response generation module. This would include processing the user's turn on a dialogue level, since the context updating module only takes care of the user's turn on an informational level. In the actual design for dialogue control, described in this report, another module was added, the dialogue updating module, to determine the consequences of the user's turn on a dialogue level.

The dialogue updating module is charged with the updating of the goalstack according to the state of the dialogue in each turn. The goalstack is used to structure the goals that define what must be achieved during the dialogue. In information retrieval dialogues the dialogue structure highly resembles the task structure. So a plan based approach is used in which the goals embody the plan to accomplish the task and subtasks in such a dialogue. After other modules have interpreted the user's turn and extracted the factual information from it, the dialogue updating module processes the consequences of the new information and the user's turn on a dialogue level. The goalstack is updated to reflect this new situation.

The task of the response generation module is to generate the actual system response. Given the goals that still must be achieved, the response generation module places communicative actions with those goals and selects and combines communicative actions in a system response. The communicative actions are linked to goals and they are designed to help achieve the goals.

The control board is used to store the goal structure and the communicative actions, both the possible ones and the ones selected for the system response. Other information related to dialogue control is also place on this blackboard, e.g. conflicts in the system beliefs of the dialogue. All the information used for dialogue control is present on this blackboard.

Because of the research nature of the Alparon system, it should be possible to experiment with different approaches to all the aspects of dialogue management. Therefore dialogue control is designed as a generic framework. Within this framework various rulesets can be defined to specify the behaviour of the dialogue control. By changing these rulesets various approaches to dialogue control can be implemented and refined. The components for dialogue control are combined into an engine. The exact operation of this engine can be specified by the rulesets. The actual strategy for dialogue control is defined by these rulesets.

9.2 Operation

One of the aims was to develop dialogue control for a mixed-initiative dialogue. By using a list of communicative actions for each goal, several communicative actions can be used in successive turns to help achieve a particular goal. By starting with a very general (passive) action (e.g. an encouragement "*Yes?*") and making the actions progressively more specific (active), the system can let the user have the initiative initially. But when the goal is not achieved after some turns, the system can take over the initiative with direct questions.

From the opposite point of view, the system can be defined to react to certain utterances (dialogue acts) from the user. These reactions can have direct consequences for the goalstack and therefore for the dialogue structure as a whole. E.g. when the system is presenting a travelplan, the user can request a reconfirmation. This mechanism can be used to define when and how the user can take over the initiative.

The framework also incorporates strategies to deal with certain problems that can arise during the dialogue. When the dialogue manager is confronted with conflicting values for concepts in the application domain, a choice must be made between the values and that choice must be verified is some cases. Another problem can be unresolved choices between possible interpretations of the user's turn. A new goal to resolve the choice between domain values, is then added to the goalstack. The usual mechanism to deal with goals can now be used to solve such problems.

The actual performance of the dialogue control mechanism depends, of course, on the mechanism of the framework and the rulesets. But it also depends on the other parts of the dialogue manager. The rulesets must be defined in relationship with the other modules of the dialogue manager.

In the previous chapter the dialogue control part of a small train timetable information system has been defined for testing the framework. This is a small system just to test and illustrate how the dialogue control can be defined to achieve desired behaviour. Example dialogues show how the framework - with adequate rulesets - can control a mixed-initiative dialogue. Further, more elaborate testing with a complete system should be done to arrive at definite conclusions for this dialogue control solution. The testing so far however shows good promise.

9.3 Future work

This last section gives some suggestions for future work for the Alparon system. These are mainly suggestions related to dialogue control of the Alparon system. The first section, however, contains some suggestions for the whole system.

9.3.1 Further testing

Dialogue systems, especially combined with automatic speech processing, have their own specific situations that have to be dealt with. Errors in the speech recognitions process and meaning extraction in a dialogue context, are examples of characteristic issues that have to be handled. In order to fully test the dialogue control solution described in this report, the dialogue control components should be tested in a complete system.

Currently, not all the modules and blackboards of the Alparon system have been developed. Some have already been developed but do not have all the functionalities other modules rely on. Furthermore the implementations of the modules must be tuned to each other. They rely on other module to perform certain tasks.

To create a complete system in which the modules can be tested in their intended environment, should be one of the first goals in the near future. In this test environment further refinement can be attempted.

In the current version of the dialogue control components, some information concerning the results of queries is supposed to be stored on the informations board. This kind of information should eventually be placed on the status board.

9.3.2 Improvements to dialogue control

Some improvements and additions that can be made to the components for dialogue control are listed in this section.

- When the actionlists become empty and the goals are not achieved, the dialogue manager can not be of service. In the current implementation the dialogue manager just generates empty responses when that happens. A scheme to fail gracefully should be implemented for such situations.

- Expressions used by the user sometimes refer to more specific concepts than their literal meaning (e.g. the user refers to Delft station when he is just speaking of Delft). In the current implementation operations can be defined to take some actions in specific situation (e.g. replace value `city(Delft)` with `station(Delft)`). The reference module could be used in future versions to help in this kind of situation. This module could give alternate refinements of a domain values and the control components can use these according to generic rules.

- The priority scheme of goals and actions is quite straightforward. Maybe a more complicated scheme is necessary to further refine the dialogue control in a complete system.
- The combinations of communicative actions is limited to two actions per response in the current implementation. The user should not get too much information each turn. But perhaps the limit of two communicative actions is too strict.
- All the blackboards should keep track of their history. The control board does not do so in the current implementation.

9.3.3 Optimization

Finally some words about optimization are given. This is not a very important issue during development and testing, but for an complete test or demonstration system this becomes more pressing.

The modular design of the dialogue manager is quite useful during development and testing, but gives no optimal performance. For a production or demonstration system, the modular approach must probably be abandoned for speed considerations.

For dialogue control, improvement in speed can be accomplished by streamlining the testing of conditions in rules. Now these conditions are tested again and again, one by one. E.g. by applying a technique like Rete-net matching, a great decrease in time spended on checking conditions, can be achieved.

Appendix A

Information board

This appendix gives an overview of the implementation of the information board used for testing dialogue control components. This blackboard is implemented in Java using the Alparon library for messaging modules. The source code can be found in appendix F.

The information board can be used to store values in slots. To do this, the information board contains a set of slots. In these slots, values can be stored and a history of these values is recorded for every slot.

A slot is specified by its name and contains a list of entries. The last entry is the current entry of the slot, the others are previous entries and make up the history of the slot.

An entry consists of a value and some attributes. A value is an arbitrary string. Several attributes are allowed:

- `empty`
  The entry has no value. This is the first entry for every slot.
- `asked`
  The entry has no value and a value for this slot is asked for by the user.
- `verified`
  The value is this slot is verified.
- *ground*
  In this attribute the ground for the value can be specified. This ground is an arbitrary string. The ground `user` is used to indicate the user mentioned the value. The ground `system` is used in the train timetable information application to indicate the system introduced the value and has some confidence in that value. The ground `default` is used to indicate the value is a default value. The ground `toverify` is used to indicate the system introduced the value but the value should be verified. A value can have more than one ground.

When the information board receives a request to do something with a certain slot and that slot does not exist at that moment, the information board first creates a slot with that name and places an empty entry in it. Then the board processes the request. So every slot has an empty entry in its history.

An overview of the operations the information board can perform on its slots and of the messages to request them, is shown in the following list.

- Putting a value in a slot.

  Message name: `put`
  Message body: slotname, optionally the value and the ground
  Reply message name: none

  This message should be used to put a new entry with the specified value and ground in a slot. When only the slotname is specified, the slot gets a new empty entry with the `asked` attribute. When only the slotname and the value are specified, a new entry with the specified value is placed in the slot with `user` as ground.

  When the specified value is the same as the current value, the specified ground is added as an attribute to the current value.

- Getting the current value of a slot.

  Message name: `get`
  Message body: slotname
  Reply message name: `Slot`
  Reply message body: slotname and value

  The information board replies with a `Slot` message with the slotname and the current value for that slot, when it receives the `get` message. When the information board does not contain the specified slot, a new slot with that name is created.

- Getting the attributes of a slot.

  Message name: `getAttr`
  Message body: slotname
  Reply message name: `SlotAttr`
  Reply message body: slotname, value with attributes

  The value and the attributes together are placed on one line in the reply message.

- Getting the previous value of a slot.

  Message name: `getPrevious`
  Message body: slotname
  Reply message name: `PreviousAttr`
  Reply message body: slotname, previous value with attributes

  When the previous value is requested, the previous value together with the attributes of that value are returned. When there is no previous value, the reply message body is `there is no previous entry`.

- Adding a `verified` attribute to a slot.

  Message name: `verified`
  Message body: slotname, value
  Reply message name: none

To add a `verified` attribute to a value in a slot, the value must also be specified in the message. The `verified` attribute will only be added to the current value of the slot when the value specified in the message is the same as the current value of the slot.

- Removing a slot.

  Message name: `remove`
  Message body: slotname
  Reply message name: none

  This message will remove the specified slot from the information board.

- Clearing the whole information board.

  Message name: `clear`
  Message body: none
  Reply message name: none

  This message will remove all the slots from the information board.

Appendix B

Send and receive module

During the testing of the dialogue control modules, the send and receive module is used to send and receive arbitrary messages. It is implemented in Java using the Alparon library for messaging modules. The source code can be found in appendix G. The module has two functions:

- Sending messages

  Sending messages can be done in two ways:

  - Specifying the message

    To send a message in this way, the whole message has to be specified on standard input. The user has to specify the name of the module to which the message must be sent, the message name and the message body. The message body consists of zero or more lines. The user ends the specification of the message and sends the message by entering a empty line.

  - Sending a predefined message

    Predefined messages can be sent by supplying a key for a predefined message when asked for the module name to send a message to. Every predefined message has its own key. A list of the keys and messages can be seen by entering a question mark ('?') when asked for a module name.

    The predefined messages are read from a file. Each line in that file begins with the key followed by a colon (:), then the module name, a comma, a message name, a comma, and finally a comma-separated list of message body lines:

    key: *module_name*, *message_name*, *message_body_line_1*, ... , *message_body_line_n*

    A new set of predefined messages can be read by entering an asterisk ('*') when asked for a module name. The module then asks for a filename with the predefined messages.

- Receiving messages

  As soon as the send and receive module receives a message, the whole message is displayed on standard output. This can happen during the specification of a message, but has no consequences for the message to be sent.

Appendix C

Java code of the dialogue updating module

File: ../alparon/dialogueupdating/DUM.java //

```
package alparon.dialogueupdating;

import alparon.lib.*;
import java.util.*;
import java.io.*;
import COM.stevesoft.pat.Regex;


/** Alparon Dialogue Updating module. This module can process the
 * consequences of the user's turn for the dialogue.
  *
  * @author Niels Andeweg
  * @version 1.0
  */
public class DUM extends MessagingModule {

    // class constants
    // *********************************************************************

    /** The generic module name of the information board.
     */
    static final String IB = "IB";

    /** The generic module name of the control board.
     */
    static final String CB = "CB";

    // instance variables
    // *********************************************************************

    /** Reaction rules
     *
     * A Collection of reaction rules.
     */
    private Collection reactionRules;

    /** Decomposition rules for complex goals.
     *
     * A Hashtable with the String representation of the goals as keys and
     * as elements DecompRules
     */
    private Hashtable decompRules;

    /** Achievement rules for atomic goals.
     *
     * A Hashtable with the String representation of the goals as keys and
```

```
     * as elements a List of AchRules. For a goal to be achieved one of the
     * AchRules must check true.
     */
    private Hashtable achRules;

    /** Operations for atomic goals.
     *
     * A Hashtable with the String representation of the goals as keys and
     * as elements a List of OperRules.
     */
    private Hashtable operRules;


    // constructors
    // *******************************************************************

    /** Construct an dialogue updating module (DUM) instance.
     * All the rules are read and messagehandlers registered.
     *
     *@param pid the process ID of de DUM
     */
    private DUM(int pid) {
        initialise(pid);
        // read the various rules
        readReactionRules("reaction.rules");
        readDecompRules("decomposition.rules");
        readAchRules("achievement.rules");
        readOperRules("operation.rules");
        // add messagehandlers
        addHandler("turn", new MessageHandler() {
                public void handleMessage(Message msg) {
                    turn(msg);
                }
            });
        addHandler("transportDiscourse", new MessageHandler () {
                public void handleMessage(Message msg) {
                    tD(msg);
                }
            });
        // add messagehandlers for testing
        addHandler("show", new MessageHandler() {
                public void handleMessage(Message msg) {
                    show();
                }
            });
        addHandler("makeProperForm", new MessageHandler(){
                public void handleMessage(Message msg) {
                    properForm();
                }
            });
        addHandler("checkGoals", new MessageHandler() {
                public void handleMessage(Message msg) {
                    checkAchievement();
                }
            });
        addHandler("applyReaction", new MessageHandler() {
                public void handleMessage(Message msg) {
                    applyReaction(msg);
                }
            });
        addHandler("checkOperations", new MessageHandler() {
                public void handleMessage(Message msg) {
                    checkOperations();
                }
            });
        addHandler("checkConflicts", new MessageHandler() {
                public void handleMessage(Message msg) {
                    checkConflicts();
                }
            });
        addHandler("checkChoices", new MessageHandler() {
                public void handleMessage(Message msg) {
                    checkChoices();
                }
```

```
        });
}


// methods for initialisation
// ******************************************************************

/** Read the reaction rules from a file.
 *
 * @param filename the name of the file
 */
private void readReactionRules(String filename) {
    // try to read decomposition rules from file
    try {
        reactionRules = ReactionRulesReader.readReactionRules(this,
                                                              filename);
    }
    catch(Exception e) {
        System.err.println(e.getMessage());
        stop();
    }
}

/** Read the decomposition rules from a file.
 *
 * @param filename the name of the file
 */
private void readDecompRules(String filename) {
    // try to read decomposition rules from file
    try {

        decompRules = DecompRulesReader.readDecompRules(this, filename);
    }
    catch(Exception e) {
        System.err.println(e.getMessage());
        stop();
    }
}

/** Read the achievement rules from a file.
 *
 * @param filename the name of the file
 */
private void readAchRules(String filename) {
    // try to read achievement rules from file
    try {
        achRules = AchRulesReader.readAchRules(this, filename);
    }
    catch(Exception e) {
        System.err.println(e.getMessage());
        stop();
    }
}

/** Read the operation-rules from a file.
 *
 * @param filename the name of the file
 */
private void readOperRules(String filename) {
    // try to read the operation-rules from file
    try {
        operRules = OperRulesReader.readOperRules(this, filename);
    }
    catch(Exception e) {
        System.err.println(e.getMessage());
        stop();
    }
}


// methods for processing messages
// ******************************************************************

/** Method to process the "turn" message.
```

```
 *
 * First check the reaction rules. Given the dialogue acts of the turn
 * check whether the reaction rules can be applied.
 * First the atomic goals on the goalstack will be check for
 * achievement and operations. At last the goalstack will be put
 * in proper form.
 *
 *@param msg this should be the "turn" message containing the dialogue
 * acts of the user's turn in the message body.
 */
private void turn(Message msg){
    /* first check whether there are reaction rules that can be applied
     */
    applyReaction(msg);
    /* check for conflicts
     */
    checkConflicts();
    /* check for choices
     */
    checkChoices();
    /* check the atomic goals for achievement (and remove them when
     * achieved); if they are not achieved, check for operations (and
     * try to execute them); if the operations succeed then check for
     * achievement again.
     */
    checkAtomicGoals();
    /* put goalstack in proper form, i.e. atomic goals on top
     */
    properForm();
    /* send reply message back
     */
    reply(msg, "turn", new String[0]);
}

/** Method to process the "transportDiscourse" message. Just send
 *it back.
 *
 *@param msg this should be the "transportDiscourse" message
 */
private void tD(Message msg) {
    reply(msg, msg.body());
}

/** Put the goalstack in proper form. When there are no atomic goals
 * on the top of the goalstack, decompose the complex goals at the
 * top of the goalstack en check them for achievement. Repeat this
 * untill there are atomic goals on the top.
 */
private void properForm() {
    while(isOnlyComplexOnTop()) {
        // Decompose complex goals on top
        // get complex goals on top
        send(CB, "popComplexGoals", new String[0]);
        Message msg = receive("ComplexGoals");
        // for all complex goals
        for(int i = 1; i < msg.body().length; i++ ) {
            decompGoal(msg.body()[i]);
        }
        // Check new goals for achievement and operations
        checkAtomicGoals();
    }
}

/** Get the atomic goals from the goalstack and check whether they are
 *achieved or not. If they are achieved, remove them from the
 *goalstack.
 */
private void checkAchievement() {
    String[] goals = getAtomicGoals();
    for(int i = 0; i < goals.length; i++) {
        checkGoal(goals[i]);
    }
}
```

```
/** Get the atomic goals from the goalstack and check whether there
 * are operations for these goals. If there are, try to execute them.
 */
private void checkOperations() {
    String[] goals = getAtomicGoals();
    for(int i = 0; i < goals.length; i++) {
        attemptOperations(goals[i]);
    }
}

/** Check the control board for any conflicts that have arisen in this
 * turn. If there are any deal with them. i.e. if the previous value
 * of the slot had a "user" or "system" ground, a goal to verify the
 * slot will be added to the goalstack.
 */
private void checkConflicts() {
    send(CB, "getConflicts", "");
    Message receiveMsg = receive("Conflicts");
    String[] body = receiveMsg.body();
    Vector goals = new Vector();
    // answer contains slotNames for which there are conflicts
    for(int i = 1; i < body.length; i++) {
        String slotName = body[i];
        send(IB, "getPrevious", slotName);
        Message msg;
        do {
            msg = receive("PreviousAttr");
        } while (!msg.body()[1].equals(slotName));
        String previousAttr = msg.body()[2];
        if ((previousAttr.indexOf("user") > -1)
            || (previousAttr.indexOf("system") > -1)) {
            goals.add("Averify(" + slotName + ")");
        }
        else {
            // do nothing
        }
    }
    if (!goals.isEmpty()) {
        send(CB, "addLevel", (String[]) goals.toArray(new String[0]));
    }
}

/** Check the control board for any choices that haveto be made. Get
 * those choices and handle them. i.e. put a goal to resolve the choice
 * on the goalstack
 */
private void checkChoices() {
    send(CB, "getChoices", "");
    Message msg = receive("Choices");
    String[] body = msg.body();
    int size = body.length;
    Vector goals = new Vector();
    int i = 1;
    while (i < size) {

        StringBuffer goal = new StringBuffer("Achoice(" + body[i++]);
        int numberOfAlts;
        try {
            numberOfAlts = Integer.parseInt(body[i++]);
        }
        catch(Exception e) {
            numberOfAlts = 0;
        }
        for(int t = 0; t < numberOfAlts; t++) {
            goal.append(", " + body[i++]);
        }
        goal.append(")");
        goals.add(goal.toString());
    }
    if (!goals.isEmpty()) {
        send(CB, "addLevel", (String[]) goals.toArray(new String[0]));
    }
}
```

```
/** Apply the reaction rules. The message contains the dialogue acts of
 * the user's turn.
 *
 * @param msg the 'applyReaction' message.
 */
private void applyReaction(Message msg) {
    String[] body = msg.body();
    int numberOfDA = body.length - 1;
    String[] das = new String[numberOfDA];
    // copy dialogue acts from msg to das
    for(int i = 0; i < numberOfDA; ) {
        das[i] = body[++i];
    }
    for(Iterator it = reactionRules.iterator(); it.hasNext(); ) {
        ReactionRule rule = (ReactionRule) it.next();
        rule.apply(das);
    }
}

/** Print all the rules on standard output.
 */
private void show() {
    System.out.println("Reactionn rules:");
    for(Iterator it = reactionRules.iterator(); it.hasNext(); ) {
        System.out.println(it.next());
    }
    System.out.println("Decomposition rules:");
    for(Enumeration e = decompRules.keys(); e.hasMoreElements(); ) {
        String key = (String) e.nextElement();
        System.out.println(key + ": " + decompRules.get(key));
    }
    System.out.println("Achievement-rules:");
    for(Enumeration e = achRules.keys(); e.hasMoreElements(); ) {
        String key = (String) e.nextElement();
        System.out.println(key + ": " + achRules.get(key));
    }
    System.out.println("Operation-rules:");
    for(Enumeration e = operRules.keys(); e.hasMoreElements(); ) {
        String key = (String) e.nextElement();
        System.out.println(key + ": " + operRules.get(key));
    }

}


// auxilary methods
// ********************************************************************

/** Check the CB for atomic goals on top of the goalstack.
 *
 * @return true if there are atomic goals on the goalstack
 */
private boolean isAtomicOnTop() {
    send(CB, "isAtomicOnTop", new String[0]);
    Message msg = receive("AtomicOnTop");
    return msg.body()[1].equals("true");
}

/** Check the CB for empty goalstack.
 *
 * @return true if the goalstack is empty
 */
private boolean isGoalstackEmpty() {
    send(CB, "isEmpty", new String[0]);
    Message msg = receive("Empty");
    return msg.body()[1].equals("true");
}

/** Check the CB for only complex goals on top of the goalstack.
 *
 * @return true if there are only complex goals on top of the goalstack
 */
private boolean isOnlyComplexOnTop() {
    send(CB, "isOnlyComplexOnTop", new String[0]);
```

```
    Message msg  = receive("OnlyComplexOnTop");
    return msg.body()[1].equals("true");
}

/** Get atomic goals from the goalstack.
 *
 * @return a array of String with the atomic goals of the goalstack
 */
private String[] getAtomicGoals() {
    send(CB, "getAtomicGoals", new String[0]);
    Message msg = receive("AtomicGoals");
    String[] body = msg.body();
    String[] goals = new String[body.length - 1];
    for(int i = 1; i < body.length; i++ ) {
        goals[i-1] = body[i];
    }
    return goals;
}

/** Decompose the supplied complex goal into subgoals.
 */
private void decompGoal(String goal) {
    Debug.println("Decompose " + goal, 5);
    DecompRule rule  = (DecompRule) decompRules.get(goal);
    if (rule == null) { // no decomposition rule for goal
        System.err.println("DUM: decompGoal(): no decomposition"
                           + " rule for " + goal);
    }
    else {
        rule.decomp(goal);
    }
}

/** Reset the operation rules of supplied goal. This means the operation
 * can be executed again if it was already executed.
 */
void resetOperRule(String goal) {
    // reset operation rules of goal if any
    if (operRules.containsKey(goal)) {
        List operRuleL = (List) operRules.get(goal);
        for(Iterator it = operRuleL.iterator();
            it.hasNext(); ) {
            ((OperRule) it.next()).reset();
        }
    }
}

/** Check the atomic goals for achievement (and remove them when
 * achieved); if they are not achieved, check for operations (and
 * try to execute them); if the operations succeed then check for
 * achievement again.
 */
void checkAtomicGoals() {
    String[] goals = getAtomicGoals();
    for(int i = 0; i < goals.length; i++) {
        String goal = goals[i];
        if (!checkGoal(goal)) { // n.b. side-effects
            if (attemptOperations(goal)) { // n.b. side-effects
                checkGoal(goal); // n.b. side-effects
            }
        }
    }
}

// regular expression for atomic goal with parameters
static private Regex paraGoal = new Regex("^A[A-Za-z0-9_]+\\((.*)\\)");

/** Check the supplied goal for achievement and remove when achieved.
 *
 * returns true if goal is removed from goalstack
 */
private boolean checkGoal(String goal) {
    Debug.println("Check " + goal, 5);
    List ruleL = (List) achRules.get(goal);
```

```
        if ((ruleL == null) && (paraGoal.search(goal))) {
            // look for generic form of goal for achievement rules
            String genGoal = paraGoal.left(1) + "*" + paraGoal.right(1);
            ruleL = (List) achRules.get(genGoal);
        }
        if (ruleL == null)
            System.err.println("DUM: checkGoal(): no achievement-rule " +
                               "for " + goal);
        else {
            // for all AchRules
            for(Iterator i = ruleL.iterator(); i.hasNext(); ) {
                if (((AchRule) i.next()).check(goal)) {
                    send(CB, "removeGoal", goal);
                    return true;
                }
            }
        }
        return false;
    }

    /** Attempt to do the operations for a goal if they exist. Once an
     * operation is executed for a goal, it can not be executed again,
     * unless the operation rule is reset.
     *
     * @return true if operations are executed.
     */
    private boolean attemptOperations(String goal) {
        Debug.println("Attempt operations for " + goal, 5);
        boolean res = false;
        if (operRules.containsKey(goal)) {
            for(Iterator rules = ((List) operRules.get(goal)).iterator();
                rules.hasNext(); ) {
                OperRule rule = (OperRule) rules.next();
                Debug.println("OperRule: " + rule, 7);
                if (rule.attempt()) { // side-effects
                    res = true;
                    Debug.println("Operation succeeded for goal "
                                  + goal, 3);
                }
            }
        }
        return res;
    }


    // main method
    // *********************************************************************

    /** The main methode run when you invoke the dialogue updating module.
     *
     *@param args a string giving the process ID
     */
    public static void main(String args[]) {
        // set debug level
        Debug.level(5);
        int pid = 0;
        try {
            pid = Integer.parseInt(args[0]);
        }
        catch(Exception e) {
            String errormsg="DUM: pid expected as argument, program stopped!";
            System.err.println(errormsg);
            System.exit(1);
        }

        DUM dum = new DUM(pid);
        try {
            while (dum.server != null) {
                try {
                    dum.listen();
                }
                catch (UnknownMessageException e) {
                    System.err.println("DUM: " + e.getMessage());
                }
```

```
                catch (MalformedMessageException e) {
                    System.err.println("DUM: " + e.getMessage());
                }
            }
        }
        finally {
            dum.stop();
        }
    }

}
// end of class DUM
//
```

File: ../alparon/dialogueupdating/ReactionRulesReader.java //

```
package alparon.dialogueupdating;

import java.util.*;
import java.io.*;
import COM.stevesoft.pat.Regex;


/** Class to read the reaction rules. Use method readReactionRules(String)
  * to read the reaction rules.
  *
  * @author Niels Andeweg
  * @version 1.0
  */
class ReactionRulesReader {

    // regular expressions
    // condition delimiter expression
    static Regex condDelimExpr = new Regex("^\\s+&\\s+");
    // 'is' or 'has' condition expression
    static Regex isHasCondExpr =
        new Regex("^(\\w+?)\\s+(is|has|hasnot)\\s+" +
                  "([\\w|*|\\(|\\)]+?)(?=\\s|$)");
    // True condition expression
    static Regex trueCondExpr = new Regex("^true(?=\\W|$)");
    // dialogue act condition
    static Regex dialCondExpr =
        new Regex("^DA\\s+([\\w|\\[|\\]|\\(|\\)|*]+?)(?=\\s)");
    // goal condition
    static Regex goalCondExpr =
        new Regex("^Goal\\s+([\\w|\\(|\\)|\\*]+?)\\s+exist(?=\\s)");
    // response condition
    static Regex respCondExpr =
        new Regex("^Goal\\s+([\\w|\\(|\\)|\\*]+?)\\s+in\\sresponse(?=\\s)");
    // conditions operations seperator
    static Regex sepExpr = new Regex("^\\s+:\\s+");
    // operation expression
    static Regex operExpr = new Regex("^(put|goal|send)((?@()))");
    // comma expression, separates the operations
    static Regex commaExpr = new Regex("^\\s*,\\s*");
    // put operation arguments expression
    static Regex putArgsExpr =
        new Regex("\\((.+),\\s*(.+),\\s*(.+)\\)");
    // send operation arguments expression
    static Regex sendArgsExpr2 =
        new Regex("\\((.+),\\s*(.+),\\s*(.+),\\s*(.+)\\)");
    // send operation arguments expression
    static Regex sendArgsExpr1 =
        new Regex("\\((.+),\\s*(.+),\\s*(.+)\\)");
    // goal operation arguments expression
    static Regex goalArgsExpr =
        new Regex("\\((.+)\\)");



    /** Read the reation rules from the specified file.
     *
     * @param dum the dialogue updating module
     * @param filename the name of the file containing the achievement rules
     * @return a Collection of reaction rules.
     */
    static Collection readReactionRules(DUM dum, String filename)
        throws FileNotFoundException, IOException {
        Vector coll = new Vector();
        BufferedReader in = new BufferedReader(new FileReader(filename));
        String line = in.readLine();
        while (line != null) {
            ReactionRule rule = parseReactionRule(dum, line);
            if (rule != null) {
                coll.add(rule);
            }
            line = in.readLine();
        }
        in.close();
```

```
    return coll;
}

/** Parse one reaction rule.
 *
 * @param dum the dialogue updating module, necessary to create
 * conditions and operations.
 * @param line the String to parse.
 */
private static ReactionRule parseReactionRule(DUM dum, String line) {
    Vector conditions = new Vector();
    while (true) {
        if (isHasCondExpr.search(line)) {
            String slot = isHasCondExpr.stringMatched(1);
            String type = isHasCondExpr.stringMatched(2);
            String val = isHasCondExpr.stringMatched(3);
            if (type.equals("is")) {
                conditions.add(new ContCond(dum, slot, val));
            }
            else if (type.equals("has")) {
                conditions.add(new AttrCond(dum, slot, val));
            }
            else if (type.equals("hasnot")) {
                conditions.add(new AttrCond(dum, slot, val, true));
            }
            line = isHasCondExpr.right();
        }
        else if (trueCondExpr.search(line)) {
            conditions.add(new TrueCond());
            line = trueCondExpr.right();
        }
        else if (dialCondExpr.search(line)) {
            String da = dialCondExpr.stringMatched(1);
            conditions.add(new DACond(da));
            line = dialCondExpr.right();
        }
        else if (goalCondExpr.search(line)) {
            String sgoal = goalCondExpr.stringMatched(1);
            conditions.add(new GoalCond(dum, sgoal));
            line = goalCondExpr.right();
        }
        else if (respCondExpr.search(line)) {
            String sgoal = respCondExpr.stringMatched(1);
            conditions.add(new RespCond(dum, sgoal));
            line = respCondExpr.right();
        }
        else {
            break;
        }
        if (condDelimExpr.search(line))
            line = condDelimExpr.right();
    }
    if (sepExpr.search(line))
        line = sepExpr.right();
    Vector operations = new Vector();
    while (operExpr.search(line)) {
        Operation oper = null;
        if (operExpr.stringMatched(1).equals("put")) {
            oper = parsePutOperation(dum, operExpr.stringMatched(2));
        }
        else if (operExpr.stringMatched(1).equals("send")) {
            oper = parseSendOperation(dum, operExpr.stringMatched(2));
        }
        else if (operExpr.stringMatched(1).equals("goal")) {
            oper = parseGoalOperation(dum, operExpr.stringMatched(2));
        }
        if (oper != null)
            operations.add(oper);
        line = operExpr.right();
        if (commaExpr.search(line))
            line = commaExpr.right();
    }
    if (conditions.isEmpty() || operations.isEmpty())
        return null;
```

```
        else
            return new ReactionRule(conditions, operations);
    }

    /** Parse a put operation.
     */
    static private PutOperation parsePutOperation(DUM dum, String line) {
        if (putArgsExpr.search(line))
            return new PutOperation(dum,
                                    putArgsExpr.stringMatched(1),
                                    putArgsExpr.stringMatched(2),
                                    putArgsExpr.stringMatched(3));
        else
            return null;
    }

    /** Parse a send operation.
     */
    static private SendOperation parseSendOperation(DUM dum, String line) {
        if (sendArgsExpr2.search(line)) {
            return new SendOperation(dum,
                                     sendArgsExpr2.stringMatched(1),
                                     sendArgsExpr2.stringMatched(2),
                                     sendArgsExpr2.stringMatched(3),
                                     sendArgsExpr2.stringMatched(4));
        }
        else if (sendArgsExpr1.search(line)) {
            return new SendOperation(dum,
                                     sendArgsExpr1.stringMatched(1),
                                     sendArgsExpr1.stringMatched(2),
                                     sendArgsExpr1.stringMatched(3));
        }
        else {
            return null;
        }
    }

    /** Parse a goal operation.
     */
    static private GoalOperation parseGoalOperation(DUM dum, String line) {
        if (goalArgsExpr.search(line))
            return new GoalOperation(dum, goalArgsExpr.stringMatched(1));
        else
            return null;
    }

    /** Unit testing routine. Try to read the reaction rules from file
     * "reaction.rules".
     */
    public static void main(String[] args)
        throws FileNotFoundException, IOException {
        // Instantiate a dummy Regex to get the startup message
        // of Regex now
        Regex r = new Regex("");
        Collection rules = readReactionRules(null, "reaction.rules");
        System.out.println(rules);
    }
}
//
```

File: ../alparon/dialogueupdating/ReactionRule.java //

```
package alparon.dialogueupdating;

import java.util.*;
import java.io.*;

/** The reaction rule. This rule specifies under what circumstances, what
 * operations must be executed. The curcumstances are usually conditions on
 * the dialogue acts of the user.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
class ReactionRule {

    // instance variables

    /** A List of Conditions. These conditions must be met for the
     * operation to be executed.
     */
    private List conds;

    /** A List of Operations. These operations are executed when the
     * conditions are met.
     */
    private List opers;


    // constructor

    /** To create a reaction rule the conditions and operations must be
     * specified.
     *
     * @param conds a List of Conditions.
     * @param opers a List of Operations.
     */
    ReactionRule(List conds, List opers) {
        this.conds = conds;
        this.opers = opers;
    }


    // instance methods

    /** Apply the reaction rule. Check whether the conditions are met and
     * execute the operation when so.
     *
     * @param das the dialogue acts of the user's turn. The conditions
     * must be tested on these dialogue acts.
     */
    void apply(String[] das) {
        boolean testconds = true;
        String arg = "";
        for(Iterator it = conds.iterator(); it.hasNext() && testconds; ) {
            Condition cond = (Condition) it.next();
            testconds &= cond.check(das);
            String tempArg = cond.getArg();
            if (!tempArg.equals("")) {
                arg = tempArg;
            }
        }
        if (testconds) {
            for(Iterator it = opers.iterator(); it.hasNext(); ) {
                Operation oper = (Operation) it.next();
                oper.execute(arg);
            }
        }
    }


    // auxiliary methods

    public String toString() {
        return "" + conds + " : " + opers;
```

```
    }
}
//
```

File: ../alparon/dialogueupdating/DecompRulesReader.java //

```
package alparon.dialogueupdating;

import java.io.*;
import java.util.*;
import alparon.Alparon;
import COM.stevesoft.pat.Regex;


/** The DecompRulesReader provides a class method to read the decomposition
  * rules, readDecompRules.
  *
  * @author Niels Andeweg
  * @version 1.0
  */
class DecompRulesReader {

    // regular expression
    // decomposition rule expression:
    static Regex ruleExpr =
        new Regex("^(C[A-Za-z0-9_\\(\\)]+):\\s*\\[(.*)\\]");
    // goal list expression:
    static Regex ListExpr = new Regex("^\\[(.*?)\\]");
    // delimiter expression:
    static Regex delimExpr = new Regex("^\\s*,\\s*");
    // goal expresssion:
    static Regex goalExpr = new Regex("^([AC][A-Za-z0-9_\\*\\(\\)]+)");

    /** Reads the decomposition rules from a file. <br>
     * <p>
     * Each line of the file should contain a decomposition rule.
     * Each rule should start with a complex goal followed by ":",
     * the goal the decomposition rule is for. The subgoals in which the
     * complex goal should be decomposed, should be specified by a comma-
     * separated list of comma-separated lists of goals (list delimiters
     * are "[" and "]").
     * The goals should be strings beginning with a "C" for complex goals
     * and a "A" for atomic goals.
     * <p>
     * Example: <br>
     * <code>
     * Ctopgoal: [[CsubgoalLevel1], [Asubgoal1Level2, Asubgoal2Level2]]
     * </code>
     * <p>
     *
     * @param filename the name of the file relatively to the current
     * directory
     * @return a Hashtable containing the decomposition rules as a List of
     * Lists of Strings, indexed by the String representation of the goals
     */
    static Hashtable readDecompRules(DUM dum, String filename)
        throws FileNotFoundException, IOException {
        BufferedReader in = new BufferedReader(new FileReader(filename));
        Hashtable decompRules = new Hashtable();
        String line = in.readLine();
        // for each line do
        while (line != null) {
            String complexGoal;
            List subgoals;
            if (ruleExpr.search(line)) {
                complexGoal = ruleExpr.stringMatched(1);
                subgoals = parseSubgoals(ruleExpr.stringMatched(2));
                DecompRule rule = new DecompRule(dum, subgoals);
                decompRules.put(complexGoal, rule);
            }
            line = in.readLine();
        }
        in.close();
        return decompRules;
    }

    /** Parse a list of lists of subgoals.
     *
     * @param subgoalsString the String to parse.
```

```
     * @return a List of Lists of string representations of goals.
     */
    private static List parseSubgoals(String subgoalsString) {
        Vector subgoals = new Vector();
        while (ListExpr.search(subgoalsString)) {
            subgoals.add(parseList(ListExpr.stringMatched(1)));
            subgoalsString = ListExpr.right();
            if (delimExpr.search(subgoalsString))
                subgoalsString = delimExpr.right();
        }
        return subgoals;
    }

    /** Parse a list of subgoals.
     *
     * @param list the string to parse.
     * @return a List of string representations of goals
     */
    private static List parseList(String list) {
        Vector goalSet = new Vector();
        while (goalExpr.search(list)) {
            goalSet.add(goalExpr.stringMatched(1));
            list = goalExpr.right();
            if (delimExpr.search(list))
                list = delimExpr.right();
        }
        return goalSet;
    }

    /** Unit test routine. It tries to read the divisionrules from file
     * "division".
     */
    public static void main(String[] args)
        throws FileNotFoundException, IOException {
        // Instantiate a dummy Regex to get the startup message
        // of Regex now
        Regex r = new Regex("");
        Hashtable decompRules = readDecompRules(null, "decomposition.rules");
        System.out.println("Decomposition rules:\n" + decompRules);
    }
}
//
```

File: ../alparon/dialogueupdating/DecompRule.java //

```
package alparon.dialogueupdating;

import java.util.*;
import alparon.lib.*;


/** The decomposition rule. This rule tell how a complex goal should
 * be decomposed. This rule is linked to a complex goal.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
class DecompRule {

    // instance variables
    // *******************************************************************

    /** A list of levels. The goal this rule is linked to must be decomposed
     * to these levels. The levels are a collection of goals.
     */
    private List levels;

    /** The dialogue updating module, necessary to send messages
     */
    private DUM dum;


    // constructor
    // *******************************************************************

    /** Create a decomposition rule.
     *
     * @param levels a List of Lists of goals.
     */
    DecompRule(DUM dum, List levels) {
        this.dum = dum;
        this.levels = levels;
    }


    // instance methods
    // *******************************************************************

    /** Decompose. The levels of this decomposition rule are placed on the
     * goalstack.
     */
    boolean decomp(String goal) {
        // for all levels in rule
        boolean res = false;
        for(Iterator li = levels.iterator(); li.hasNext(); ) {
            List level = (List) li.next();
            if ((level.size() > 0) &&
                (((String) level.get(0)).indexOf('*') > -1 )) {
                // first new goal in level contains '*'
                // get number from infoboard and make number levels
                // to add
                String subgoal = (String) level.get(0);
                dum.send(DUM.IB, "get", goal);
                Message msg = dum.receive("Slot");
                try {
                    int number = Integer.parseInt(msg.body()[2]);
                    for(int i = number; i > 0; i--) {
                        int index = subgoal.indexOf('*');
                        int size = subgoal.length();
                        String s1 = subgoal.substring(0, index);
                        String s3 = subgoal.substring(index + 1, size);
                        Vector level2 = new Vector();
                        level2.add(s1 + i + s3);
                        addLevel(level2);
                        res = true;
                    }
                }
                catch(Exception e) {}
```

```
            }
            else {
                addLevel(level);
                res = true;
            }
        }
        return res;
    }

    /** Auxiliary method the add just one level to the goalstack.
     *
     * @param level the level to add.
     */
    private void addLevel(List level) {
        // new level to add
        String[] body = (String[]) level.toArray(new String[0]);
        dum.send(DUM.CB, "addLevel", body);
        // reset operation rules of new goals if any
        for(int i = 0; i < body.length; i++) {
            dum.resetOperRule(body[i]);
        }
    }

    // auxiliary methods
    public String toString() {
        return levels.toString();
    }
}
//
```

File: ../alparon/dialogueupdating/AchRulesReader.java //

```
package alparon.dialogueupdating;

import java.util.*;
import java.io.*;
import alparon.lib.MessagingModule;
import COM.stevesoft.pat.Regex;


/** Class to read the achievement rules. Use method readAchRules(String)
  * to read the achievement rules.
  *
  * @author Niels Andeweg
  * @version 1.0
  */
class AchRulesReader {

    // regular expressions
    // achievement rule expression
    static private Regex achRuleExpr =
new Regex("^(A[A-Za-z0-9_*\\(\\)]+):");
    // delimiter expression
    static Regex delimExpr = new Regex("^\\s*&\\s*");
    // 'is' or 'has' condition expression
    static Regex isHasCondExpr =
        new Regex("^([\\w|*]+?)\\s+(is|has|hasnot)\\s+" +
                  "([\\w|*|\\(|\\)]+?)(?=\\s|$)");
    // 'communicated' condition expression
    static Regex commCondExpr = new Regex("^communicated(?=\\W|$)");
    // True condition expression
    static Regex trueCondExpr = new Regex("^true(?=\\W|$)");


    /** Read the achievement rules from the specified file.
     *
     * @param dum the dialogue updating module
     * @param filename the name of the file containing the achievement rules
     * @return a hashtable with String representations of goals as keys
     * and a List of achievement rules as elements.
     */
    static Hashtable readAchRules(DUM dum, String filename)
        throws FileNotFoundException, IOException {
        Hashtable ht = new Hashtable();
        BufferedReader in = new BufferedReader(new FileReader(filename));
        String line = in.readLine();
        while (line != null) {
            if (achRuleExpr.search(line)) {
                String key = achRuleExpr.stringMatched(1);
                Vector v = new Vector();
                line = in.readLine();
                while (line != null && !(achRuleExpr.search(line))) {
                    AchRule achRule = parseAchRule(dum, line);
                    if (achRule != null)
                        v.add(achRule);
                    line = in.readLine();
                }
                ht.put(key, v);
            }
            else {
                line = in.readLine();
            }
        }
        in.close();
        return ht;
    }

    /** Parse a list of conditions.
     *
     * @param dum the dialogue updatingmodule.
     * @param andConds the String to be parsed.
     */
    private static AchRule parseAchRule(DUM dum, String andConds) {
        Vector v = new Vector();
        while (true) {
```

```
            if (isHasCondExpr.search(andConds)) {
                String slot = isHasCondExpr.stringMatched(1);
                String type = isHasCondExpr.stringMatched(2);
                String val = isHasCondExpr.stringMatched(3);
                if (type.equals("is")) {
                    v.add(new ContCond(dum, slot, val));
                }
                else if (type.equals("has")) {
                    v.add(new AttrCond(dum, slot, val));
                }
                else if (type.equals("hasnot")) {
                    v.add(new AttrCond(dum, slot, val, true));
                }
                andConds = isHasCondExpr.right();
                if (delimExpr.search(andConds))
                    andConds = delimExpr.right();
            }
            else if (commCondExpr.search(andConds)) {
                v.add(new CommCond(dum));
                andConds = commCondExpr.right();
                if (delimExpr.search(andConds))
                    andConds = delimExpr.right();
            }
            else if (trueCondExpr.search(andConds)) {
                v.add(new TrueCond());
                andConds = trueCondExpr.right();
                if (delimExpr.search(andConds))
                    andConds = delimExpr.right();
            }
            else
                break;
        }
        return v.isEmpty() ? null : new AchRule(v);
    }

    /** Unit testing routine. Read achievement rules from the file
     * "achievement.rules".
     */
    public static void main(String[] args)
        throws FileNotFoundException, IOException {
        // Instantiate a dummy Regex to get the startup message
        // of Regex now
        Regex r = new Regex("");
        Hashtable rules = readAchRules(null, "achievement.rules");
        System.out.println(rules);
    }
}
//
```

File: ../alparon/dialogueupdating/AchRule.java //

```
package alparon.dialogueupdating;

import java.util.*;
import alparon.lib.*;

/** An achievement rule. When all the conditions are met, the goal this
 * rule belongs to is achieved.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
class AchRule {

    // instance variables
    // *******************************************************************

    /** The List of Conditions. When these conditions are met, the goal this
     * rule belongs to, is achieved.
     */
    private List conds;


    // constructor
    // *******************************************************************

    /** Create a new achievement rule.
     *
     * You have to supply a List of Conditions. The achievement rule checks
     * true if all the Conditions of the List are met.
     *
     * @param conds the List of Conditions
     */
    AchRule(List conds) {
        //this.dum = dum;
        this.conds = conds;
    }


    // instance methods
    // *******************************************************************

    /** Check this achievement rule, i.e. check all the conditions.
     *
     * @return true if all the Conditions are met
     */
    boolean check(String goal) {
        boolean acc = true;
        // for all conditions as long as conditions are true
        for(Iterator i = conds.iterator(); i.hasNext() && acc; ) {
            Condition cond = (Condition) i.next();
            String[] args = {goal};
            acc = cond.check(args);
        }
        return acc;
    }


    // auxiliary methods
    // *******************************************************************

    public String toString() {
        return conds.toString();
    }
}
//
```

File: ../alparon/dialogueupdating/OperRulesReader.java //

```
package alparon.dialogueupdating;

import java.util.*;
import java.io.*;
import COM.stevesoft.pat.Regex;

/** This class can be used to read the operation rules from a file.
 * To read the operation rules invoke the readOperRules method.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
class OperRulesReader {

    // regular expressions
    // operation-rule expression
    static private Regex operRuleExpr =
        new Regex("^(A[\\w|*|\\(|\\)]+):");
    // delimiter expression, separates the conditions
    static Regex delimExpr = new Regex("^\\s*&\\s*");
    // condition expression
    static Regex condExpr =
        new Regex("^(\\w+?)\\s+(is|has|hasnot)\\s+" +
                  "(.+?)(?=\\s*[&|:])");
    // True condition expression
    static Regex trueCondExpr = new Regex("^true(?=\\W|$)");
    // separator between conditions and operations
    static Regex sepExpr = new Regex("^\\s*:\\s*");
    // operation expression
    static Regex operExpr = new Regex("^(put|goal|send)((?@()))");
    // comma expression, separates the operations
    static Regex commaExpr = new Regex("^\\s*,\\s*");
    // put operation arguments expression
    static Regex putArgsExpr =
        new Regex("\\((.+),\\s*(.+),\\s*(.+)\\)");
    // send operation arguments expression
    static Regex sendArgsExpr1 =
        new Regex("\\((.+),\\s*(.+),\\s*(.+)\\)");
    // send operation arguments expression
    static Regex sendArgsExpr2 =
        new Regex("\\((.+),\\s*(.+),\\s*(.+),\\s*(.+)\\)");
    // goal operation arguments expression
    static Regex goalArgsExpr =
        new Regex("\\((.+)\\)");


    /** Read the operationrules from the file.
     *
     * @param dum the dialogue updating moduel, necessary to create
     * conditions and operation.
     * @param filename the file with the operation rules.
     * @return a Hashtable with the goals as keys and the operation rules
     * as elements. (The elements are Lists of operation rule, all operation
     * rules in a List belong to the same goal.)
     */
    static Hashtable readOperRules(DUM dum, String filename)
        throws FileNotFoundException, IOException {
        Hashtable operRules = new Hashtable();
        BufferedReader in = new BufferedReader(new FileReader(filename));
        String line = in.readLine();
        while (line != null) {
            if (operRuleExpr.search(line)) {
                String key = operRuleExpr.stringMatched(1);
                Vector v = new Vector();
                line = in.readLine();
                while (line != null && !(operRuleExpr.search(line))) {
                    OperRule operRule = parseOperRule(dum, line);
                    if (operRule != null)
                        v.add(operRule);
                    line = in.readLine();
                }
                operRules.put(key, v);
            }
```

```
        else
            line = in.readLine();
    }
    in.close();
    return operRules;
}

/** Parse one operation rule.
 *
 * @param dum the dialogue updating module.
 * @param line the String to parse.
 */
private static OperRule parseOperRule(DUM dum, String line) {
    Vector conditions = new Vector();
    while (true) {
        if (condExpr.search(line)) {
            String slot = condExpr.stringMatched(1);
            String type = condExpr.stringMatched(2);
            String val = condExpr.stringMatched(3);
            if (type.equals("is")) {
                conditions.add(new ContCond(dum, slot, val));
            }
            else if (type.equals("has")) {
                conditions.add(new AttrCond(dum, slot, val));
            }
            else if (type.equals("hasnot")) {
                conditions.add(new AttrCond(dum, slot, val, true));
            }
            line = condExpr.right();
        }
        else if (trueCondExpr.search(line)) {
            conditions.add(new TrueCond());
            line = trueCondExpr.right();
        }
        else {
            break;
        }
        if (delimExpr.search(line))
            line = delimExpr.right();
    }
    if (sepExpr.search(line))
        line = sepExpr.right();
    Vector operations = new Vector();
    while (operExpr.search(line)) {
        Operation oper = null;
        if (operExpr.stringMatched(1).equals("put")) {
            oper = parsePutOperation(dum, operExpr.stringMatched(2));
        }
        else if (operExpr.stringMatched(1).equals("send")) {
            oper = parseSendOperation(dum, operExpr.stringMatched(2));
        }
        else if (operExpr.stringMatched(1).equals("goal")) {
            oper = parseGoalOperation(dum, operExpr.stringMatched(2));
        }
        if (oper != null)
            operations.add(oper);
        line = operExpr.right();
        if (commaExpr.search(line))
            line = commaExpr.right();
    }
    if (conditions.isEmpty())
        return null;
    else
        return new OperRule(conditions, operations);
}

/** Parse a put operation.
 */
static private PutOperation parsePutOperation(DUM dum, String line) {
    if (putArgsExpr.search(line))
        return new PutOperation(dum,
                                putArgsExpr.stringMatched(1),
                                putArgsExpr.stringMatched(2),
                                putArgsExpr.stringMatched(3));
```

```
        else
            return null;
    }

    /** Parse a send operation.
     */
    static private SendOperation parseSendOperation(DUM dum, String line) {
        if (sendArgsExpr2.search(line)) {
            return new SendOperation(dum,
                                     sendArgsExpr2.stringMatched(1),
                                     sendArgsExpr2.stringMatched(2),
                                     sendArgsExpr2.stringMatched(3),
                                     sendArgsExpr2.stringMatched(4));
        }
        else {
            if (sendArgsExpr1.search(line)) {
                return new SendOperation(dum,
                                         sendArgsExpr1.stringMatched(1),
                                         sendArgsExpr1.stringMatched(2),
                                         sendArgsExpr1.stringMatched(3));
            }
            else {
                return null;
            }
        }
    }

    /** Parse a goal operation.
     */
    static private GoalOperation parseGoalOperation(DUM dum, String line) {
        if (goalArgsExpr.search(line))
            return new GoalOperation(dum, goalArgsExpr.stringMatched(1));
        else
            return null;
    }

    /** Unit testing routine. Try to read the operation rules from file
     * "operation.rules".
     */
    public static void main(String[] args)
        throws FileNotFoundException, IOException {
        // Instantiate a dummy Regex to get the startup message
        // of Regex now
        Regex r = new Regex("");
        Hashtable rules = readOperRules(null, "operation.rules");
        System.out.println(rules);
    }
}
//
```

File: ../alparon/dialogueupdating/OperRule.java //

```
package alparon.dialogueupdating;

import java.util.*;

/** The operation rule. This rule tells for what conditions, what operations
 * should be executed.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
class OperRule {

    // instance variable


    /** A List of Conditions. These conditions must be met in order to
     * execute the operations.
     */
    private List conditions;

    /** A List of Operations that must be excuted when the conditions
     * are met.
     */
    private List operations;

    /** This Collection contains all the argument for which the opreations
     * have already been executed. The operations should not be executed
     * with the same arguments twice unluss the operation rul is reset.
     */
    private Collection executedArgs;


    //constructor

    /** To create an operation rule the conditions and the operations must
     * be specified.
     *
     * @param conditions the Conditions.
     * @param operations the Operations.
     */
    OperRule(List conditions, List operations) {
        this.conditions = conditions;
        this.operations = operations;
        //this.executed = false;
        this.executedArgs = new Vector();
    }


    // accessors

    final List conditions() {
        return conditions;
    }

    final List operations() {
        return operations;
    }


    // instance methods

    /** Test the conditions. If the are met execute the operations. The
     * conditions can be met for a certain argument (when the generic
     * conditions are instatiated with that argument, the conditions are
     * met). This argument is used to instatiate the generic operations
     * and stored so the operation rule will not be executed for the same
     * argument.
     */
    boolean attempt() {
        boolean testConds = true;
        boolean res = false;
        String arg = "";
        Iterator conds = conditions.iterator();
```

```
        while (testConds && conds.hasNext()) {
            Condition cond = (Condition) conds.next();
            String[] dummy = {""};
            testConds &= cond.check(dummy);
            String tempArg = cond.getArg();
            if (!tempArg.equals("")) {
                // actually all arg must be the same
                arg = tempArg;
            }
        }
        // only execute the operations if the conditions are met and if
        // the operations have not been executed with the argument arg
        if (testConds && !(executedArgs.contains(arg))) {
            for(Iterator ops = operations.iterator(); ops.hasNext(); ) {
                Operation oper = (Operation) ops.next();
                oper.execute(arg);
            }
            executedArgs.add(arg);
            res = true;
        }
        return res;
    }

    /** Reset the operations rule. The operationrule can be executed
     * agian for all arguments.
     */
    void reset() {
        this.executedArgs = new Vector();
    }

    // auxiliary methods
    public String toString() {
        return conditions + ": " + operations;
    }
}
//
```

File: ../alparon/dialogueupdating/Condition.java //

```
package alparon.dialogueupdating;


/** The abstract class for all conditions.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
abstract class Condition {

    /** The dialogue updating module, used to send messages.,
     */
    protected DUM dum;

    /** A variable to store the replacement of the * for which a generic
     * condition is met.
     */
    protected String arg = "";

    public String toString() {
        return "condition";
    }

    abstract boolean check(String[] args);

    String getArg() {
        return this.arg;
    }

}
//
```

File: ../alparon/dialogueupdating/AttrCond.java //

```
package alparon.dialogueupdating;

import alparon.lib.*;
import COM.stevesoft.pat.Regex;

/** The attribute condition. This condition is true if a slot on the
 * information board has a certain attribute. This condition can be
 * negated.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
class AttrCond extends Condition {

    /** The name of the slot that must be check for a certain attribute.
     */
    String slotName;

    /* The attribute that the slot must have.
     */
    String attr;

    /** A flag to indicate whether the condition should be negated.
     */
    boolean neg = false;

    /** Create a new attribute condition. This condition is not negated.
     *
     * @param dum the dialogue updating module, used to send messages to
     * the information board.
     * @param slotName the name of the slot.
     * @param attr the attribuut the slot should have.
     */
    AttrCond(DUM dum, String slotName, String attr) {
        this.dum = dum;
        this.slotName = slotName;
        this.attr = attr;
    }

    /** Create a new attribute condition.
     *
     * @param dum the dialogue updating module, used to send messages to
     * the information board.
     * @param slotName the name of the slot.
     * @param attr the attribuut the slot should have.
     * @param neg a flag indicating whether the condition should be
     * negated (if true, the condition is negated).
     */
    AttrCond(DUM dum, String slotName, String attr, boolean neg) {
        this(dum, slotName, attr);
        this.neg = neg;
    }

    static Regex parGoal = new Regex("^\\w+\\((.*)\\)");

    /** Check whether this condition is true.
     */
    boolean check(String[] args) {
        String slotName;
        if (this.slotName.equals("*") && parGoal.search(args[0])) {
            slotName = parGoal.stringMatched(1);
        }
        else {
            slotName = this.slotName;
        }
        String[] body = {"getAttr", slotName};
        dum.send(DUM.IB, body);
        Message ans = dum.receive("SlotAttr");
        while (!ans.body()[1].equals(slotName))
            ans = dum.receive("SlotAttr");
        String slotAttr = ans.body()[2];
        boolean res = (slotAttr.indexOf(attr) > -1);
```

```
        return neg ? !res : res;
    }

    public String toString() {
        return slotName + " has" + (neg ? "not " : " ") + attr;
    }

}
//
```

File: ../alparon/dialogueupdating/CommCond.java //

```
package alparon.dialogueupdating;

import alparon.lib.*;
import java.io.*;


/** The communicated condition. This condition is true when the goal it
 * belongs to has an action in its actionlist that was part of the
 * previous system response.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
class CommCond extends Condition {

    /** Create a ne communicated condition.
     *
     * @param dum the dialogue updating moduel, used to send messages.
     */
    CommCond(DUM dum) {
        this.dum = dum;
    }

    /** Check this conditions.
     */
    boolean check(String[] args) {
        String goal = args[0];
        dum.send(DUM.CB, "isCommunicated", goal);
        Message msg = dum.receive("Communicated");
        String ans = msg.body()[1];
        return ans.equals("true");
    }

    public String toString() {
        return "communicated";
    }
}
//
```

File: ../alparon/dialogueupdating/ContCond.java //

```
package alparon.dialogueupdating;

import alparon.lib.*;
import COM.stevesoft.pat.Regex;


/** The contents condition. This condition is met if the slot at least
 * contain the specified value in its value.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
class ContCond extends Condition {

    /** The slotname of the information board that should contain a certain
     * value.
     */
    private String slotName;

    /** The value the slot should at least contain.
     */
    private String value;

    /** Flag to indicate the condition is generic.
     */
    private boolean gen = false;

    /** The part of the value without the *.
     */
    private String genValue = "";

    private Regex genValueExpr = new Regex("^(\\w+)\\(\\*\\)$");

    /** Create a new contents conditions.
     *
     * @param dum the dialogue updating module.
     * @param slotName the slot name.
     * @param value the value that should be checked. This value may be
     * generic, i.e. contain an asterisk(*). This asterisk matches
     * everything. When the information board contain an value in the
     * slot that is at least the value of this condition with the asterisk
     * replaced with anything, this condition is met. What matches the
     * asterisk  is then placed in the variable arg.
     */
    ContCond(DUM dum, String slotName, String value) {
        this.dum = dum;
        this.slotName = slotName;
        this.value = value;
        this.gen = genValueExpr.search(value);
        this.genValue = genValueExpr.stringMatched(1);
        Debug.println(slotName + value + gen + genValue, 7);
    }


    private Regex slotValueExpr = new Regex("(\\w+)\\((.+)\\)");

    /** Check this condition.
     */
    boolean check(String[] args) {
        String[] body = {"get", slotName};
        dum.send(DUM.IB, body);
        Message ans = dum.receive("Slot");
        while (!ans.body()[1].equals(slotName))
            ans = dum.receive("Slot");
        String slotValue = ans.body()[2];
        if (gen) {
            // check whether slotvalue is equal to genValue(...) and
            // put ... in arg
            if ((slotValueExpr.search(slotValue)) &&
                slotValueExpr.stringMatched(1).equals(genValue)) {
                arg = slotValueExpr.stringMatched(2);
                return true;
            }
```

```
            return false;
        }
        else {
            // check whether slotvalue contains value
            // Debug.println("Check: " + this + slotValue.indexOf(value), 7);
            return slotValue.indexOf(value) > -1;
        }
    }

    // return arg
    String getArg() {
        return arg;
    }

    public String toString() {
        return slotName + " is " + value;
    }
}
//
```

File: ../alparon/dialogueupdating/DACond.java //

```
package alparon.dialogueupdating;

import java.io.*;
import COM.stevesoft.pat.Regex;


/** The dialogue act condition. This condition is met when the user's
 * turn contains this dialogue act.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
class DACond extends Condition {

    /** The dialogue act the user's turn should contain.
     */
    private String da;

    /** Flag top indicate whether the dialogue act is generic.
     */
    private boolean gen= false;

    /** The front part of a generic dialogue act (part before the *).
     */
    private String front = "";

    /** The end part of a generic dialogue act (part after the *).
     */
    private String end = "";

    static private Regex genExpr = new Regex("^(.*)\\*(.*)$");

    /** Create a new dialogue act.
     *
     * @param da the dialogue act that should be part of the user's turn.
     */
    DACond(String da) {
        this.da = da;
        this.gen = genExpr.search(da);
        if (this.gen) {
            this.front = genExpr.stringMatched(1);
            this.end = genExpr.stringMatched(2);
        }
        else {
            this.front = da;
        }
    }

    /** Check this condition. When the conditionis generic, arg will
     * contain the replacement for the * for which this conditions
     * is met.
     *
     * @param das the dialogue acts of the user's turn.
     */
    boolean check(String[] das) {
        for(int i = 0; i < das.length; i++) {
            String da = das[i];
            if (da.startsWith(front)) {
                if (gen) {
                    int endIndex = da.lastIndexOf(end);
                    if (endIndex > 1) {
                        arg = da.substring(front.length(), endIndex);
                        return true;
                    }
                }
                else {
                    return true;
                }
            }
        }
        return false;
    }
```

```
    public String toString() {
        return "DA " + da;
    }
}
//
```

File: ../alparon/dialogueupdating/GoalCond.java //

```
package alparon.dialogueupdating;

import java.io.*;
import COM.stevesoft.pat.Regex;
import alparon.lib.Message;


/** The goal condition. This condition is met when the specified goal is
 * part of the goalstack.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
class GoalCond extends Condition {

    /** The goal that should be on the goalstack.
     */
    private String goal;

    /** A flag to indicate a generic rule.
     */
    private boolean gen= false;

    static private Regex genExpr = new Regex("^(.*)\\*(.*)$");

    /** Create a new goal condition.
     *
     * @param dum the dialogue updating module.
     * @param goal the goal that should be on the goalstack
     */
    GoalCond(DUM dum, String goal) {
        this.dum = dum;
        this.goal = goal;
        this.gen = genExpr.search(goal);
    }

    static private Regex genGoalExpr = new Regex("^.+\\((.+)\\)");

    /** Check whether the goal is on the goalstack. If the specified goal
     * in this condition is generic (contains a *), then what replaces the
     * asterisk to meet this condition, is placed in the variable arg.
     */
    boolean check(String[] args) {
        dum.send(DUM.CB, "existGoal", goal);
        Message ans = dum.receive("Goal");
        String[] body = ans.body();
        String ansgoal = body[1];
        if (!ansgoal.equals("")) { // message contains goal
            if (gen && genGoalExpr.search(ansgoal)) {
                arg = genGoalExpr.stringMatched(1);
            }
            return true;
        }
        else {
            return false;
        }
    }

    public String toString() {
        return "Goal " + goal + " exists";
    }
}
//
```

File: ../alparon/dialogueupdating/RespCond.java //

```
package alparon.dialogueupdating;

import java.io.*;
import COM.stevesoft.pat.Regex;
import alparon.lib.Message;


/** The in response condition. This condition is true when the specified
 * goal has an action in its actionlist that was part of the previous
 * system response.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
class RespCond extends Condition {

    /** The goal that should have an action in its actionlist that is part
     * of the system response.
     */
    private String goal;

    /** A flag to indicate whether this is a generic condition.
     */
    private boolean gen= false;

    static private Regex genExpr = new Regex("^(.*)\\*(.*)$");

    /** Create a in reposne condition.
     *
     * @param dum the dialogue updating module.
     * @param goal the goal
     */
    RespCond(DUM dum, String goal) {
        this.dum = dum;
        this.goal = goal;
        this.gen = genExpr.search(goal);
    }

    static private Regex genGoalExpr = new Regex("^.+\\((.+)\\)");

    /** Check whether this condition is met. Place the replacement for the
     * asterisk in arg when this is a generic condition.
     */
    boolean check(String[] args) {
        dum.send(DUM.CB, "inResponse", goal);
        Message ans = dum.receive("Goal");
        String[] body = ans.body();
        String ansgoal = body[1];
        if (!ansgoal.equals("")) { // message contains goal
            if (gen && genGoalExpr.search(ansgoal)) {
                arg = genGoalExpr.stringMatched(1);
            }
            return true;
        }
        else {
            return false;
        }
    }

    public String toString() {
        return "Goal " + goal + " in response";
    }
}
//
```

File: ../alparon/dialogueupdating/TrueCond.java //

```
package alparon.dialogueupdating;

import alparon.lib.*;
import java.io.*;


/** The true condition. This condition is always met.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
class TrueCond extends Condition {

    TrueCond() {
        super();
    }

    /** Check  this conditions
     *
     * @return true always
     */
    boolean check(String[] args) {
        return true;
    }

    public String toString() {
        return "true";
    }
}
//
```

File: ../alparon/dialogueupdating/Operation.java //

```
package alparon.dialogueupdating;

import java.io.*;

/** The abstract class Operation. All operations are subclasses of this
 *  class.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
abstract class Operation {

    /** The dialogue updating module. This is necessary to send messages.
     */
    protected DUM dum;

    Operation(DUM dum) {
        this.dum = dum;
    }

    abstract public String toString();

    /** This method should be implemented by the subclass.
     *
     * @param arg the argument for which * must be substituted in generic
     * operations.
     */
    abstract void execute(String arg);

}
//
```

File: ../alparon/dialogueupdating/GoalOperation.java //

```
package alparon.dialogueupdating;

import COM.stevesoft.pat.Regex;

/** The goal operation. When this operation is executed a goal is placed
 *  on top of the goalstack.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
class GoalOperation extends Operation {

    /** The goal that must be placed on the goalstack.
     */
    private String goal;

    /** A flag indicating the goal to be placed is generic and must
     * be instantiated.
     */
    private boolean gen = false;

    /** The front part of the generic goal.
     */
    private String front = "";

    /** The end part of the generic goal.
     */
    private String end = "";

    static private Regex genExpr = new Regex("^(.*)\\*(.*)$");

    /** To create a goal operation the dialogue updating module and
     * a goal must be specified. A generic goal is a goal with an asterisk
     * in it. This asterisk should be replace, the goal instantiated.
     *
     * @param dum the dialogue updating module, necessary to send messages.
     * @param goal the goal to be placed on the goalstack
     */
    GoalOperation(DUM dum, String goal) {
        super(dum);
        this.goal = goal;
        this.gen = genExpr.search(goal);
        if (gen) {
            this.front = genExpr.stringMatched(1);
            this.end = genExpr.stringMatched(2);
        }
        else {
            this.front = goal;
        }
    }

    /** Execute the operation, place the goal on the goalstack.
     *
     * @param arg the replacement of the asterisk in a generic goal.
     */
    void execute(String arg) {
        String newGoal;
        if (gen) {
            newGoal = front + arg + end;
        }
        else {
            newGoal = goal;
        }
        dum.send(DUM.CB, "addLevel", newGoal);
    }

    public String toString() {
        return "goal(" + goal + ")";
    }

}
//
```

File: ../alparon/dialogueupdating/PutOperation.java //

```
package alparon.dialogueupdating;

import COM.stevesoft.pat.Regex;

/** The put operation. This operation put a certain value in a certain
 * slot of the information board.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
class PutOperation extends Operation {

    // instance variables

    /** The slot of the information board.
     */
    private String slot;

    /** The value to be placed in the slot.
     */
    private String value;

    /** A flag indication the operation is generic. The value must be
     * instantiated, i.e. the asterisk substituted for a certain value.
     * The value must be of the form ....(*).
     */
    private boolean gen = false;

    /** The generic part of the generic value.
     */
    private String genValue;

    /** The ground for the value in the slot.
     */
    private String ground;

    private Regex genValueExpr = new Regex("^(\\w+)\\(\\*\\)$");


    // constructor

    /** @param dum the dialogue updating module.
     * @param slot the slot.
     * @param value the value.
     * @param ground the ground.
     */
    PutOperation(DUM dum, String slot, String value, String ground) {
        super(dum);
        this.slot = slot;
        this.value = value;
        this.ground = ground;
        this.gen = genValueExpr.search(value);
        this.genValue = genValueExpr.stringMatched(1);
    }


    // methods

    /** Execute the put operation.
     *
     * @param arg the argument for which the asterisk must be substituted
     * in a generic operation.
     */
    void execute(String arg) {
        String newValue;
        if (gen) {
            newValue = genValue + "(" + arg + ")";
        }
        else {
            newValue = value;
        }
        String[] body = {slot, newValue, ground};
        dum.send(DUM.IB, "put", body);
```

```
    }

    public String toString() {
        return "put(" + slot + ", " + value + ", " + ground + ")";
    }

}
//
```

File: ../alparon/dialogueupdating/SendOperation.java //

```
package alparon.dialogueupdating;

import alparon.lib.Message;
import COM.stevesoft.pat.Regex;

/** The send operation. This operation sends an arbitrary message to an
 * arbitrary module.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
class SendOperation extends Operation {

    // instance variables

    /** The receiver of the message, a module.
     */
    private String receiver;

    /** The name of the message to send.
     */
    private String name;

    /** The body of the message. This body is just one String for now,
     *  not an array of Strings.
     */
    private String body;

    /** Flag to indicate whether the operation must wait for a reply.
     */
    private boolean wait = false;

    /** When the operation must wait for a reply, this is the reply message
     * name to wait for.
     */
    private String returnMsgName;

    /** Flag to indicate this is an generic operation that must be
     * instantiated.
     */
    private boolean gen = false;

    /** The front part of a generic operation.
     */
    private String front = "";

    /** The end part of a generic operation.
     */
    private String end = "";

    static private Regex genExpr = new Regex("^(.*)\\*(.*)$");


    // constructors

    /** @param dum the dialogue updation module, necessary to send messages.
     * @param receiver the receiver module.
     * @param name the message name.
     * @param body the message body.
     */
    SendOperation(DUM dum, String receiver, String name, String body) {
        super(dum);
        this.receiver = receiver;
        this.name = name;
        this.body = body;
        this.wait = false;
        this.gen = genExpr.search(body);
        if (gen) {
            this.front = genExpr.stringMatched(1);
            this.end = genExpr.stringMatched(2);
        }
        else {
            this.front = body;
```

```
        }
    }

    /** @param dum the dialogue updation module, necessary to send messages.
     * @param receiver the receiver module.
     * @param name the message name.
     * @param body the message body.
     * @param returnMsgName the name of the return message to wait for.
     */
    SendOperation(DUM dum, String receiver, String name,
                  String body, String returnMsgName) {
        super(dum);
        this.receiver = receiver;
        this.name = name;
        this.body = body;
        this.wait = true;
        this.returnMsgName = returnMsgName;
        this.gen = genExpr.search(body);
        if (gen) {
            this.front = genExpr.stringMatched(1);
            this.end = genExpr.stringMatched(2);
        }
        else {
            this.front = body;
        }
    }


    // methods

    /** Send a Message to a module and wait for a reply when necessary.
     *
     * @param arg the argument which replaces the asterisk in the generic
     * operation.
     */
    void execute(String arg) {
        String newBody;
        if (gen) {
            newBody = front + arg + end;
        }
        else {
            newBody = body;
        }
        Message sendMsg = dum.send(receiver, name, newBody);
        if (wait) {
            // wait untill reply, but ignore reply yet
            Message receiveMsg;
            receiveMsg = dum.receive(returnMsgName);
        }
    }

    public String toString() {
        return "send(" + receiver + ", " + name + ", " + body +
            (wait ? (", " + returnMsgName) : "") + ")";
    }

}
//
```

Appendix D

Java code of the response generation module

File: ../alparon/responsegeneration/RGM.java //

```
package alparon.responsegeneration;

import alparon.lib.*;
import java.util.*;
import java.io.*;
import COM.stevesoft.pat.Regex;


/** Alparon Response Generation Module.
  *
  * This module generates the system response. It uses the goals on the
  * control board as a basis. For all the atomic goals communicative action
  * list will be formed. Some of these action will be combined into a
  * system response.
  *
  * @author Niels Andeweg
  * @version 1.0
  */
public class RGM extends MessagingModule {

    // instance variables
    // ********************************************************************

    /** Action rules for the atomic goals
     *
     * A Hashtable with the String representations of the goals as keys and
     * as elements ActionRules
     */
    private Hashtable actionRules;

    /** Combination rules for the communicative actions
     *
     * A hashtable with the communicative actions (String representation)
     * as keys and as elements CombiRules
     */
    private Hashtable combiRules;


    // constructors
    // ********************************************************************

    /** Construct an response generation module (RGM) instance. Read the
     * action and combination rules and register the message handlers.
     *
     *@param pid the process ID of de RGM
     */
```

```
private RGM(int pid) {
    initialise(pid);
    // read the action rules
    readActionRules("action.rules");
    // read combination rules
    readCombinationRules("combination.rules");
    // add messagehandlers
    addHandler("turn", new MessageHandler() {
            public void handleMessage(Message msg) {
                turn(msg);
            }
        });
    addHandler("transportDiscourse", new MessageHandler () {
            public void handleMessage(Message msg) {
                tD(msg);
            }
        });
    // add messagehandlers for testing
    addHandler("putActionLists", new MessageHandler() {
            public void handleMessage(Message msg) {
                putActionLists();
            }
        });
    addHandler("show", new MessageHandler() {
            public void handleMessage(Message msg) {
                show();
            }
        });
}

// methods for initialisation
// *******************************************************************

/** Read the achievement rules from a file.
 *
 * @param filename the name of the file
 */
private void readActionRules(String filename) {
    // try to read action rules from file
    try {
        actionRules = ActionRulesReader.readActionRules(this, filename);
    }
    catch(Exception e) {
        System.err.println(e.getMessage());
        stop();
    }
}

/** Read the combination rules from a file.
 *
 * @param filename the name of the file
 */
private void readCombinationRules(String filename) {
    // try to read combination rules from file
    try {
        combiRules = CombiRulesReader.readCombiRules(this, filename);
    }
    catch(Exception e) {
        System.err.println(e.getMessage());
        stop();
    }
}


// methods for processing messages
// *******************************************************************

/** Method to process the "turn" message.
 *
 *@param msg this should be the "turn" message
 */
private void turn(Message msg){
    // remove previous response
    send("CB", "removeResponse", new String[0]);
```

```
    // give new atomic goals action lists
    putActionLists();
    /* not here
       // remove previous response
       send("CB", "removeResponse", new String[0]);
    */
    // get candidate communicative actions for response
    List cands = getCandidates();
    if (cands.isEmpty()) {
        // if there are no candidate abort dialogue
        // nog implementeren
        send("CB", "putResponse", new String[0]);
    }
    else {
        // generate the response and place it on the control board
        generateResponse(cands);
    }
    reply(msg, "turn", new String[0]);
}

/** Method to process the "transportDiscourse" message. Just send
 *it back.
 *
 *@param msg this should be the "transportDiscourse" message
 */
private void tD(Message msg) {
    reply(msg, msg.body());
}

/** Print all the rules on standard output.
 */
private void show() {
    System.out.println("Communicative action rules:");
    for(Enumeration e = actionRules.keys(); e.hasMoreElements(); ) {
        String key = (String) e.nextElement();
        System.out.println(key + ": " + actionRules.get(key));
    }
    System.out.println("Combination rules:");
    for(Enumeration e = combiRules.keys(); e.hasMoreElements(); ) {
        String key = (String) e.nextElement();
        System.out.println(key + ": " + combiRules.get(key));
    }
}


static private Regex paraGoalExpr =
    new Regex("^(A[A-Za-z0-9_]+)\\((.*)\\)");

/** Put actionList at all atomic goals without one.
 */
private void putActionLists() {
    send("CB", "getAtomicWithoutActionList", new String[0]);
    Message msg = receive("AtomicWithoutActionList");
    String[] body = msg.body();
    for(int i = 1; i < body.length; i++) {
        String goal = body[i];
        String genGoal;
        String parameter = "";
        ActionRule rule = (ActionRule) actionRules.get(goal);
        if ((rule == null) && (paraGoalExpr.search(goal))) {
            genGoal = paraGoalExpr.stringMatched(1);
            parameter = paraGoalExpr.stringMatched(2);
            rule = (ActionRule) actionRules.get(genGoal + "(*)");
        }
        if (rule == null) { // no action rule for goal
            System.err.println("RGM: putActionLists(): no action"
                               + " rule for " + goal);
        }
        else {
            rule.putActionList(parameter);
        }
    }
}
```

```
// auxilary methods
// ********************************************************************

/** Get a List of Candidates (communicative actions) for the system
 * response from the control board.
 *
 * @return a List of Candidates ordered by decending priority
 */
List getCandidates() {
    send("CB", "getCandidates", new String[0]);
    Message msg = receive("Candidates");
    Vector cands = new Vector();
    String[] body = msg.body();
    for(int i = 1; i < body.length; ) {
        try {
            int priority = Integer.parseInt(body[i++]);
            Action action = new Action(body[i++]);
            cands.add(new Candidate(priority, action));
        }
        catch(NumberFormatException e) {}
    }
    return cands;
}

static private Regex genActionExpr = new Regex("(\\w+)\\(.*\\)");

/** Generate the system response. The response consists of one or more
 * communicative actions specified in the List of candidates 'actions'.
 *
 * @param a List of Candidates (ordered by priority, decending); the
 * List can contain multiple occurances of Candidates with the same
 * action.
 */
private void generateResponse(List cands) {
    // vector for selected cadidates
    Vector response = new Vector();
    if (!cands.isEmpty()) {
        // select candidate with highest priority and which occurs the
        // most
        Candidate first = selectHighest(cands);
        response.add(first);
        String action = first.getAction().getAction();
        // look for possible combination with rest of candidates
        CombiRule combiRule = (CombiRule) combiRules.get(action);
        if ((combiRule == null) && genActionExpr.search(action)) {
            String genAction = genActionExpr.stringMatched(1);
            combiRule = (CombiRule) combiRules.get(genAction + "(*)");
        }
        if (combiRule != null) {
            Debug.println("er is combirule", 5);
            Candidate second = combiRule.getCombination(cands);
            if (second != null) {
                response.add(second);
            }
        }
    }
    // generate string array of selected candidates
    String[] actionSA = new String[response.size()];
    int i = 0;
    for(Iterator it = response.iterator(); it.hasNext(); ) {
        actionSA[i++] = ((Candidate) it.next()).getAction().getAction();
    }
    send("CB", "putResponse", actionSA);
}

/** Select the candidate with the highest priority. If there are
 * more than one, then choose between the ones that occur the most.
 *
 * @param cands a non-empty List of Candidates, ordered by decending
 * priority.
 */
private Candidate selectHighest(List cands) {
    //System.out.println(cands);
```

```
    if (cands.isEmpty()) {
        // this should not happen
        return null;
    }
    // vector with Candidates with the highest priority
    Vector highest = new Vector();
    Iterator it = cands.iterator();
    Candidate cand = (Candidate) it.next();
    highest.add(cand);
    int highestPriority = cand.getPriority();
    while (it.hasNext()) {
        cand = (Candidate) it.next();
        if (cand.getPriority() == highestPriority) {
            highest.add(cand);
        }
        else {
            break;
        }
    }
    Debug.println(highest, 7);
    cand = (Candidate) highest.get(0);
    int occurances = getNumberOfOccurances(cand, highest);
    for(int i = 1; i < highest.size(); i++) {
        Candidate nextCand = (Candidate) highest.get(i);
        int nextOccurances = getNumberOfOccurances(nextCand, highest);
        System.out.println("" + nextCand + nextOccurances);
        if (nextOccurances > occurances) {
            cand = nextCand;
            occurances = nextOccurances;
        }
    }
    //System.out.println(cand);
    return cand;
}

/** Return the number of occurances of Candidates in the List with the
 * same action as the specified Candidate.
 */
private int getNumberOfOccurances(Candidate cand, List cands) {
    Action action = cand.getAction();
    int occurances = 0;
    for(Iterator it = cands.iterator(); it.hasNext(); ) {
        if (((Candidate) it.next()).getAction().equals(action)) {
            occurances++;
        }
    }
    return occurances;
}


// main method
// *********************************************************************

/** The main methode run when you invoke the response generation module.
 *
 *@param args a string giving the process ID
 */
public static void main(String args[]) {
    // set debug level
    Debug.level(5);
    int pid = 0;
    try {
        pid = Integer.parseInt(args[0]);
    }
    catch(Exception e) {
        String errormsg="RGM: pid expected as argument, program stopped!";
        System.err.println(errormsg);
        System.exit(1);
    }
    RGM rgm = new RGM(pid);
    try {
        while (rgm.server != null) {
            try {
                rgm.listen();
```

```
                }
                catch (UnknownMessageException e) {
                    System.err.println("RGM: " + e.getMessage());
                }
                catch (MalformedMessageException e) {
                    System.err.println("RGM: " + e.getMessage());
                }
            }
        }
        finally {
            rgm.stop();
        }
    }

}
//
```

File: ../alparon/responsegeneration/ActionRulesReader.java //

```java
package alparon.responsegeneration;

import java.io.*;
import java.util.*;
import COM.stevesoft.pat.Regex;


/** The ActionRulesReader provides a class method to read the action rules,
  * readActionRules.
  *
  * @author Niels Andeweg
  * @version 1.0
  */
class ActionRulesReader {

    static private Regex ruleExpr =
        new Regex("^(A[\\w|\\*|\\(|\\)]+):\\s*\\[(.*)\\]");
    static private Regex actionExpr =
        new Regex("([\\w|\\*|\\(|\\)]+)");
    static private Regex delimExpr =
        new Regex("\\s*,\\s*");

    /** Read the action rules from file filename
     *
     * @param rgm the response generation module, necessary to create
     * action rules
     * @param filename the filename
     * @return a Hashtable with the actionrules, the keys are the goals
     * thee rules belong to and the elements are ActionRules.
     */
    static Hashtable readActionRules(RGM rgm, String filename)
        throws FileNotFoundException, IOException {
        BufferedReader in = new BufferedReader(new FileReader(filename));
        Hashtable actionRules = new Hashtable();
        String line = in.readLine();
        // for each line do
        while (line != null) {
            String goal;
            List actions;
            if (ruleExpr.search(line)) {
                goal = ruleExpr.stringMatched(1);
                actions = parseActionList(ruleExpr.stringMatched(2));
                if (!actions.isEmpty()) {
                    actionRules.put(goal, new ActionRule(rgm, goal, actions));
                }
            }
            line = in.readLine();
        }
        in.close();
        return actionRules;
    }

    /** Parse a actionlist.
     */
    static private List parseActionList(String line) {
        Vector actions = new Vector();
        while (actionExpr.search(line)) {
            actions.add(actionExpr.stringMatched(1));
            line = actionExpr.right(1);
            if (delimExpr.search(line)) {
                line = delimExpr.right();
            }
        }
        return actions;
    }

    /** Unit test routine.
     * Try to read the action rules from file "action.rules".
     */
    public static void main(String[] args)
        throws FileNotFoundException, IOException {
        // Instantiate a dummy Regex to get the startup message
        // of Regex now
```

```
        Regex r = new Regex("");
        Hashtable actionRules = readActionRules(null, "action.rules");
        System.out.println("Action rules:\n" + actionRules);
    }
}
//
```

File: ../alparon/responsegeneration/ActionRule.java //

```
package alparon.responsegeneration;

import java.util.*;
import COM.stevesoft.pat.Regex;

/** The action rule. This rule specified what communicative actions can be
 * used to (help) achieve a goal.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
class ActionRule {

    // instance variables
    // ********************************************************************

    /** Goal the action List is for.
     *
     * This should be the goal under which this rule is stored in the
     * Hashtable of the RGM
     */
    private String goal;

    /** For generic goals, this contains the generic part.
     */
    private String genGoal = "";

    /** List with communicative actions for a goal.
     */
    private List actions;

    /** Response generation module, necessary to send messages
     */
    private RGM rgm;


    // constructor

    static private Regex paraGoalExpr =
        new Regex("^(A[A-Za-z0-9_]+)\\((.*)\\)");

    /** @param rgm the response generation module.
     * @param goal the goal the action rule belongs to.
     * @param actions a List of Actions.
     */
    ActionRule(RGM rgm, String goal, List actions) {
        this.rgm = rgm;
        this.goal = goal;
        if (paraGoalExpr.search(goal)) {
            this.genGoal = paraGoalExpr.stringMatched(1);
        }
        this.actions = actions;
    }


    // instance methods

    static private Regex paraActionExpr =
        new Regex("^([A-Za-z0-9_]+)\\(\\*\\)");

    /** Set the action list of the goal. Parameter contains the value for
     * which the generic rule matches. This parameter should be substituted
     * in de action for *. No parameter will be signaled by "" (empty
     * string).
     *
     * @param parameter the substitution of the asterisk in generic actions.
     */
    void putActionList(String parameter){
        String[] body = new String[actions.size()+1];
        body[0] = goal;
        String[] actionSA = (String[]) actions.toArray(new String[0]);
        for(int i = 0; i < actionSA.length; i++) {
            String action = actionSA[i];
```

```
            if ((paraActionExpr.search(action)) && (!parameter.equals(""))) {
                String genAction = paraActionExpr.stringMatched(1);
                body[0] = genGoal + "(" + parameter + ")";
                action = genAction + "(" + parameter + ")";
            }
            else {
                body[0] = goal;
            }
            body[i+1] = action;
        }
        rgm.send("CB", "putActionList", body);
    }


    // auxiliary methods

    public String toString() {
        return actions.toString();
    }

}
//
```

File: ../alparon/responsegeneration/CombiRulesReader.java //

```
package alparon.responsegeneration;

import java.io.*;
import java.util.*;
import COM.stevesoft.pat.Regex;


/** The CombiRulesReader provides a class method to read the combination
  * rules, readCombiRules.
  *
  * @author Niels Andeweg
  * @version 1.0
  */
class CombiRulesReader {

    static private Regex ruleExpr =
        new Regex("^([\\w|\\*|\\(|\\)]+):\\s*\\[(.*)\\]");
    static private Regex actionExpr =
        new Regex("([\\w|\\*|\\(|\\)]+)");
    static private Regex delimExpr =
        new Regex("\\s*,\\s*");

    /** Read the combination rules from file filename. The keys are Actions
     * (String representation) and the elements and CombiRules.
     *
     */
    static Hashtable readCombiRules(RGM rgm, String filename)
        throws FileNotFoundException, IOException {
        BufferedReader in = new BufferedReader(new FileReader(filename));
        Hashtable combiRules = new Hashtable();
        String line = in.readLine();
        List actions;
        // for each line do
        while (line != null) {
            Action action;
            if (ruleExpr.search(line)) {
                action = new Action(ruleExpr.stringMatched(1));
                actions = parseActionList(ruleExpr.stringMatched(2));
                if (!actions.isEmpty()) {
                    combiRules.put(action.getAction(),
                                   new CombiRule(rgm, action, actions));
                }
            }
            line = in.readLine();
        }
        in.close();
        return combiRules;
    }

    /** Parse an actionlist.
     */
    static private List parseActionList(String line) {
        Vector actions = new Vector();
        while (actionExpr.search(line)) {
            actions.add(new Action(actionExpr.stringMatched(1)));
            line = actionExpr.right(1);
            if (delimExpr.search(line)) {
                line = delimExpr.right();
            }
        }
        return actions;
    }


    /** Unit test routine.
     * Try to read the combination rules from file "combination.rules".
     */
    public static void main(String[] args)
        throws FileNotFoundException, IOException {
        // Instantiate a dummy Regex to get the startup message
        // of Regex now
        Regex r = new Regex("");
        Hashtable combiRules = readCombiRules(null, "combination.rules");
```

```
        System.out.println("Combination rules:\n" + combiRules);
    }
}
//
```

File: ../alparon/responsegeneration/CombiRule.java //

```java
package alparon.responsegeneration;

import java.util.*;
import COM.stevesoft.pat.Regex;


/** The combination rule. A conbination rule tell with waht other actions a
 * certain action can be combined in the system response.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
class CombiRule {

    // instance variables

    /** The action the rule belongs to.
     */
    private Action action;

    /** A list of action which can be combined with the action this rule
     * belongs to. Actions are Strings.
     */
    private List actions;

    /** The Response Generation Module.
     */
    private RGM rgm;


    // constructor

    /** @param rgm the response generation module.
     * @param action the action.
     * @param actions the List of Action the action may be combined with.
     */
    CombiRule(RGM rgm, Action action, List actions) {
        this.rgm = rgm;
        this.action = action;
        this.actions = actions;
    }


    // instance methods

    /** Look in the List of Candidates for possibles actions to combine
     * with. Choose the one with the highest priority. If there isn't
     * any then return null.
     *
     * @param cands a List of Candidates, check whether one of these
     * candidates may be combined with the action.
     */
    Candidate getCombination(List cands) {
        int highestPriority = -1;
        Candidate result = null;
        for(Iterator it = actions.iterator(); it.hasNext(); ) {
            Action action = (Action) it.next();
            // look for Candidate with that action and highest priority
            Candidate cand = getHighestCandidateWith(cands, action);
            if ((cand != null) && (cand.getPriority() > highestPriority)) {
                highestPriority = cand.getPriority();
                result = cand;
            }
        }
        return result;
    }

    static private Regex genActionExpr = new Regex("(\\w+)\\(.*\\)");

    /** Select the candidate with the highest priority that contains
     * the specified action.
     *
     * @param cands a List of Candidates.
```

```
     * @param action a communicative action.
     */
    private Candidate getHighestCandidateWith(List cands, Action action) {
        int highestPriority = -1;
        Candidate result = null;
        for(Iterator it = cands.iterator(); it.hasNext(); ) {
            Candidate cand = (Candidate) it.next();
            if (action.isGen()) { //action is generic
                if (genActionExpr.search(cand.getAction().getAction())) {
                    String candAction = genActionExpr.stringMatched(1);
                    if ((candAction.equals(action.getGenAction())) &&
                        highestPriority < cand.getPriority()) {
                        result = cand;
                        highestPriority = cand.getPriority();
                    }
                }
            }
            else { // action is not generic
                if ((action.equals(cand.getAction()))
                    && (cand.getPriority() > highestPriority)) {
                    result = cand;
                    highestPriority = cand.getPriority();
                }
            }
        }
        return result;
    }

    // auxiliary methods

    public String toString() {
        return actions.toString();
    }
}
//
```

File: ../alparon/responsegeneration/Candidate.java //

```
package alparon.responsegeneration;


/** Candidate action for the system response. A Candidate is an Action with
 * a priority.
 *
 * @author Niels Andeweg
 * @version 1.0
  */
class Candidate {

    // instance variables

    /** The communicative action.
     */
    Action action;

    /** The priority of this candidate.
     */
    int priority = 0;

    /** @param priority the priority.
     * @param action the communicative Action.
     */
    Candidate(int priority, Action action) {
        this.priority = priority;
        this.action = action;
    }

    Action getAction() {
        return action;
    }

    int getPriority() {
        return priority;
    }

    public String toString() {
        return priority + ": " + action;
    }
}
//
```

Appendix E

Java code of the control board

File: ../alparon/controlboard/CB.java //

```
package alparon.controlboard;

import alparon.lib.*;
import java.util.*;


/** The control board <br>
  *
  * <p>
  * The control board contains all the information for dialogue control, the
  * goalstack, the actionlists, the system response, choices and conflicts.
  *
  * <p>
  * The control board can handle the following messages:
  * <ul>
  * <li> Message name: clear <br>
  *      Message body: none <br>
  *      Reply message name: none <br>
  * <li> Message name: putConflict <br>
  *      Message body: slotname <br>
  *      Reply message name: none <br>
  * <li> Message name: getConflicts <br>
  *      Message body: none <br>
  *      Reply message name: Conflicts <br>
  *      Reply message body: slotnames <br>
  * <li> Message name: putChoice <br>
  *      Message body: slotname and values <br>
  *      Reply message name: none <br>
  * <li> Message name: getChoices <br>
  *      Message body: none <br>
  *      Reply message name: Choices <br>
  *      Reply message body: slotnames and values <br>
  * <li> Message name: addLevel <br>
  *      Message body: goals <br>
  *      Reply message name: none <br>
  * <li> Message name: removeGoal <br>
  *      Message body: goal <br>
  *      Reply message name: none <br>
  * <li> Message name: popComplexGoals <br>
  *      Message body: none <br>
  *      Reply message name: ComplexGoals <br>
  *      Reply message body: complex goals <br>
  * <li> Message name: getAtomicGoals <br>
  *      Message body: none <br>
  *      Reply message name: AtomicGoals <br>
  *      Reply message body: atomic goals <br>
  * <li> Message name: isAtomicOnTop <br>
```

```
 *      Message body: none <br>
 *      Reply message name: AtomicOnTop <br>
 *      Reply message body: true or false <br>
 * <li> Message name: isEmpty <br>
 *      Message body: none <br>
 *      Reply message name: Empty <br>
 *      Reply message body: true or false <br>
 * <li> Message name: isOnlyComplexOnTop <br>
 *      Message body: none <br>
 *      Reply message name: OnlyComplexOnTop <br>
 *      Reply message body: true or false <br>
 * <li> Message name: existGoal <br>
 *      Message body: goal <br>
 *      Reply message name: Goal <br>
 *      Reply message body: goal <br>
 * <li> Message name: putActionList <br>
 *      Message body: atomic goal and actions <br>
 *      Reply message name: none <br>
 * <li> Message name: getAtomicWithoutActionList <br>
 *      Message body: none <br>
 *      Reply message name: AtomicWithoutActionList <br>
 *      Reply message body: atomic goals <br>
 * <li> Message name: getCandidates <br>
 *      Message body: none <br>
 *      Reply message name: Candidates <br>
 *      Reply message body: actions <br>
 * <li> Message name: putResponse <br>
 *      Message body: actions <br>
 *      Reply message name: none <br>
 * <li> Message name: getResponse <br>
 *      Message body: none <br>
 *      Reply message name: Response <br>
 *      Reply message body: actions <br>
 * <li> Message name: removeResponse <br>
 *      Message body: none <br>
 *      Reply message name: none <br>
 * <li> Message name: isCommunicated <br>
 *      Message body: goal <br>
 *      Reply message name: Communicated <br>
 *      Reply message body: true or false <br>
 * <li> Message name: inResponse <br>
 *      Message body: goal <br>
 *      Reply message name: Goal <br>
 *      Reply message body: goal <br>
 *
 *
 * @author Niels Andeweg
 * @version 1.0
 */
public class CB extends MessagingModule {

    // class constants
    // *******************************************************************

    /** The initial goal for the goalstack */
    static final String INITIALGOAL = "Ctop";


    // instance variables
    // *******************************************************************

    /** The goalstack with actionlists appended to the goals, so also the
     * agenda.
     */
    private GoalStack goalstack = new GoalStack();

    /** The system response.
     */
    private Response response;

    /** The conficts
     */
    private Collection conflicts;
```

```
/** The choices
 */
private Collection choices;


// constructors
// ********************************************************************

/** Construct an control board instance and register the handlers for
 *  the messages.
 *
 * @param pid the process ID of de CB
 */
private CB(int pid) {
    initialise(pid);
    clear();
    //Vector goals = new Vector();
    //goals.add(INITIALGOAL);
    //goalstack.addLevel(goals);
    // add messagehandlers
    addHandler("clear", new MessageHandler() {
            public void handleMessage(Message msg) {
                clear();
            }
        });
    addHandler("addLevel", new MessageHandler() {
            public void handleMessage(Message msg) {
                addLevel(msg);
            }
        });
    addHandler("addGoal", new MessageHandler() {
            public void handleMessage(Message msg) {
                addGoal(msg);
            }
        });
    addHandler("putActionList", new MessageHandler() {
            public void handleMessage(Message msg) {
                putActionList(msg);
            }
        });
    addHandler("putResponse", new MessageHandler() {
            public void handleMessage(Message msg) {
                putResponse(msg);
            }
        });
    addHandler("putConflict", new MessageHandler() {
            public void handleMessage(Message msg) {
                putConflict(msg);
            }
        });
    addHandler("putChoice", new MessageHandler() {
            public void handleMessage(Message msg) {
                putChoice(msg);
            }
        });
    addHandler("removeGoal", new MessageHandler() {
            public void handleMessage(Message msg) {
                removeGoal(msg);
            }
        });
    addHandler("removeResponse", new MessageHandler() {
            public void handleMessage(Message msg) {
                removeResponse(msg);
            }
        });
    addHandler("popComplexGoals", new MessageHandler() {
            public void handleMessage(Message msg) {
                popComplexGoals(msg);
            }
        });
    addHandler("getAtomicGoals", new MessageHandler() {
            public void handleMessage(Message msg) {
                getAtomicGoals(msg);
            }
```

```
        });
    addHandler("getAtomicWithoutActionList", new MessageHandler() {
            public void handleMessage(Message msg) {
                getAtomicWithoutActionList(msg);
            }
        });
    addHandler("getCandidates", new MessageHandler() {
            public void handleMessage(Message msg) {
                getCandidates(msg);
            }
        });
    addHandler("getResponse", new MessageHandler() {
            public void handleMessage(Message msg) {
                getResponse(msg);
            }
        });
    addHandler("getConflicts", new MessageHandler() {
            public void handleMessage(Message msg) {
                getConflicts(msg);
            }
        });
    addHandler("getChoices", new MessageHandler() {
            public void handleMessage(Message msg) {
                getChoices(msg);
            }
        });
    addHandler("isAtomicOnTop", new MessageHandler() {
            public void handleMessage(Message msg) {
                isAtomicOnTop(msg);
            }
        });
    addHandler("isEmpty", new MessageHandler() {
            public void handleMessage(Message msg) {
                isEmpty(msg);
            }
        });
    addHandler("isOnlyComplexOnTop", new MessageHandler() {
            public void handleMessage(Message msg) {
                isOnlyComplexOnTop(msg);
            }
        });
    addHandler("isCommunicated", new MessageHandler() {
            public void handleMessage(Message msg) {
                isCommunicated(msg);
            }
        });
    addHandler("existGoal", new MessageHandler() {
            public void handleMessage(Message msg) {
                existGoal(msg);
            }
        });
    addHandler("inResponse", new MessageHandler() {
            public void handleMessage(Message msg) {
                inResponse(msg);
            }
        });
    addHandler("show", new MessageHandler() {
            public void handleMessage(Message msg) {
                show(msg);
            }
        });
}


// methods for processing messages
// ******************************************************************

/** Clear the entire control board.
 */
private void clear() {
    goalstack = new GoalStack();
    response = new Response();
    conflicts = new Vector();
    choices = new Vector();
```

```
}

/** Add the level specified in the message.
 *
 * @param msg the 'addLevel' message containing the goals to be placed
 * on a new level.
 */
private void addLevel(Message msg) {
    Vector goals = new Vector();
    for(int i = 1; i < msg.body().length; i++)
        goals.add(msg.body()[i]);
    goalstack.addLevel(goals);
}

/** Add a goal to a specified level.
 *
 * @param msg the 'addGoal' message containing the goal and the level.
 */
private void addGoal(Message msg) {
    try {
        goalstack.addGoal(msg.body()[1], Integer.parseInt(msg.body()[2]));
    }
    catch(ArrayIndexOutOfBoundsException e) {}
    catch(NumberFormatException e) {}
}

/** Remove the specified goal from the goalstack.
 *
 * @param msg the 'removeGoal' message containing the goal to be
 * removed.
 */
private void removeGoal(Message msg) {
    goalstack.removeGoal(msg.body()[1]);
}

/** Remove the response. Actually the response itself is not removed,
 * but all the actions in the current response are removed from the
 * actionlists.
 *
 * @param msg the 'removeResponse' message.
 */
private void removeResponse(Message msg) {
    Collection actions = response.getResponse();
    for(Iterator it = actions.iterator(); it.hasNext(); ) {
        String action = (String) it.next();
        goalstack.removeAction(action);
    }
    // the response will not be actually deleted
    // response = new Response();
}

/** Put the specified actionlist at the specified goal.
 *
 * @param msg the 'putActionList' message containing the goal and
 * the action to be placed with that goal.
 */
private void putActionList(Message msg) {
    String[] body = msg.body();
    String goal = body[1]; // body[0] == "putActionList"
    Vector actions = new Vector();
    for(int i = 2; i < body.length; i++) {
        actions.add(body[i]);
    }
    goalstack.putActionList(goal, actions);
}

/** Put the specified response on the control board. The previous
 * response is replaced with this new one.
 *
 * @param msg the 'putResponse' message containing the action to be
 * placed in the system response.
 */
private void putResponse(Message msg) {
    String[] body = msg.body();
```

```
    Vector actions = new Vector();
    for(int i = 1; i < body.length; i++) {
        actions.add(body[i]);
    }
    Collection goals = goalstack.getGoalsWithActions(actions);
    response = new Response(actions, goals);
}

/** Put a conflict on the control board. The conflict is specified as
 * the slot on the informationboard for which there is a conflict.
 *
 * @param msg the 'putConflict' message containing the slotname for
 * which there is a conflict.
 */
private void putConflict(Message msg) {
    String[] body = msg.body();
    String slotName = body[1]; // body[0] is "putConflict"
    if (!conflicts.contains(slotName)) {
        conflicts.add(slotName);
    }
}

/** Put a choice on the control board. A choice is specified by a slot
 * of the information board for which a choice should be made for
 * several possible values for this slot. These possible values are
 * specified after the slotname.
 *
 * @param msg the 'putChoice' message containing the choice.
 */
// eigenlijk mag er nog geen choice bestaan voor dit slot.
private void putChoice(Message msg) {
    String[] body = msg.body();
    String slotName = body[1]; // body[0] is "putChoice"
    String[] alts = new String[body.length - 2];
    for(int i = 2; i < body.length; i++) {
        alts[i-2] = body[i];
    }
    choices.add(new Choice(slotName, alts));
}

/** Remove all the complex goals from the top of the goalstack and
 * return them in the 'ComplexGoals' reply message.
 *
 * @param msg the 'popComplexGoals' message.
 */
private void popComplexGoals(Message msg) {
    Collection cgc = goalstack.popComplexGoals();
    String[] cgsa = new String[cgc.size()];
    int i = 0;
    for(Iterator ci = cgc.iterator(); ci.hasNext(); )
        cgsa[i++] = ((ComplexGoal) ci.next()).id();
    reply(msg, "ComplexGoals", cgsa);
}

/** Return all the atomic goals on the goalstack in the 'AtomicGoals'\
 * reply message.
 *
 * @param msg the 'getAtomicGoals' message.
 */
private void getAtomicGoals(Message msg) {
    Collection agc = goalstack.getAtomicGoals();
    String[] agsa = new String[agc.size()];
    int i = 0;
    for(Iterator ci = agc.iterator(); ci.hasNext(); )
        agsa[i++] = ((Goal) ci.next()).id();
    reply(msg, "AtomicGoals", agsa);
}

/** Return all the atomic goals without an actionlist in the
 * 'AtomicWithoutActionList' reply message.
 *
 * @param msg the 'getAtomicWithoutActionList' message.
 */
private void getAtomicWithoutActionList(Message msg) {
```

```
    Collection agc = goalstack.getAtomicWithoutActionList();
    String[] agsa = new String[agc.size()];
    int i = 0;
    for(Iterator ci = agc.iterator(); ci.hasNext(); )
        agsa[i++] = ((Goal) ci.next()).id();
    reply(msg, "AtomicWithoutActionList", agsa);
}

/** Return all the action that are at the end of the actionlist,
 * together with the level number/priority of the goals they
 * belong to. The reply message is 'Candidates'.
 *
 * @param msg the 'getCandidates' message.
 */
private void getCandidates(Message msg) {
    Vector cands = goalstack.getCandidates();
    String[] candsS = new String[cands.size()];
    int i = 0;
    for(Iterator it = cands.iterator(); it.hasNext(); ) {
        candsS[i++] = (String) it.next();
        candsS[i++] = (String) it.next();
    }
    reply(msg, "Candidates", candsS);
}

/** Return the action of the current system response in the
 * 'Response' reply message.
 *
 * @param msg teh 'getResponse' message.
 */
private void getResponse(Message msg) {
    Collection actions = response.getResponse();
    String[] body = new String[actions.size()];
    int i = 0;
    for(Iterator it = actions.iterator(); it.hasNext(); ) {
        body[i++] = (String) it.next();
    }
    reply(msg, "Response", body);
}

/** Reply on msg with the conflicts and remove the conflicts. The
 * reply message is 'Conflicts'.
 *
 * @param msg the 'getConflicts' message.
 */
private void getConflicts(Message msg) {
    String[] body = (String[]) conflicts.toArray(new String[0]);
    conflicts.clear();
    reply(msg, "Conflicts", body);
}

/** Reply with the choices and remove them. The reply message is
 * 'Choices'.
 *
 * @param msg the 'getChoices' message.
 */
private void getChoices(Message msg) {
    Vector vbody = new Vector();
    for(Iterator it = choices.iterator(); it.hasNext(); ) {
        Choice choice = (Choice) it.next();
        vbody.add(choice.slotName());
        String[] alts = choice.alts();
        vbody.add("" + alts.length);
        for(int i = 0; i < alts.length; i++ ) {
            vbody.add(alts[i]);
        }
    }
    String[] body = (String[]) vbody.toArray(new String[0]);
    choices.clear();
    reply(msg, "Choices", body);
}

/** Check whether there is an atomic goal on top of the goalstack.
 * The reply message is 'AtomicOnTop' with 'true' or 'false' in the
```

```
     * body.
     *
     * @param msg the 'isAtomicOnTop' message.
     */
    private void isAtomicOnTop(Message msg) {
        boolean aot = goalstack.isAtomicOnTop();
        reply(msg, "AtomicOnTop", aot ? "true" : "false");
    }

    /** Check whether the goalstack is empty.
     * The reply message is 'Empty' with 'true' or 'false' in the
     * body.
     *
     * @param msg the 'isEmpty' message.
     */
    private void isEmpty(Message msg) {
        boolean ie = goalstack.isEmpty();
        reply(msg, "Empty", ie ? "true" : "false");
    }

    /** Check whether there are only complex goals on top of the goalstack.
     * The reply message is 'OnlyComplexOnTop' with 'true' or 'false' in the
     * body.
     *
     * @param msg the 'isOnlyComplexOnTop' message.
     */
    private void isOnlyComplexOnTop(Message msg) {
        boolean iocot = !(goalstack.isEmpty()) &&
            !(goalstack.isAtomicOnTop());
        reply(msg, "OnlyComplexOnTop", iocot ? "true" : "false");
    }

    /** Checks whether the specified goal (in the message) has an action
     * that was part of the system response. The reply message is
     * 'Communicated' with 'true' or 'false' in the
     * body.
     *
     * @param msg the 'isCommunicated' message containing a goal.
     */
    private void isCommunicated(Message msg) {
        String goal = msg.body()[1];
        boolean communicated = response.isCommunicated(goal);
        reply(msg, "Communicated", communicated ? "true" : "false");
    }

    /** Check whether goal, specified in the body of the message, exists.
     * If it exist than send it back in reply message 'Goal', else send
     * back message 'Goal' with empty body. Generic goals
     * are allowed of the form 'gengoal(*)' where * may be everything.
     *
     * @param msg the 'existGoal' message.
     */
    private void existGoal(Message msg) {
        String[] body = msg.body();
        Goal goal = goalstack.existGoal(body[1]);
        if (goal == null) {
            reply(msg, "Goal", "");
        }
        else {
            reply(msg, "Goal", goal.id());
        }
    }

    /** Check whether goal, specified in the body of the message, was part
     * of the last response. If so, send the goal back in reply message
     * 'Goal', else send message 'Goal' back with empty body. Generic goals
     * are allowed of the form 'gengoal(*)' where * may be everything.
     *
     * @param msg the 'inResponse' message.
     */
    private void inResponse(Message msg) {
        String[] body = msg.body();
        Goal goal = response.checkFor(body[1]);
        if (goal == null) {
```

```
            reply(msg, "Goal", "");
        }
        else {
            reply(msg, "Goal", goal.id());
        }
    }


    // auxiliary methods
    // ******************************************************************

    public String toString() {
        return "Goalstack:\n" + goalstack + "Response: " + response +
            "\nConflicts for slots: " + conflicts + "\nChoice: " +
            choices;
    }

    /** Show the entire control board on standard output.
     */
    private void show(Message msg) {
        System.out.println(this);
    }

    private String[] collection2StringArray(Collection coll) {
        String[] stringArray = new String[coll.size()];
        int i = 0;
        for(Iterator it = coll.iterator(); it.hasNext(); ) {
            stringArray[i++] = (String) it.next();
        }
        return stringArray;
    }


    // *****************************************************************

    /** The main methode run when you start the control board.
     *
     * @param args a string giving the process ID
     */
    public static void main(String args[]) {
        //set debug level
        Debug.level(3);
        int pid = 0;
        try {
            pid = Integer.parseInt(args[0]);
        }
        catch(Exception e) {
            System.err.println("Control board: pid expected as argument, " +
                               "program stopped!");
            System.exit(1);
        }
        CB cb = new CB(pid);
        try {
            while (cb.server != null) {
                try {
                    cb.listen();
                }
                catch (UnknownMessageException e) {
                    System.err.println("Control board: " + e.getMessage());
                }
                catch (MalformedMessageException e) {
                    System.err.println("Control board: " + e.getMessage());
                }
            }
        }
        finally {
            cb.stop();
        }
    }

}
//
```

File: ../alparon/controlboard/GoalStack.java //

```
package alparon.controlboard;

import java.util.*;
import alparon.Alparon;
import COM.stevesoft.pat.Regex;


/** This is the actual goalstack. The control board contains one such
 * goalstack.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
public class GoalStack {

    // instance variables
    // *******************************************************************

    /** The goalstack is a Vector of Collections of Goals
     */
    private Vector stack;


    // constructor
    // *******************************************************************

    GoalStack() {
        stack = new Vector();
    }


    // instance methods
    // *******************************************************************

    // methods that change the goalstack
    // *********************************

    /** Add a level to the goalstack with the given goals.
     *
     * The goals must be supplied in a Collection. The Collection
     * should contain String representations of the goals. Those
     * String representations should be transformed to
     * alparon.controlboard.Goal representations. If there allready exists
     * the same goal, then the goal will be tranfer to the new level. (So
     * not added.)
     *
     * @param goals a Collection containing String representations of
     * goals
     */
    void addLevel(Collection sgoals) {
        Vector tg = new Vector();
        for(Iterator it = sgoals.iterator(); it.hasNext(); ) {
            String sgoal = (String) it.next();
            Goal newGoal = null;
            Goal oldGoal = getGoal(sgoal);
            if (oldGoal == null) { // goal didnot exist allready
                if (sgoal.charAt(0) == Alparon.COMPLEXFLAG) {
                    newGoal = new ComplexGoal(sgoal);
                }
                else if (sgoal.charAt(0) == Alparon.ATOMICFLAG) {
                    newGoal = new AtomicGoal(sgoal);
                }
            }
            else { // goal did exist allready
                newGoal = oldGoal;
                removeGoal(oldGoal);
            }
            if (newGoal != null) {
                tg.add(newGoal);
            }
        }
        if (!tg.isEmpty())
            stack.add(tg);
```

```
}

/** Add a goal to a specific level. The level must exist.
 *
 * @param sgoal the goal to add.
 * @param level the level in which the goal must be placed.
 */
void addGoal(String sgoal, int level)
    throws ArrayIndexOutOfBoundsException {
    Goal goal;
    if (sgoal.charAt(0) == Alparon.COMPLEXFLAG) {
        goal = new ComplexGoal(sgoal);
        ((Collection) stack.get(level)).add(goal);
    }
    else if (sgoal.charAt(0) == Alparon.ATOMICFLAG) {
        goal = new AtomicGoal(sgoal);
        ((Collection) stack.get(level)).add(goal);
    }
}

/** Remove the specified goal. If the goal is the last one on a level,
 * remove that now empty level as well.
 *
 * @param sgoal the goal to be removed.
 */
void removeGoal(String sgoal) {
    for(Iterator si = stack.iterator(); si.hasNext(); ) {
        Collection level = (Collection) si.next();
        for(Iterator li = level.iterator(); li.hasNext(); )
            if (((Goal) li.next()).isEqual(sgoal))
                li.remove();
        if (level.isEmpty())
            si.remove();
    }
}

/** Remove the specified goal. If the goal is the last one on a level,
 * remove that now empty level as well.
 *
 * @param goal the goal to be removed.
 */
void removeGoal(Goal goal) {
    for(Iterator si = stack.iterator(); si.hasNext(); ) {
        Collection level = (Collection) si.next();
        for(Iterator li = level.iterator(); li.hasNext(); )
            if (((Goal) li.next()) == goal) {
                li.remove();
            }
        if (level.isEmpty())
            si.remove();
    }
}

/** Remove the specified action (String) from all the actionLists.
 *
 * @param action the action to be removed
 */
void removeAction(String action) {
    Collection atomicGoals = getAtomicGoals();
    for(Iterator it=atomicGoals.iterator(); it.hasNext(); ) {
        AtomicGoal atomicGoal = (AtomicGoal) it.next();
        if (atomicGoal.hasActionList()) {
            atomicGoal.removeAction(action);
        }
    }
}

/** Remove the complex goals from the top of the goalstack and return
 * them in a Collection.
 *
 * @return a Collection with the ComplexGoals
 */
Collection popComplexGoals() {
    Vector cgv = new Vector();
```

```
    try {
        Collection toplevel = (Collection) stack.lastElement();
        for(Iterator li = toplevel.iterator(); li.hasNext(); ) {
            Goal goal = (Goal) li.next();
            if (goal instanceof ComplexGoal) {
                cgv.add(goal);
                li.remove();
            }
        }
        if (toplevel.isEmpty())
            stack.removeElementAt(stack.size() - 1);
    }
    catch(NoSuchElementException e) {}
    catch(ArrayIndexOutOfBoundsException e) {}
    return cgv;
}

/** Put the action list at the specified goal.
 *
 * @param goal the goal for which the actionlist is.
 * @param actions the actions to be placed at the goal.
 */
void putActionList(String goal, List actions) {
    AtomicGoal atomic = getAtomicGoal(goal);
    if (atomic != null) {
        atomic.setActionList(actions);
    }
}


// inspection methods
// ******************

/** Check whether there is an atomic goal on top of the goalstack.
 */
boolean isAtomicOnTop() {
    boolean atomic = false;
    try {
        Collection toplevel = (Collection) stack.lastElement();
        for(Iterator li = toplevel.iterator(); li.hasNext(); ) {
            if (li.next() instanceof AtomicGoal) {
                atomic = true;
                break;
            }
        }
    }
    catch(NoSuchElementException e) {}
    return atomic;
}

/** Check whether the goalstack is empty.
 */
boolean isEmpty() {
    return stack.isEmpty();
}

static private Regex genGoalExpr =
    new Regex("^(.+)\\(\\*\\)");
static private Regex parGoalExpr =
    new Regex("^(.+)\\(.+\\)");

/** Check whether the goalstack contains the 'goal' and return it
 * else return null. Generic goals of form 'gengoal(*)' will be
 * matched with anything for * and specific goal returned.
 *
 * @param sgoal the goal to be checked.
 */
Goal existGoal(String sgoal) {
    boolean gen = genGoalExpr.search(sgoal);
    String front = "";
    if (gen) {
        front = genGoalExpr.stringMatched(1);
    }
    for(Iterator si = stack.iterator(); si.hasNext(); ) {
```

```
        for(Iterator li = ((Collection) si.next()).iterator();
            li.hasNext(); ) {
            Goal goal = (Goal) li.next();
            if (gen) {
                if (parGoalExpr.search(goal.id())) {
                    String goalfront = parGoalExpr.stringMatched(1);
                    if (goalfront.equals(front)) {
                        return goal;
                    }
                }
            }
            else {
                if (goal.isEqual(sgoal)) {
                    return goal;
                }
            }
        }
    }
    return null;
}

/** Return all the atomic goals on the goalstack in a Collection.
 *
 * @return a Collection of AtomicGoals
 */
Collection getAtomicGoals() {
    Vector agv = new Vector();
    for(Iterator si = stack.iterator(); si.hasNext(); )
        for(Iterator li = ((Collection) si.next()).iterator();
            li.hasNext(); ) {
            Goal goal = (Goal) li.next();
            if (goal instanceof AtomicGoal)
                agv.add(goal);
        }
    return agv;
}

/** Return the atomic goals without a action list  on the goalstack
 * in a Collection.
 *
 * @return a Collection of AtomicGoals.
 */
Collection getAtomicWithoutActionList() {
    Vector gc = new Vector();
    Collection goals = getAtomicGoals();
    for(Iterator i = goals.iterator(); i.hasNext(); ) {
        Goal goal = (Goal) i.next();
        if ((goal instanceof AtomicGoal) &&
            !(((AtomicGoal) goal).hasActionList())) {
            gc.add(goal);
        }
    }
    return gc;
}

/** Return the atomic goals that one of the specified actions in its
 * actionlist.
 *
 * @param actions a Collection of actions.
 * @return a Collection of AtomicGoals.
 */
Collection getGoalsWithActions(Collection actions) {
    Collection atomicGoals = getAtomicGoals();
    Collection answer = new Vector();
    for(Iterator ai = actions.iterator(); ai.hasNext(); ) {
        String action = (String) ai.next();
        for(Iterator agi = atomicGoals.iterator(); agi.hasNext(); ) {
            AtomicGoal atomicGoal = (AtomicGoal) agi.next();
            if (atomicGoal.inActionList(action)) {
                answer.add(atomicGoal);
                break;
            }
        }
    }
```

```
    return answer;
}

/** Return the ends of the action lists. These are the candidates to be
 * part of the system response. The return vector contains alternately
 * a number (in a String) (the priority) followed by an action (with
 * that priority).
 */
Vector getCandidates() {
    Vector cands = new Vector();
    int levels = stack.size() - 1; // levels: 0 to levels
    for(ListIterator li = stack.listIterator(levels + 1);
        li.hasPrevious(); ) {
        Collection level = (Collection) li.previous();
        for(Iterator it = level.iterator(); it.hasNext(); ) {
            Goal goal = (Goal) it.next();
            if ((goal instanceof AtomicGoal)
                && ((AtomicGoal) goal).hasActionList()) {
                // every atomic goal should have an actionlist
                String action = ((AtomicGoal) goal).getLastAction();
                cands.add(levels + "");
                cands.add(action);
            }
        }
        levels--;
    }
    return cands;
}


// auxiliary methods
// ********************************************************************

/** Get the atomic goal with the specified name.
 *
 * @param atomicGoal the goalname.
 * @return the AtomicGoal with that name or null if it does not exists.
 */
private AtomicGoal getAtomicGoal(String atomicGoal) {
    for(Iterator si = stack.iterator(); si.hasNext(); ) {
        Collection level = (Collection) si.next();
        for(Iterator li = level.iterator(); li.hasNext(); ) {
            Goal goal = (Goal) li.next();
            if ((goal instanceof AtomicGoal) &&
                goal.isEqual(atomicGoal)) {
                return (AtomicGoal) goal;
            }
        }
    }
    return null;
}

/** Get the goal with the specified name.
 *
 * @param sgoal the goalname.
 * @return the Goal with that name or null if it does not exists.
 */
private Goal getGoal(String sgoal) {
    for(Iterator si = stack.iterator(); si.hasNext(); ) {
        Collection level = (Collection) si.next();
        for(Iterator li = level.iterator(); li.hasNext(); ) {
            Goal goal = (Goal) li.next();
            if (goal.isEqual(sgoal)) {
                return goal;
            }
        }
    }
    return null;
}

// overrides Object.toString()
public String toString() {
    StringBuffer buff = new StringBuffer();
    int levelnummer = 0;
```

```
        for(Iterator vi = stack.iterator(); vi.hasNext(); ) {
            buff.append(levelnummer++);
            buff.append(": ");
            for(Iterator li = ((Collection) vi.next()).iterator();
                li.hasNext(); ) {
                buff.append(li.next().toString());
                buff.append(", ");
            }
            buff.delete(buff.length() - 2, buff.length());
            buff.append("\n");
        }
        return buff.toString();
    }

}
//
```

File: ../alparon/controlboard/Goal.java //

```
package alparon.controlboard;

/** Goal, the abstract class for all goals.
  *
  * @author Niels Andeweg
  * @version 1.0
  */
abstract class Goal {

    // instance variables
    // *********************************************************************

    /** The type of the goal. The string representation without the C or A.
     */
    protected String type;

    /** The String representation of the goal. This is used in messages and
     * also for the constructor.
     */
    protected String id;


    // auxiliary methods
    // *********************************************************************

    /** Check whether this goal is equal to another goal. It is equal if
     * the argument is also a Goal object and if they have the same String
     * representation.
     */
    public boolean equals(Object o) {
        if (o instanceof Goal)
            return ((Goal) o).toString().equals(toString());
        else
            return false;
    }

    /** Check whether this goal is equal to another goal that is specified
     * with its String representation.
     */
    boolean isEqual(String sgoal) {
        return sgoal.equals(id);
    }

    public String toString() {
        return this.id;
    }

    String id() {
        return this.id;
    }

}
//
```

File: ../alparon/controlboard/ComplexGoal.java //

```
package alparon.controlboard;
import alparon.Alparon;


/** Complex goal
  *
  * @author Niels Andeweg
  * @version 1.0
  */
class ComplexGoal extends Goal {

    /** To create a complex goal, specify the name of the goal.
     */
    ComplexGoal(String type) {
        if (type.charAt(0) == Alparon.COMPLEXFLAG) {
            this.type = type.substring(1);
            this.id = type;
        }
    }

}
//
```

File: ../alparon/controlboard/AtomicGoal.java //

```
package alparon.controlboard;
import alparon.Alparon;
import java.util.*;


/** Atomic Goal.
  *
  * @author Niels Andeweg
  * @version 1.0
  */
class AtomicGoal extends Goal {

    // instance variables
    // ********************************************************************

    /** Communicative actionlist of the atomic goal
     */
    private ActionList actionList = null;


    // constructor
    // ********************************************************************

    AtomicGoal(String type) {
        if (type.charAt(0) == Alparon.ATOMICFLAG) {
            this.type = type.substring(1);
            this.id = type;
        }
    }


    // instance methods
    // ********************************************************************

    /** Place teh actions in the actionlist attached to this atomic goal.
     *
     * @param actions a List of actions.
     */
    void setActionList(List actions) {
        actionList = new ActionList(actions);
    }

    /** Remove the specified action from the actionlist.
     *
     * @param action the action to be removed.
     */
    void removeAction(String action) {
        if (actionList != null) {
            actionList.removeAction(action);
        }
    }

    /** Check whether this atomic goal has an actionlist.
     */
    boolean hasActionList() {
        return (actionList != null);
    }

    /** Check whether the specified action is part of the actionlist of
     * this atomic goal.
     *
     * @param action the action.
     */
    boolean inActionList(String action) {
        if (actionList == null) {
            return false;
        }
        else {
            return actionList.inActionList(action);
        }
    }

    /** Get the last action of the actiolist.
```

```
     *
     * @return the last action of the actionlist of thisatomic goal.
     */
    String getLastAction() {
        return actionList.getLastAction();
    }

    // overrides Goal.toString()
    public String toString() {
        if (actionList == null) {
            return id;
        }
        else {
            return id + actionList;
        }
    }
}
//
```

File: ../alparon/controlboard/Choice.java //

```
package alparon.controlboard;


/** The choice object.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
class Choice {

    /** The name of the slot for which there is a choice.
     */
    private String slotName;

    /** The possible values for the slot.
     */
    private String[] alts;

    /** To create a choice, the slotname and the possible values must be
     * specified.
     *
     * @param slotName the name of the slot on the information board.
     * @param alts an array of the possible values for the slot.
     */
    Choice(String slotName, String[] alts) {
        this.slotName = slotName;
        if (alts.length < 2) {
            System.out.println("alparon.controlboard.Choice: Choice should" +
                               " have 2 or more alternatives");
        }
        this.alts = alts;
    }

    /** @return the slotname of this choice.
     */
    String slotName() {
        return slotName;
    }

    /** @return the possible value in an array.
     */
    String[] alts() {
        return alts;
    }

    public String toString() {
        StringBuffer ans = new StringBuffer(slotName);
        ans.append(": ");
        for(int i = 0; i < alts.length; i++) {
            ans.append(alts[i]);
            if (i < (alts.length - 1)) {
                ans.append(", ");
            }
        }
        return ans.toString();
    }

}
//
```

File: ../alparon/controlboard/Response.java //

```
package alparon.controlboard;

import java.util.*;
import COM.stevesoft.pat.Regex;

/** The response of the system.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
class Response {

    // instance variables
    // *********************************************************************

    /** The response is a collection of comminicative actions (Strings).
     */
    private Collection response;

    /** The goals from which there is an action in the response
     */
    private Collection goals;


    // constructors
    // *********************************************************************

    /** Create a empty Response.
     */
    Response() {
        this.response = new Vector();
        this.goals = new Vector();
    }

    /** Create a new Response instance with the actions in the collection.
     * These actions are Strings.
     *
     * @param a collection with actions (Strings)
     */
    Response(Collection actions, Collection goals) {
        this();
        this.response = actions;
        this.goals = goals;
    }


    // instance methods
    // *********************************************************************

    /** @return a collection containing the actions of the response.
     */
    Collection getResponse() {
        Vector ans = new Vector();
        for(Iterator it = response.iterator(); it.hasNext(); ) {
            ans.add(it.next());
        }
        return ans;
    }

    /** Check whether the specified goal had an action in its actionlist
     * that was part of this response.
     *
     * @param goal the goal to check for.
     */
    boolean isCommunicated(String goal) {
        for(Iterator it = goals.iterator(); it.hasNext(); ) {
            Goal g = (Goal) it.next();
            if (g.isEqual(goal)) {
                return true;
            }
        }
        return false;
    }
```

```
    static private Regex genGoalExpr =
        new Regex("^(.+)\\(\\*\\)");
    static private Regex parGoalExpr =
        new Regex("^(.+)\\(.+\\)");

    /** Check whether the specified goal had an action in its actionlist
     * that was part of this response. The specified action is allowed to
     * be generic, containing an *. The return value is the actual goal
     * that had an action of its actionlist in the response (an
     * instatiation of the generic goal).
     */
    Goal checkFor(String sgoal) {
        boolean gen = genGoalExpr.search(sgoal);
        String front = "";
        if (gen) {
            front = genGoalExpr.stringMatched(1);
        }
        for(Iterator it = goals.iterator(); it.hasNext(); ) {
            Goal goal = (Goal) it.next();
            if (gen) {
                if (parGoalExpr.search(goal.id())) {
                    String goalfront = parGoalExpr.stringMatched(1);
                    if (goalfront.equals(front)) {
                        return goal;
                    }
                }
            }
            else {
                if (goal.isEqual(sgoal)) {
                    return goal;
                }
            }
        }
        return null;
    }

    public String toString() {
        return response.toString();
    }

}
//
```

Appendix F

Java code of the information board

File: ../alparon/infoboard/IB.java //

```
package alparon.infoboard;

import alparon.lib.*;
import java.util.*;

/** The information board. This is an implementation of the information
 * board. This implementation was made to help testing the dialogue
 * control components of the Alparon system. It does not have all the
 * functionalities it should have in the Alparon system. Only the
 * functionalities concerning dialogue control were implemented.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
public class IB extends MessagingModule {

    // instance variables
    // hashtable of slots
    Hashtable ib = new Hashtable();

    // constructors
    // *******************************************************************
    /** Construct an infoboard instance.
     *
     * @param pid the process ID of de DUM
     */
    private IB(int pid) {
        initialise(pid);
        // add messagehandlers
        // put/ slot/
        // put/ slot/ value/
        // put/ slot/ value/ ground/
       addHandler("put", new MessageHandler() {
                public void handleMessage(Message msg) {
                    put(msg);
                }
            });
        // get/ slot/
        addHandler("get", new MessageHandler() {
                public void handleMessage(Message msg) {
                    get(msg);
                }
            });
        // getAttr/ slot/
        addHandler("getAttr", new MessageHandler() {
                public void handleMessage(Message msg) {
                    getAttr(msg);
```

```
            }
        });
    addHandler("getPrevious", new MessageHandler() {
            public void handleMessage(Message msg) {
                getPrevious(msg);
            }
        });
    // verified/ slot/ value/
    addHandler("verified", new MessageHandler() {
            public void handleMessage(Message msg) {
                verified(msg);
            }
        });
    // remove/ slot/
    addHandler("remove", new MessageHandler() {
            public void handleMessage(Message msg) {
                remove(msg);
            }
        });
    // clear/
    addHandler("clear", new MessageHandler() {
            public void handleMessage(Message msg) {
                clear();
            }
        });
    // show
    addHandler("show", new MessageHandler() {
            public void handleMessage(Message msg) {
                show();
            }
        });
}


// instance methods for message handling
// *******************************************************************

private void put(Message msg) {
    String slotName = msg.body()[1];
    String value = msg.body().length > 2 ? msg.body()[2] : "";
    String ground = msg.body().length > 3 ? msg.body()[3] : "";
    // if not exist, create
    if (!ib.containsKey(slotName))
        ib.put(slotName, new Slot(slotName));
    Slot slot = (Slot) ib.get(slotName);
    if (value.equals(""))
        slot.putAskedValue();
    else if (ground.equals(""))
        slot.putUserValue(value);
    else
        slot.putGroundValue(value, ground);
}

private void get(Message msg) {
    String slotName = msg.body()[1];
    if (!ib.containsKey(slotName))
        ib.put(slotName, new Slot(slotName));
    Slot slot = (Slot) ib.get(slotName);
    String[] body = {slotName, slot.getValue()};
    reply(msg, "Slot", body);
}

private void verified(Message msg) {
    String[] body = msg.body();
    String slotName = body[1];
    String value = body[2];
    if (ib.containsKey(slotName)) {
        Slot slot = (Slot) ib.get(slotName);
        slot.verified(value);
    }
}

private void getAttr(Message msg) {
    String slotName = msg.body()[1];
```

```
        if (!ib.containsKey(slotName))
            ib.put(slotName, new Slot(slotName));
        Slot slot = (Slot) ib.get(slotName);
        String[] body = {slotName, slot.getAttr()};
        reply(msg, "SlotAttr", body);
    }

    private void getPrevious(Message msg) {
        String slotName = msg.body()[1];
        if (!ib.containsKey(slotName))
            ib.put(slotName, new Slot(slotName));
        Slot slot = (Slot) ib.get(slotName);
        String[] body = {slotName, slot.getPrevious()};
        reply(msg, "PreviousAttr", body);
    }

    private void remove(Message msg) {
        String slotName = msg.body()[1];
        if (ib.containsKey(slotName))
            ib.remove(slotName);
    }

    private void clear() {
        ib = new Hashtable();
    }

    private void show() {
        System.out.println("Infoboard:");
        for(Enumeration e = ib.keys(); e.hasMoreElements(); ) {
            String slotName = (String) e.nextElement();
            Slot slot = (Slot) ib.get(slotName);
            System.out.println(slot);
        }
    }

    // ****************************************************************
    /** The main methode run when you start the information board.
     *
     * @param args a string giving the process ID
     */
    public static void main(String args[]) {
        //set debug level
        Debug.level(3);
        int pid = 0;
        try {
            pid = Integer.parseInt(args[0]);
        }
        catch(Exception e) {
            System.err.println("Infoboard: pid expected as argument, " +
                               "program stopped!");
            System.exit(1);
        }
        IB ib = new IB(pid);
        try {
            while (ib.server != null) {
                try {
                    ib.listen();
                }
                catch (UnknownMessageException e) {
                    System.err.println("Infoboard: " + e.getMessage());
                }
                catch (MalformedMessageException e) {
                    System.err.println("Infoboard: " + e.getMessage());
                }
            }
        }
        finally {
            ib.stop();
        }
    }

}
//
```

File: ../alparon/infoboard/Slot.java //

```
package alparon.infoboard;

import java.util.*;

/** The slot. A slot is a place on the information board where values can
 * be stored.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
class Slot {

    // instance variables
    // *******************************************************************

    // name of slot
    protected String name;
    // contains a list of entries
    protected List history;


    // constructors
    // *******************************************************************

    Slot(String name) {
        this.name = name;
        this.history = new Vector();
        Entry entry = new Entry();
        this.history.add(entry);
    }


    // instance methods
    // *******************************************************************

    void putUserValue(String value) {
        Entry oldEntry = (Entry) history.get(history.size() - 1);
        String oldValue = oldEntry.getValue();
        if (!value.equals(oldValue)) {
            Entry entry = new Entry(value);
            history.add(entry);
        }
        else {
            oldEntry.addGround("user");
        }
    }

    void putGroundValue(String value, String ground) {
        Entry oldEntry = (Entry) history.get(history.size() - 1);
        String oldValue = oldEntry.getValue();
        if (!value.equals(oldValue)) {
            Entry entry = new Entry(value, ground);
            history.add(entry);
        }
        else {
            oldEntry.addGround(ground);
        }
    }

    void putAskedValue() {
        Entry entry = new Entry(true);
        history.add(entry);
    }

    String getValue() {
        int size = history.size();
        if (size > 0) {
            Entry entry = (Entry) history.get(size - 1);
            return entry.getValue();
        }
        else { // this can not happen, allways empty entry in history
            return "this is not supposed to happen";
        }
```

```
    }

    String getAttr() {
        int size = history.size();
        if (size > 0 ) {
            Entry entry = (Entry) history.get(size - 1);
            return entry.getAttr();
        }
        else { // this can not happen, allways empty entry in history
            return "this is not supposed to happen";
        }
    }

    String getPrevious() {
        int size = history.size();
        if (size > 1) {
            Entry entry = (Entry) history.get(size - 2);
            return entry.getAttr();
        }
        else { // there is only one entry
            return "there is no previous entry";
        }
    }

    void verified(String value) {
        Entry entry = (Entry) history.get(history.size() - 1);
        if (value.equals(entry.getValue()))
            entry.verified();
    }


    // auxiliary methods
    // ********************************************************************

    // overriding Object.toString()
    public String toString() {
        return name + ": " + history.toString();
    }
}
//
```

File: ../alparon/infoboard/Entry.java //

```
package alparon.infoboard;


/** Entry: a certain string value with attributes.
  *
  * An Entry is either empty or has a value.
  *
  * An empty Entry has a value of "". It can be asked for or not.
  *
  * An Entry with a value is introduced either by the system or by the
  * user. It can be verified and has a ground. The ground specifies on what
  * the value is based (user-intrduced, default-value by the system, etc.).
  * The default ground is "user" (value is introduced by the user).
  *
  * new Entry(): creates a default Entry, empty and not asked for.
  *
  * new Entry(boolean): creates a empty Entry, asked for if parameter is
  * true.
  *
  * new Entry(String): creates an Entry with the supplied value, with the
  * ground "user" and not verified
  *
  * new Entry(String, String): creates an Entry with the supplied value,
  * with the supplied ground and verified.
  *
  *
  * @author Niels Andeweg
  * @version 1.0
  */
class Entry {

    // instance variables
    // ******************************************************************

    // value of entry
    protected String value = "";
    protected boolean empty = true;
    protected boolean asked = false;
    protected boolean verified = false;
    protected String ground = "";
    // turnnumber of creation or last modification
    protected int turn = 0;


    // constructors
    // ******************************************************************

    Entry() {}

    Entry(boolean asked) {
        this.asked = asked;
    }

    Entry(String value) {
        this(value, "user");
    }

    Entry(String value, String ground) {
        this.empty = false;
        this.value = value;
        this.ground = ground;
    }


    // accessors
    // ******************************************************************
    String getValue() {
        return value;
    }

    String getAttr() {
        return toString();
    }
```

```
    boolean isEmpty() {
        return empty;
    }

    boolean isAskedFor() {
        return asked;
    }

    boolean isVerified() {
        return verified;
    }

    String ground() {
        return ground;
    }


    // modifiers
    // ********************************************************************

    void verified() {
        verified = true;
    }

    void verified(boolean verified) {
        this.verified = verified;
    }

    void addGround(String newGround) {
        if (ground.indexOf(newGround) == -1) {
            ground = ground + ", " + newGround;
        }
    }


    // auxiliary methods
    // ********************************************************************

    // overriding Object.toString()
    public String toString() {
        if (empty) {
            if (asked)
                return "empty(asked)";
            else
                return "empty";
        }
        else {
            String verified = "";
            if (this.verified)
                verified = ", verified";
            return value + "(" + ground + verified + ")";
        }
    }
}
//
```

Appendix G

Java code of the send and receive module

File: ../alparon/test/SRModule.java //

```
package alparon.test;
import alparon.lib.*;
import java.io.*;
import java.util.*;
import COM.stevesoft.pat.Regex;


/** The send and receive module. Thsi module can be used to send and
 * receive arbitrary messages. Predefined messages to send can be read
 * from a file. Give '?' for the modulemane and you get an overview of
 * the predefined messages. Give '*' for the modulename and you can
 * specify a file from where to read predefined messages.
 *
 * @author Niels Andeweg
 * @version 1.0
 */
public class SRModule extends MessagingModule implements Runnable {

    private Thread sendThread = new Thread(this, "sendThread");
    private Thread receiveThread = new Thread(this, "receiveThread");
    //    private Hashtable msgs = new Hashtable();
    private Vector msgs = new Vector();

    private SRModule(int pid){
        initialise(pid);
        readMsgs("messages");
        showMsgs();
        sendThread.start();
        receiveThread.start();
    }

    private static Regex msgExpr =
        new Regex("(\\w+):\\s*");
    private static Regex elemExpr =
        new Regex("([\\w|\\(|\\)|\\*|\\:|\\s]+?)\\s*(?=,|$)");
    private static Regex delimExpr =
        new Regex(",\\s*");

    private void readMsgs(String filename) {
        //msgs = new Hashtable();
        msgs = new Vector();
        try {
            BufferedReader in = new BufferedReader(new FileReader(filename));
            String line = in.readLine();
            while (line != null) {
                if (msgExpr.search(line)) {
                    String key = msgExpr.stringMatched(1);
```

```
                line = msgExpr.right();
                Vector msg = new Vector();
                msg.add(key);
                while (elemExpr.search(line)) {
                    String elem = elemExpr.stringMatched(1);
                    msg.add(elem);
                    line = elemExpr.right();
                    if (delimExpr.search(line)) {
                        line = delimExpr.right();
                    }
                }
                msgs.add(msg);
            }
            line = in.readLine();
        }
    }
    catch(Exception e) {}
}

private void readNewMsgs() {
    try {
        System.out.print("Filename for messages: ");
        byte[] buff = new byte[100];
        int lbuff = System.in.read(buff);
        String filename = new String(buff, 0, lbuff-1);
        readMsgs(filename);
        showMsgs();
    }
    catch(Exception e) {}
}

private void showMsgs() {
    System.out.println();
    for(Iterator it = msgs.iterator(); it.hasNext(); ) {
        Vector msg = (Vector) it.next();
        Iterator mit = msg.iterator();
        String key = (String) mit.next();
        StringBuffer msgString = new StringBuffer(key);
        msgString.append(": ");
        while (mit.hasNext()) {
            String elem = (String) mit.next();
            msgString.append(elem).append(", ");
        }
        System.out.println(msgString.substring(0, msgString.length() - 2));
    }
}

public void run(){
    if (Thread.currentThread() == sendThread ) {
        try {
            sendMsgs();
        }
        catch(Exception e) {};
    } else {
        receiveMsgs();
    }
}

private void sendMsgs() throws IOException {
    byte[] buff = new byte[100];
    int lbuff;
outer: do {
    String module, msgname, body[];
    Vector bodyvector = new Vector();
    System.out.print("Module name: ");
    if (server == null) break outer;
    lbuff = System.in.read(buff);
    module = new String(buff, 0, lbuff-1);
    if (module.equals("?")) {
        showMsgs();
        continue;
    }
    if (module.equals("*")) {
        readNewMsgs();
```

```
            continue;
        }
        Vector msg = predefined(module);
        if (msg != null) {
            module = (String) msg.get(1);
            msgname = (String) msg.get(2);
            body = new String[msg.size() - 3];
            for(int i = 3; i < msg.size(); i++) {
                body[i - 3] = (String) msg.get(i);
            }
        }
        else {
            System.out.print("Message name: ");
            if (server == null) break outer;
            lbuff = System.in.read(buff);
            msgname = new String(buff, 0, lbuff-1);
            System.out.print("Message body: ");
            if (server == null) break outer;
            lbuff = System.in.read(buff);
            while (lbuff > 1) {
                bodyvector.add(new String(buff, 0, lbuff-1));
                System.out.print("Message body: ");
                if (server == null) break outer;
                lbuff = System.in.read(buff);
            }
            body = new String[bodyvector.size()];
            for(int i = 0; i < bodyvector.size(); i++) {
                body[i] = (String) bodyvector.get(i);
            }
        }
        send(module, msgname, body);
        try {
            sendThread.sleep(1500);
        }
        catch(Exception e) {};
    } while (server != null);
    }

    private void receiveMsgs() {
        Message receivedMsg;
        MessageHandler msgHandler;
        while (server() != null) {
            receivedMsg = receive();
            msgHandler = (MessageHandler)
                msgHandlers.get((receivedMsg.body())[0]);
            if (msgHandler != null) {
                try {
                    msgHandler.handleMessage(receivedMsg);
                }
                catch(Exception e) {}
            }
        }
    }

    private Vector predefined(String msgkey) {
        for(Iterator it = msgs.iterator(); it.hasNext(); ) {
            Vector msg = (Vector) it.next();
            String key = (String) msg.get(0);
            if (key.equals(msgkey)) {
                return msg;
            }
        }
        return null;
    }

    public void stop() {
        try {
            if (server != null) {
                server.stop();
                Debug.println(settings.gmn() + " stopped.", 2);
            }
            server = null;
            //kill sendThread
            //sendThread.stop();
```

```
        }
        catch (IOException e) {
            System.err.println("Error stopping " + settings.gmn());
            System.err.println(e.getMessage());
        }
    }

    public static void main(String[] args) throws IOException {
        Debug.level(9);
        SRModule srm = new SRModule(Integer.parseInt(args[0]));
    }
}
//
```

Bibliography

[Allen, 1995] J. Allen (1995), *Natural Language Understanding*, The BenjaminCummings Publings Company, Inc., second edition.

[Andeweg, 1999] A.C. Andeweg (1999), Dialogue Control Literature survey, Technical report, Alparon, Delft University of Technology.

[Aust and Oerder, 1995] H. Aust and M. Oerder (1995), Dialogue Control in Automatic Inquiry Systems, in *Corpus–based Approaches to Dialogue Modelling*, 9th Twente Workshop on Language Technology, pp. 45–49, University of Twente.

[Carletta et al., 1995] J. Carletta, A. Isard, S. Isard, J. Kowtko, G. Doherty-Sneddon, and A. Anderson (1995), The Coding of Dialogue Structure in a Corpus, in *Corpus–based Approaches to Dialogue Modelling*, 9th Twente Workshop on Language Technology, pp. 25–34, University of Twente.

[Churcher et al., 1997] Gavin E. Churcher, Eric S. Atwell, and Clive Souter (1997), Dialogue Management Systems: a Survey and Overview, Technical Report 97.06, School of Computer Studies, University of Leeds.

[Cohen, 1994] P.R. Cohen (1994), Models of Dialogue, in *Cognitive Processing for Vision & Voice*, 4th NEC Research Symposium, pp. 181–204, NEC & SIAM.

[Cole et al., 1997] Ronald Cole, Joseph Mariani, Hans Uszkoreit, Annie Zaenen, and Victor Zue, editors (1997), *Survey of the state of the art in human language technology*, Cambridge University Press.

[Flanagan, 1997] David Flanagan (1997), *Java in a nutshell*, O'Reilly, second edition.

[Jokinen, 1996] Kristiina Jokinen (1996), Cooperative Response Planning in CDM: Reasoning about Communicative Strategies, in *Dialogue Management in Natural Language Systems*, 11th Twente Workshop on Language Technology, pp. 159–168, University of Twente.

[Oldervoll, 1998] Thomas Oldervoll (1998), Information Presentation in a Automatic Speech Processing System, Master's thesis, Alparon, Delft University of Technology.

[Rothkrantz et al., 1996] L.J.M. Rothkrantz, R.J. van Vark, J.P.M. de Vreught, J.W.A. van Wees, and H. Koppelaar (1996), Automatische Spraakherkenning en OVR–dialogen, Technical Report Alparon series 96–01, Delft University of Technology, Knowledge Based Systems, in Dutch.

[Vark, 1997a] R.J. van Vark (1997a), Designing Dialogue Management in the Alparon Project, Technical Report Alparon series 97–01, Delft University of Technology.

[Vark, 1997b] R.J. van Vark (1997b), Dialogue Manager Framework & Test environment Source documentation, Technical Report Alparon 97–14, Delft University of Technology.

[Vark, 1997c] R.J. van Vark (1997c), DiMaCS 1.0, Technical Report Alparon series 97–12, Delft University of Technology.

[Vark, 1997d] R.J. van Vark (1997d), Task Coordinator version 1.0, Technical Report Alparon series 97–09, Delft University of Technology.

[Vark et al., 1996] R.J. van Vark, J.P.M. de Vreught, and L.J.M. Rothkrantz (1996), Classification of Public Transport Information Dialogues using an Information Based Coding Scheme, in *Workshop Dialogue Processing in Spoken Language Systems*, European Conference on Artificial Intelligence, pp. 92–99, revised paper has also appeared in Lecture Notes in Artificial Intelligence 1236, pp. 55–69, 1997.

[Vromans et al., 1999] Bram Vromans, Robert Jan van Vark, B. Rueber, and A. Kellner (1999), Extending the SUSI System with Negative Knowledge, in *Proceedings of Eurospeech 1999 (to appear)*, Budapest, Hungary.