WP6D2—Dialogue Manager Design Documentation

Eric Bilange, Wieland Eckert,
Norman Fraser, Nigel Gilbert,
Marc Guyomard,
Paul Heisterkamp,
Jean-Yves Magadur,
Scott McGlashan, Jacques Siroux,
Jutta Unglaub, Robin Wooffitt
and Nick Youd

July 1991

TITLE	:	WP6D2—Dialogue Manager Design Documentation
PROJECT	:	P2218 - SUNDIAL
AUTHOR	:	Eric Bilange, Wieland Eckert, Norman Fraser, Nigel Gilbert, Marc Guyomard, Paul Heisterkamp, Jean-Yves Magadur, Scott McGlashan, Jacques Siroux, Jutta Unglaub, Robin Wooffitt and Nick Youd
ESTABLISHMENT	:	Cap Gemini Innovation, Daimler-Benz Forschunginstitut, IRISA, Logica Cambridge Ltd, University of Erlangen, University of Surrey
ISSUE DATE	:	July 1991
DOCUMENT ID	:	
VERSION	:	4
STATUS	:	Final
WP NUMBER	:	6
LOCATION	:	Mikhail
KEYWORDS	:	Deliverable, Dialogue, Design, Documentation

Abstract

This report is the second deliverable from Work Package 6: Dialogue Management. It consists of design documentation for each of the modules of the dialogue manager of the intermediate Sundial demonstrator system. Each chapter is organised in a common format. An initial Overview describes the overall functionality of the module. A section on Scope delimits the behaviour of the module. The Principles section describes the theoretical basis of the module. A section on Algorithms describes how the functions or the module are carried out in the implementation. The knowledge bases used by the module are described next. What needs to be done to 'customise' these knowledge bases to work with a specific application (e.g. flight enquiries or reservations, or train timetable enquiries) is indicated in the next section. Finally, a section describes the implementation. An appendix provides an annotated trace of the internal activity of the Dialogue Manager when handling a simple dialogue.

The design described in this document is an implementation of the functional specification contained in the first deliverable from WP6: the *WP6 Dialogue Manager Functional Specification* of July 1990 (**?**)), to which reference should be made for details of the architecture of the dialogue manager.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This document details the design and implementation of the 'Intermediate Demonstration' version of the Sundial Dialogue Manager. The requirements for the Dialogue Manager were described in the first deliverable from Work Package 6 (**?**)), which also indicated the relationship of the Dialogue Manager to the rest of the Sundial system.

Briefly, the Dialogue Manager is responsible for all pragmatic and conversational aspects of the system, including organising the dialogue, interacting with the application database which holds task specific knowledge, and maintaining the current beliefs of the system about the user's requests and situation.

As noted in **?**), the objective of the SUNDIAL project is to design and implement prototype computer systems which will interact with users over telephone lines to answer queries within restricted task domains such as flight enquiries, flight reservations and train enquiries. Previous designs for speech information service systems have not realised the need for a Dialogue Manager—they have linked a database directly to a language parsing component.

The problems that arise with such designs provide a rationale for the incorporation of a Dialogue Manager within the SUNDIAL system. The general problem is that without some means of managing the interaction between the system and the user, the resulting interaction is 'unnatural' from the user's point of view—the system does not conduct a dialogue with the user in the same, or similar, way as people conduct information service conversations with each other. This 'unnaturalness' is evidenced by the lack of interactional properties such as the following:

- **Extended Interaction**. The largest unit of interaction is frequently a single question answer pair: the user asks a question and the system gives an answer. Naturally occurring dialogues, on the other hand, are extended across more than one sand these dialogues display a sequential organization. For example, dialogues consist of an opening sequence, the body of the dialogue and a closing sequence.

- **Indexicality** Utterances are interpreted (and produced) in a 'context-independent' manner. In natural dialogue interpretation is 'context-dependent': participants oriented to the content and structure of previous utterances, or past context, in order to determine the (situated) significance of the current utterance. Ellipsis, for example, is commonplace in natural dialogue since participants make use of past context in order to interprete 'incomplete' utterances.

- **Mixed Initiative** The initiative is the sole responsibility of one participant. In natural dialogue, either participant can take the initiative. For example, an agent may initiate a clarification of the user's request.

- **Co-operative Reponses** If the system fails to find the information requested by the user, a simple negative response is generated. In natural dialogues, co-operative responses such as suggestive answers may be given in such circumstances.

- **Repair Strategies** If the system is unable to interprete an acoustic sequence or fails to understand an utterance, the result is a simple error message. In natural dialogues, one particpant may correct the problem so allowing the dialogue to the dialogue can continue.

Incorporating this sort of functionality within an information service system is a manifestation of a user-centred design strategy: rather than simply design the system from point of view of information *exchange*, the functionality of the system is increased so as to give an *interaction* between system and user, an interaction which displays properties of natural conversation.

The main purpose of the Dialogue Manager is therefore to increase the extent to which the dialogue is natural, comfortable, comprehensive and comprehensible for the user.

The overall architecture of the SUNDIAL system is shown in figure 1.1.

The Dialogue Manager communicates with the Linguistic Processor (LP), receiving from it semantic and other information derived from the user's utterances, and sending back predictions about expected future utterances. It also communicates with the Message Generator (MG), sending it messages for conversion into system utterances, and with the application database. Messages to and from the LP and MP are encoded in the project's "Semantic Interface Language".

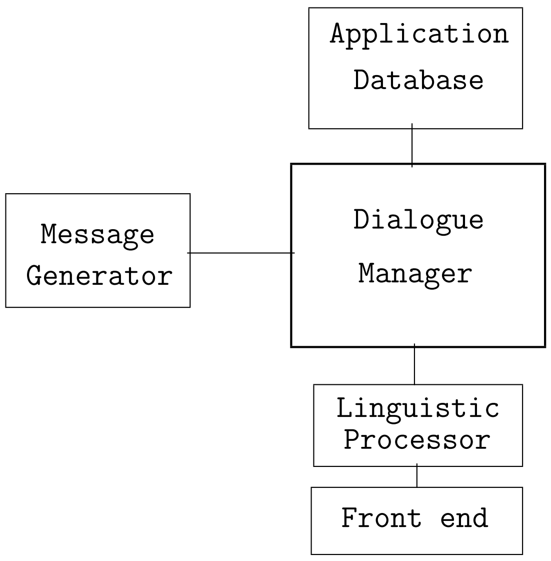

Figure 1.1: The SUNDIAL Information Service System

Communication with the application database is in the database's own query language[1].

The Dialogue Manager is intended to be 'generic', that is, it is intended that only data, not code, should need to be changed in order to switch the Manager to information enquiries for another domain. The succcess of this is shown in the ease with which the current implementation can be changed from dealing with flight enquiries (for the UK demonstrator) to flight reservations (for the French demonstrator) and to train enquiries (for the German demonstrator). Some experiments at Logica with the very different domain of Home Banking have confirmed that changing domains is not complicated.

The Dialogue Manager is also language independent. Indeed, the Dialogue Manager does not need to be aware of the language understood or used by the system, because all its input and output are at the semantic level.

Because of its application and language independent nature, the same Dialogue Manager is being used for the French, German and English language demonstrators. It is hoped that it will also be used for the next version of the Italian demonstrator[2].

[1]For the Intermediate Demonstration, the application database has been simulated using a 'flat' file or Prolog terms, and Prolog predicates are used for searching the file.

[2]The Italian partners did not plan to contribute to Work Package 6 or to the design of the Dialogue Manager until after the date of this deliverable

The design and implementation of the Dialogue Manager has been a truly European effort, with portions being specified and coded by British, French and German partners. The problems of managing such a distributed development task were lessened slightly by the fact that the architecture of the Dialogue Manager divides it up into five modules, which communicate by message passing superintended by a very simple scheduler, or 'postie'. Nevertheless, coordination of the development has not been easy. Those involved deserve some acknowledgement for their willingness to cooperate with other developers even when communication was only possible through a sometimes unreliable email service, and their willingness to spend time away from home and family at the three 'integration weeks' which were held to try to speed up the development process.

Most of this document is concerned with describing how the functional specification proposed in **?**) has been implemented in the Intermediate Demonstrator. There is one chapter for each of the modules which together compose the Dialogue Manager. The chapters all follow a common format. An initial Overview describes the overall functionality of the module. A section on Scope delimits the behaviour of the module. The Principles section describes the theoretical basis of the module. A section on Algorithms describes how the functions or the module are carried out in the implementation. The knowledge bases used by the module are described next. What needs to be done to 'customise' these knowledge bases to work with a specific application (e.g. flight enquiries or reservations, or train timetable enquiries) is indicated in the next section. Finally, a section describes the implementation.

1.1 The Dialogue Module

The dialogue module is responsible for maintaining the coherence of the dialogue. Thus it is responsible for:

- dialogue structure,
- turn management,
- motivating system utterances,
- predicting the caller's next utterance,
- interpreting the caller's utterances,
- extending the dialogue history and,

- adjusting the dialogue strategy according to the current dialogue success level.

At the heart of the dialogue module is a dialogue grammar which describes at a fairly high level of generality the range of possible dialogue structures which can be handled by the Sundial system.

1.2 The Belief Module

The belief module is at the center of the semantic interpretation process, as well as being the major mechanism whereby discourse coherence (at a level within the utterance) is maintained. The belief module interprets input from the user and assists in the generation of the system's utterances. It does this by maintaining a structured model of knowledge instances, access to which varies both contextually (according to the current foci) and modally (according to the world view chosen). This model (alternatively referred to as the *belief model*, and the *discourse model*) contains representations of that material referred to in conversation, as well as its closure under inference, and various hypothetical extensions which may aid the reasoning/planning process. So far as other components of the system are concerned, the belief module constitutes a dynamic knowledge base, to which queries and updates may be addressed.

1.3 The Task Module

The present implementation of the generic Task Module is able to:

- Establish Tasks and Task Parameters,
- Check Specificity,
- Access the Application Database,
- Present Responses from the Database,
- Relax Constraints on the User's Requests,
- Make Task Specific Inferences,
- Supply Default Knowledge and
- Check Reliability.

The general idea has been to provide a generic module which can be used for different applications by providing the respective adaptations. The design of the Task Module is based on the principle of a clear separation between application dependent and generic, ie. application independent knowledge. Thus the interfaces between the generic mechanisms and the application database on the one hand and the task dependent knowledge on the other hand are clear cut. There is a layer of generic data base accessing procedures, which make the representation of application dependent knowledge transparent to the generic modules. When changing the format of knowledge representation only the respective accessing procedures have to be adapted accordingly. Task dependent knowledge can be defined in a rather flexible way, since the representation has been kept in a declarative mode, apart from rules for inferences, consistency checking and constraint relaxation.

As a another general feature, the Task Module allows handling of several tasks at a time. Different task instances are being stored in a queue and processed according to the principle of first-in-first-out.

1.4 The Message Planner

The message planning module interfaces the Dialogue manager with the message generator in the Sundial system. As such, it is responsible for providing descriptions of system messages within the interface language. These descriptions are then passed to the message generation module for detailed linguistic realization and synthesis. When generation is complete, the message planning module updates the rest of the Dialogue Manager.

Global dialogue planning is carried out by the dialogue module. The rationale for this concerns predictions. Predictions about user's utterances are constructed by the dialogue module on the basis of system move information sent to the message planning module. If the message planning module were to perform dialogue planning, i.e. expand, contract or add system moves, then the predictions exported to the linguistic processing module (via the linguistic interface module) may be inappropriate for the system move information generated by the message planning and message generation modules. Given the design of the dialogue module, our strategy has been to remove global dialogic planning from the message planning module and thereby ensure that the predictions are appropriate to what the dialogue module expects.

With global semantic planning, i.e. building a semantic representation for each system move, while the message planning module decides, on the basis of its static knowledge definitions, how objects are to be described, the construction of semantic representations is carried out by the belief module. The reason for this is that all the relevant semantic information is contained in the belief model.

Finally, local planning—deciding whether objects are to be given a linguistic realization and, if so, whether they are to be anaphoric—is carried out by the message generation module itself. The message planning module passes the message generation module all (Dialogue Manager) information relevant to this aspect of generation: in particular, prosodic, ellipsis and accessiblity information about the system utterance as well as the current linguistic history are passed to the message generation module.

1.5 The Linguistic Interface

The Linguistic Interface is the module responsible for managing the interface between the Dialogue Manager and the Linguistic Processor (LP). It maintains a local database, carries out structural transformations, and superintends communications between the Dialogue Manager and the LP.

The Linguistic Interface has a number of basic functions. These can be grouped under the following headings:

- linguistic history management
- prediction building
- interface management

A record of everything that has been said so far is kept in the Linguistic History. This is currently used for two main purposes. Firstly, it is used in order to add detail to next-utterance predictions. The Dialogue Module predicts the dialogue acts and task/lexical types and objects which are expected to occur in the next utterance. The Linguistic Interface examines the History to see if any of the objects have been mentioned before and, if so, it retrieves the referring expressions used to identify them and builds these into the prediction. Secondly, the History is used in order to retrieve linguistic forms for reuse by the Message Planner/Message Generator.

The Dialogue Module sends the Linguistic Interface a set of predictions consisting of one or more structures. The Linguistic Processor expects a corresponding list of one or more predictions in a somewhat different form, so the Linguistic Interface carries out the transformations.

The French and British Linguistic Processors are implemented in Quintus Prolog. This allows the interface to be implemented by means of simple Prolog predicate calls. The German Linguistic Processor is implemented in Lisp. Interactions take place via the Quintus Prolog foreign language interface to an intermediate C program. In order to provide a maximally accessible interface, the Linguistic Interface also supplies the option of writing its output to and reading its input from text files.

1.6 The 'Postie'

The modules of the Dialogue Manager communicate solely by passing messages from one to another. The 'Postie' (called after the colloquial name for a "post person") is a very simple module which administers the message passing. Because the Postie is at the hub of the Dialogue Manager and because all messages between modules pass through it, it is also the place for additional code to help with tracing, debugging and testing the Manager as a whole.

The Postie is intentionally very simple. The temptation to load it with any functionality beyond passing on the messages it receives has been resisted, in order not to compromise the modularity of the architecture.

The Postie receives messages from the modules and places them into a first–in, first–out (FIFO) buffer. Messages are taken from the front of this buffer and despatched to their destination modules. This process continues until the buffer is empty. The Postie then does nothing until another message is received.

The Postie starts off the Dialogue Manager's operation by clearing its message buffer and sending an 'init' message to one of the modules (arbitrarily chosen to be the dialogue module). This module processes the message and generates further messages destined for other modules. Thus the endless processing of messages is begun, to continue forever (or until one of the modules sends a 'closedown' message to the Postie).

Chapter 2

The Dialogue Module

2.1 Overview

This chapter describes the theoretical and implementation principles of the Dialogue Module. First of all, it is important to note that a significant choice has been made from the last deliverable [WP6D1000]: the 'I/O Decider' and the Dialogue Module have been merged into one. The reader is referred to [WP6D1000 Chap 5, pp. 86-87] to see the justification for this arrangement. However, this doesn't mean that the functionalities that this submodule was responsible for have disappeared.

2.2 Scope

In Wp6D1000 section 5.4.1 we said:

> The dialogue module is responsible for maintaining the coherence of the dialogue. Thus it is responsible for:
>
> - dialogue structure,
> - turn management,
> - motivating system utterances,
> - predicting the caller's next utterance,
> - interpreting the caller's utterances,
> - extending the dialogue history and,

- adjusting the dialogue strategy according to the current dialogue success level.

In the present version of the dialogue module, all these functionalities are more or less covered. More or less means that we have, during our implementation and tests discovered some new functionalities which were not stated in our first functional specifications.

2.3 Principles

As we said in the WP6D1000 (chap 5, section 5.4.1.1 page 93) *at the heart of the dialogue module is a dialogue grammar which describes at a fairly high level of generality the range of possible dialogue structures which can be handled by the Sundial system.* This is this option we considered in developing our module but also we considered criticisms about this approach (**?**)). The fact is that first faced to the analysis of the WP3 dialogues and second, faced to the linguistic theory we used (**?**)), it appeared that a clear conversational organization emerged. An open question is of course whether this organization is universal (since we didn't tested it for languages other than familiar European ones). This organization has been captured more with the concept of *conversational rules* rather than on a pure dialogue grammar. This is probably because the word *grammar* has a non-wishful connotation: a grammar is supposed to recognize (or build a syntactic representation) a pattern and a dialogue grammar is supposed to do something else since dialogue is dynamic and it is not our intention to build a system which builds the dialogue structure after the dialogue occurred but during the dialogue it should build that structure. This is certainly why some linguistic theories are not always receivable because they are concerned with Conversation Analysis and therefore relevant for *a posteriori* analyses but not always adapted to *a priori* analysis and even less well-suited for making predictions. Therefore, it is not a contradiction to say that our dialogue organization is based on a linguistic theory, since it was useful for the corpora analyses, but adapted to first dynamic aspects and second to computational tractability.

2.3.1 Basis

In the light of corpora analyses, task oriented dialogues (such as flight reservation, time table enquiries, appointment taking ...) contain negotiations organized in two possible generic patterns:

1. Negotiation opening + Reaction

2. Negotiation opening + Reaction + Evaluation of the negotiation

Moreover negotiations may enter sub–negotiations for mainly two possible reasons. First the topic of a negotiation can be complex such that several steps are necessary to solve the full negotiation. Second, negotiations may need clarification sequences for explanations or misunderstandings, e.g. some negotiations are repair sequences.

A dialogue model should take into account this general observation keeping in mind the task that must be achieved. An oral dialogue system should also take into consideration speech understanding limitations (?)) mainly due to acoustic confusion e.g. repairing techniques to avoid misleading situations due to misunderstandings should be provided (?)).
Finally, the model must be habitable — not rigid — so that the two locutors can take initiative whenever they want or need to.

This basis lead us to define the structural description of a dialogue that refines the concept of negotiation. It consists of four levels similar to the linguistic model of Roulet and Moeschler (?); ?)): *transaction* (T), *exchanges* (E), *interventions* (It), and *dialogue acts* (DA). The structural principle follows the following description:

$$\begin{array}{lcl} T & ::= & \{E\}^+ \\ E & ::= & \{It\{E\}^*\}^+ | \{E\}^+ \\ It & ::= & \{DA\}^+ \end{array}$$

In each level specific functional aspects are assigned:

- *Transaction level* : task oriented dialogues are a collection of transactions. For example, in the domain of travel planning, transactions could be : book a one-way, a return, etc. The transaction level is tied to plan achievement. They may also be meta-plans such as “problem formulation” or “problem resolution” ...
- *Exchange level* : transactions are achieved through exchanges. Exchanges may be embedded (sub-exchanges). Exchange topics may concern task objects or the dialogue itself (meta-communication).
- *Intervention level* : An exchange is made up of interventions. Three possible illocutionary functions[1] are attached to interventions: *initiative* (I), *reaction* (R), and *evaluation* (Ev).

[1]This *speech act theory* terminology will be explained in section 2.3.2.

- *Dialogue acts* : A dialogue act (*DA*) could be defined as a *speech act* augmented with structural effects on the dialogue (thus on the dialogue history) (**?**)). There are one or more DAs in an intervention. Possible secondary DAs denote the argumentation attached to main ones. For the purpose of this paper, we will not describe argumentation analysis.

  DAs represent the minimal entities of the conversation.

Now we will describe each dialogue entity.

2.3.2 Dialogue acts

If we assume that when uttering something a locutor performs an action, then minimal dialogue constituents are **dialogue acts**. This concept is derived from the one of speech acts. This is not particularly new and it is not our intention to introduce the full state of the art concerning *speech acts*. We refer the reader to (**?**)) who offers an interesting study about speech acts. Levinson concludes with: "speech acts should be defined according to the modification effects on the context". An interesting development on this claim is (**?**)). For Bunt, the context is modelled by the knowledge of the agents, more precisely what they know and what they suspect. Thus (what he calls) a dialogue act is defined by its appropriateness conditions and effects, denoted by the couple <effects on the Hearer, effects on the Speaker>. The context is defined by the couple <estate of H, estate of S>. A dialogue act may occur for three reasons: the speaker wants to know something, wants to make known something or knows the hearer wants to know something. Then Bunt describes various communicative functions for questions, informs, and answers[2]. He identified 22 communicative functions (wh-question, yn-question, alt-question, wh-answer, yn-answer, inform . . .).

We enriched the context notion with the current dialogic situation, e.g. the dialogue structure. Actually, it is clear that the dialogue continuation after the use of a wh-question is neither the same that the one after a yn-question nor the one after an order. Thus, a dialogue act has also effects on the dialogue structure, which indirectly means that, taken together they have effects on the flow of the conversation. Moreover, as an appropriateness condition a dialogue act can be use in certain dialogue situation depending on the previously uttered dialogue acts.

Introducing the dialogue structure as a third parameter for contextual definition is important because this also genuinely captures the effects of what Grosz

[2]The term "illocutionary force" is used in the speech act theory to refer to distinctions like "questions", "answers", etc. The term "communicative function" is preferred for the mathematical sense of function and because it is more concrete than "illocutionary force".

and Sidner called “linguistic indicators” (“woops”, “now back to”) (?)). For us, these utterance fragments correspond to kind of **meta-communicative dialogue acts**.

For us, a dialogue act is made up of the following information:

- Appropriateness conditions:
  - A task function,
  - A dialogue situation,
  - Estate of H and S.
- Effects:
  - Effects on H and S,
  - Effects on the task,
  - Structural effect(s).

The *task function* is the kind of task goal the dialogue act is able to ‘perform’. The word ‘perform’ here means: the dialogue act transfer the knowledge to the addressee that the speaker wants to achieve this specific task goal. It may be empty as far as some dialogue acts are not concerned with the task.
The appropriateness conditions offer the possibility to deeply understand an utterance. An illustration of that can be found in Beun (?)) with the problem of declarative questions. For example, the sentence “flight BA753” could be both a question or an answer and then the understanding of such and act depends on the value of the communicative function. As Beun illustrated, one major difficulty in dialogue is to affect the correct communicative function(s) to an utterance. In our definition of a dialogue act, once the appropriateness conditions are committed then there is no understanding problem. However, the most difficult point is certainly to correctly characterize the dialogue situation. We will show through the description of the rest of our model how this is solved.

We provide in appendix the list of dialogue acts and their definition. No matter of how many dialogue acts we have defined, but the main aspect is that the more we have dialogue acts the more we are able to endow the system with dialogue sensibility.
Defining a list of speech acts is certainly dependent on how the computational model of the dialogue manager is envisaged. For example, in the Allen, Cohen and Perrault, the motivation was clearly based on belief representation in a formal framework: designing a new theory. Other authors (?), ?), ?)) seem to have been less influenced by a pure theoretical motivation but more influenced by a pragmatic approach based on corpora analyses. Another important divergence

point could be the type of application: is it an information seeking application or task oriented one. For example, the act "recapitulation" in the LOKI project is clearly due to the task and has nothing to do in the Allen et al. application on train enquiries in a station. Also, the act "renup" in the YP is motivated by the fact that multiple answers are possible which is not envisaged in the LOKI project. One possible summary could be:

> The more we define dialogue acts for an application, the more we provide functionalities (i.e. dialogue possibilities).

Another dimension to consider is the differences between oral dialogues and written dialogues. In the latter the percentage of phatic management (or dialogue control acts (**?**))) is less important than in oral. This is more obvious with human machine interaction due to non robust speech understanding systems and non entirely satisfying synthesis systems which involve more clarification subdialogues and various types of misrecognitions different from human-human interaction.

> Dealing with oral dialogues obliges us to have a precise definition of dialogue control acts and their use will be more intensive than in written dialogues.

2.3.3 The intervention level

As stated in the model introduction there are three types of interventions: initiatives, reactions and evaluations, made up of DAs. An intervention belongs to one and only one exchange. Therefore there is no isomorphic relation between a "utterance" and an intervention. An utterance may contain more than one intervention and thus belong to one or more exchanges.

Since initiatives, reactions and evaluations have precise functions, the dialogue acts they contain fulfill these functions. This induces a DA taxonomy according to the intervention categories. For the sake of simplicity, we will compare[3] these categories to Searle's (**?**)) and not introduce our taxonomy in detail with the list of our DAs.

As we will see, some interventions open exchanges. Therefore, we will also introduce in that section the notion of exchanges.

[3]This comparison is abusive since dialogue acts and speech acts have not the same definition but gives readability insights.

2.3.3.1 Initiatives

Initiatives open exchanges. As we will see an exchange carries a specific intention. Thus initiatives explicitly introduce exchange intentions. We can already distinguish two kinds of initiatives: initiatives that concern factual information and initiatives that concern communicative information. **Factual initiatives** in informative dialogues are mainly requests about facts, e.g. made of DAs whose communicative function is of type request, similar to the Searle's *directive* category, with a non empty task function. The focus spaces they manipulate are concrete world objects (the task function of the used DAs). **Communicative initiatives** concern the communication itself, therefore the possible dialogue acts are of the expressive and directive categories, and have no task function.

Moreover, the locutor who performs an initiative *is responsible for properly closing the exchange s/he opened.* This doesn't mean that the addressee must remain passive, s/he is also allowed to take initiative. This remark is important to show that free interaction is possible in this model and also that our model takes into consideration dialogue conventions that participants must respect.

2.3.3.2 Reactions

Reactions react to initiatives. This incorporates the adjacency pair paradigm (**?**)) where, for example, an answer is the second part of a pair initiated by a question. Reactions include DAs whose communicative function are of type assertive and declarative.

However, reactions are not always a simple intervention. Actually, they can be a full exchange as shown in figure 2.1.

- A_1: I offer you the flight which leaves at 10.30. Is that ok ?
  - B_1: is it a TAT flight ?
  - A_2: yes it is
  - B_2: at what time does it arrive ?
  - A_3: 11.30
- B_3: I'd like a return ticket the day after in the evening

Figure 2.1: A complex reaction

Surely, B_3 is not the answer to A_1. So what happened during the interaction, made of (B_1, A_2, B_2, A_3), is simply a digression interruption which plays the role of a reaction. Therefore, in the light of this dialogue, our model considers exchanges as possible reactions— **reacting exchanges**. In our example, there is an exchange (E_1^R) made of two exchanges: E_2 and E_3. The structure of this excerpt is shown in figure 2.2 (we intentionally exclude B_3's intervention: its place in the dialogue structure will be shown later):

E_0	I_{A_1}			*I offer you the flight which leaves at* 10.30. *Is that ok?*
	E_1^R	E_2	I_{B_1}	*is it a TAT flight?*
			R_{A_2}	*yes it is*
		E_3	I_{B_2}	*when does it arrive?*
			R_{A_3}	*at* 11.30

Figure 2.2: Example of a reacting exchange

Our principle of properly closing exchanges (see 2.3.3.1) is not violated here. E_2 is properly closed by B since s/he opened E_3 and A didn't contest the exchange shift. E_3 and E_1^R are properly closed according to B since s/he wants to shift the focus of attention by introducing the "return". However, at this stage, we cannot claim that E_0 is properly closed, since this will depend on A's next intervention. This is discussed later.

2.3.3.3 Evaluations

Evaluations evaluate the exchange they belong to. Different kinds of DAs may be used to perform evaluations such as requests (for confirmation), inform (of one's understanding), or dialogue control acts (**?**)) which may have simple cue expressions in their surface realization such as "yes", "ok", "fine" etc., simple echoes, or non linguistic surface realization such as pauses.

Firstly, evaluation are optional: for example, in E_2 and E_3 there is no evaluation. Secondly, evaluations may be full exchanges, in which case we call them **evaluating exchanges**. However, evaluations have a particular behaviour since their effect(s) are not immediate. Once an evaluation is performed, it is not possible to exactly decide on the nature of the evaluation between a simple evaluation or an evaluating exchange. For example, in the dialogue excerpt in figure 2.3, A's evaluation (the DA A_2^1) may have two possible continuations, as presented in figure 2.4[4].

[4]A_i^j stands for the jth DA in the ith A's intervention.

A_1	When would you like to leave ?
B_1	November the 13th
A_2	A_2^1 November the 13th
	A_2^2 at what time would you like to leave ?

E_0	E_1	I_{A_1}	*When would you like to leave?*
		R_{B_1}	*November the* 13*th*
		$Ev_{A_2^1}$	*November the* 13*th*
	E_2	$I_{A_2^2}$	*at what time would you like to leave?*

Figure 2.3: An open dialogue

Continuation I

A_1	When would you like to leave ?
B_1	November the 13th
A_2	A_2^1 November the 13th
	A_2^2 at what time would you like to leave ?
B_2	at 10

E_0	E_1	I_{A_1}	*When would you like to leave?*
		R_{B_1}	*November the* 13*th*
		$Ev_{A_2^1}$	*November the* 13*th*
	E_2	$I_{A_2^2}$	*at what time would you like to leave?*
		R_{B_2}	*at*10

Continuation II

A_1	When would you like to leave ?
B_1	November the 13th
A_2	A_2^1 November the 13th
	A_2^2 at what time would you like to leave ?
B_2	B_2^1 No, November the 30th
	B_2^2 at 10

E_0	E_1	I_{A_1}		*When would you like to leave?*
		R_{B_1}		*November the* 13*th*
		E_3^{Ev}	$I_{A_2^1}$	*November the* 13*th*
			$R_{B_2^1}$	*No, November the* 30*th*
	E_2	$I_{A_2^2}$		*at what time would you like to leave?*
		$R_{B_2^2}$		*at* 10

Figure 2.4: Illustration of a simple evaluation ($Ev_{A_2^1}$) and an evaluating exchange (E_3^{Ev}).

In continuation **I** ("at 10") the evaluative intervention remains an evaluation whereas in continuation **II** ("no, november the ...") it becomes an initiative of an **evaluating exchange** (shortened as E^{Ev}). Thus, when uttering A_2, A is not sure of the effect of the A_2^1 dialogue act. However, as a matter of fact, A conjectures that it could be a simple evaluation and therefore A enters a new exchange.

The example we gave here is of course an **oral** dialogue. We emphasize that the oral dimension is of prime importance since situations like continuation II are typically due to acoustic problems. That raises dialogue phenomena which don't appear in written dialogues (**?**), Chapter 2). Evaluations by echoing information (what is called echo dialogue acts) never damage the dialogue 'fluidity'. On the contrary, they enhance the level of cooperation between the two locutors for two reasons:

1. as a human factor, this underlines a mindful attitude of a locutor,

2. and this blocks possible dialogue ruptures involving complex repair strategies once a dialogue participant, sooner or later, realizes that a misunderstanding occurred. If there is a misunderstanding, as in continuation II, it is repaired as soon as the evaluation occurs[5].

One can notice also that the representation of the dialogue in continuation II doesn't respect the dialogue synchronicity ($R_{B_2^1}$ appears before $I_{A_2^2}$ and was uttered after). This offers the interesting advantage of capturing the isomorphic relation between the dialogue presented here and a dialogue where the contestation occurs just after the confirmation. In these two dialogue situations, the communicative functions are the same but expressed in a different order and the model provides the same representation for both dialogues.

2.3.4 The exchange level

Actually, the notion of exchange has been introduced while presenting interventions. In this section, we will summarize what has been said, and also explicitly state the necessary information attached to exchanges.

Basically exchanges are made up of interventions, and of exchanges. In fact, they may have several dialogic function such as: factual information motivation, reactive motivation (such as the one presented in figure 2.2) or an evaluative motivation. This is somewhat simplified description since there is no reason why an initiative could not be an exchange. We don't consider this particular case as in person–machine interaction this very rarely occurs.

Therefore, we considered that basically four notions are attached to an exchange, whatever its nature:

- its intention,
- its main attention,
- its level of activity,
- its level of embedding.

Intentional and attentional information is not a new notion and is introduced by Grosz and Sidner (**?**), **?**)). However, this information is distributed over the four levels of the model (e.g. transaction, exchange, intervention and dialogue act).

[5]Moreover, B_2^1 could be uttered in overlap with A_2^2

The level of activity of an exchange is related to the notion of an open exchange (subject to modification in the continuation of the dialogue) and a closed exchange (a stable one, that is closed for the rest of the dialogue). Our example presented in figure 2.4 perfectly illustrates that during the dialogue more than one exchange can be opened (e.g. exchanges E_1 and E_2). Moeschler and Roulet pointed out that an exchange can be closed whenever it has reached "completeness", but unfortunately they don't offer formal computerized descriptions to decide on the "completeness" of an exchange.

In human–machine interaction such a description can be formalized. The following principles are relevant to exchange closure:

- when its intention is realized – Intentionality principle —
- when the next exchange in sequence (e.g. not embedded) contains a reaction – Closure implicit acceptance principle –
- when the evaluation is explicitly accepted by a closed evaluating exchange – Closure explicit acceptance –

The intentionality principle simply stipulates that the intention of the exchange owner (the one who opened the exchange) is attained. This achievement may be positive (true satisfaction) or negative. For example a negative achievement occurs in the following exchange:

A:	At what time do you want to leave ?
B:	I don't mind

This has *no side effect at the level of the exchange* but it may have side effects at the level of the transaction.

The closure implicit acceptance principle is very powerful because it allows the closure for structural reasons even if there is no cue expressions that express exchange boundaries. This doesn't mean that the topic of the exchange cannot be re-entered after.

Last, the closure explicit and implicit acceptance principles are not valid for *evaluating exchanges.* These are only closed by the intentionality principle if and only if there is a positive achievement.

The level of embedding is only used as a meta-control resource. We were inspired by Luzzati's work (**?**)) who stated that whenever discourse reaches a high level of embedding, there is a communication problem. As we will see, it is very useful in human–machine communication for the system to decide when it has to enter repairing sequences. This will be discussed in the next section, concerning the computer suitability of the model.

2.3.5 The transaction level

Transactions are directly tied to the management of the task that must be performed during the dialogue. Of course, there could be only one transaction, and in such a case the transaction matches exactly the task to perform.

However, Amalberti et al. (**?**)) studied telephonic appointment-taking in a medical center and observed the secretary's strategy to tackle this task. They concluded that dialogue could be broken down into three parts:

1. Request formulation.
2. Request satisfaction.
3. Agenda updates.

It appears that these observations are valid for many task-oriented problems. According to these observations, we decided to adopt Roulet and Moeschler's transaction level as part of ours (Roulet's et al. studies were not focused on task–oriented dialogues). This offers our model a conception of dialogue management strategy dependent on the task currently processed. In the "request formulation" transaction, it is accepted that the agent who solves the problem has the initiative to obtain the necessary information. In the request formulation transaction, it is the agent who proposes a solution, but it is commonly accepted that the caller often takes the initiative to be sure that the solution is acceptable. The last transaction (updates) is concerned with solution recapitulation from the agent where both dialogue participants have initiative. In the task we are interested in – flight reservation – this transaction is where the agent requests miscellaneous parameters such as the telephone number of the caller.

Of course, this sequence may be disrupted since the solution(s) may be rejected and the formulation problem reentered. However, having the transaction level in our model offers the benefit of respecting coherence in the task achievement.

2.3.5.1 Transaction shifts.

The transaction closure problem is similar to the exchange closure problem. The principles are as follows:

- A transaction can be closed whenever the intention is reached.
- A transaction can be closed whenever all the exchanges in that transaction are **or can be** closed.

Looking back to our example in figure 2.1, the structure will be as shown in figure 2.5.

$$\begin{array}{ll} T_0 & \left[\begin{array}{ll} \ldots & \\ E_k & \left[\begin{array}{ll} I_{A_1} & \textit{I offer you the flight which leaves at } 10.30\textit{. Is that ok?} \\ E_{k+1} & \left[\begin{array}{ll} E_{k+2} & \left[\begin{array}{ll} I_{B_1} & \textit{is it a TAT flight?} \\ R_{A_2} & \textit{yes it is} \end{array}\right. \\ E_{k+3} & \left[\begin{array}{ll} I_{B_2} & \textit{when does it arrive?} \\ R_{A_3} & \textit{at } 11.30 \end{array}\right. \end{array}\right. \end{array}\right. \end{array}\right. \\ & \\ T_1 & \left[\ E_0 \quad \left[\ I_{B_3} \quad \textit{I'd like a return ticket the day after in the evening}\right.\right. \end{array}$$

Figure 2.5: Example of transaction shift

B, when uttering B_3, explicitly shifts the transaction in introducing the return ticket ("I'd like a return…"). From B's point of view E_k can be closed since its intention is positively realized. However, A, at this stage, could contest the transaction shift since s/he is the owner of E_k, and as such responsible for the closure. B is of course aware of that and will consider both a flashback to T_0 or a continuation of T_1 as normal.

2.3.6 Overview of the implementation

We will not describe the implementation of this model in this section, see 2.5. In this section, we will only report interesting aspects of this model and show how it tackles difficult problems in human–machine oral dialogue. The dialogue module implementation is declarative and contains mainly two interesting sources of knowledge:

- conversational rules: which are able to plan possible dialogue evolutions according to the model we described.
- dialogue acts: which are declaratively modelled such that one can suppress or add new DAs or adapt appropriateness conditions in some DAs according to specific dialogue strategies.

2.3.6.1 Reliable contextual interpretation

The most problematic aspect of human–machine dialogues is certainly the interpretation part. We do not claim that the generation part is a simple problem but many aspects have been well formalized and solved (**?**), **?**)).

Interpretation space Keeping track of the discourse has already proved its adequacy for many problems such as ellipses and anaphora resolution (**?**)). However, our claim is that the model presented provides a well grounded restriction of the interpretation space.

At any stage of the dialogue, the system has precise views on what is currently happening and it knows specifically where problems could occur. For example, when closing an exchange it knows that it is **temporarily** closed and, thanks to the closure principles, it knows exactly when situations are clarified. Thus there is a clear relationship between discourse level and utterance level intentions which provides solid bases for recognition of utterance intentions. To illustrate this, suppose the system has just uttered A_2 in our example in figure 2.3. The state of the system (A) is:

(i) The dialogue structure is the one presented in the figure
(ii) A THINK B WANTS (departure date is 13.11)
(iii) A WANT-TO-KNOW departure-time

The effect of $Ev_{A_2^1}$ and the effect $I_{A_2^2}$ are that B KNOWS (ii) and (iii) and therefore A KNOWS that B KNOWS (ii) and (iii). From this knowledge, the system can hypothesize, on B's next move, the following general considerations:

(a) from (ii) B can *contest or acknowledge* (departure date is 13.11)
(b) from (iii) B is to provide A with information concerning the departure time

And from (i) it can hypothesize any incidental exchange into E_2 such as an exchange introduced by "could you repeat" etc.

Thus there is a clear circumscription on the necessary referential material to interpret B's next move. Moreover, heuristics can be added such that hypothesized interventions can be preferentially ordered according to contextual patterns.

The current implementation of our system gives the ordered dialogue predictions shown in the following table.

Order	Communicative Function	Topic	% Corpus
1	answer	Departure Time	84
2	confirm + answer	departure date + departure time	5
3	contest	departure date	0
4	contest + answer	departure date + departure time	0
5	meta-communicative question		2
6	initiative	flight reservation	9

The ordering (first column) reflects the order established by the system. What has been observed in the corpus is (1, 6, 2, 5, 3, 4). However, the proportion we observed for item 6 is biased in more than one dialogue since off lines conversations occurred and users were not attentive. We found no example of item 3 and 4 only because the machine was simulated and no misunderstandings have been simulated for departure times.

However, for other dialogic situations we adapted our system predictions to observed situations. But, fortunately none of them were in contradiction with our model.

Dialogue act identification It is important for the system to tag the user's utterance with the correct dialogue act(s) but ambiguities may occur. For example the intervention: "november the 13th no november the 30th" (taken from our corpus) could be interpreted as a *correction* or an *inform* with a self-repair. The contextual situation disambiguate such an utterance: if the user is in position of reacting then the system interprets that as a self-repair and considers the last part of the utterance as *inform* otherwise, the user is in the situation of possibly contesting a system evaluation and considers the last part as a *correction* to the system believes.

Moreover, we use the surface form of the user's utterance since cue expressions can disambiguate also the utterance. For example, "please" added at the end of an utterance highly tags a question. Therefore, whenever doubts remain due to speech understanding degradation, the system can strengthen hypotheses. This is possible only if aggregation rules are specified which is not the case at present in our system.

2.3.6.2 Natural behaviour and repair procedures

We recall that our model is used in an oral dialogue system. Consequently, it faces frequent acoustic failures since available speech recognition systems are not robust (**?**)). Therefore, evaluations are intensively used which has been shown as a natural behaviour. This offers also the merit of not embarking the system into misleading situations due to misunderstandings. The system is capable of directly opening an evaluating exchange (and then blocking the dialogue progression until the evaluation is achieved) whenever the speech understanding results are badly scored, or capable of uttering a simple evaluation (and allowed to progress in the transaction resolution) whenever the results are good or averagely scored. Opening evaluating exchanges are realized with different repair procedures:

- ask for confirmation with yes-no questions. Ex: “did you say November the 13th ?”
- ask for word spelling whenever the failure concerns a name of an object (such as a city, an airport, a proper noun, etc.). Ex: “Could you spell the name of the arrival city ?”
- ask for repetition whenever the recognition is very bad. Ex: “Could you repeat”

2.4 Knowledge Bases

There are four knowledge bases in the dialogue module:

1. conversational rules,
2. system dialogue acts definitions,
3. user dialogue act definitions,
4. dialogue acts properties.

A first point to clarify is probably why there are two different views of DAs: one for the system’s and one for the user’s. A first justification is that there was no attempt to write these concepts elegantly. The second justification is that dialogue act interpretation (user’s DA) do not involve the same knowledge as the generation even if all in all they are of the same nature, e.g. their representation are not the same. Simply, identifying caller’s DA means identifying them from a SIL representation, e.g. a surface semantic representation, whereas generating dialogue acts works with deeper representations, such as for example task goals.

2.4.1 Conversational rules

2.4.2 Dialogue Organization – Dialogue History

> The dialogue structure is based on a triplet paradigm: *Initiative-Reaction-Evaluation* (IRE in the rest)

The structural description is: a dialogue is made of transactions; a transaction is made of exchange; an exchange is made of interventions and/or exchanges; an intervention is made of dialogue acts.

Basically, a simple exchange is made of an initiative (to open the exchange), a reaction to the initiative and an optional evaluation. Complex exchanges contain exchanges such as: evaluative exchanges or reactive exchanges which in turn are either simple or complex (see section 2.3.4).

The user of the dialogue module should keep in mind also that the dialogue history is stored in two formats:

- a tree-like structure representation (stored in *dm_memory* via the prolog fact *dialog_struct*),
- prolog facts representing dacts, exchanges and transactions (namely: *move/9, exchange/6*).

2.4.2.1 The tree structure representation

This representation is supported by a prolog list representation. The tree-like structure is to be read from right to left. A full exchange looks like:

$$[evaluation(X, Id_1), reaction(Y, Id_2), initiative(X, Id_3), ExchangeId]$$
which is the same as:
$$ExchangeId \left[\begin{array}{l} X : initiative\ Id_3 \\ Y : reaction\ Id_2 \\ X : evaluation\ Id_1 \end{array} \right.$$

X and Y refers to dialogue participants, Id_i are pointers to dialogue acts in the predicate-like representation.

This tree means, in a ‘lispean’ jargon, that the CAR of the list represents the most recent event in the dialogue, while the CDR represents the past. Therefore, pattern matching in the dialogue history is easy when we have to refer to recent events, otherwise, it is more complex when we have to extract events in the past. To exemplify this, let us see the following sequence of expressions:

1. $[[evaluation(X, \_) \mid Rest] \mid PastEvents]$
2. $reverse(Rest, [Subexchange \mid \_])$
3. $reverse(PastEvents, [Exchange \mid \_])$

Expression 1 tries to find a subexchange in an exchange or a transaction. The searched exchange is the last one and ends up in an evaluation performed by dialogue participant X. The exchange identifier of this subexchange is obtained via expression 2. The reverse action looks for the first element in the reversed list, which is namely the wanted identifier. During the matching we don't mind whether there is any sub-subexchanges. Expression 3 looks for the root exchange or transaction pointer. In a BNF form, these three expressions are equivalent to:

$$[[evaluation(X, \_)\ \{Substruct_1\}^*\ Subexchange]\ \{Substruct_2\}^*\ Exchange]$$

The star convention means that stared item can be empty. Here it is implicitly assumed that $Substruct_1$ contains at least one initiative and one reaction.

2.4.2.2 The predicate representation

As we said two dynamic prolog predicates are used: move/9 and exchange/6. Descriptions are as follow:

- $move(Id, Type, Owner, IllocFunction, ACtLabel, AttentionState, Exchange, Succ, Pred)$
  The terminology is certainly tricky since *move* represents in fact a performed dialogue act. *Type* could be either main or subarg (see later a discussion about that). *Exchange* is the exchange in which the act is performed. *Succ* and *Pred* are not useful for our purpose.

- $exchange(Id, Sons, Father, Attention, Intention, Owner—task)$
  Sons is the list of subexchanges in the exchange. Father is of course the pointer to either the father exchange or the transaction. Owner is either system or user, and task is a marker which stipulate that in fact this refers to a transaction.

2.4.3 Dialogue rules

Dialogue rules could be categorized as follow:

1. dialogue grammar rules,
2. dialogue control rules,
3. dialogue coherence rules[6].

Dialogue rules are used in the following way: given a dialogue situation the rules are supposed to tell what could occur next (by both dialogue participants). When a rule infers that dialogue participant X can perform an exchange opening then if:

- $X = system$ it is what we called a *dialogue goal*,
- $X = user$ then it is a generic prediction.

2.4.3.1 Dialogue rules: categories

Dialogue grammar rules capture the hierarchical aspect of the above exposed model. Thus, they are somehow blind and their conclusion lead to what we call pure dialogue allowances both for the system and the user (see section 3.1 for the grammar).

Dialogue control rules directly express the need for a control in the dialogue such as for example a request for spelling a word. According to the current dialogue situation, and mainly in evaluative substructures, their conclusions directly indicate the kind of action to perform in the conversation, e.g. a restricted set of dialogue control acts.

Dialogue coherence rules or also called **conversational rules** are used to introduce rhetoric markers in the conversation and topic shift/reintroduce management.

2.4.3.2 Syntax

The syntax is shown in figure 2.6 and an example is provided in figure 2.7.

[6]This class can be seen also as the equivalent to the previous one.

```
< RuleName > =>
Dialogue Pattern
===>
dialogue(Allowance , ExchangeId)
     <- Conditions and/or Actions
```

Figure 2.6: Syntax of a dialogue rule

```
evaluation =>
[reaction(Y,_),{Incidental},initiative(X,N),Exchange]
===>
dialogue(evaluation(X,_), Exchange)
     <- not meta-discursive(Exchange).
```

Figure 2.7: An *evaluation* allowance rule

2.4.4 System dialogue acts

Each dialogue act is modelled as a rule-like formula as shown in figure 2.8.

```
< Dialogue act label > =>
List of Goals
==>
Dialogue act effects
<- Conditions and/or actions
```

Figure 2.8: System Dialogue declaration syntax

Now, let us describe each component.

2.4.4.1 Goals

Goals are a conjunction of three kinds of knowledge. Goals are indexed by the module which delivered it.

1. Dialogue allowance: dialogue(Allowance, In-exchange)
2. Task goal(s): task(Task-goal)
3. Communication goal(s): communication(Communication management goal)

The term “goal” is certainly not appropriate. In fact, it includes what each module in the dialogue manager has to say.

For now, we identified different possibilities for each module messages[7]

- Task:
  - Get_parameter(X): obtain the parameter X (i.e. the task module is requesting for a parameter value).
  - Get_parameter(X,x): obtain the parameter X with the assumption that its value is x.
  - inferred(O,o): the object O has been inferred to have the value o.
  - Check(X,x): check that the value of X is x[8].
  - Correct_parameter(X,x,y,r): X’s value cannot be x but y for the reason r.
  - answer($S_1, \ldots, S_n$): solutions are $S_1 \ldots S_n$. S_k could be a simple solution (a tuple) or a complex solution of the form $[s_k, w_k]$ where w_k is a list of warnings associated to the solution s_k. s_k can be [] in which case, these means that there is no solution for the reason w_k.[9]

  Normally, that is all. But, one can imagine that a request could be sent to the task module such as “give me the value of X”. We consider that it is the role of the belief module to answer such a question[10].

- Communication:
  - reco-rate(null): nothing had been recognised.
  - reco-rate(O,S): the object O has been recognised with the score S. Here, O can be abstract or concrete.

A dialogue act example is provided in figure 2.9.

[7]This is not a claim for any normalisation. We just want to state what seems to be necessary. Moreover, we are not claiming that this list is exhaustive.

[8]This should reflect the degree of delicacy of the running of the task module. We assume that default inferences, normal inferences and necessity are not at the same level of management in the task module and this must transpire through the dialogue.

[9]We have no idea about the relevance of ordering solutions. According to the French simulation this was of no importance.

[10]This means that when the I/O decider has to know something about the task it asks it directly to the belief module which may ask it to the task module

```
confirm =>
dialogue(evaluation(X,Id), Exchange) &
communication(reco-rate(Object, avg))
==>
move(Id,main,X,evaluation,confirm,Object, Exchange,_,_)
¡- exchange(Exchange, _,_,Attention,_,_),
in_attention(Attention,Object).
```

Figure 2.9: A Dialogue Act example

2.4.5 Caller's dialogue acts

User's dialogue acts are less declarative than the system's. They are procedures where the body of the procedure reflects the preconditions of the act. The identification of an act is based on three aspects:

1. matching of discourse markers or expressions (which could be theoretically called "conventional implicature").
2. contextual interpretation according to the predictions
3. system's knowledge and user's knowledge (that is the assumptions of the system about the user's knowledge).

Typically, the interpretation takes into consideration surface elements then the predictions and thereafter models.

The surface interpretation requires heuristics on formulations such as for example the add of "please" at the end of a "can you" utterance is typically a request for the system to perform an action.

The models of both dialogue participants are particularly important for correctly interpreting user's initiatives (it is necessary for example to make a distinction between a request or a request for confirmation).

2.4.6 Dialogue acts properties

Dialogue act properties are relevant for both system and caller's dialogue acts. They express their illocutionary properties by different means. Also, adjacency pairs paradigm is expressed in this list of properties.

A dialogue act may have the following properties:

- First, it can be assigned an illocutionary function. For this, you just have to add a new fact stating the function it realizes between:

  1. correction,
  2. clarification,
  3. politeness,
  4. offer,
  5. statement,
  6. positive.

- Then, It must be put in at least one of the lists giving its IRE value (see section 2.3.3). The declaration is made in the predicate *type_of_act1*.

  Syntax of a DA type declaration:
  $type\_of\_act1(List, IREClass)$
  where $Class\ in\ initiative, reaction, evaluation$

  therefore, you just have to add the act in the list of the class it belongs to.

- Preference between acts. This is necessary when possible conflicts could occur in the generation cycle. For example, it is possible that a confirm-def and a request-confirm-def are planned in the same cycle for the same object. Then the preference should state which of these two dialogue acts is to be selected.

  Declaration of preferences:
  $prefer\_between(LabelOfAct1, LabelOfAct2, PreferredAct)$
  $prefer\_between(LabelOfAct2, LabelOfAct1, PreferredAct)$

- And it must be made known which user dact(s) it may be followed by, by creating one or more *pair/3* predicates with the appropriate preference value. For this, the dialogue description of the new dialogue must yield all the possible dialogue continuations.

  Declaration of adjacency pairs:
  $pair(Act, PairedAct, Preference)$
  $Preference\ in\ preferred, dispreferred$

  The preference value is subject to caution since it may evolve in the future.

2.5 Algorithms

2.5.1 Generation Cycle

The generation cycle is concerned with system dacts generation. It works as follow:

1. Trigger conversational rules. This produces System Dialogue allowances (SDA) and User dialogue allowances (UDA).
2. Compute possible candidate dialogue acts on the basis of SDA.
3. Select best dialogue acts
4. Generate Dialogue Acts (GDA)
5. Make predictions
6. If GDA is/are one or two turns then
   step 1
   else Send messages.

2.5.1.1 Triggering conversational rules

Input	:	dialogue history
Output	:	System Dialogue Allowances – SDA & User Dialogue Allowances – UDA

1. Try all conversational rules (CR) on the current dialogue history
2. Try all CR on the current transaction
3. Try all CR on the current exchange

2.5.1.2 Computing candidate Dacts

Input	:	SDA and Task, Belief and Dialogue goals
Output	:	List of Candidate dialogue acts – LC

1. Save the current dialogue history and the current stacks of exchanges and transactions.
   Set the list of candidates LC to [].

2. Instantiate a dialogue act
   If no dialogue act can be instantiated then go to 4
   else

   - Add this dact to the list of candidates LC.
     store effects on dialogue history, exchange and transaction stacks, and the goals it achieves in the effects of the act.
     Backtrack on the same act (e.g. Go to 2 with the same act label).
   - Mark the act as used

3. Reestablish the state saved in 1. Go to step 2.

4. Mark all dacts as unused. End.

2.5.1.3 Select best candidates

Input	:	list of candidate dialogue acts LC
Output	:	list of dialogue acts to generate LG

1. Suppress, in LC, dialogue acts that overlap in intention according to the preference stated in dialogue act properties.

2. Append in LG clarification Dacts (see dacts properties).
   If LG is not empty

   - then Select initiatives in LG to LG otherwise select evaluations in LG to LG

   else

   - prefer reactions to initiatives. E.g. select the act which has the least incidental effect.

2.5.1.4 Dact Generation

Input	:	LG
Output	:	messages for the Message Planner

1. Merge compatible dialogue acts (in fact they are all compatible at this stage).
2. Apply their effects (structural) on the dialogue history (the merge is useful to not duplicate effects).
3. Erase goals they achieve and store them as "trying_to_solve" with two arguments: namely the list of goals and the entry point in the dialogue structure.
4. Store effects as old_effects. This care may be not useful in the future.
5. Send virtually a message to the message planner. Virtually means that we stack the message.

2.5.1.5 Making predictions

Output : Messages for the Linguistic Interface

1. Trigger conversational rules
2. Affect UDA with plausible contexts and dialogue act labels.
    - For Reaction UDA then the plausible context is the context of the exchange in which the DA belongs to. Dialogue act labels are the paired labels with the last system dacts.
    - For initiative UDA then
        - the plausible context could be the one of the exchange in which the initiative will build a subexchange,
        - the plausible context can be contexts of postponed exchanges (unlikely more than one),
        - lastly, the plausible context can be the context of the transaction.

- For evaluation UDA, the principle are the same as with reaction UDA.

3. Virtually send a message containing these predictions to the Linguistic Interface.

2.5.1.6 Send Messages

Sending message means effectively send all messages which have been stacked during the generation cycle.

2.5.2 Interpretation Cycle

1. Compute the caller's dact,
2. Compute the effects of the caller's dact.

2.5.2.1 Computing the caller's dact

The following algorithm gives a sketchy idea of what is really done. The problem is that it is not evident at this stage to express simply $< if \cdots then \cdots else >$ statements where backtrack is allowed.

Input: A SIL representation I

```
Request the BM to strip off from I what is "understandable" (C) and what is not (D).
if C = []
        then Affect(D, [])
        else
            if there is a prediction concerning a task identification
                        then
                             Request the BM to find a task in C.
                             if it succeeds then the act is a task identification
                        else
                             Request the BM to find an interpretation in current contexts.
                             Status is the returned status from the BM
                             case Status is
        direct, indirect : dact is a wh-answer
        modify : dact is a correction
        repeat : dact is an echo
        fail : Affect(D,C)
    esac
```

The output of this algorithm is in fact a track in the memory which states the label of the act and the system output it reacts to.

The procedure Affect with two parameters mainly find interpretation of pure phatic utterances and initiatives. This is done using linguistic cues (**?**)), and knowledge of the two dialogue participants (system and user).

2.5.2.2 Computing the effects of the caller's act

1. Insert the act in the dialogue history,
2. collect the goals that the system move it answers was trying to achieve,
3. compute the task effects,
4. compute the dialogic effects,
5. send messages

Inserting the move in the dialogue history is rather simple when we have clearly identify the context, and therefore the exchange it belongs to. When it is not clear where to insert the caller's act, then by default an incidental exchange is created at the current point of the conversation (e.g. the last system act).

Computing the task effects means that according to the goal the system was trying to solve and the nature (semantic content + dialogue act property(ies)) of the reaction, a message is sent to the task module. It is of course possible that this step has no effect: especially when the act is meta-communicative.

Computing dialogue effects means updating the current status of the open/closed exchanges and transactions. As said in the section about principles, when exchanges are closed (resp. transactions) evaluation are implicitly accepted. On the contrary, exchanges may be explicitly reentered and then the effect is that if there were one recently opened exchange, it is then postponed.

Send message is the same procedure as in the generation cycle.

2.6 Customization

Four main files are useful for customizing the dialogue module:

1. *rules.pl* contains the dialogue rules declaration,
2. *dacts.pl* contains the system's dialogue acts declaration,
3. *interpret.pl* contains the necessary knowledge for user's dialogue act computation,
4. *interpret_effects.pl* declares the messages to send to the Task Module once a Task goal is realized.

2.6.1 Dialogue Description: a methodology for customizing

To customize the dialogue module, to make it handle another dialogue than it was originally designed for, it is first necessary to have a clear idea of what should be customized: e.g., different tasks/languages, different recognition front ends may require an altogether different dialogue or at least a different behaviour in the dialogue strategy. What is necessary is to have beforehand a thorough description of that dialogue.

In our case, the intercity information dialogue for the german application had been designed simultaneously with the airline reservation dialogue, but at that early stage it was not yet clear how these two would match. So, the basis of the adaptation was a flowchart description of the german dialogue, which had, at the first glance, hardly anything to do with the airline reservation dialogue. One of the astounding things to see was how easy the dialogue module was adaptable to the other dialogue.

Because of this ease, I think that it is best to concentrate on the task, language, and strategy when designing a dialogue, rather than to worry about the dialogue module. So:

Step 1 *Describe your dialogue, e.g. in a flowchart, without taking notice of the dialogue module.*

2.6.2 More about Dialogue Acts

Relevant file :	**dacts.pl**

The dialogue module's dialogue rules are based on the IRE-schema. For every dialogue act (dact in the following) of system or user, one of these values must be specified, e.g. illocutionary function attached to a dact may be of type initiative, reaction and evaluation. It turned out that, as several dacts (in the dialogue module terminology) may be realized by one single utterance, these dacts do not correspond to "Dialogue Steps", but rather to the "Dialogue functions" (Bunt's terminology) in our dialogue description. Still, there was no possible 1-1 mapping, as not all of the functions and dacts are on the same level of abstraction. In finding appropriate correspondences, the IRE-distinction proved crucially helpful.

What we did was to look for dacts which had as a precondition an initiative of the system and to identify one of them as corresponding roughly to our first dialogue function of the system. If there had been no correspondence, a closer look at the degree of complexity (or rather abstractness) of out "first move" would have been appropriate. So:

Step 2 *(Re-)design your dialogue (acts—functions—moves—...) to fit the abstractness level of the dacts; make sure an IRE value can be attached (remember also that a dacts may be both an I and E[11]).*

Step 3 *Identify the first dact: it is the one that has as precondition the dialogue goal dialogue-opening.*

E.g. for the German dialogue manager, the act looks like:

```
s_initial ==>
    dialogue(dialogue_opening(system,Id),E)
===>
    create_transaction([s_initial, _,_,_], Transaction),
    create_exchange([initiative(system,Id),Exchange1],
                      Transaction, open_dialogue(Task_id) , Task_id),
    create_move(Id,subarg,system,initiative, s_initial, _, Exchange1,_,_)
<-
    get_new_move_id(Id).
```

[11]See Technical discussion at the end of the paper

s_initial is the name (label) of the dact.
The condition to perform such an act is that there is an existing dialogue goal *dialogue_opening* from the system.

Definition 1 *The syntax of a dialogue goal is:*
$dialogue(DialogueGoal(Locutor, DummyID), Exchange)$

where *DialogueGoal* is a dialogue goal label which expresses of course the nature of the goal, *Locutor* is the label of the dialogue participant to perform that goal (e.g. system or user), DummyId is in fact the identifier of the dact which will perform that dialogue goal, and lastly *Exchange* is the identifier of the exchange in which the dact is performed.
The conclusions of the act denote the effects. In our example, there are three effects:

1. a transaction creation,
2. an exchange creation,
3. and a dact insertion as the exchange initiative.

Describe the use of create_transaction and the use of create_exchange.

The created dact here is tagged as a *subarg*. This means that this act can be combined in the same intervention (but doesn't mean that it is obliged to specify such an information to combine dacts in the same turn!) (see section 2.6.4).

2.6.2.1 Looping through preconditions & effects

It is important to note that the user-dacts and the system-dacts are treated differently in the dialogue module. This reflects the asymmetry of the interpretative process of assigning communicative function (or dact) labels to user utterances vs. the process of interpreting the dialogue history as requiring from the system the generation of an utterance with a certain communicative function to properly continue the dialogue. While user dacts can be predicted, but not totally foreseen, the dialogue module has to take into account all effects of the system dact to choose the right one(s).

After the identification of the first dact, the procedure of customizing accordingly continues with replacing it by the first new dialogue act (it is assumed that this is a dact of the system). First, the name of the dact is replaced by the name of the new dact. Then, in the *create_move*[12] predicate of the dact-description, the name is also replaced. This introduces a new (system-)dialogue act. Now, it is important to propagate this new dact throughout the dialogue module.

This new dact must be assigned all of the necessary properties:

- First, it can be assigned an illocutionary function. For this, you just have to add a new fact such as:

  $politeness(greeting)$, or $clarification(request - for - confirm) \ldots$

  You have the following known categories: *statement, politeness, clarification, offer, positive.*

- Then, It must be put in at least one of the lists giving its IRE value (see section 2.6.6). The declaration is made in the predicate *type_of_act1.*

  Definition 2 *Syntax of a dact type declaration':*
  $type\_of\_act1(List, IREClass)$
  where $Class$ *in* $initiative, reaction, evaluation$

  therefore, you just have to add the act in the list of the class it belongs to.

- Preference between acts. This is necessary when possible conflicts could occur in the generation cycle. For example, it is possible that a confirm-def and a request-confirm-def are planned in the same cycle for the same object. Then the preference should state which of these two dialogue acts is to be selected.

  Definition 3 *Declaration of preferences:*
  $prefer\_between(LabelOfAct1, LabelOfAct2, PreferredAct)$
  $prefer\_between(LabelOfAct2, LabelOfAct1, PreferredAct)$

- And it must be made known which user dact(s) it may be followed by, by creating one or more *pair/3* predicates with the appropriate preference value. For this, the dialogue description of the new dialogue must yield all the possible dialogue continuations.

  Definition 4 *Declaration of adjacency pairs:*
  $pair(Act, PairedAct, Preference)$
  $Preference$ *in* $preferred, dispreferred$

[12]Choose between *insert_move* and *create_move* is discussed in the technical discussion section

The preference value is subject to caution since it may evolve in the future.

In our case, as we did find a corresponding dact for virtually every one of our communicative functions, most of this could be done by simply replacing the strings of the dact names.

Step 4 *Create the new dact; give it values in "illocutionary function" and IRE.*

Step 5 *Create predicate pair for the new dact and every possible user dact that may follow it.*

2.6.2.2 Creating complex dialogue moves

Complex dialogue moves are utterances that realize more that one dact. System complex moves can be created in three ways. First, the generation cycle merges duplicates of dacts: this has the effect that e.g. two system dact with the same illocutionary function but not the same attentional state can be merged, as for example a confirm of the departure date and a confirm of the departure city can be merged into a confirm for departure date and city.

Second, the *pair* predicate may contain as a possible user continuation an *epsilon* value, which means that the user is allowed to say nothing, and that the system may continue its turn (e.g. 1-or-2-turns acts have a paired epsilon and 2-turns acts don't). Accordingly, if the generation cycle encounters an epsilon dialogue allowance for the user, it continues to generate the next system dact, adding it to the previous one before sending the whole to the MG.

Third, a system dact may be typed as a subact. In that case, the generation cycle will continue to find an appropriate "main"-dact. This can be very helpful to combine dacts according to the status of parameter values, as sub-acts will always build complex dacts (see for example in the german dialogue module the acts: *s_give_info* and *s_indic_last* and *s_request_continue*).

2.6.2.3 Realizing task goals

It is one of the great advantages of the dialogue module that it is task independent; however, the advantage results in the fact that one has to be careful to add to the changed or newly introduced dact the appropriate task effects. That means, that the messages to and from the task module must harmonize with the changed dialogue. E.g., in the German dialogue, the dact *s_propose_continue*, which replaces the airline application dact alts-question, has as a task effect the task message *continue_task*, so it looks like this:

```
s_propose_continue ==>
    dialogue(initiative(S, SubAct-_), Exchange),
    task(continue_task(TaskId, Solutions))
===>
    insert_move(Id,main,system,initiative(system,SubAct-Id),
              s_propose_continue,[id:TaskId, solution:Solutions], Exchange,_,SubAct)
<-
    get_new_move_id(Id),
    move(SubAct,_,_,_,_,_Solution,Exchange,_,_).
```

2.6.3 What is dependent to the interpretation of dacts

Relevant file :	**interpret.pl, dm_interface.pl**

User's dialogue acts have the same properties as the system's. This means that one user's dialogue act is also included in the *type_of_act1* declaration, *statement, politeness* etc.

However, it is not so clear that system generation of dacts could be the reverse function of the user interpretation. This is why user's dact declaration (their body: such as conditions and effects, but not their properties) doesn't follow the same rules and conventions stated for system's dact declaration.

2.6.3.1 The interpretation cycle

The first step in the interpretation is the dact computation: what dact to assign to an intervention. This dact assignment is negotiated with the Belief Module, therefore, part of the interpretation is made in the module *dm_interface*. The first applied heuristic is that the interpretation is context dependent, hence predictions are used. A first attempts is then to match the user's input against one prediction. Only one dact label is assigned in dm_interface: the act which corresponds to a task identification (in the French dialogue module this act is named *forpb* and in the German it is *u_give_task*). Apart that, the dact computation is made in the module *interpret*. The interpretation is really not declarative for now. New extensions are planned in the future to declare user's dialogue acts (examples of such declarations are provided at the end of this document).

2.6.4 Dialogue rules

Relevant file :	**rules.pl**

Dialogue rules are here to explicitly organize the dialogue according to a dialogue grammar they mirror. Moreover, they can be used to express dialogue control. Let us see some dialogue rule examples:

```
example 1: reaction
                    reaction =>
                         [initiative(X,_), Exchange]
                    ===>
                        dialogue(reaction(Y,_), Exchange)
                    <-
                        dialoguee(X,Y).

example 2: initiative
                    initiative =>
                         [[evaluation(X,_) | _] Histo]
                    ===>
                        dialogue([initiative(X,_),_], Exchange)
                    <-
                        reverse(histo, [Exchange | _]).

example 3: control
                    embedding_control =>
                         [[evaluation:[evaluation:[reaction(X,_) | Reminder] | _] | _] | _]
                    ===>
                        dialogue(control:evaluation(Y,_), Clarification)
                    <-
                        reverse(Reminder, [Clarification | _]),
                        \+closed(Clarification),
                        dialoguee(X,Y).
```

Here again there is a dedicated syntax: the name of the rule, the premises and the conclusion (the dialogue goal) and also a list of conditions and/or actions.

The premises try a mapping between the stipulated dialog substructure or structure and the current dialogue history (in the tree-like representation). Dialogue rules are applied at three levels:

- at the level of the dialogue,
- at the level of the current transaction,

- and at the level of the current exchange.

This choice is simple to justify: we may want to have only a local view of the dialogue, e.g. in the current exchange, to express dialogue conventions such as the one in example 1. But, we may want also global views at the transaction or even more at the dialogue level, e.g. this is necessary for the dialogue rule in example 3.

Moreover, dialogue rules are generally valid for both dialogue participants, such as *evaluation* where X denotes a dialogue participants (system or user). This is why they can be both predictive (when X is instantiated by 'user') and useful for generation (X instantiated by 'system').

2.6.4.1 Customising dialogue rules

As we may foresee, two possibilities are offered:

- add, suppress or modify existing rules,
- characterize rules according to the dialogue participant.

More about rule premises and conclusions Adding a new rules signifies that you are able to identify a dialogue history pattern in which you want to allow something which is not allowed yet. Once the dialogue history pattern is identified, then you have to write it in the same format as in examples. You have also to write the conclusion of the rule which follows the syntax of a dialogue goal (see section 1). In fact the goal can be a little bit more complex.

Definition 5 *A simple dialogue goal is of the form:*
$dialogue(Predicate, ExchangeId)$

where *predicate* is a two placed predicate (initiative/2, reaction/2 or evaluation/2). A complex goal is of the following format:

Definition 6 *A complex dialogue goal is of the form:*
$dialogue(Substructure, Exchange)$

Here $Substructure$ is the new substructure that will be added at $Exchange$[13] if the dialogue goal is achieved. We need such artefact to tell the dialogue acts what

[13]This is to be considered as a position. *Exchange* may be either an exchange pointer or a transaction entry point.

are their effects on the dialogue history. For example, when you allow an initiative, of course the dact which will realize that initiative is of type initiative but also the side effect is that you open a new exchange. This is why the conclusion of the dialogue rule *initiative* shown in example 2 is:

$$dialogue([initiative(X, \_), \_], Exchange)$$

the substructure tells explicitly that we will create a subexchange of *Exchange* for which we don't have yet an identifier (the reason for the use of "_").

Adding new rules When you add a new dialogue control rule then you may wish to characterise precisely the issued dialogue goal (you'll find examples with repetitions and reintroductions). You have the possibility to add what you want in the goal description as follow:

Definition 7 *Dialogue control rules produce a goal of the form:*
$dialogue(Marker : Function, Exchange)$

where Marker is the new feature you may add. But now, remember that if you wish that this goal is to be performed then you must have the corresponding dact(s) with the same marker in their dialogue conditions. Marker can be of any atomic (in the prolog sense) value.

Step 6 *Add new dialogue rules if needed.*

Splitting rules Splitting a rule means that you don't want to have the same behaviour for the system and the user. For example, the system may always try to evaluate exchanges while users don't. This mean that you may have specialized dialogue rules for the user to allow her to open new exchanges while not having evaluated a previous one.

Step 7 *Split rules if predictions are not satisfying.*

2.6.5 Task effects

Relevant file :	**interpret_effects.pl**

In the module *interpret_effects* are declared what are the messages to send to the Task Module (TM) according the task goal. This is done via the predicate *task_effects/3* and the definition is:

Definition 8 *A task effect is declared as follow:*

task_effects(TaskGoal, Semantics, UserMove):-
    Optional traces,*
    Optional tests,*
    send_message(tm, TheReply),
    *Optional descriptor updates **

This part of the system is not really declarative, but it should be in the future. The purpose of $task\_effects$ is according to a task goal and a user reply concerning that goal to compute the reply to the TM and optionally update descriptors in the dialogue history.
$MOVE$ is the id to the dact in the predicate *move*. You may refer to that to get the dialogue act label in order to know, for example, whether it is *positive* to see if the user confirmed or not a value (see the answer to the task goal $update\_default$).
You may also want to extract information in the *Semantic* object, which is in a SIL representation **?**) and manipulate it with standards *get_path* etc.
Finally, you have to choose the correct answer, and format, to compute the *TheReply* (see documentation on the TM **?**)).
And also, if the answer is an update to a parameter, then use the predicate *update_task_value* (see examples in the file).

2.6.6 Technical discussion

This is an overview of what could be unclear while customizing the dialogue module.

2.6.6.1 What to use between insert_move and create_move?

The function *insert_move* has the property to update both the tree-like structured dialogue history and the predicate representation of the dialogue history while *create_move* only updates the latter representation.

So far, dacts which only create simple moves (such as confirm for example), you have to use *insert_move* otherwise, for a dacts which creates for example a subexchange you have to use *create_move*. The reason is simple: when you create the exchange (via the use of the predicate *create_exchange*) you must have the first act of the exchange but since it will be created later (with create_move) you don't have it ! Thus, you create the exchange with a new identifier for the first move of it and create it as if it existed. This will have side effects on the tree-like representation of the dialogue history which will introduce an item denoting the move. Therefore, when it is the turn to create the move you should not duplicate its entry in the tree-like structure, this is the reason why you have to use *create_move* and not *insert_move*.

2.6.6.2 Why an act can be both declared as I and E?

For example, the act *confirm* is both declared as an evaluation and as an initiative. The evaluation role is trivial. What is not trivial is that it is also an initiative. In fact, you have to remember that it is not an initiative but it may have the status of an exchange opener which is by default an initiative. Thus initiative means both: open an exchange and take the initiative and open an exchange only. The following dialogue will exemplify this principle:

Ex:

S: when do you want to leave ?
U: January the 23rd *the system understood 24th instead*
S: January the 24th, at what time ?

This dialogue is represented as:

$$\begin{array}{ll} E_1 & \left[\begin{array}{l} I(S,1):\ \mathit{when\ do\ you\ want\ to\ leave?} \\ R(U,2):\ \mathit{January\ the\ 23rd} \\ E(S,3):\mathit{January\ the\ 24th} \end{array} \right. \\ E_2 & \left[\begin{array}{l} I(S,4):\ \mathit{at\ what\ time?} \end{array} \right. \end{array}$$

Dynamically, at this time the system's evaluation *January the 24th* is a simple act, not more than an evaluation. But, if you anticipate the next user's intervention, he will probably contest the system evaluation with something like *no I want to leave on January the 23rd* and this contest will assign a new function to the system's evaluation: an act which had opened an exchange, hence an initiative.

This is due to the non-deterministic nature of evaluations.

2.6.7 Appendix 1: Language used in the dialogue rules

- reestablish_postponed_goals
- cycle
- find_pattern
- act
- dialoguee
- reverse
- ensure_postponed
- closed
- move
- exchange
- postponed
- closed_explicit
- dependency

2.6.8 Appendix 2: Language used in the dialogue acts

2.6.8.1 Actions

- insert_move
- create_move
- create_exchange
- create_transaction
- temporary_close_task
- get_new_move_id
- open_task

2.6.8.2 Tests

- move
- exchange
- in_attention
- member
- completed_task
- current_task
- task_status
- card

2.7 Implementation

The dialogue manager is implemented in QUINTUS Prolog. It consists of various prolog submodules which are described below.

2.7.1 Submodule descriptions

Module name	*Description*
dact_properties.pl	States the list of properties attached to dialogue acts.
dialog.pl	Algorithm described in section 2.5.1.1 “triggering conversational rules”
dm.pl	Initiate procedures and loads performed when starting the dialogue module.
dm_interface.pl	Responsible for reacting according to all messages received by the dialogue module from the LI, TM, BM and MP.
dm_memory.pl	Declaration and maintenance of the dynamic memory of the dialogue module.
interpret.pl	Responsible for computing the effects of the user’s dacts. See section 2.5.2.2 for details on the algorithm.
interpret_effects.pl	Responsible for computing the task effects of the user’s dacts (part of the algorithm described in 2.5.2.2).
iodecide.pl	Responsible for the generation cycle: top level, and predictions (see algorithms 2.5.1).
meta.pl	Responsible for selecting the appropriate dacts in generation (algorithm 2.5.1.3) and for computing meta discursive effects (step 4 of algorithm 2.5.2.2).
rules.pl	The dialogue rules and the compiler.
system_dacts.pl	System dialogue acts declaration and the compiler.
tools.pl	A tool box. Mainly used for manipulating the dialogue history, thus, some macro instructions used in the dialogue rules and system dacts are declared here.
user_dacts.pl	Declaration of user’s dacts and the surface interpretation algorithm described in 2.5.2.1.

2.7.2 Main predicates

We describe here the main predicates according to the submodule organisation.

2.7.2.1 dact_properties

- **pair(?First-Member, ?Second-Member, ?Nature)**
  first-member and *second-member* are dialogue act labels paired by the fact *pair* and the nature of the pairing is given as a last parameter. Nature can be *preferred* or *dispreferred* as suggested by Levinson (**?**)) for adjacency pairs.
- **type_of_act(+Act, -Function)**
  For a given act it return the illocutionary function, e.g. initiative, reaction or evaluation.

Various other properties are declared, please refer to the listing and section 2.4.6.

2.7.2.2 dialog

Only one main predicate here.

trigger_dialogue_rules/0
which triggers all dialogue rules on the current whole dialogue structure, transaction and exchange.

2.7.2.3 dm_interface

- **receive_from_pm_dm/3**
  various instantiations according to the sender:
  - from the TM: assert the message as a tm goal, of the form $tm(message)$
  - from the MP: assert the message as a mp goal, of the form $mp(message)$
  - from the LI: start the process of interpretation in sending the BM a request stripped message.
  - from the BM: various actions are performed according to the kind of message.
- **send_message(+Dest, +Message)**
  buffers a message that must be delivered to Dest.

- **send_messages/0**
  send all buffered messages in a format requested by their recipients (e.g. in a list, or one by one and so on).

2.7.2.4 interpret

- **insert_act/0**
  the algorithm described in 2.5.2.2
- **insert_move_in_history/5**
  insertion of the user's dialogue act at the correct place in the dialogue history.
- **dialogue_effects/3**
  Computes the direct dialogic effects and sends to "meta" relevant information to compute meta-dialogue effects.

2.7.2.5 interpret_effects

The main predicate is:

- **task_effects/3**
  which sends the task module task information derived from the interpretation of the user's dacts.

2.7.2.6 iodecide

- **dialogue_module/0**
  the top level of the generation cycle described in 2.5.1.
- **compute_dialogue_act/1**
  computes the list of candidate dialogue acts for the system: the top level of algorithm 2.5.1.2.
- **io_decide/1**
  called by the previous one. For one candidate dialogue act it computes its effects and store them (step 2 of algorithm 2.5.1.2).
- **generate/1**
  top level of algorithm 2.5.1.4, e.g. build the message for the MP.

- **predict/1**
  computes the predictions attached to one system dialogue act. (algorithm 2.5.1.5).

2.7.2.7 meta

- **select(+List, -Selectedacts)**
  selects the most relevant acts from a list of candidates (algorithm 2.5.1.3).
- **meta_effects/6**
  computes meta discursive effects.

2.7.2.8 user_dacts

User's dialogue acts are not declared the same as the system's. Each dact is a procedure **user_dialog_act/2** which gets the dialogic aspect and the informative aspect of a message (resp. obtained via the request_stripped and request_anchor negotiations with the BM) and according to their contents and the user model selects what is the most probable uttered dialogue act.

2.8 Conclusion

The actual implementation has reached a reasonable stage where it is obvious that the theoretical model is tractable. The model exposed in the beginning of this chapter show a hierarchical and functional description of conversation that can cover most of the dialogues obtained during the simulation (WP3 phase 1). One major criticism that one can formulate about generic models is that there is a lack of predictive power. This is not the case in our model model, and thus this is a benefit for the implementation, since we paid attention to the non trivial principles of exchange closure. These principles when applied, give the model powerful predictive mechanisms for contextual interpretation.

Moreover, we covered different aspects of human-computer interaction and more specifically with the natural way the system has to provide feed back of its understanding which avoid complex repairs. Another interesting feature is that implicitness and explicitness phenomena have been investigated.

Despite these positive strong points, less efforts have been spent on robustness aspects and more specifically we have not yet investigate failures due to inconsistencies and the associated repair mechanisms. Also, if the model seems now stable and quite satisfying, we wish to redesign some parts of the implementation to have a more clear "dialogue description language" such as it will be easier for naive users of this module to update the knowledge and investigate new researches in human-computer oral dialogues.

Chapter 3

The Belief Module

3.1 Overview

The belief module is at the center of the semantic interpretation process, as well as being the major mechanism whereby discourse coherence (at a level within the utterance) is maintained. This chapter begins by discussing the functionalities of semantic interpretation, and of message conceptualisation, performed by the belief module. Section 3.3 discusses some of the issues, involving knowledge representation and inference, contextually-bound interpretation, the use and maintenance of focal registers, and the planning of descriptions. In Section **??** we consider in some detail the internal workings of the various components.

There are currently two forms of knowledge definition available for use with the BM; these are presented in Section **??**. Section **??** considers the work needed to modify the BM for a new application. It is forseen that this corresponds mainly to extending SIL to include definitions of task-relations and parameters, then establishing the necessary constraints to render these inferrable.

Section **??** turns to implementation. The interfaces are described, and the internal workings of NOOP, the backbone of the current implementation, are discussed. The use of a test suite is described, and tentative recommendations made as to how this could be extended. Finally, Section **??** outlines what work is still required, and what design decisions still have to be made.

3.2 Scope

The BM interprets input from the user and assists in the generation of the system's utterances. It does this by maintaining a structured model of knowledge instances, access to which varies both contextually (according to the current foci) and modally (according to the world view chosen). This model (alternatively referred to as the *belief model*, and the *discourse model*) contains representations of that material referred to in conversation, as well as its closure under inference, and various hypothetical extensions which may aid the reasoning/planning process. So far as other components of the system are concerned, the BM constitutes a dynamic knowledge base, to which queries and updates may be addressed. Here we consider these functionalities in more detail.

Input from the user is that which is built up by strict compositionality, without any additional inference, by the parser, using the constraints of the lexicon. It may be used to build new structure in the belief model, which will mirror closely the structure and content of the input. Alternatively, it may fully or partly reference existing structure.

While some substructures within the input expression will map directly onto discourse objects, available for later reference by either the system or user, others may act as indicators about how it can be contextually anchored to existing material. In fact interpretation of utterances is a double act, carried out in part by the belief module, and in part by the dialogue module:

- the dialogue module may use its knowledge of current goals and conversational structure to propose how user inputs may be anchored;
- the success or failure of the belief module in carrying this out may be used by the dialogue module in refining its account of the conversational significance of the utterance. For example, the input may simply repeat earlier discourse material, or it may contradict what has been said previously; in either case, dialogue expectations need to be updated.

Aside from success or failure with regard to interpreting the input at some given context, if there is unpredicted information in the input, the dialogue module is notified appropriately. This is how 'overloaded replies' are dealt with.

It is clearly necessary to represent and store discourse objects whose structure and environment within the knowledge base correspond closely to that produced from the utterance by the parser. On the other hand, knowledge at the level of the application, which reflects its competence, must be maintained: both in order to pass on to the application what the user's requirements are, and in order to channel knowledge transmitted back by the application. This means that inferential bridges need to be built between 'surface', linguistically oriented, representations, and 'deep', task-oriented representations. The BM may be thought of as responding on both levels, to requests and updates. However, interaction with the application is much simpler than interaction with the user, because there is no ambiguity or uncertainty.

Turning from input to output, the belief module works in tandem with the message planner in the planning of appropriate descriptions which will lead to appropriate system utterances. This process of *message conceptualisation* may be thought of in terms of extraction of those fragments of a semantic network which are sufficient to enable the hearer to build appropriate structure in his own discourse model. The amount of structure which needs to appear in a message depends on how great an overlap of context the two speakers may assume—this may be sufficient to allow fragmentary (elliptical) descriptions (for example). A speaker may use more or less full descriptions, again depending on the amount of shared structure assumed. A speaker also uses prosodic marking to signal the extent to which discourse entities can be taken as part of the shared discourse model, the extent to which differences exist between different instantiations of the discourse model—such as when the system's capabilities do not exactly match the user's requirements—and the extent to which the current utterance builds on the resources of recent utterances.

The discourse model is therefore used by an agent as a resource promoting discourse coherence, both in speaking and listening modes. But, together with a number of computational and psycholinguistic accounts (**?**, **?**)), we treat this coherence as an emergent property of the autonomous process of internal model-building. The discourse model retains the gist of natural language utterances, while omitting surface features, or the order in which information was acquired. It may be thought of as a semantic network which instantiates the generic knowledge available for the domain. Such instantiation is generally monotonic over the course of the dialogue. However, there will be occasions where contradictory information is present, for example because of communication difficulties, or where the system cannot provide an exact solution to the caller's requirements, or where the caller wishes to entertain more than one instantiation of his requirements. To deal with this such situations, information in the discourse model is made relative to some 'world', or *view*. For example, in the case of modified information—which may be caused by acoustic failure—the system can maintain two views on the discourse model whose contents differ only on the details under dispute, and so ask which the caller intended.

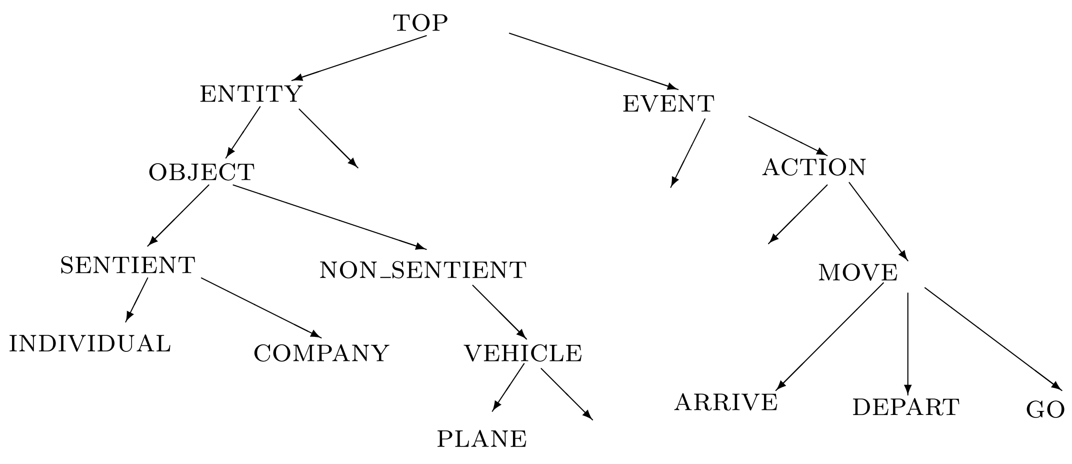

Figure 3.1: The semantic class hierarchy

3.3 Principles

3.3.1 Knowledge representation

Representing discourse entities and their interrelationships can be done in a relatively straightforward manner using a *semantic network* (eg **?**)). Unlike predicate calculus, these have the advantage that persistence of objects is built into the representation. They are also amenable to graphical representation. [1] Monotonic addition of information can be effected by simply growing the relevant portion of the network to include the new entities and relations (see Figure 3.2 and 3.3 for examples).

Deciding the semantic primitives to be used in the knowledge representation language is to some extent arbitrary. The *SIL* language presented here, makes use of the following principles:

1. discourse entities are *typed*; their types, or *classes*, are partially ordered by subsumption, according to a *semantic class hierarchy.* Figure 3.1 shows a portion of this hierarchy. In the representation used here, all nodes inherit from a single node, TOP. Discourse entities are defined to behave as instances of nodes in the class hierarchy. The contents of the discourse model at any one time consists of whatever such instances currently exist, together with the relations between them.

[1]There are disadvantages too. In particular, extensional representations are favoured above intensional ones. Meta-references to objects and quantification are difficult to represent. See section 3.3.2 for a discussion of some of these issues.

2. A discourse entitity has *roles* (or 'links')—two place relations linking it to other discourse entities. What roles a discourse entity is capable of having are defined for it at the class level. Considerations of compactness argue that such role definitions be derived using inheritance. For example, if the class GO has a role *thegoal*, then a class which specialises it such as DRIVE should also have this role, or a specialisation of it. Using inheritance-based principles to organise knowledge has also been argued for on linguistic principles, notably by **?**).

3. The choice of what classes of object should be represented is to some extent an arbitrary one. The classification used in a particular natural language can be exploited: for example, by defining classes corresponding to common nouns such as "plane", and unique individual for proper nouns such as "paris". Verbs present more of a problem—we follow usual semantic network practice, and recent literature on linguistic semantics (eg **?**)) in *reifying* these—ie, representing the relevant events, states, and relations by tokens denoting discourse entities. This greatly facilitates representation, and may be partially argued for on grounds of anaphoric usage; consider, for example the following:

   (1)

A	i want to travel to perros
B	sorry. where is (the journey/that) to?

   To the extent that nouns and verbs reference the same event, they may be represented as the same discourse entity. In (1), "travel" and "journey" might both refer to the same discourse entity, SINGLE_JOURNEY1.

4. The process of reification, taken to its conclusions, can provide for the representation of instance of any complex relation, even those not directly expressible in natural language, as discourse entities.

Using a semantic network representation, we may illustrate how the discourse model is extended over two successive inputs:

(2)	a	fly from london to paris
	b	travelling from heathrow at 17:15

As a result of (2a), with semantic representation:

$$\begin{tikzcd}
& \textsc{go}1 \arrow[dl, "theinstrument"'] \arrow[d, "thesource"] \arrow[dr, "thegoal"] & \\
\textsc{plane}1 & \textsc{location}1 \arrow[d, "thecity"] & \textsc{location}2 \arrow[d, "thecity"] \\
& \textsc{london} & \textsc{paris}
\end{tikzcd}$$

Figure 3.2: Discourse model after the utterance: "fly from london to paris"

(3)

$$\begin{bmatrix} \mathit{type} : \mathit{go} \\ \mathit{theinstrument} : \mathit{type} : \mathit{plane} \\ \mathit{thesource} : \mathit{thecity} : \mathit{london} \\ \mathit{thegoal} : \mathit{thecity} : \mathit{paris} \end{bmatrix}$$

the discourse model contains the information shown in Figure 3.2. The semantic representation of (2b) is:

(4)

$$\begin{bmatrix} \mathit{type} : \mathit{go} \\ \mathit{thesource} : \mathit{theairport} : \mathit{heathrow} \\ \mathit{thesourcetime} : \begin{bmatrix} \mathit{thehour} : 17 \\ \mathit{theminutes} : 30 \end{bmatrix} \end{bmatrix}$$

As a result, the discourse model gets extended as shown in Figure 3.3. This extension assumes that we can infer that "fly" and "travelling" refer to the same event. See section 3.3.3 for how this can be done.

The power of representation may be further extended by the addition of *rules of inference*, or constraints. For example, world knowledge tells us that flying entails a journey, and that the latter has arrival and departure events associated with it. It would then be possible to extend the utterances in (1) by the inputs:

(5) a leaving from terminal 4
b arriving at charles de gaulle airport at 1730

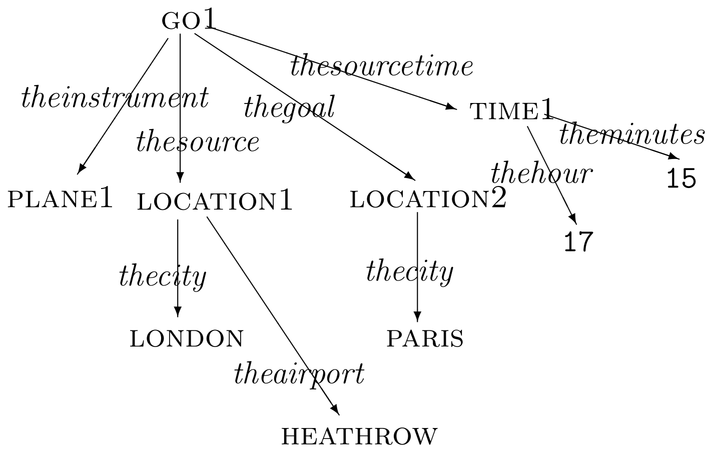

Figure 3.3: Discourse model after the utterance: "travelling from heathrow at 17:15"

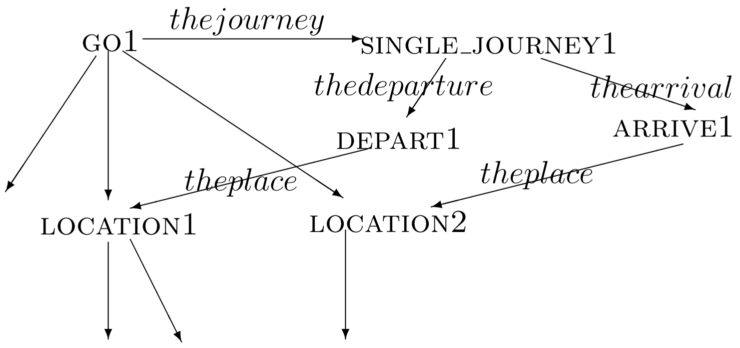

Figure 3.4: Discourse model after the utterance: "travelling from heathrow at 17:15"

Figure 3.4 shows the results of operating an inference rule that adds the discourse events corresponding to SINGLE_JOURNEY, ARRIVAL and DEPARTURE. If all applicable constraints have been imposed, we call the resulting state of the discourse model *inferentially complete*. We need however to distinguish between explicitly mentioned entities, and inferred ones, in case referring to the latter is done in a different way—for example they may be presented as being less accessible.

The knowledge representation discussed so far is partial—it doesn't make the 'closed world assumption', whereby if information isn't present, it is known not to be the case. Instead, there is no distinction between falsehood and incomplete knowledge. This is satisfactory as long as the addition of information doesn't violate monotonicity, either by incompatibility with existing information, or because

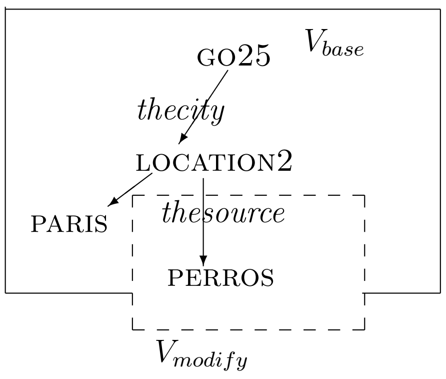

Figure 3.5: Adding incompatible information at a modified view

it violates some constraint, such as one insisting that for a GO event, <thesource theairport> and <thegoal theairport> should be different. We circumvent this problem in our representation by distinguishing between the contents of the discouse model as seen from different *views*, or 'worlds'. Consistency then only needs to be maintained with respect to a given view. Consider for example the case where the input "fly from paris" is followed by "flying from perros at five". This could happen because the interlocutor is inconsistent with himself, or careless. In speech-recognition-based dialogue systems, it happends frequently because of imperfections in speech-decoding technology. The discourse model before the second utterance, represented in Figure 3.5 at V_{base}, was internally consistent. Consistency may be maintained by extending the discourse model, as the result of the second input, not at the current view but at an alternative view V_{modify}. To avoid duplication, information not specifically in V_{modify} is inherited from V_{base}.

The mechanism of recording dubious extensions to the discourse model as it were provisionally, in separate views, may be used as well in cases where there is no contradiction, but the reliability of information is in doubt, or default knowledge has been used before checking with the interlocutor. It is also used for representing disjoint instantiations.

3.3.2 The interpretation of input

Distinct from the informational content of an utterance, viewed as a portion of the discourse model, the *semantic representation* is closer to the surface, being compositionally derived from natural language expressions, with the aid of the lexicon. Assuming that interpretation is modular, this representation is not anchored in context; the semantic representation is ambiguous as to what discourse entities are being referred to. We may assume nonetheless that expressions of the semantic representation language make use of the same primitives as those used in describing the contents of the discourse model. However the representation is closer to predicate logic, allowing in particular the possibility of quantifiers, determiners, disjunction and negation. Using the binary-valued feature DEF to denote definiteness, for example, we can represent: "the flight from paris is delayed" as follows:

$$\begin{bmatrix} \mathit{type} : \mathit{delay} & \\ \mathit{thetheme} : & \begin{bmatrix} \mathit{type} : \mathit{single\_journey} \\ \mathit{def} : + \\ \mathit{thedeparture} : \mathit{theplace} : \mathit{thecity} : \mathit{paris} \end{bmatrix} \end{bmatrix}$$

In interpreting this input as an assertion, it would be necessary to find out to what extent it asserted new information, and to what extent it simply referenced existing information. In this example, SINGLE_JOURNEY and PARIS are existing information, and need to be linked to existing discourse entities. DELAY is new information, requiring the addition of a new discouse entity, with the appropriate links. In many European languages, where a common noun is used to describe a discourse entity, whether or not it is assumed to be already present in the discourse model is signalled by the use of a definite or indefinite determiner, respectively. When interpreting input, definiteness signals, and other accessibility signals such as the use of pronouns, are used to guide the search for an anchoring to existing discourse entity. Once found however they are not retained as part of the discourse model's information.

3.3.3 Accessibility within the discourse model

There are two ways in which the accessibility of already-existing discourse objects can be used, in production and interpretation:

1. Explicitly: *accessibility markers* (**?**)) are used in descriptions; this has the effect both of economising on the effort of encoding—since accessibility markers usually involve reduced descriptions—and the effect of assisting the interlocutor to find a referent in his own discourse model;

2. Implicitly: the speaker may build descriptions, on the basis of the assumed shared discourse model, which depend on contextual inference for their interpretation.

These may be illustrated in the following examples:

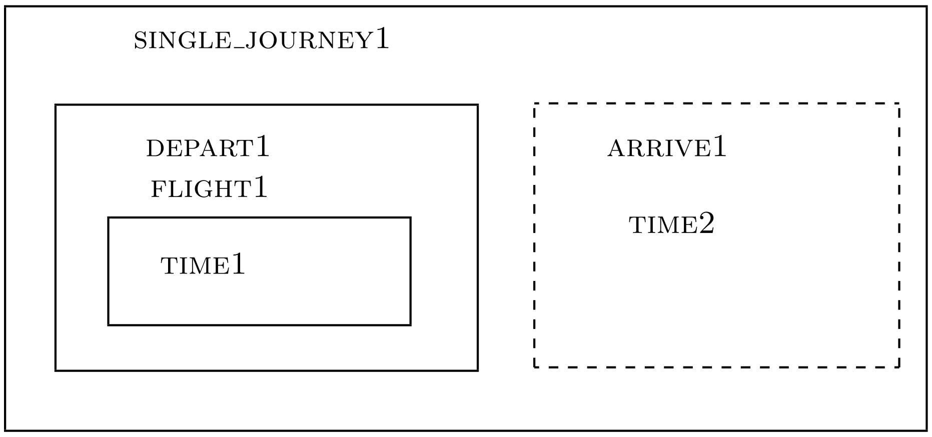

Figure 3.6: Embedded topic spaces within the discourse model

(6) a there's a flight to paris at seven thirty.
it arrives there at eight forty five

b $C1$: I want to fly to edinburgh
$A1$: what time do you want to leave

In example (6a), the underlined words act as high accessibility markers, indicating that their antecedents are to be found in the very recent discourse context. In this example, there is enough semantic evidence to link "it" to the only recent object of type FLIGHT, and "there" to the only recent object of type LOCATION. In example (6b), the referent of "leave" will be a discourse object of type DEPART. Although this hasn't previously been explicitly mentioned in the dialogue, the current context is sufficient to link it to the discourse object of type GO introduced in $C1$. [2]

A simple strategy for locating antecedents during interpretation uses an *accessibility index*, in which all discourse objects are referenced in three ways: by recency, by type, and according to discourse topic. The notion of discourse topic is not altogether straightforward to define; as **?**) points out, one may think of topics as embedded: that is, in discussing a certain time, the narrow topic is that time, whereas wider topics are the departure event with which the time is associated, and the journey with which the departure event is associated. With such a topic structure, an object of type TIME would be immediately accessible, while objects of type FLIGHT and DEPART would be accessible from the next higher topic space, as in Figure **??**. In the figure, the object TIME2 is less accessible, since there is a more immediately accessible object of that type. Topic spaces

[2] An alternative view, not currently followed here, is that $A1$ underlyingly contains argument positions, for example, for departure location and arrival location, but that these have been ellipsed. The solution however is similar—the referents have to be found within the discourse model.

must take views into account: for example, if there are alternative instantiations of a caller's requirements, and one particular flight is being considered, then that flight is currently topical, and the others are not. This requirement is easiest handled by using the views mechanism in the implementation of topics. At any time, the current topic tree can be defined as a subtree of the current views tree. A routine for interpretation then is as follows:

- if the accessibility of an object is explicitly encoded, say in a pronoun or definite description, then use the information this conveys about recency, together with the inferred type and current topic, to locate the correct antecedent.
- if accessibility information is not explicitly given, it has to be decided whether or not to create a new discourse object. A conservative approach to this is to use an existing discourse object within the current topic space or accessible from it, provided that this is of the correct type and (locally) unique. If assuming this causes constraints to be violated, as is the case in assuming that the departure and arrival places are the same, then backtracking can be used to create a new object of the correct type.

In generating descriptions, it is necessary to provide definiteness and accessibility markers in all cases, even when their use may be prevented because a (linguistically) marked utterance is required, or because the lexicon does not allow their generation.

Generating prosodic accessibility markers follows the same principles outlined above. In this case, The distinctions that are made are as follows, by the values assigned to the feature [PFOCUS] (prosodic focus):

new : in the case of entirely new discourse objects, this needs to be signalled prosodically. For English, prosodic accents are placed on the relevant material. However, if they are unique in the current context and their existence could have been inferred by the hearer, there is the option of marking them as 'given';

contrastive : if the current mention of a discourse object contrasts (in type and role) with the most recent mention of an object of the same semantic type, then this marking is appropriate;

referring : a weaker case of the above occurs when the contrast is less marked: for example, the antecedent has been previously referred to, but there has been an intervening reference, not explicitly contrastive, to an object of the same type;

given : the catch all condition, for discourse objects which have been previously mentioned

Work is currently underway on the preciser definition and implementation of these pfocus features.

3.3.4 Building descriptions

Let us first consider in isolation, the problem of building descriptions—SIL representations which partially cover the contents of the discourse model. Several requirements emerge:

1. specific objects may need to be described, or referred to;
2. we may distinguish between *intensional* descriptions, which indicate where the discourse entities they denote are to be found in relation to other material in the discourse model, and *extensional* descriptions, which describe what information their objects contain;
3. descriptions may be more or less verbose, in the amount of information they contain. This is not simply a matter of using anaphora appropriately; it may not be appropriate for example to base utterances on all the database information found;
4. descriptions may be about the contents of one or more views. If we consider views other than the current one as its hypothetical extensions, then we may see utterances based on such descriptions as attempts to advance the shared pool of information, by presenting the interlocutor with assumptions to accept or reject;
5. descriptions are only useful for a language generator, if their contents are lexically realisable.

Requiring that certain objects be mentioned in a description may entail that these be given prominence, say prosodically or in terms of word-order. Assignment of such prominence may depend for example on the speaker's intention; if the primary intention is to find the value of some discourse entity, and a secondary value to mention some other, as a form of implicit confirmation, then utterances like (7) may be produced:

(7) $\underline{\text{what time}}$ do you want to leave london

—where the query element is both prosodically and syntactically marked, but the confirmation element may nonetheless receive some prosodic prominence. Determining such a scale of importance is a function of the context in which the material was introduced, and the relative importance assigned to it by the dialogue manager. With contexts represented as views, the relative importance of “what time” and “london” may be a function of the compound dialogue act which references them.

The basis for building descriptions, intensional or extensional, leads to a number of possible description types:

- an anchoring description describes an object by reference to known material to which it is related; it does not necessarily describe the object itself. For example, “what time” in “what time do you want to leave”, and “the departure time”, both describe a point in time with respect to a departure event.
- a value description describes the contents of an object. For example, “five oclock”. By itself it is only useful if context has been established;
- a full description contains the value description, or an anaphoric reference to it, as well as anchoring it. For example: “leaving at five oclock”.
- a compound description which explicitly combines anchoring with a value is also possible, for example: “the departure time is five oclock”.

Value descriptions will normally occur in cases of new information, or uncertainty about the reliability of information. Anchoring may be necessary when the context cannot be relied on. In this case it is important to anchor to material which the interlocutor knows about. Anchoring therefore works by searching for such material, under the constraint that it, and links back to the described, need to be lexically realisable. Determining how much to put in an utterance is difficult. In the case where the caller asks explicitly for certain information, such as the arrival time of a flight, the problem becomes simplified: describe what has been asked for. But this is not always the case, and it may be necessary to rely on a ‘user model’, or ‘user stereotype’, to control what information needs to be added to the description. With a dynamic component, such a model could also prove useful in formatting information: for example, in determining whether to use the twenty-four or twelve-hour clock, whether to refer to times as “our time” or “their time”, etc.

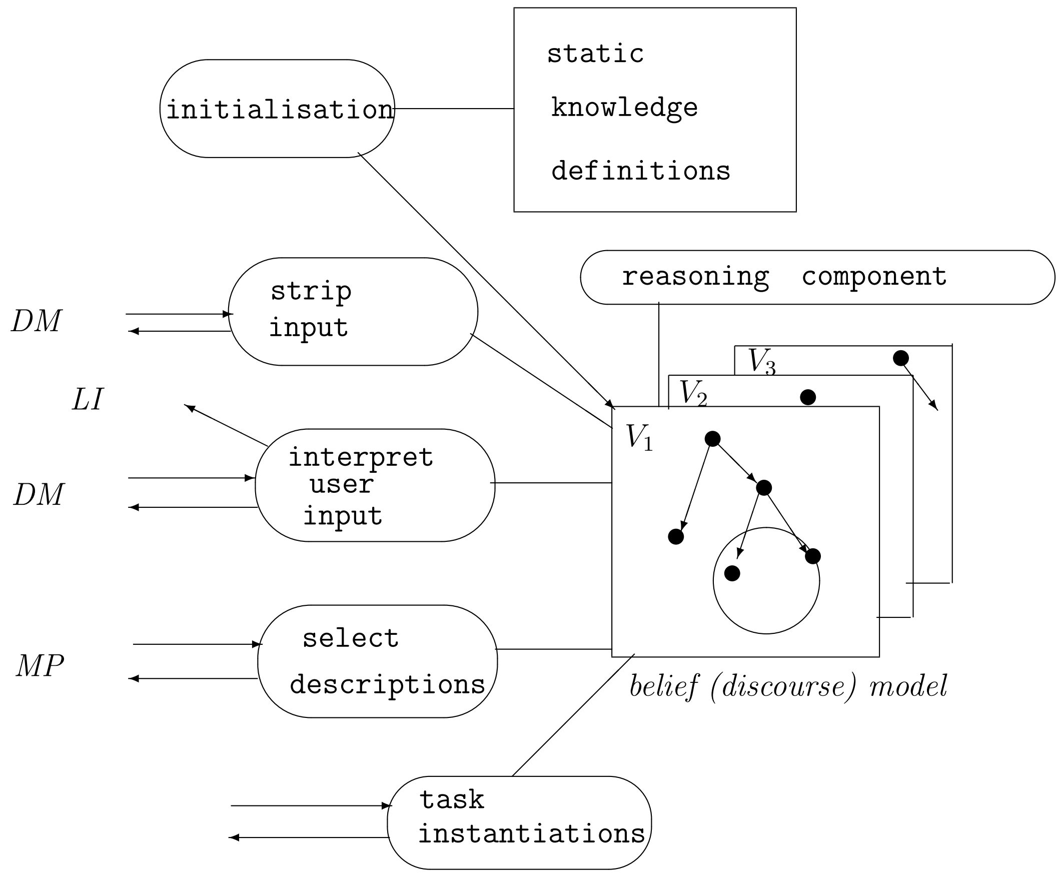

Figure 3.7: Belief module: main components

3.4 Algorithms

Figure **??** presents an overview of the functionalities of the BM, showing its interactions with the DM, LI, MP and TM.

3.4.1 Initialisation

The knowledge definitions provide generic information about classes; an object-centred representation is built to hold this: class objects are linked by inheritance links to parents, and by argument links to objects representing value restrictions on roles. Where constraints indicate path equations, code is 'planted' in the root classes of such constraints which ensure that if an instantiation of one path is built, instantiations of the remaining paths in the equivalence set are ensured.

3.4.2 Stripping input

The input is decomposed until a portion is found whose head type belongs to the belief knowledge definitions, and returned split into a meaningful part, and a 'rubbish' part. (Only rubbish so far as the belief module is concerned). This latter would include fragments such as the representation of "would you mind repeating ...": this can be interpreted directly in the dialogue module as indicating a particular class of dialogue acts, and does not require representation in the discourse model.

3.4.3 Interpreting utterances

We choose to describe the interpretation functionality of the belief module, in terms of a generic paradigm of writing information to the belief model, and reading task-relevant material. This is because the various functionalities encoded in the remote-request messages share a considerable amount of code in their inplementation.

3.4.3.1 Writing at an anchor point

An *anchor point* defines a node in the belief representation, in terms of a root id, and a path from that root. The anchor point is defined to be that unique object reached by following the path from the root. It may or may not exist previously as a belief model object. The basic routine is the following:

1. Establish a start point at the anchor point, creating a new object if necessary;
2. Start the recursive interpret procedure from that anchor point.

The recursive interpret procedure takes an Id and a piece of SIL representation, checks that the type in the SIL representation corresponds to the class of the Id, and then searches through the attribute-value pairs in the SIL representation, checking that the attributes are permissible for the start object, and if so, recursively calling the interpret procedure for the values. Bottoming-out occurs when leaf nodes are reached. Failure takes place if the class of object, or one of its roles, is incoherent with the knowledge definitions embodied in the class information for that object.

3.4.3.2 Searching for anchors

With the basic recursive interpret procedure in place, it is possible to search a list of anchor-contexts, one in turn, until a successful recursive interpret has taken place. However, it may be the case that no successful anchoring is possible given the proposed set of contexts. In this case, a search is made from the local task focus (defined to be the id of the current task), attempting anchoring at all paths (including the null path) from that root that have not already been visited, in a breadth-first manner.

Searching for a new task in the input uses variants of this mechanism: objects for each candidate task label are created in turn, and attempts to anchor take place, until a successful write has been performed. For the non-task-specific request, a more exhaustive search, as described above, is used.

3.4.3.3 Side-effects: constraint daemons

Currently the only constraints that are implemented, other than value restrictions, are ones that can take the form of path equations, for example

(8) Node: $reservation$
Constraint: $< theflight\ \ thedeparture\ \ thetime > \equiv < departure\_time >$

This says that for an object of type reservation, if the time of the departure of the flight is defined, it should be the same as the object in the (top-level) departure_time slot. This is a basic mechanism for converting between surface (linguistic-oriented) semantic descriptions, and task-oriented ones. It is implemented by the use of constraint daemons: whenever information is to be written at an attribute for an object, if this is a possible constraint root, then the particular branches of all matching constraint equations are recorded, together with residues from their equivalence sets. Otherwise, if the object is on a currently active constraint path, a check is made to determine if the attribute extends that path. When the end of a constraint path is reached, building of the remainder of the equivalence set is undertaken, creating new objects where necessary.

3.4.3.4 Interpreting descriptions

The term *description* is used narrowly here, to indicate semantic representations built like lamda abstracts. These have the general form:

$$\begin{bmatrix} ID : id_n \\ TYPE : \_ \\ MODUS : \_ \\ THEDESC : \begin{bmatrix} (expression \\ containing \;\; id_n) \end{bmatrix} \end{bmatrix}$$

The referent id_n is described in terms of some of the relations it partakes in. A description is invalid if it doesn't reference its ID somewhere within its $THEDESC$. Otherwise, it may be:

*referential***:** a discourse object already exists which complies with the descriptions within the $THEDESC$;

*attributive***:** no such discourse object exists; the description is then taken to specify a new discourse object, which should satisfy the constraints within the description. It must be such that the new object could come about via the system's knowledge—ie, as a result of interacting with the application;

*unacceptable***:** it is impossible to find a discourse referent, and the description makes no sense as an attributive expression. In this case it is to be treated as a case of presupposition failure.

In interpreting descriptions, whether embedded or not, the BM proceeds as follows: firstly, an attempt is made to find a discourse referent. This is done by finding out whether the variable acting as placeholder for such a referent becomes instantiated to an existing object as a result of interpreting the description. The description cannot be interpreted beginning from the anchoring context; instead, an object in the *accessibility index* (see Section **??** below) of the appropriate type, and with the appropriate uniqueness properties, if these are indicated in the $MODUS$, is taken to be the root of the description.

3.4.3.5 Modification of existing data

If part of the input contradicts existing data, a new *modify* view, inheriting from the current view, is created, and the information which is inconsistent written at that view. Thanks to inheritance, that information which remains unmodified is still accessible at the new view. The DM is subsequently informed about the inconsistency, and the two conflicting views.

3.4.3.6 Alternatives in the input

A similar mechanism is used when two closely competing alternatives arrive in the input from the user, in a packed representation—if, for example, there is uncertainty about whether “Paris” or “Perros” was spoken. For each alternative, a new view is created, inheriting from the current view. The DM is then informed about the existence of this ‘branching’ in the belief model, and all the (new) alternative views are returned.

3.4.3.7 Reading from the belief model

The core routine traverses objects in the belief module recursively, starting from a specified root object. For a given root object, only those objects which are accessible at paths specified in that object’s need-to-know definitions are visited, and the information found there is returned, as an attribute-value (or *dag*) structure. That part of the information corresponding to the anchor context of the request is returned to the DM as its result; any residue is sent separately, and may be used in the interpretation of overloaded answers.

3.4.3.8 Status flags

The need for status flags arises because for correct dialogic interpretation of an utterance it is necessary to know the status of its referential interpretation. For example, if the material is a repeat, it may be interpreted as a request for confirmation, or a confirmation, depending on context. After successful writing and reading of information requiring interpretation, a status is assigned as follows:

- if interpretation took place successfully at a proposed context, $status = direct$.
- if interpretation took place successfully, but not at a proposed context, $status = indirect$.
- if interpretation took place successfully, but no new information was found in the input, $status = repeat$.
- if interpretation took place, but previously existing structure had to be modified, $status = modify$.
- similarly, if the interpretation leads to a branching under alternative views, $status = alts$.

- in cases of complete failure, $status = fail$.

The direct/indirect distinction is made by looking at the nature of the search, both for writing and reading, that was required. The repeat and modify values are capable of overriding direct/indirect values, and are obtained by looking at the set of objects traversed, to find out (respectively) whether they are all old objects, or whether they contain at least one modify object.

3.4.4 Generating descriptions

Because descriptions need to be lexically realisable, the portions of the semantic network they describe need to be built of links and nodes which are lexically realisable. There are three major kinds of description:

1. a *null* description of a node, consists of description of the contents (or value) of the node. It is only meaningful if the node can be evaluated—ie, there is something in the node to describe;
2. a *local* description is an extension of a null description, which contains a description of the node contents, with enough of its environment to include some node which is mutually known. A local description of a set of nodes attempts to locate them all within a single spanning environment;
3. an *anchoring* description does not attempt to describe the contents of a node, but the constraining environment of which the node is part. It corresponds closely to the Natural Language phenomenon of functional definite descriptions. For example, if **flight_3** is a flight with associated departure event **departure_01**, which in turn has a time role **time24**, the NL equivalent of an anchoring description for **time24** would be

   (9)
   the departure time of the flight

   This would be successful so long as *the flight* was capable of specifying uniquely, given the discourse context, and so long as the path

   $<thedeparture\ thetime>$

   uniquely defined an object, given **flight_3**.

A final kind of description is an alternative to a local description; it describes both the value of an object and its environment, by producing the pair:

$<$*anchoring_description*, *null_description*$>$.

This is useful for producing copular and existential descriptions, for example:

(10) *the departure time of the flight is six fifty five*

It is produced by running both the null and the anchoring description routines for the object in question.

3.4.5 Contextualisation

There are a number of quasi-orthogonal ways in which context dependence is represented:

- through the use of anchoring paths, the dm may require that certain information, such as the answer to a query, is anchored at a particular place;
- through the use of views. Interpretation and generating descriptions is normally relevant to the current view;
- through the use of focal registers which maintain knowledge about the *accessibility* of discourse objects.

Accessibility information is particularly useful when generating or interpreting anaphora, or reduced definite descriptions. An *accessibility index* is maintained for the current task: every instance which is made during the current task, or which is transferred to it, is registered, by type. Accessibility will then depend on whether a discourse object is unique to its type or not. 'Accessibility frames' for earlier tasks are also kept—however, objects which are only available through these are considered to be less accessible.

3.5 Knowledge bases

3.5.1 What knowledge is needed

Knowledge is loaded at initialisation time from a file (known as the **knowledge definitions file**); this contains knowledge about the organisation of classes and constraints for a particular domain. The BM itself is entirely domain independent.

The contents of the knowledge definitions file may be regarded as comprising *role definitions, type definitions, constraint definitions,* and *need-to-know definitions*:

role definitions for an object of a given type, define what attributes it can have, and what types the value of these attributes are restricted to have. In addition, they specify whether a role is functional, or one-one. In the latter case, the role function can be taken to have an inverse, but this is not currently used;

type definitions define for every object class, what superclass it inherits from. Currently the use of multiple inheritance is restricted to the cross-classification of some classes as *non-informational* (and therefore not permitting instantiation within the BM), and *task*. Single inheritance ensures that properties defined higher up the hierarchy appear on descendants. There is no local overriding of defaults;

constraint definitions are defined on object classes, and stipulate the constraint equations that have to apply to instances of a given class;

need-to-know definitions stipulate for a given class of objects, what properties of the object need to be 'known' (ie recursively), for the object itself to be deemed known. For example, for a date, currently the month and the day need to be known. This is the main way of declaring that knowledge is sufficiently specified to be known to the task module.

Two forms of knowledge definition file are currently available: **presil** and **sil**. The presil form exists because the BM was first implemented before the SIL language had been adequately developed and described. Although the syntax of the two knowledge definition file formats is different, they contain approximately the same knowledge, and the difference is transparent to the BM.

3.5.2 Presil knowledge definitions

A presil knowledge definitions file is treated as a data file providing definitions for four predicates:

1. **type_isa(SubType,SuperType)**: describes the ISA links between classes.

2. **roledef(Type,SlotLab,SlotType,RelationType)**: describes the slot **SlotLab** of **Type**. A type will in general have a number of roledef/4 clauses defined for it. These do not necessarily exhaust its roles, because

it may inherit others. **SlotType** references another type which must be described in the knowledge definitions file; alternatively, if the role takes an atomic (string or number) value, it may enumerate the set of possible values, using the following syntax:

litval(ListOfValues)

If **ListOfValues** is variable, no restriction is put on the set of values. **RelationType** may be used to specify if the relation so defined is one-one, or simply functional. This feature is not currently used.

3. **constraint(Type,PathEquation)**: describes a path equation which is required to hold for instances of **Type**. PathEquation is a list of two or more lists, each representing a path defined on an instance of **Type**. Care must be taken to ensure that every such path can exist; the initialisation routine which reads the knowledge definitions file does not currently complain if this is not so—but inferences based on that constraint will not of course be possible. Any number of constraints may be defined for a given class of object.

4. **need_to_know(Type,Paths)**: **Paths** is a list of paths which must be known, for **Type** to be known. Only one such clause per type.

There are currently two presil knowledge definitions files in existence: one in the domain of flights, and one in the domain of home-banking transactions. Example clauses for flights are given below:

```
type_isa(temporal,entity).
type_isa(event,temporal).
type_isa(reservation,event).

roledef(reservation, theticket, ticket, functional).
roledef(reservation, theflight, flight, functional).
roledef(reservation, departure_date,date,functional).
roledef(reservation, arrival_place, place, functional).

constraint(reservation,[[theflight,thedeparture,thetime],
        [departure_time]]).

need_to_know(reservation,
[[departure_date],[arrival_place],[departure_place],
         [departure_time]]).
```

3.5.3 Sil knowledge definitions

Sil knowledge definitions are contained in a single file, **sildefns.pl**. Because the SIL language has been defined to be extensible to a number of different domains, it is not currently envisaged for there to be more than one knowledge definitions file.

Issues regarding compatibility between TM and BM representations, and the inferences which will allow the former to be derived, are discussed in Section **??**.

3.6 Customization

It is currently the case that only the knowledge, and not the behaviour of the BM, is customizable. As will be clear from Section **??**, at least two forms of knowledge definitions are possible. However, certain generic attribute labels are referenced in the code and care must be taken if they are changed.

These are:

id
type
modus
thedesc
thevalue

If presil knowledge definitions are being used, only **id** and **objname** are used. The latter label is the precursor of **type**, and is retained for downwards compatibility. However this has been factored out from the BM code. On initialisation, the user is prompted to choose between sil and presil knowledge definitions, and the fact **type_label/1** is defined accordingly.

As proof of its portability, it should be mentioned that the BM has been tested fairly rigorously, both in isolation and with the full dialogue manager in place, with presil knowledge definitions for both flights and home-banking transactions. However this depended on well understood customisations of the task module existing for the two domains. In the following subsection issues of customisation for a particular application are discussed.

3.6.1 Customisation for a particular application

The SIL knowledge definitions as they currently stand are aimed at defining possible semantic representations as they appear at the linguistic interface. These may be referred to as *lexically-oriented* representations —see Section **??**. However, as we have discussed, representations which are suitable for presentation to the application may not coincide with lexically-oriented ones. We may speak of *task-oriented* representations, with the caveat that in general the intersection between these and lexically-oriented representations need not be empty. Figure **??** exemplifies the situation. I will assume that a *task object* is an n-place relation which the application database knows about. For ease of exposition, suppose that it is 'flat' in the sense that all of the parameters may be treated by the application as atomic values. The BM can maintain an instance of a query, as a structured object whose root is an instance of the relation name (**taskobj1** in Figure **??**), along with the parameters of the query as fillers of its roles (**param1** . . . **param3** in Figure **??**). As values of unknown parameters become known, they may be added, where necessary using alternative views to permit a number of alternative solutions.

On the other hand, the portion of the belief model which maps directly onto a lexically-oriented representation (the *lexical projection*), will tend to be embed-

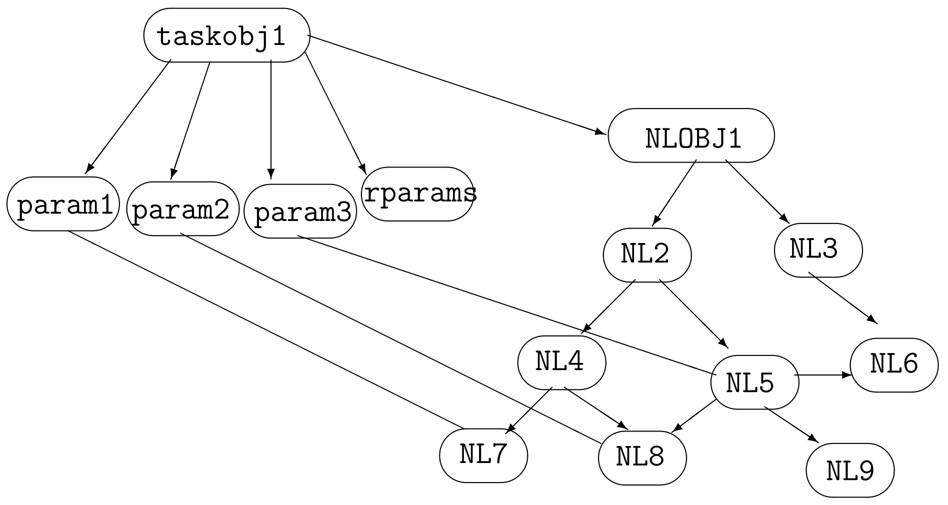

Figure 3.8: The problem of task inference—matching representations

ded and complex. The constraints that form part of the knowledge definitions therefore need to:

1. define which params of a given task object the system needs to attempt to evaluate;
2. define the correspondences between lexically-oriented nodes and task-oriented nodes, if these should differ.

In Figure **??**, for example, this is done by having **NLOBJ1**—and instances of other lexically-oriented classes from which tasks can be inferred—as a slot of **taskobj1**. Supposing that the constraints on **param1** and **NL7**, and **param2** and **NL8** require them to be identical, this can be enforced by the appropriate path-equation defined on the parent class of **taskobj1**. **Param3**, on the other hand, may be derivable as a function of some lexically-oriented node or nodes—in this case, **NL9**.

Even if the definition of relations in the application doesn't allow them to be entirely flat—this would be the case for example, for parameters which take time points (hrs, mins), or time-intervals (time-point1,time-point2) as values —the above principles still apply.

The implications of the above discussion for customisation can be summarised as follows:

- It is necessary to define the basic relations and their parameters for a given application. This has already been done, to a certain extent in SIL, but will inevitably need redoing for a new application. For the sake of compatibility, task relation names and parameter labels should be uniform across the BM and the TM.

- For instances of a task-relation type so defined, it is necessary to specify which slots are knowable/need to be known by the TM.
- Sufficient constraints need to be established, to enable task-oriented information to be inferred from lexically-oriented information.

It should be mentioned that in principle these constraints will be bi- directional: that is, a lexically-oriented representation of task-oriented information is inferrable.

3.7 Implementation

3.7.1 Overview of messages affecting the belief module

The request/return_anchor* messages currently form the backbone of the BM's work. These may be characterised as attempts to interpret input information, in SIL form, according to the internal belief model and current context, and issue interpretations which are relevant to the task module, and status information. Information is written to the belief module, or if present checked; at this stage, constraints apply and unanticipated structure may be built. Finally, that information which is relevant to the task is read off; typically this is divided into two parts: that part relevant to the context of the question, which is returned with the appropriate 'return-' message; and the residue, which is returned with a task_info message. Further side effects are the sending of doubtful_id messages for objects with doubtful scores, and the issuing of bindings information which establishes an equivalence between the arbitrary identifiers used for objects in the input, and canonical internal object identifiers which can be guaranteed to persist. In the following sections, we consider in more detail the relevant functionalities.

We may class messages in the Dialogue Manager as a whole, as either requests, responses, or updates, and cross classify these as 'local' or 'remote', depending on whether they originate in this module, or in some other wp6 Module. The messages affecting the BM can then be described as follows:

message predicate	module interaction	classification
request_stripped/2	$dm \rightarrow bm$	remote-request
return_stripped/4	$bm \rightarrow dm$	local-response
request_anchor_task/3	$dm \rightarrow bm$	remote-request
return_anchor_task/3	$bm \rightarrow dm$	local-response
request_anchor/3	$dm \rightarrow bm$	remote-request
return_anchor/4	$bm \rightarrow dm$	local-response
doubtful_id/3	$bm \rightarrow dm$	local-update
task_info/2	$bm \rightarrow dm$	local-update
bindings/1	$bm \rightarrow li$	local-update
request_knowledge/2	$dm \rightarrow bm$	remote-request
return_knowledge/2	$bm \rightarrow dm$	local-response
request_description_null/2	$dm \rightarrow bm$	remote-request
return_description_null/2	$bm \rightarrow dm$	local-response
request_description_local/2	$mp \rightarrow bm$	remote-request
return_description_local/2	$bm \rightarrow mp$	local-response
request_description_anchoring/2	$mp \rightarrow bm$	remote-request
return_description_anchoring/2	$bm \rightarrow mp$	local-response
request_description_anchoring_null/2	$mp \rightarrow bm$	remote-request
return_description_anchoring_null/2	$bm \rightarrow mp$	local-response
request_view_ids/2	$tm \rightarrow bm$	remote-request
return_view_ids/2	$bm \rightarrow tm$	local-response

The **request_** and **return_** messages are related in the obvious sense.

A number of messages make use of a *context expression.* This is generally a triple: **ctx(RootId,View:Path)**; it is taken to mean: "use the object specified at **Path** from **Root** as the point at which information is intended to be written to or read from the bm, and adopt **View** as the current view while this is taking place. **View** may be omitted, in which case the current default view is used. In some requests, a list of views may be specified, in which case the request is carried out for each view in the list, and a corresponding list of results is returned.

Considering the messages in more detail:

request/return_stripped: Takes a surface semantic representation (SIL), and strips off that part which is foreign to the BM's knowledge definitions. This will be the part which is of value to the dialogic interpretation, not the referential interpretation, of an utterance. Both the stripped part, and the 'meaningful' core are returned unaltered, and the core is classed as being a direct representation of a task, or otherwise as 'unknown';

request/return_anchor_task: Takes a set of possible tasks (ie, a list corresponding to possible task classes), and returns if successful an id for one of these, and the information that was predicated about that task instance in the input. Also returned is a status flag (see below);

request/return_anchor: Takes a set of *anchor contexts* (rootid–path pairs) and returns, if possible one of these at which it was possible to anchor the input. The order of the anchor contexts is significant, in that it orders the search, so it is useful to put the more likely anchor contexts first. Also returned are a representation of the task relevant portion of the input, at the successful context, and a status flag, whose value may be one of:

direct: the input was successfully written to the context;

indirect: the input was successfully written; however retrieving the information at context could only be done indirectly via inference.

repeat: the information contains nothing new;

modify: the information is inconsistent with previous information, and has required a modification in order for it to be written;

fail: it has been impossible to interpret the input

If none of the input contexts are successful candidates, some further search is carried out in an attempt to anchor the material from the current task focus.

doubtful_id: This is a side-effect of any interpretation request, whereby if an object with doubtful score is detected, the DM is notified. It is proposed that in future releases, this may be changed to a message indicating that doubtful information has been interpreted at a conditional view;

task_info: This update message returns that part of the input which is task-relevant but which was not specified by the successful anchor context;

bindings: This update message is sent after all interpretation has taken place (typically to the LI module), to tell it what the internal ids for objects is to be.

request/return_knowledge: This is a general-purpose accessing facility; currently only its use by the dm is forseen. Input pattern: **know(Agent,Ctx)**; where **Agent** may be one of *system, user, shared*, or a variable to indicate don't care. Ctx is the expression **ctx(RootId,View:Path)**: **RootId** and **Path** are essential, but **View** may be omitted: in this case the current view is taken as the default. Output expression: **know(Agent,Ctx,ObjectId)**: where **Agent** corresponds to that stipulated in the input pattern. If this was underspecified, it becomes instantiated to one of *system, user, shared.*

request/return_description_*: These all return SIL expressions, which are lexically realisable. They differ according to the mode of description used:

null: describe only the structure referenced by the object at **Ctx**;

local: describe more than the structure referenced by the object at **Ctx**. If several contexts are specified, build a spanning description;

anchoring: produce a description that doesn't reveal any of the internal contents of the object at **Ctx**—this may not be known—but anchors it for the user in terms of discourse objects which s/he does know about. This anchoring may be likened to that performed by a context expression. The principle difference is that the anchoring must be to an object the user knows about, and must be via links and objects that are lexically realisable;

anchoring_null: Build two descriptions, one as for anchoring, and one as for null (for use in generating copular and existential descriptions).

request/return_view_ids: given one or more inputs, which are instantiations for some current requirement, return (in that order), a set of views at which the information has been written to the BM.

3.7.2 NOOP

The first version of the BM used **protalk**, a Quintus library designed principally for extending **proWindows**; this had a number of disadvantages. The current version uses Nigel's Object Oriented Package (NOOP), designed and implemented by Nigel Gilbert. Features of this which the current implementation has taken advantage of include:

Multiple inheritance: This is used to implement the class hierarchy. Nodes representing classes inherit from nodes representing their super-classes. Instances inherit from the class immediately above them. Although NOOP does not make a principled distinction between classes and instances, the local definitions of roles enable this distinction to be drawn when needed. Multiple inheritance does not currently play a large part in SIL. It has been used however in the PRESIL knowledge definitions, to distinguish between informational and non-informational, and task and non-task classes.

Daemons: These cause specific actions to be triggered when values are put in slots. They are used to propagate path equations, as follows: when a value is placed in a particular slot, if the establishment of this link could possibly contribute towards some path-constraint, then a daemon is planted at the slot. This is a *parameterised daemon* which records something of the conditions under which it was planted—in this case, the possible extensions of the current constraint path. Triggering it will cause further constraint daemons to be planted further down the path, or may result in the finished constraint-path being triggered, leading to a functional evaluation taking place, or an equivalent path (pointing to the same target object) being built.

Views: What portion of the information in the belief model is currently visible depends on the view taken. Views are inheritance-based, so that information visible in an earlier view $View_1$ is visible from $View_2$ inheriting from $View_1$, unless it is locally overridden by information in $View_2$ or some intervening view. This facility is used to implement modification and alternatives, and has other potential uses. Because daemon creation and propagation is view-dependent, side effects which are caused by constraints are only relevant to the view in which the triggering information was added, or one of its descendants.

Backtracking: Creation of new objects, and slot assignment, are all reversible on backtracking. This is a very powerful feature which allows the side-effects of unproductive search to be undone, without any additional search. It does require the implementer to be very careful with the use of cuts however. A cut should never block return to a side-effect creating call which may need to be undone. The facility has been extended within the BM implementation, by a backtrackable *gensym*. The advantage of this is that the results of test runs remain relatively stable, from run to run, even when the search strategy has been considerably modified (though this cannot of course be guaranteed).

Although chronological backtracking is thus facilitated, backtracking which makes use of longer term dependencies has to be implemented manually. Currently this is only done to the extent that certain side effects during interpretation are saved at a temporary view, which is expunged after every interpretation cycle.

Multiple inheritance of views could be implemented in NOOP, in a manner analogous to that of multiple inheritance for objects. This would enable the intersection of two orthogonal choices to be brought into focus—as for example, if the user wanted to explore one of three departure times, on one of two possible dates. For each choice combination, a view which inherited from both the relevant alternative views could be created.

3.7.3 Testing

A full mechanism for testing components of the SUNDIAL Dialogue Manager has yet to be evolved. Ideally, this should involve the testing of components in isola-

Test name	Description
test_bm1	tests request_stripped with two examples
test_bm2	tests request_anchor_task, then two request_anchor's, the last involving a modify
test_bm3	tests request_anchor_task, followed by 3 request_anchor's.
test_bm4	similar to the above, with a lot of search to find suitable anchor contexts
test_bm5	an example based on the calls to the bm from an actual run of the system, which needed debugging
test_bm6	similar example, with modification and descriptions
test_bm7	alternatives in the input, and descriptions
test_bm8	alternatives from the TM, and descriptions

Table 3.1: Examples currently available in the BM test harness

tion, as well as tests of the integrated system. Testing a component in isolation poses certain challenges: test material should correspond to mechanisms which have been designed to cope with specific phenomena. However such phenomena are usually dependent on a certain preceding discourse having taken place. For example, to test the ability of the BM to handle modifications or repeats, similar or identical material to the current input must have been uttered on some previous occasion. In the same way, to test the ability to cope with a task shift, a previous task must have been established. The current set of tests (see Table **??**) establish previous contexts for a given phenomenon by chaining together a set of invocations of the belief module, so as to cumulatively build up a context. A more principled solution, in which the tests are directly designed in terms of the phenomena they are supposed to cover, would be to determine, for a given phenomenon, exactly what context it presupposed, and to compile a representation of that context, modulo any arbitrary irrelevancies, into the belief knowledge base, at an initialisation phase prior to running the test. Such a mechanism could be extended to testing some larger subsystem of the dialogue manager, so long as it was possible to define both the test material and its presupposed context, and compile a schematic version of the latter into the relevant knowledge bases.

3.7.4 The current software structure

The belief module software consists of the following files:

accessible.pl
bm.pl
bm_test.pl
bminit.pl
bminterp.pl
bmsys.pl
bmtools.pl
constraint_daemons.pl
describe.pl
silcustom.pl
sildefns.pl
silio.pl

The root module, which handles the interfaces with the rest of the Dialogue Manager, and which must be loaded to load the other modules, is bm.pl. bm_test.pl is a test harness, which is logically separate. Loading it will load bm.pl and the other modules. The principle modules, with their entry points, are summarised in Table **??**

3.8 Conclusion

A number of implementation and design tasks still lie ahead. On the implementation side, **get_description_local** and **get_description_anchoring** are not yet fully implemented and tested. Currently a number of search strategies are being explored for these. The **request/return_knowledge** protocol with the DM is also in need of implementation. The breadth-first search undertaken as a result of failing to interpret an input at any of a set of contexts currently halts after the first level. A deeper search could be implemented, but considering the present cost of search, would need to be guided by heuristic means.

Only those constraints representable as path-equations are currently implemented. Functional constraints represent a challenge—not so much in the implementation, but because they need to be reversible. Constraints, it should be remembered, are used symmetrically, both to derive task-oriented information from lexically-oriented, and vice versa. A simpler solution than to try to make functions reversible (they may after all have more than one argument) is to define where necessary reverse mappings. For example, the function which maps the pair $<hour,\ time\_of\_day>$ onto 24-hour values, could have as its inverse a pair of mappings, one mapping a 24-hour value onto a (12) hour value, the other mapping a 24-hour value onto a time of day. This increases the considerable burden on the knowledge engineer on ensuring consistency between constraints.

Module Name	entry points
accessible	accessibility/3 accessibility_type_indices/2 add_frame_to_accessibility_index/1 add_to_accessibility_index/2
bm	receive_from_pm_bm/2
bm_test	bm_init/0 run_all_tests/1 run_tests/1 run_tests/2 test_bm1–8/0
bminit	init_bm/0
bminterp	interpret_a_task/5 interpret_in_ctxs/5 strip_rubbish/4
bmsys	record_tm_alts/3
constraint_daemons	propagate_at/3 start_constraint_daemons/1 value_restrict/4
describe	get_description_local/2 get_description_null/2 get_description_anchoring/2
silcustom	silcustom/0
silio	dag_id_to_obj/4 dag_to_objs/3 desc_id_to_obj/3 write_need_to_knows/3 write_need_to_knows/4

Table 3.2: BM modules, with main entry points

Describing objects within alternative views is another challenge. We want to be able, for example, to refer to a set of alternatives as:

the first
the second
... the last

and so forth, and to interpret such descriptions. This suggests that part of the interpretation of an utterance may involve the selection of a view from a number of alternatives.

The use of views, and their management, is increasingly becoming an issue. They are currently only used for modifications and alternatives, and for temporary information. Their use could be extended to include:

- information with doubtful acoustic scores;
- default values;
- negative information: if the user says "I don't want to travel on monday", an instantiation of the user requirements but with the departure day as monday could be kept at a view, and future interpretation would then have to guard against information special to that view becoming validated;
- hypothetical alternatives.

However, before the views mechanism is exploited much further, it will become necessary to establish a system of *views management*. Thus, it is necessary to ensure that a proliferation of views do not unnecessarily block access to information. Rules constraining the creation of new views could help here—for example, don't create alternative views while there is a modify view still in existence. Views need to be typed, according to how they relate to one another, and to the current view. An alternative view inherits (with no overriding) from a previous view; a modify view overrides. However alternative views are mutually incompatible with one another.

A mechanism also needs to be established for ratifying information. It is suggested that, to the extent that ambiguity in the belief model is represented via branching views, ratification comes down to choosing, either as the result of interaction with the user, or as a default, between competing views.

Chapter 4

The Task Module

4.1 Overview

This chapter describes the implementation for the intermediate demostration system of the Task Module, which is designed to be a submodule of the Dialogue Manager in the SUNDIAL system. In chapter five of the first deliverable of WP6 one of the main features of the architecture of the Dialogue Manager is concurrency, ie. the modules are to be viewed as communicating experts. The following specification shows that the present implementation of the Task Module fits well into this concept.

The Task Module maintains its own history and channels lookup/update requests from other modules. Part of the knowledge resulting from the interpretation of an utterance is being received and processed by the Task Module. So, the Belief Module sends a context–dependent referential interpretation of the user's utterance to the Task Module — via the Dialogue Module. This interpretation should be suitable for the Task Module to make a task interpretation. The Dialogue Module may merge it with a dialogue goal before passing it on.

The Task Module does not determine the dialogue strategy of the SUNDIAL dialogue manager nor interact with the user even if the terminology in the present document might suggest this in some parts. The general mechanisms of the Task Module on the one hand try to support as much as possible a cooperative system dialogue strategy towards the user and on the other hand not to impose unnecessary constraints on the other modules' strategies.

4.2 Scope

The design and implementation of the Task Module has been realized according to the functional description included in the functional specification of WP6. The present implementation of the generic Task Module is able to handle **all the functions** described in the document:

- Establish Tasks and Task Parameters,
- Check Specificity,
- Access the Application Database,
- Present Responses from the Database,
- Relax Constraints on the User's Requests,
- Make Task Specific Inferences,
- Supply Default Knowledge and
- Check Reliability.

The general idea has been to provide a generic module which can be used for different applications when providing the respective adaptations. The architecture of the Task Module includes a proper submodule which defines the interface to the application specific knowledge.

4.3 Principles

The design of the generic Task Module is based on the principle of a clear separation between application dependent and generic, ie. application independent knowledge. Thus the interfaces between the generic mechanisms and the application database on the one hand and the task dependent knowledge on the other hand are clear cut. There is a layer of generic data base accessing procedures, which make the representation of application dependent knowledge transparent to the generic modules. When changing the format of knowledge representation only the respective accessing procedures have to be adapted accordingly. Task dependent knowledge can be defined in a rather flexible way, since the representation has been kept in a declarative mode, apart from rules for inferences, consistency checking and constraint relaxation.

As a another general feature, the Task Module allows handling of several tasks at a time. Different task instances are being stored in a queue and processed according to the principle of first-in-first-out. This processing strategy may be subject to future change and to parameterization.

4.4 Algorithms

4.4.1 Overview

The generic Task Module has been designed to work with different application domains, such as information seeking enquiries on *flights* and *trains*. Several different services may be distinguished within these information areas. For example, an airline information system may include enquiries on *flight connections*, *fares* and requests for *ticket reservation*. The design of the Task Module allows to specify these as different *task classes* together with certain features needed to process the enquiries. A *task instance* or shortly *task* is being established as soon as sufficient information to be able to identify a task class is being given to the Task Module.

Each task created undergoes certain stages of processing with the goal of performing the respective database transaction. Generally certain basic mechanisms, such as *collecting parameters*, *querying a database* and *presenting the result* of the query are involved with all task classes.

It is the responsibility of the task module to provide these basic mechanisms, supporting cooperative behaviour of the system in its dialogue strategy towards the user. This cooperative behaviour is achieved by providing features such as *default values* and *parameter inferences* to abbreviate the parameter collecting dialogue, *constraint relaxation* in order to improve the effectiveness of database queries, and *ordering*, *reduction* or *summarizing* of query results.

Additionally, a mechanism has been designed for the management of *multiple task instances* at a time, which could be needed if the user wants to provide anticipated information on his next request while the current one has not yet been finished. So, while talking about flight connections a user might say that (s)he wants a booking later which in the respective application dependent knowledge may be defined as a different task. (S)he may then give her/his address which will be needed only for the following booking task. In that case the Task Module saves this information, storing it in a new instance of the task class "booking" in its internal memory. After the finishing the current "flight" task, it will then take up the waiting "booking" task and process it. The sequence of processing task instances has been implemented according to the principle of *first-in-first-out.* It may be an issue for further development of the Task Module to parameterize the strategy for this sequence.

Once a task has been finished or ended, the generic mechanism allows to continue with another task which has been defined to have a senseful relation to it. This *task continuation* or "linking" facility allows the transfer of known parameters from one task to another. For example, this would permit a comfortable fare enquiry or ticket reservation for a connection specified in a previous timetable enquiry, because these task classes have many common parameters.

The processing or "lifecycle" of a task has been divided into five main parts or *task stages*:

- *Initialize task:* Initializes the internal memory of the TM for the respective new task instance. According to the application knowledge, defaults for obligatory parameter values and asked parameters are being entered into the internal memory, as far as they could not be transferred from a task already processed. In the latter case they would have already been stored in the task instance.

- *Collect parameters:* Requests for the necessary parameters are generated until the incoming answers have provided enough information for a database query.

- *Database query:* A database query is performed. If it was not successful, parameter values are relaxed or a request to change some of them is being issued.

- *Present result:* The database entries that have been found are being presented. The user may request additional information on these or may select a specific one. The TM also offers possible tasks to continue with from which one may be chosen.

- *Database update:* After an unique entry has been selected and the user has provided an authorization, additional parameters needed for a reservation, e.g. name and adress of the user, are collected, a reservation transaction is performed, and a confirmation message about its success or failure is issued. Here, too, continuation tasks are offered on termination if any have been specified.

A more detailled description of the implementation of the different task stages will be given in the following sections. The task stages are intended to be executed subsequently, but there are some points where execution can return to a former stage. After the presentation of a query result or while collecting additional parameters for an update, it is still possible to change the values of some obligatory parameters, which will cause the processing of the current task instance to return to the database query stage again, because the current query result has become inconsistent with the new parameter values.

4.4.2 Saving state

There are different kinds of information that the task module has to gather and store for each service that the user requests. An object-oriented notion allows to store this information in a coherent manner, making the current state of each task instance accessible through a task instance identifier and a set of access and modification procedures. The task state essentially consists of the following parts:

- The identifiers and class names of the existing task instances.
- The parameters which are necessary to specify the requested services. For each of these parameters, its value and origin have to be stored. The generic task module is able to store parameter values without imposing any constraints on their format. When values have to be compared or manipulated, for example during constraint relaxation or database queries, application specific procedures are used for these purposes. It has been considered that there are three kinds of parameters:

    1. *Obligatory parameters* are the parameters needed to guarantee that a database query or update operation can sensefully take place. The task module will actively ask for any of these parameters as soon as a specific service has been requested. In the application domain of train connections, for example, no senseful database enquiry can be realized without knowing a departure and arrival place and at least either a time of departure or one of arrival.
    2. *Optional parameters* are those which can contribute to make a database query or update more specific, but will not cause failure of these operations if they are missing. They provide additional specifying information for the database query. For example, for a query on train connections certain itineraries may be specified. The task module will not ask for these parameters, but leave the initiative to the user.

       The set of optional parameters is per definitionem disjunct to the set of obligatory parameters.

3. *Asked parameters* are parameters which the user expects to receive as part of the answer to an enquiry. They can be set up either by requirement of the user or by default (i.e. the application specific knowledge of what parts of a database entry should be presented to the user as a minimum). This set of parameters is not disjunct to the sets of obligatory and optional parameters. It may support the cooperativeness of the system strategy to repeat some of the given parameters to the user together with the result of a database query.
4. *Update parameters* are additional parameters needed for tasks that allow an update operation, e.g. name or adress of the user. They will be treated like obligatory parameters. However, they will only be requested from the user after (s)he has selected a unique item (flight, hotel room etc.), i.e. after the "present result" task stage.

- Control information about the development of the task. It is necessary to store which of the task stages is currently active for each task instance, and whether some parameters have been excluded from constraint relaxation.

- The database entries found as result of an eventual database query. They can be used on request to provide further information if the user has not asked for all available parameters at first. A single entry could be selected as argument of an update to a reservation database.

4.4.3 Processing messages and scheduling tasks

The messages which other modules send to the task module can be subdivided into two categories: Messages that affect the task state, and Messages that have only informational character but do not contribute directly to the progress of a task. The latter are requests by other modules for the current state of processing of a given task and the messages exchanged with the Belief Module to maintain coherency between task parameters and task processing state and the respective knowledge representation(s) in the "views" (see below) in the Belief Module. This kind of messages is processed using a comparatively simple suspend/resume mechanism for the whole Task Module, as processing has to be resumed at the same place where the request message for a view-identifier has been sent out.

Each time a message of the first category arrives, the task module's top–level procedure `process_message/1` is activated. This procedure has two main duties: It controls the *creation and termination* of task instances, and it performs *consistency checking* and assertion of all incoming *new parameter values*, confirmation or selection of messages. As a control feature, the *task continuation* mechanism is also implemented in this procedure. It should be emphasized that the TM is able to process any input at any time it can be called by the "Postie" module. Messages can lead to the following operations performed by the top-level procedures (for a formal description of the messages see “external interfaces”):

- *Creating a new task instance:* If a message states that a new instance of a specified task class is to be created, the TM will create this task instance in its dynamic memory and assign to it the unique identifier given by the BM or the DM. The dialogue Module will be informed of the identifier, and any new parameters delivered together with the instance creation message will be stored. Being able to create a new task instance at any time, even if the former task has not been finished, means that the TM has to be able to manage multiple tasks simultaneously. For this reason, any information and messages concerning a task instance will be tagged with its identifier in order to allow indexed storage of the tasks.
- *Terminating a task instance:* On the respective message, all information concerning a specified task instance is being erased from memory, making it thus unavailable for further processing. An issue for further development may be to save the information of the previous task instances.
- *Continuation of a task with another:* If a specific task is to be continued with a task of another class, an application-specific list of parameters will be transferred as default values for the new task, and the other values will be deleted. The list of transferred parameters consists of correspondency pairs which allow, for example, exchanging the destination and departure airport if a flight enquiry is followed by another enquiry for the return flight. The new task instance will have the same identifier, but a new class, and start in the task stage “initialise task”.
- *Selecting a solution* or *Authorizing a database update:* These two actions are only expected to happen when a database query has already been performed, and they will cause the TM to switch execution of the specified task instance to the next stage.
- *Excluding a parameter from constraint relaxation:* Some parameters can be excluded from constraint relaxation, e.g. because the user has stressed them particularly. *Remark:* This feature has not yet been implemented, but the respective message has been defined as it could become very useful in the future.
- *New parameters:* If a message delivers one or more parameter values or names of asked parameters, they will be checked for consistency—according to task dependent consistency rules—and then stored in the TM memory. If an inconsistency occurs, an error message (suggesting a default value

if possible) will be issued and the TM will be suspended. If there are pending parameters from a multi-parameter input in such a case, they will be preserved for processing together with the next incoming parameters.

If the task has already progressed so far that a query result has been obtained and saved in the task status, this query result will become inconsistent if any of the parameters used in the query is being changed. For this reason, the saved list of solutions will be removed and the task stage is being set back to "database query" in such case.

After the information contents of an incoming message have been processed, the task scheduler procedure `task_control/0` is being called. This will then select one of the currently existing task instances and resume its execution, which will continue until an information request message is sent out and the task module suspends itself again. If there is no executable task instance, a `request_start_task` message describing all existing task classes is being sent out. The algorithms for the different task stages will terminate successfully if they have not sent out any request messages. Thus the top-level algorithm of the TM is the following:

1. Receive an incoming message and determine the changes in the task state that it contains. If a new task instance is created, send out a notifying message:
   `update_start_task/2`
   If an inconsistent parameter is being detected, send out one of two messages asking to correct it:
   `request_correct_parameter/4` or
   `request_confirm_correct_parameter/5`

2. Select one of the existing task instances (according to the principle first-in-first-out); if none exists, send out a `request_start_task` message and terminate.

3. Call the task stage algorithm for the current stage of the selected task.

4. If the task stage algorithm terminates successfully, switch to the next stage and go to previous step; else terminate.

4.4.4 The task stages

The following subsections describe the functionality, algorithms and some details of the implementation of the submodules for the five task stages. Each of them contains one main procedure which is called by the scheduler procedure `task_control/0` when a selected task instance has reached the corresponding stage. The main procedures run the basic algorithms of each of the stages. After being called, each of the submodules can either achieve its result successfully, or send out a request message because an interaction with the user seems to be necessary. In the first case, the scheduler procedure will immediately call the next stage, in the second case the task module will be suspended. Even if there may exist a desired reply message, it has to be remembered that all other messages to the TM are also allowed at any time.

4.4.4.1 Task initialisation

In this stage, initial values of control parameters and defaults are being written into the memory of a newly created task instance. The following steps are performed:

1. The control information is initialised and the query result is set to be empty.
2. Some of the parameters of the task instance may already have default values at this time, because they might have been copied from a former task instance from which the currently active one was called as a continuation. For this reason the existing values are checked first, and then only the default values for those parameters that do not have a value are retrieved from the application knowledge and written into the task instance memory. For each applied default, send out a notifying message:
   `update_default/3`
3. Finally, all default asked parameters are retrieved and written into memory.

4.4.4.2 Collecting parameters

The aim of this stage is to collect enough parameter values to enable a senseful database query, or at a later step to collect additional parameters for a database update. For each of these two cases, a set of minimal parameter sets is specified in the application knowledge. If the values of all parameters contained in at least one of the minimal parameter sets are known, the query for a database access is assumed to be well enough specified and thus possible.

All the parameters that appear in these minimal parameter sets are obligatory parameters, and the procedure will request their values one by one in the order specified in the task-class-specific obligatory parameter list (this does not mean that the TM could not receive them in a different order). This will be repeated until a database access is possible. Each time a request is sent out, the TM will suspend itself.

Before a request message for a certain parameter is sent out, the procedure tries to derive its value by making use of application-specific inference rules. Inferred values are assumed to be as reliable as those provided by the user and thus they are not presented for confirmation. There can be cases where an inference does not really provide a value, but only states that one or more obligatory parameters has become obsolete because another parameter implies a certain value for it. For example, knowing a flight number can make asking for departure and destination place unnecessary, although it might be preferred not to infer the values of the latter parameters directly because this could involve a costly database access.

If a request message is necessary, a default value will be provided as suggestion if possible, but default values always need a confirmation. Thus there is the following algorithm:

1. Check if a database access is possible.
2. If not, find all unknown obligatory parameters and try to derive their values through inferences. If an inference succeeds, send out a notifying message:
   `update_inferred/3`
   Check again if a database access is possible.
3. If not, check if the next unknown obligatory parameter has a default value, and send out a request message with suggested value for that parameter:
   `request_confirm_parameter/3`
4. Else send out a request message without suggestion for the value of the parameter:
   `request_parameter/e`

Inference rules are represented as prolog predicates in the application knowledge submodule.

Values are being provided for all known parameters that have been supplied or confirmed by the user.

It would also be desirable to export inference rules as predictions to the Dialogue Module. They could be passed on to the linguistic processing as top down predictions limiting the range of possible hypotheses generated. So, there may be a prediction saying that inferable information will be uttered with less prominance, ie. focus marking by the user, since it is assumed to be known information.

An easy solution would be to allow only pairs or groups of corresponding parameters, i.e. only inference rules of the type "If Params A and B are known, C is not being needed". This would make it possible to export these inference rules as predictions in a simple form, while a more general kind of inference rules could lead to a very complex representation for predictions. The main problem is that if there is a rule that does not enable the TM to predict a set of possible values for a parameter directly, it could only send a syntactical representation of the rule to the DM, and the DM would have to be able to interpret this representation or to push it down into linguistic processing.

4.4.4.3 Querying the Database

In order to present the user a reasonable number of solutions for a database query, some different cases have to be distinguished. The interval of the number of entries that result from a database query has been divided into four subintervals which are characterized by the constants `MinGoal`, `MaxGoal` and `Threshold`. The intervals look like that:

```
0 ... MinGoal ... MaxGoal ... Threshold ...
```

`MinGoal` and `MaxGoal` describe the optimum range for the number of solutions, i.e. the range of numbers of solutions that can be presented to the user directly without requiring a summary. A number of entries within the interval from `MaxGoal` to `Threshold` will be tolerated too, and the result will be presented by a summary. The numbers of solutions below `MinGoal` or above `Threshold` are not acceptable for the presentation and thus the query has to be improved either by constraint relaxation or by sending a message requesting more specific or alternative values for some parameters. The algorithm of the procedure is as follows:

1. Build a query from the known parameters and get the number of resulting database entries.
2. If the number of entries is between `MinGoal` and `Threshold`, terminate successfully.
3. If it is below `MinGoal`, try to increase it by constraint relaxation. In case of success update the relaxed parameters in the task knowledge base and terminate successfully, after sending out messages informing of the relaxation:

`update_relaxed/3`.

4. In case of failure send out a message requesting alternative parameter values for some of the parameters used in the query: `request_alternative_values/2`.

5. If the number of queries is above `Threshold`, send out a message asking for more specific values for some of the parameters of the query: `request_specific_values/2`.

In case of too many results only parameters with values that describe an interval or are unknown are suggested for closer specification. If no results are found, all known parameters are suggested to be changed.

4.4.4.4 Constraint Relaxation[1]

The Constraint Relaxation Algorithm has been developed by IRISA. However, the University of Erlangen team has integrated this mechanism into the Task Module framework. Some application dependent information has been transferred to the module *application_knowledge*. A relaxation algorithm for intervals of days has been added, as well as the time relaxation algorithm has been replaced by a more sophisticated one. Other new specific relaxation algorithms have been added as well.

The main characteristics of the Constraint Relaxation Submodule (CRS) are the following :

- The CRS is designed to work on the flight database of the first version of the Task module (TM). This database is based on a single relation (universal relation).
- The CRS's aim is to provide new values parameters allowing a non empty response from the database. The CRS is called by the TM when the user's query has not answer from the database.
- The CRS contains some general algorithms for relaxing parameters values (ex : for relaxing `sourcetime` and `date` parameters).
- The CRS offers to the application designer the ability to define his own constraints relaxation algorithms.

[1]This subsection has been written mainly by IRISA.

Calling the constraint relaxation In the Task module, for calling the CRS, the following predicate will be used `tm_irisa_relax(+LP,-LP)`

Input parameters (+LP) The input parameter is a list with the following format: `[[par,value]...]`
`par` is a parameter name,
`value` is the value for the parameter.
Example: `[[sourcetime,(9,15)][class,economy]]`

Output parameters (-LP) The output parameter is a list with the following format : `[[par,value]...]`
`par` is a parameter name,
`value` is the new value for the parameter
The parameter(s) relaxed has (have) new value(s) and give(s) a non empty response from the database.

Customizing the constraint relaxation

- declarating the parameter to be relaxed.
  This declaration is made putting in the predicate
  `application_relaxable_parameter/2` the list of the parameters with the following format :
  list of $sup - elem_i$

  where $sup - elem_i$ has one of the two following formats :

  - $[elem_i]$
  - $\{elem_i\}$

  and $elem_i$ is :

  - a parameter name,
  - or a sequence of parameter names or $sup-elem_i$ separated by commas.

  Semantics : using {} means a specific algorithm has to be provided by the designer for the parameter or for the group of parameters.
  Example:

```
application_relaxable_parameter
        ( [ [flightid],
          {class},
          {[date,sourcetime]},
          [flightid,{[date,sourcetime]}] ]).
```

- associating a data type to each parameter using the predicate `tm_def_type_parameter`
  Example:

  ```
  tm_def_type_parameter(flightid,flightid).
  tm_def_type_parameter(sourcetime,time).
  tm_def_type_parameter([date,time],date_time).
  ```

  Remark: These declarations have been included in module `application_knowledge.pl` now, see comments in implementation files.

  Note: date and time types have specific relaxation algorithms within the CRS.

- declarating the specific algorithms using the predicate :
  `application_relax_algorithm(type name, algorithm name)`
  example : application_relax_algorithm(date_time, relaxe_date_time)

General method When a new value for a parameter is found, a research in the database is carried on; if the response from the database is empty, a new value will be computed. If a new value can't be found, the program will deal with an other parameter.

Date relaxation The date (encoded by an integer) is relaxed depending a relaxation order provided by the designer. The CRS program contains two examples of use of the predicate (namely tm_relaxe_date_in_order).

Time relaxation — the New Algorithm[2] Given an input time interval $[T_1, T_2]$ the algorithm computes the output interval $O^n = [T_1 - \Delta t_n, T_2 + \Delta t_n]$ with:

- $0.00 \leq T_1 - \Delta t_n$,
- $23.59 \geq T_2 + \Delta t_n$,
- $\Delta t_n = 2 \cdot \Delta t_{n-1}$,
- $\Delta t_0 = 5min$.

[2]This algorithm has been implemented by the Erlangen team.

Time relaxation — the Old Algorithm Given an input temporal interval [(hour1, minute1),(hour2,minute2)] the algorithm computes the output interval [TR1,TR2] as :
$0 =< TR1 \, with \, TR1 = hour1 - I$
$23 >= TR2 \, with \, TR2 = TR2 + I$
where I takes successively its value in [1..23].

Examples The CRS program provides two examples of specific constraints relaxation algorithms. The first one deals with the relaxation of the class parameter. The second one deals with the relaxation of a couple of parameters (date and sourcetime. Depending on the hour (morning or evening), it relaxes differently the date and the time.

Remark: These specific algorithms are application dependent and should be transferred to module `application_knowledge.pl`.

4.4.4.5 Presenting results

At this stage the TM is ready to present the result of the query. This is done by a summary if the number of entries found in the previous stage is above `MaxGoal`, else all database entries (solutions) found are presented.

Remark: The summarization of results mechanism will eventually be implemented by IRISA.

After the first presentation the user can add asked parameters or change parameter values in order to retrieve more information and thus repeat the presentation stage. If only asked parameters are added now, the further information can be presented without making a new database query, as the TM will store the entries which have been found in the memory of the task instance. The presentation uses the following algorithm:

1. Retrieve the database entries for the query built in the previous stage, and store them into the task instance memory.
2. If their number exceeds `MaxGoal`, generate a summary; else use all of them.
3. Sort the result of the previous step in an application-dependent order.
4. Send the solutions to the BM, receive a `solution_view_id` for every one of them.

5. Send out a message containing the `view_ids`:
   `reply_solutions/2`

6. If the active task includes a database update, send out a message asking for authorization to perform it:
   `request_confirm_database_update/2`;
   If more than one solution had been presented, ask for selection of one of them instead:
   `request_select_solution/2`

7. Else get the list of possible continuation tasks and send out a message suggesting the selection of one of them:
   `request_confirm_continue_task/2`
   If none exist, send out a message suggesting to finish the active task:
   `request_confirm_end_task/1`

4.4.4.6 Making database updates

Once the dialogue has lead to the selection of an unique solution, and the authorization for a database update has arrived, additionally needed parameters such as e.g. the name or address of the user will be collected, using the same algorithm as in the parameter collecting stage. Then a database update transaction will be performed, trying to insert the desired reservation entry into the corresponding database.

After this transaction has taken place, an information message about its success or failure is being sent out, and additionally a message offering the possible continuation tasks is issued if there are any. This leads to the following algorithm:

1. Collect the update parameters (the parameter collecting algorithm described before is used here too).
2. Perform the database update transaction.
3. If it was successful, send out a success message:
   `reply_database_update/2`.
   Else send out a failure message:
   `reply_no_database_update/2`.

4. If continuation task classes exist, send out a message inviting to choose one of them:

`request_confirm_continue_task/2`.
Else send out a message suggesting to end the current task:
`request_confirm_end_task/2`

Remark: In our current example database, the database update transaction has not been fully implemented, there is only a dummy procedure. The complete update transaction is supposed to be performed within the application database and as such is not part of the generic Task Module's mechanisms, but rather to be implemented by application designers.

4.5 Customization

In this section the representation of application knowledge and the application database interface is described in detail, giving also some examples for the implementation of task classes and their components.

The module files that have to be customized in order to add task classes or to change the application domain are the following:

- `application_knowledge.pl`: Contains all knowledge about the application that is needed by the generic TM modules. It is only accessed through the module `tm_knowledge.pl` in order to keep its structure transparent to the generic part.
- `application_database.pl`: Contains the database interface procedures to the application database. It is only accessed through the module `tm_database.pl`.
- `dbflight_db.pl dbtrain_db.pl`: Contain the respective databases for filghts and trains.
- `tm_const_relax.pl`: Contains some application dependent constraint relaxation algorithms provided by IRISA. However, they should be transferred to module
`application_knowledge.pl` for the next version in order to keep only generic procedures in this module.
- `tm_integrate.pl`: Contains some utility procedures for translating parameters between the DM and TM. These procedures are necessary to do some conversion (e.g. date and time formats) and to strip off parts in SIL-descriptions not needed by the TM.

As the application-specific knowledge is not too voluminous, it can easily be kept in one single file `application_knowledge.pl`, with just the task classes as a dynamic predicate. Upon loading the system the choice is offered between the flights task and the trains task, the appliation knowledge is switched accordingly and the respective database is loaded.

4.5.1 Application knowledge

To start, the names of all task classes within an application domain have to be defined, together with the fact whether each task class contains additionally a database update operation apart from the query operation. In our example flight application, there are:

```
application_task_class( dbflight, query ).
application_task_class( dbreservation, update ).
```

Next, all parameters of each task class have to be described. Each parameter belongs to one of the categories *obligatory*, *optional* or *update*. The TM will always actively ask for obligatory parameters before requesting a database query, and for update parameters before requesting a database update. Optional parameters can have default values, be supplied voluntarily by the user, or be inferred, but the TM will not ask for them. Furthermore, each parameter has a type, e.g. *time*, *place*, etc., and a database flag. If the application database server is only able to check values of a certain parameter for equality, it has to receive the flag *discrete*. If values can be tested for being within an interval, it gets the flag *interval*. The ordering of obligatory and update parameters in the following facts specifies in which order they will be asked for. For example, take the parameters of the reservation task:

```
/* Parameters coming from previous dbflight-task */
application_param(dbreservation, date, obligatory, date,
                                 interval ).
application_param(dbreservation, flightid,obligatory,flightid,
                                discrete).
application_param(dbreservation, sourcecity, obligatory, place,
                                discrete ).
application_param(dbreservation, goalcity, obligatory, place,
                                discrete ).
application_param(dbreservation, sourcetime, obligatory, time,
                                interval ).
application_param(dbreservation, goaltime, optional, time,
                            interval ).
/* genuine dbreservation-parameters for which the user will give
   values */
application_param(dbreservation, passengername,obligatory, name,
                            discrete).
application_param(dbreservation, passengernumber,obligatory, number,
                            discrete).
application_param(dbreservation, class, obligatory, number,
                            discrete).
application_param(dbreservation, meal, optionnal, name,
                            discrete).
```

In addition to the parameter types, types must also be associated with groups of parameter types if a specific constraint relaxation algorithm for such a group will be needed, e.g. for the `[date,time]` pair:

```
application_type( _, [date,time], date_time ).
```

With most task classes it may be sufficient to know only a subset of all parameters specified to be obligatory. Thus, several different subsets may be specified. The aim then is to "instantiate" at least one of these subsets. Inferences are included implicitely within these subsets.

In the example given below, knowing the *departure place* (i.e. the database column `sourcecity`), the *destination place* (i.e. the database column `goalcity`), and the *date* would be enough to start a senseful database enquiry, as it is possible that on a given day the number of flights between two places is less or equal the maximum number of results provided in th application knowledge. So, the TM does not request a departure of arrival time to start a database query, though this would be done if the number of entries exceeds the limit and no summarization is possible.

```
application_minimal_queries(  dbflight,
[ [ sourcecity, goalcity, date],
  [ sourcecity, goalcity, date, sourcetime ],
  [ sourcecity, goalcity, date, goaltime ],
  [ date, flightid ]] ).
```

After a database query has been carried out, the resulting entries are filtered so that the final solution contains only those parameter slots which the user has asked for. Some parameters, however, can be assumed to appear in any senseful answer to a question of the application domain, and are therefore considered to be *asked parameters* by default:

```
application_default_asked_parameters( dbflight,
        [ flightid, date, sourcecity, sourceairport, sourcetime,
                         goalcity, goalairport, goaltime ]).
application_default_asked_parameters( dbreservation,
        [ date, pasengernumber, passengername, class, flightid]).
```

For each parameter of each task class, a default value can be supplied:

```
application_default( dbflight, sourcecity, london).
application_default( dbflight, date, Today ).
application_default( dbreservation, pasengernumber, 1 ).
application_default( dbreservation, class, business ).
```

Inference rules can also be provided. They receive a list of `[Param,Value]` pairs in the `Known` variable, containing all parameters that have already been provided or confirmed by the user. Each inference rule can specify under which conditions a parameter value must not be asked for anymore, and it can also provide an inferred value. For example, knowing the `flightid` obviates the need to ask for the departure and destination place although they are still unknown. That is, if a *flightid* is contained in the list `Known` of the known parameters, either the `sourcecity` or the `goalcity` may be returned as "known" or "given", with the value `unknown`, which means the real value is not important at this stage and will be known by the database then.

```
application_inference( dbflight, sourcecity, unknown, Known ) :-
        member( [ flightid, _ ], Known ).
application_inference( dbflight, goalcity, unknown, Known ) :-
        member( [ flightid, _ ], Known ).
```

A second kind of inference rules specifies the parameter values that will be copied as defaults from one task instance to another if the `continue_task/2` message is sent to the task module. For example, if a reservation for a return ticked follows the current reservation, the destination and departure places are exchanged. Some of its parameters could also be used for a fare enquiry:

```
application_continuation( dbreservation, dbreservation,
        [[ sourcecity, goalcity ],
         [ goalcity, sourcecity ],
         [ date, date ],
         [ passengername, passengername ],
         [ class, class ],
         [ passengernumber, passengernumber]] ).
```

The maximum and minimum number of solutions that will be presented explicitely to the user must be specified, as well as a threshold number of database entries beyond which a summarization attempt would be too expensive:

```
application_query_min_goal( dbflight, 1 ).
application_query_max_goal( dbflight, 3 ).
application_query_threshold( dbflight, 20 ).
```

dbflight For the database query and update operations, templates for the parameter lists that will be passed to the application database interface have to be given. The name of the database (e.g. *dbflight*) has to be specified too. The number, names and order of parameters must correspond exactly to those specified in the application database interface:

```
application_query_form( dbflight, flight_dbflight,
   [ flightid, sourcecity, goalcity, date, sourcetime, goaltime,
          sourceairport, sourceterminal, goalairport, goalterminal,
          carrierid, vehicle, class] ).
application_update_form( dbreservation, reservation,
        [ flightid, pasengername, passengernumber, class, meal, date ] ).
```

A list of parameters by which the solutions presented to the user will be sorted is also required. Their importance decreases from left to right. For each of them, the `application_less/2` relation must be defined:

```
application_sort_by( dbflight, [ goaltimesource, sourcetime ] ).

application_less( time( H1, _ ), time( H2, _ )) :- H1 < H2, !.
application_less( time( H, M1 ), time( H, M2 )) :- M1 < M2, !.
application_less( time( _, _ ), time( _, _ )) :- !, fail.
application_less( Num1, Num2 ) :-
        number( Num1 ),
        number( Num2 ), !,
        Num1 < Num2.
```

A number of consistency rules can be defined for parameters. If any of these rules succeeds, a new parameter value will be rejected by the TM, sending a message asking to correct it. The rules must also instantiate symbols indicating the reason for the inconsistency (e.g. `before_sourcetime`. Consistency rules receive a list of *known* parameters in the same way as inference rules do. For example, an arrival time interval is inconsistent if it ends before the beginning of the departure time interval:

```
application_inconsistent(dbflight,goaltime,[_,A2],Known,
    before_sourcetime) :-
        member( [ sourcetime, [D1,_]], Known ),
        application_less( A2, D1 ).
```

For each task class, a list of relaxable parameters must be specified. They will be relaxed, beginning with the leftmost group, if no solutions to a query are found in the first step. Groups marked by additional brackets (here: *class* and *[date,sourcetime]*) have a special relaxation algorithm, whose name must be specified too:

```
application_relaxable_parameters( dbflight,
        [ [flightid],
  [carrierid],
          [sourcetime],
          [goaltime],
          [date],
         {[date,sourcetime]},
          [carrierid,date],
          [carrierid,class,date],
          [flightid,sourcetime],
          [flightid,{[date,sourcetime]}] ]).

application_relax_algorithm( dbflight, class, tm_relaxe_class ).
application_relax_algorithm( dbflight, date_time, tm_relaxe_date_time ).
```

Remark: These special relaxation algorithms should follow here as they are obviously application-dependent. They have been left in module `tm_const_relax.pl` for now (as provided by IRISA), but they should be separated from the generic module in the next version.

4.5.2 Application database interface

The application database interface has to implement the following three procedures:

- `application_query_entries(+Database,+PVQuery,-Entries)`
  Returns a list of all database entries that have been found for the specified query. `Database` is the name of the database as defined in `application_query_from/3`. `PVQuery` is a list of `[Param,Value]` pairs corresponding to the query parameters defined in the query template in `application_query_form/3`. `Entries` is the returned list of database entries, each of them being a list of `[Param,Value]` pairs of the same form as the query.

- `application_query_number_of_entries(+Database,`
  `+PVQuery,+Threshold,-Count)`
  Counts the number of entries that a query would find. `Database` and `PVQuery` have the same meaning as above. `Threshold` is the threshold value defined in `application_query_threshold/2`, `Count` is the number of entries found. This procedure should improve the behaviour of constraint relaxation if it has to be iterated many times and calls to a remote database have to be made on a slow transmission line, as only numbers have to be transmitted instead of complete entries.

- `application_update_transaction(+Database,+Update,-ReturnValue)`
  Performs the update operation on the database. Update is a list of `[Param,Value]` pairs for update parameters ordered as in `application_update_form/3`. The return value is `success` if the update operation has been carried out successfully, otherwise `failure`.

4.5.3 Train application database

The German application database contains the schedule for intercity trains. Three methods of accessing the timetable entries are possible:

- storing all city-to-city-connections explicitly with the departure and arrival times and the citynames,
- storing the timetables of all trains and generating the connections by request on runtime, searching all timetables for the given citynames—and
- preparing an request to an connected external database system which is specialised and trimmed in answering this requests.

The benefits of the first method is obviously the very fast access to the required information but the memory space used for string all possible train connections grows by $O(n^2)$ in the number of trains.

Using a directed graph an a search mechanism the memory increases linear by the number of trains but the searching for the respective train connection could be very time consumpting. The computindg time will increase dramatically if stopover points are allowed. The search time would grow in the order of $O(n^{s+1})$ where s is the number of stopover points.

Using commercial available software for enquiries on the timetable has the best performance but leads to the problem of connecting this system to the PROLOG-based TM. Licensing conditions have to be taken into consideration.

For the demonstrating system the first method of using explicit connections was implemented. Considering the constraints of finite[1] memory capacity the used timetable is a small subset of the possible train connections in Germany.

4.5.4 The database entries

The database contains facts which are of the following form:

```
data(dbtrainconnection,
        [ trainid, sourcecity, goalcity, date,
          sourcetime, goaltime,
          sourcestation, sourceplatform,
          goalstation, goalplatform,
          viacity, changecity, carrierid]).
```

Types of the parameters:

- `trainid`: string (e.g. *IC_683 Herrenhausen*)
- `sourcecity, goalcity`: string
- `date`: sun, mon, . . . sat
- `sourcetime, goaltime`: `time(Hour,Min)`
- `sourcestation, goalstation`: string (e.g. *Hamburg-Altona*)
- `sourceplatform, goalplatform`: integer

[1]not only finite — even very small

- `viacity, changecity`: string
- `carrierid`: bundesbahn, reichsbahn, sncf, öbb ...

4.5.5 Extension of the database

If an new train has to be added to the database, then the connection from each `Sourcecity` to each `Goalcity` has to be included as a new Prolog fact. The `Sourcecity` and `Goalcity` are determined by the implicit order given by the train direction.

4.6 Implementation

4.6.1 Module files

The implementation of the task module has been divided into a number of Quintus Prolog module files. The following summary describes briefly the contents of each of those files:

- `tm.pl`
  This file is intended to be imported by the "postie" module, as it contains the top-level procedures `process_message/1` and `receive_from_pm_tm/2`, and the specifications of all messages that can be received by the task module.

- `tm_integrate.pl`
  Contains some procedures for translation of message formats between DM and TM.

- `tm_control.pl`
  This file contains the task scheduler procedure `task_control/0`.

- `tm_initialize_task.pl`
  `tm_collect_params.pl`
  `tm_database_query.pl`
  `tm_present_result.pl`
  `tm_database_update.pl`
  `tm_const_relax.pl`
  These files contain the implementations of the algorithms for the five task stages, and the constraint relaxation mechanism.

- `tm_send_message.pl`
  This file contains the specifications of all messages that are sent out by the task module.

- `tm_status.pl`
  Contains the dynamic predicates in which the task module state is stored, and a set of access and modification procedures for them.

- `tm_knowledge.pl`
  `application_knowledge.pl`
  The first file contains a set of interface procedures to the application-dependent facts and rules that are stored in `application_knowledge.pl`. The latter is an example file which shows how application task classes can be described for the task module.

- `tm_database.pl`
  `application_database.pl`
  The first file contains the specification of the database access procedures that have to be provided by the application database. The second file is intended to be the application-specific database interface module which will build appropriate queries (e.g. SQL or C procedure calls) and access the application database. For now, it contains an the interface to the `dbtrain_db.pl` and `dbflight_db.pl` example application databases for the train and flights domain respectively, in which the database entries are represented as prolog facts.

  The following figure shows the main architecture of the TM implementation as explained above.

4.6.2 Messages with *Views*

The TM is able to communicate with other modules using *views* (see 3.3). These views implement a communication strategy which uses descriptors (viewids) rather than sets of raw data. For each set of data the TM is talking about, a viewid is requested and this viewid refers to the given instances of variables in the mentioned task.

Some basic notions used in this document

ViewId is an index for view assigned by the BM. SetOfViewIds is a list of ViewId; e.g. [view1,view2,view3].

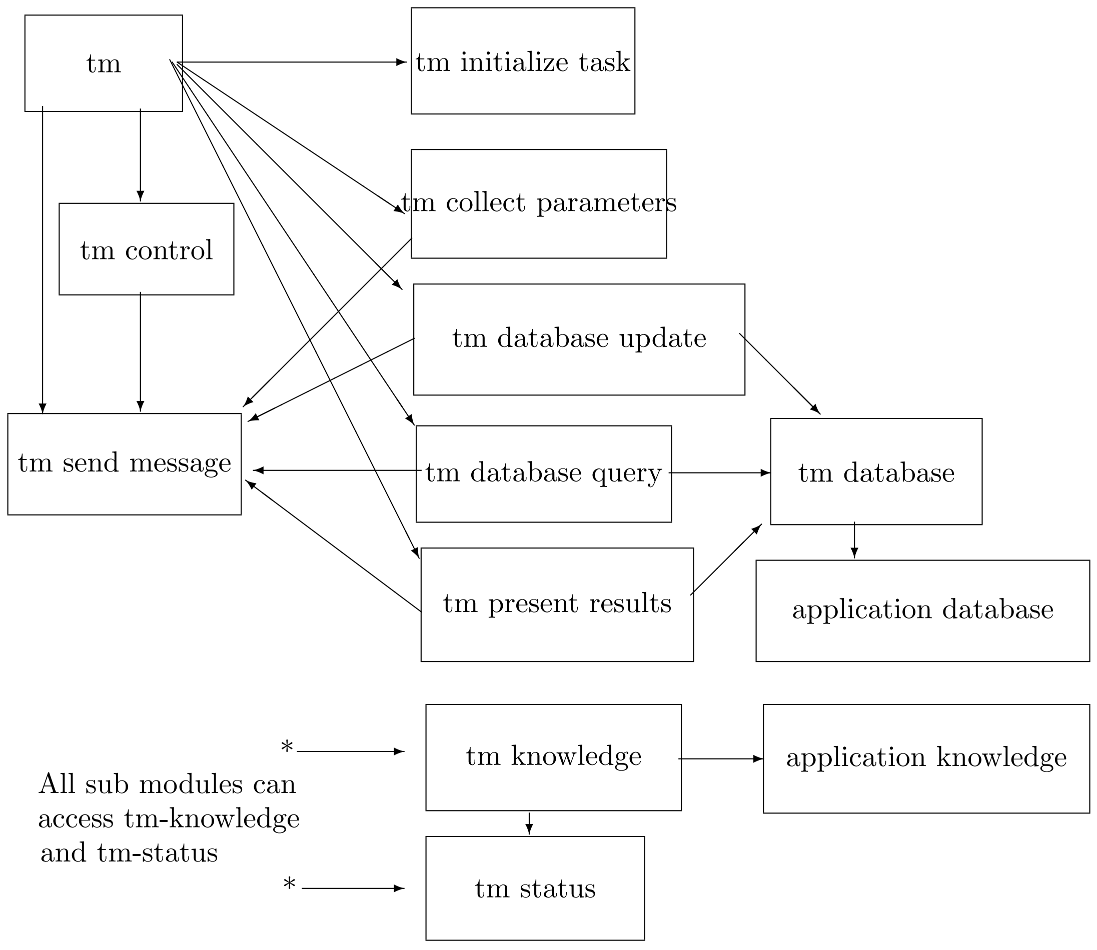

Figure 4.1: Task Module Architecture

TaskId is an index for the task. The index is assigned by the BM and the task is determined by the TM.

TaskType is the type of the task. Possible types are: *dbreservation*, *dbflight* and *dbtrain*. Each of these types is associated with a number of parameters described below.

TaskParameters are parameters associated with a given task type. Each parameter is given as a pair: *ParameterName:ParameterValue.*

dbreservation has the following parameters

Parameter Name	**Value Type**
flightid	string
date	integer
sourcecity	string
goalcity	string
sourcetime	integer
goaltime	integer
passengername	string
passengernumber	integer
class	string
meal	string

dbflight has the following parameters

Parameter Name	**Value Type**
flightid	string
sourcecity	string
sourceairport	string
sourceterminal	integer
goalcity	string
goalairport	string
goalterminal	integer
sourcetime	integer
goaltime	integer
date	integer
carrierid	string
vehicle	string
class	string

dbtrain has the following parameters

Parameter Name	Value Type
sourcecity	string
sourcestation	string
sourceplatform	integer
goalcity	string
goalstation	string
goalplatform	integer
sourcetime	integer
goaltime	integer
date	integer
viacity	string
changecity	string
carrierid	string
trainid	string

4.6.3 TM Interfaces to other modules

4.6.3.1 DM → TM

The TM is capable of handling two different types of messages. Simple messages perform parameters, opening and closing of tasks, whereas compound messages could be used to send several modifications at the same time. This should allow the DM to incorporate all new information gathered from a user utterance at once, avoiding synchronizing problems with the TM. In gerneral it should be possible, to break a compound message into several simple messages, which could on the other side lead to perform queries on partial information.

The compound message `reply([Msg1, Msg2, ...])` is recognized as an ordered sequence `Msg1`, `Msg2`, ... of simple messages. Each simple message must conform to the below given types of simple messages.

The recognized simple messages from the DM are of the following forms:

1. `init`
   This message is sent only once in order to initialize the internal memory space.

2. `start_task([id:TaskId,type:TaskType,TaskParameters])`
   A new task has been identified. `TaskType` is the task class identifier, `TaskParameters` is a list of settings for obligatory or optional parameters representing values that have been provided by the user together with the requirement for the new task. The TM will now create an instance

of the specified task type, label it with the provided identifier and notify the DM with a `update_start_task/2` message. Then it will store the parameters that have been provided initially and start sending out `request_parameter/2` messages for the missing obligatory parameters. A possible example is:
`start_task([id:task1, type:dbflight, sourcecity:paris, goalcity:london]).`

3. `reply_parameter(TaskId,TaskParameters)`
   The DM has identified some new parameters for an already existing task instance. `TaskId` is the task instance identifier, `TaskParameters` are formatted entries of provided parameters as described above. A possible example is:
   `reply_parameter(id:taskid1, [date:2006]).`

4. `select_solution(id:TaskId,ViewId)`
   Notifies the TM that the user has selected *the* solution. ViewId has been obtained by a call to BM and was given to DM by a message of type request_select_solution.

5. `database_update(TaskId)`
   Notifies the TM that the user has authorized a database update operation (a reservation). This message will be ignored if the set of solutions has not yet been restricted to a single solution.

6. `continue_task(TaskId,TaskType)`
   Notifies the TM that the user wants to start a task of the class `TaskType` which will replace the current task instance, but preserving the known parameters (e.g. a price enquiry referring to the same flight as the current timetable enquiry, or a return ticket for the same route).

7. `end_task(TaskId)`
   Notifies the TM that this task instance can be destroyed.

4.6.3.2 TM → DM

The following messages are sent ot the DM:

1. `request_start_task(TaskTypes)`
   where TaskTypes is a list of permissible task types together with any dependencies they may have. For example, `[dbreservation/[dbflight], dbflight, dbtrain]` says that three task types are permissible where the first, *dbreservation*, is dependent upon the task *dbflight* for its resolution.

2. `request_confirm_continue_task(TaskId,TaskTypes)`
   Suggest to continue the given task with an instance of one of the given continuation task types. For example, an enquiry task could be followed by another enquiry task or a reservation task:
   `request_confirm_continue_task(task42, [dbflight, dbreservation/[dbflight]])`.

3. `request_confirm_end_task(TaskId)`
   Ask for authorization to finish this task instance.

4. `request_confirm_database_update(TaskId)`
   Ask for authorization of a database update.

5. `request_confirm_parameter(TaskId,ViewId)`
   Ask for confirmation of a default value for a specific parameter of the given task instance. The ViewId has been obtained by a call to the BM for a *default view*.

6. `request_parameter(Id,ViewId)`
   Ask for the value of this parameter of the given task instance. The ViewId has been obtained by a call to the BM for a *query view*.

7. `request_confirm_correct_parameter(Id,OldViewId,Reason,NewViewId)`
   Ask for a new value for a specific parameter instead of the given erroneous value. This can be done by confirmation of the default value, or by providing a new value. The given reason is defined by the application-dependent consistency-check rules. The NewViewId has been obtained by a call to the BM for a *suggested view*. OldViewId is th ViewId of the problematic parameter value(s).

8. `request_correct_parameter(Id,ViewId,Reason)`
   Ask for a new value for a specific parameter instead of the given erroneous value. The given reason is defined by the application-dependent consistency-check rules.

9. `request_wait`
   Ask to wait for a moment.

10. `reply_solutions(Id,SetOfViewIds)`

    Show the result of a database query. `SetOfViewIds` is a list of ViewIds obtained by a call to the BM for *alternative views*. For example, a single solution would be `reply_solutions(taskid2, [view1])` and multiple solutions `reply_solutions(taskid3, [view2, view3, view4])`.

11. `request_select_solution(TaskId,SetOfViewIds)`
    Acknowledge the outside that a task of type *update* needs an unique solution. The reservation is only possible if *one* of the solutions is selected.

12. `reply_database_update(TaskId)`
    Acknowledgement that a database update operation has been performed successfully.

13. `reply_no_database_update(TaskId,Reason)`
    Acknowledgement that a database update operation has failed and why.

14. `update_start_task(TaskType,TaskId)`
    A task instance of this class has been created with the given identifier.

15. `update_inferred(Id,ViewId)`
    The given parameter has been inferred. The ViewId has been obtained by a call to the BM for a *inferred view.*

16. `update_default(Id,ViewId)`
    The given parameter has received a default value. The ViewId has been obtained by a call to th BM for a *default view.*

17. `update_relaxed(Id,ViewId)`
    The given parameter has been relaxed. The ViewId has been obtained by a call to th BM for a *relaxed view.*

18. `request_alternative_values(Id,ViewId)`
    Ask for different values for some parameters because no datasets were found. The ViewId has been obtained by a call to th BM for a *alternative view* on a list of parameter names.

19. `request_specific_values( Id, Params)`
    Ask for more specific values for some parameters because too many datasets were found. `Params` is a list of parameter names.

4.6.3.3 TM → BM

In each of the following, the bm requests a ViewIndex for parameters, parameter values, solutions and tasks.

1. `request_query_view(MsgId,TaskId,ParameterList)`
   where MsgId is the index for the message (also returned with the reply from the bm), TaskId is the index for the task and ParameterList is a list of one or parameter (appropriate to the task type) which are to be queried.

2. `request_default_view(MsgId,TaskId,Parameter:Value)`
   where Parameter:Value specifies a default Value for Parameter.
3. `request_inferred_view(MsgId,TaskId,Parameter:Value)`
   requests a ViewId for an inferred Value of a Parameter.
4. `request_suggested_view(MsgId,TaskId,Parameter:Value)`
   requests a ViewId for a suggested Value of a Parameter.
5. `request_alternative_value_view(MsgId,TaskId,Parameter,ValueList)`
   requests a ViewId for a list of alternative Values of a Parameter.
6. `request_alternative_solution_view(MsgId,TaskId,SolutionsList)`
   requests ViewId for one or more solutions: i.e. solutions is one or more sublists each giving a solution.
7. `request_relaxed_view(MsgId,TaskId,Parameter:Value)`
   requests a ViewId for a relaxed Value of a Parameter.
8. `request_continue_task(MsgId,TaskId,TaskType)`
   where TaskId is the task id of the current task and TaskType is the type of the next task. For example, in flight reservations, when the enquiry phase was complete and the system wished to move onto the reservation phase the message would be `request_continue_task(MsgId, flight1, dbreservation)`

4.6.3.4 BM → TM

In each case, the bm returns a MsgId corresponding to the 'request' MsgId, the TaskId and the either a single ViewId or a list of ViewIds.

1. `return_query_view(MsgId,TaskId,ViewId)`
2. `return_default_view(MsgId,TaskId,ViewId)`
3. `return_inferred_view(Msgid,TaskId,ViewId)`
4. `return_suggested_view(MsgId,TaskId,ViewId)`
5. `return_alternative_value_view(MsgId,TaskId,SetofViewIds)`
6. `return_alternative_solution_view(MsgId,TaskId,SetofViewIds)`
7. `return_relaxed_view(MsgId,TaskId,ViewId)`
8. `return_continue_task(MsgId,TaskId,ViewId)`
   where TaskId and ViewId are those of the new task.

4.6.4 An example message sequence

The following sequence of messages should show the strategy of establishing tasks, requesting parameters using views, finding a solution and closing down.

The notion is: `msg(receiving_module, sending_module, message)`

the whole process starts:
```
msg(tm,dm,init)
```

tm offers a set of start tasks:
```
msg(dm,tm,request_start_task([dbreservation/[dbflight],
    dbflight]))
```

dm starts the enquiry task and provides some variables:
```
msg(tm,dm,start_task([id:dbflight1, type:dbflight,
    sourcecity:london, goalcity:paris]))
```

tm notifies dm about the new task:
```
msg(dm,tm,update_start_task(dbflight, dbflight1))
```

tm needs another parameter — the bm is able to generate a view:
```
msg(bm,tm,request_query_view(tmmsg1, dbflight1, [date]))
```

the requested view arrives:
```
msg(tm,bm,return_query_view(tmmsg1, dbflight1, qview1))
```

dm should provide a value for this parameter / view:
```
msg(dm,tm,request_parameter(dbflight1, qview1))
```

the parameter is returned:
```
msg(tm,dm,reply_parameter(dbflight1, [date:2006]))
```

the same procedure for another parameter:
```
msg(bm,tm,request_query_view(tmmsg2, dbflight1, [sourcetime]))
msg(tm,bm,return_query_view(tmmsg2, dbflight1, qview2))
msg(dm,tm,request_parameter(dbflight1, qview2))
```

the requested parameter is given together with a value for an additional parameter:
```
msg(tm,dm,reply_parameter(dbflight1,[sourcetime:1715,
    sourceairport:heathrow]))
```

tm found a solution and needs a view to refer to it:
```
msg(bm,tm,request_alternative_solution_view(tmmsg3, dbflight1,
    [[flightid:ba123, goaltime:1839,
    goalairport:charles_de_gaulle]]))
```

the view is given:

```
msg(tm,bm,return_alternative_solution_view(tmmsg3, dbflight1,
    [aview1]))
```

dm will gets the solution:

```
msg(dm,tm,reply_solutions(dbflight1, [aview1]))
```

the needs of this task type are satisfied:

```
msg(bm,tm,request_continue_taskid(tmmsg4, dbflight1,
    [dbflight,dbreservation]))
```

continuing with a task of type update*:*

```
msg(tm,bm,return_continue_taskid(tmmsg4, dbreservation1,
    dbreservation))
```

inform the DM about the new task:

```
msg(dm,tm,update_start_task(dbreservation1,dbreservation))
```

TM can offer a solution in the new task domain:

```
msg(bm,tm,request_alternative_solution_view(tmmsg5,
    dbreservation1,[[flightid:ba123,date:thu,class:business]]))
msg(tm,bm,return_alternative_solution_view(tmmsg5,
    dbreservation1,[aview2]))
msg(dm,tm,reply_solutions(dbreservation1,[aview2]))
```

the solution found in the first step is preserved, but an update parameter is needed:

```
msg(bm,tm,request_query_view(tmmsg6, dbreservation1,
    [passengername]))
msg(tm,bm,return_query_view(tmmsg6, dbreservation1, qview3))
msg(dm,tm,request_parameter(dbreservation1, qview3))
msg(tm,dm,reply_parameter(dbreservation1,
    [passengername:'quintus unix']))
```

all required parameters are ok, TM could do the reservation:

```
msg(dm,tm,request_confirm_database_update(dbreservation1))
```

user authorized the system to do the reservation:

```
msg(tm,dm,database_update(dbreservation1))
```

the updating is successfully finished:

```
msg(dm,tm,reply_database_update(dbreservation1,
    [flightid:ba123,passengername:'quintus unix',
    class:business,date:thu]))
```

closing down:

```
msg(bm,tm,request_confirm_end_task(dbreservation1))
msg(tm,dm,end_task(dbreservation1))
```

Chapter 5

The Message Planner

5.1 Overview

The message planning module interfaces the Dialogue manager with the message generator in the Sundial system. As such, it is responsible for providing descriptions of system messages within the interface language. These descriptions are then passed to the message generation module for detailed linguistic realization and synthesis. When generation is complete, the message planning module updates the rest of the dialogue manager.

This chapter begins by clarifying the functionality of the message planning module since it does not support all of the functionality described in WP6000 D1 (section **??**). Section **??** elaborates on the principles underlying the message planning module. The main algorithms are described in section **??** and the knowledge base in section **??**. Section **??** describes how the message planning module can be customized for different languages and different types of message generators. The current implementation is outlined in section **??** and section **??** describes some possible extensions of the message planning module.

5.2 Scope

The message planning module described in WP6000 D1 carried out three main functions

1. global dialogic planning
2. global semantic planning

3. local semantic planning

In the current version, none of these functions are carried out by the message planning module.

Global dialogue planning is carried out by the dialogue module (see chapter on dialogue module). The rationale for this concerns predictions. Predictions about user's utterances are constructed by the dialogue module on the basis of system move information sent to the message planning module. If the message planning module were to perform dialogue planning, i.e. expand, contract or add system moves, then the predictions exported to the linguistic processing module (via the linguistic interface module) may be inappropriate for the system move information generated by the message planning and message generation modules. Given the design of the dialogue module, our strategy has been to remove global dialogic planning from the message planning module and thereby ensure that the predictions are appropriate to what the dialogue module expects.

With global semantic planning, i.e. building a semantic representation for each system move, while the message planning module decides, on the basis of its static knowledge definitions, how objects are to be described, the construction of semantic representations is carried out by the belief module. The reason for this is that all the relevant semantic information is contained in the belief model. If the message planning module were to build semantic descriptions then, as we discovered with early versions, then there would be duplication of knowledge definitions and algorithms for accessing objects in the belief model. A cleaner design and implementation emerged with this redistribution of functionality.

Finally, local planning—deciding whether objects are to be given a linguistic realization and, if so, whether they are to be anaphoric—is carried out by the message generation module itself. The principal reason for this is that the original design envisaged planning and generation to be interactive: i.e. give a schematic plan of the system utterance, the message generation module linguistically realized some of it and where it required further information, such as information about the linguistic or belief context, this information would be request from the dialogue manager via the message planning module. Current implementations of the message generation module, however, are not interactive: they are not suspendible processes in prolog. As a result, the message planning module passes the message generation module all (dialogue manager) information relevant to this aspect of generation: in particular, prosodic, ellipsis and accessiblity information about the system utterance as well as the current linguistic history are passed to the message generation module. Future implementations of the message generation module may be interactive and so avoid the need to pass this information—it would be avaliable to the message generation module if requested.

5.3 Principles

The message planning module performs two simple functions: it constructs a plan of the system utterance which the message generation module realizes; and it informs the relevant dialogue manager modules of the realized system utterance. Two principles by which these functions are achieved have already been mentioned: the dialogue module performs dialogic planning; and the belief module constructs the semantic descriptions. The remaining principles are described below.

5.3.1 System Utterances As Utterance Field Objects

In the interface between the dialogue manager module and both the linguistic processing module and the message generation module, linguistic information is described in terms of Utterance Field Objects (UFOs). Rather than the message planning module construct intermediate structures on the basis of the input from the dialogue module, the input, in form of a list of moves to be generated, is initially transformed into UFOs. The resulting UFOs are then matched against a set of planning rules which determine the semantic and syntactic information added to the UFOs. Depending upon which rules are activated, this may result in a request to the belief module to build a semantic description for the UFO which can then be realized by the message generation module. Here we describe the structure of UFOs and how their content is determined.

5.3.1.1 The structure Of UFOs

UFOs represent one or more phrases in an utterance as (closed) dags. Each phrase in the utterance is represented as a multi-level structures with fields for dialogue, semantic and syntactic information with the potential for cross-level reentrancy.

UFOs can be complex or atomic. Complex UFOs are used to represent system or user utterances which consist of more than one phrase. Utterances which consist of a single utterances, and utterances within complex UFOs, are represented as atomic UFOs. For example, the utterance *flight ba234 arrives at 5pm. is that suitable?* is represented as complex UFO with each phase represented as an atomic UFO.

A complex UFO is defined in (11).

(11)

$$\begin{bmatrix} id : \_ \\ card : \_ \\ N : atomic\_UFO^* \end{bmatrix}$$

The value of **id** is a unique index for the whole utterance. The value of **card** is an integer specifying the number of the atomic UFOs which make up the complex UFO. Each **N**: atomic_UFO pair describes an atomic UFO, where the number of these pairs corresponds to the value of the **card** attribute; e.g. if the **card** has the value 2, then there will be two **N**: atomic_UFO pairs.

In an atomic UFO, utterance information is organized in three fields: dialogue, semantics and syntax. In addition to these fields, each UFO is has a unique index for the utterance. An atomic UFO is thus defined in (12).

(12)

$$\begin{bmatrix} id : \_ \\ dialogue : \_ \\ semantics : \_ \\ syntax : \_ \end{bmatrix}$$

The value of the **id** attribute is an index for the atomic UFO. The values of the remaining attributes are discussed below.

5.3.1.2 Transforming Moves to UFOs

Requests from the dialogue module to generate a system utterance are representated as a structured list of

```
move(MoveId, MoveType, Owner, DialogueFunction,
     DialogueArguments, SemanticArguments, CurrentExchangeId,
     NextExchangeId, PreviousExchangeId)
```

Accordingly, the message planning module has the task of transforming these lists into UFO representations. The conversion between these representations is given below

1. `[move/9]` is transformed into a single atomic UFO. For example,

```
generate([move(5, main, system, reaction, reintroduce,
                            [open\_request, view1], 8, [], 4)])
```

is transformed into (13)

(13)

$$\begin{bmatrix} id : ufo34 \\ dialogue : \begin{bmatrix} dact : open\_request \\ reintroduce : 1 \end{bmatrix} \\ semantics : \begin{bmatrix} view : view1 \end{bmatrix} \end{bmatrix}$$

2. `[[move/9]|_]` is transformed into a complex UFO with atomic UFOs for each
   `[move/9]`. For example,

```
generate([[move(1, main, system, initiative, greeting,
                   \_190211, 3, [], [])],
          [move(2, main, system, initiative, open\_task,
                   [dbreservation/[dbflight]], 5, [], [])]])
```

   is transformed into (14)

(14)

$$\begin{bmatrix} id : ufo34 \\ card : 2 \\ 1 : \begin{bmatrix} id : ufo35 \\ dialogue : \begin{bmatrix} dact : greeting \end{bmatrix} \\ semantics : \_ \end{bmatrix} \\ 2 : \begin{bmatrix} id : ufo36 \\ dialogue : \begin{bmatrix} dact : open\_task \end{bmatrix} \\ semantics : \begin{bmatrix} dbreservation : \_ \\ dbflight : \_ \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

3. `[[move/9,move/9|_]|_]` is a move structure where two or more moves are sublists within a list of moves. These structures are either merged into a single UFO if their dialogue information is identical or, if not, they are expanded into two separate UFOs. For example, the sublist in

```
generate([[move(5, main, system, evaluation, confirm,
                   sourcecity=paris, 5, [], 4),
           move(6, main, system, evaluation, confirm,
                   goalcity=london, 5, [], 4)],
          [move(8, main, system, initiative, open\_request,
                   view2, 9, [], [])]])
```

is merged into a single UFO since they are both confirmations

(15)

$$\begin{bmatrix} id: ufo67 \\ card: 2 \\ 1: \begin{bmatrix} id: ufo68 \\ dialogue: \begin{bmatrix} dact: confirm \end{bmatrix} \\ semantics: \begin{bmatrix} sourcecity: paris \\ goalcity: london \end{bmatrix} \end{bmatrix} \\ 2: \begin{bmatrix} id: ufo69 \\ dialogue: \begin{bmatrix} dact: open\_request \end{bmatrix} \\ semantics: \begin{bmatrix} view: view2 \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

It should be apparent from these examples that not all information in `move/9` is converted into UFO information. In particular, MoveType, Owner, Dialogue-Function, CurrentExchangeId, NextExchangeId, PreviousExchangeId are omitted from the UFOs since the information are not relevant to planning and generation. Only the MoveIdentifier, DialogueArguments and SemanticArguments are converted[1].

The DialogueArguments in moves are transformed into the value of the dialogue attribute in UFOs. DialogueArguments consists of a (system) dialogue act and, optionality, a dialogue act modifier. The dialogues acts, given in figure **??**, are represented in the UFO with attribute value pair dact:DialogueAct, where DialogueAct is the dialogue act label extracted from the move.

These dialogue acts can be modified by one of the three modifiers: rep, fail, reintroduce. Again, these modifier are represented as attribute value pairs where the attribute label is the modifier and the value is an integer which, unless otherwise specified in the move, defaults to 1.

rep:N	the N^{th} repetition of the act
fail:N	the system failed to understand the act for the N^{th} time
reintroduce:N	the act is being reintroduced into the dialogue for the N^{th} time

(13) illustrates how a move with the DialogueArgument `reintroduce: open_request` is converted to a UFO where the dialogue attribute has a complex value consisting of dact: open_request and reintroduce:1.

[1]The MoveIdentifier is not part of the UFO and is stored separately.

greeting	opening the dialogue
request_confirm	request for a confirmation of given value
request_confirm_def	request for a confirmation of default value
renup	information about the available solution(s)
posi_check	yes-no question
open_request	wh-question
hallo	hello
open_task	request for a task
req_spec	request for further information
check	verification of a parameter value
alts_question	question proposing a set of alternatives
open_request_chgt_topic	open question with a change of subject
reppar	parameter repetition
demreppar	request for parameter repetition
demrep	request for repetition
wh_answer	answers to an open request
confirm	confirmation of a given value
confirm_def	confirmation of a default value
confirm_weak	a merged confirmation
eval	request for evaluation of a solution
ph_closure	closing the dialogue
inform_not_exist	information that no solution is available
suggrenup	proposal of a solution after constraint relaxation
recap	recapitulation of a solution

Figure 5.1: System dialogue acts

The SemanticArgument in a move is likewise converted into an attribute value representation in the UFO as shown below.

SemanticArgument	semantics in UFO
_Variable	semantics:_Variable
Attribute=Value	semantics:[Attribute:Value]
ViewId	semantics:[view:ViewId]
Task/[SubTask]|_]	semantics:[Task:_,SubTask:_|_]

5.3.1.3 Building Semantics Descriptions

Once the message planning module has converted moves to UFOs, each atomic UFO is checked against its planning rules to determine how, if at all, these structures are to be extended for generation. These rules, described in section ??, specify a template used for building the semantics and/or syntactic descriptions which conform to sil (see WP5 deliverable for definitions and details), what sort of semantic description is required as well as default prosodic and ellipsis information. Here we focus on the construction of semantic descriptions.

Templates in these rules determine the semantic description of the system utterance. With some rules, such as the ones used for UFOs whose dialogue act is *greeting*, no semantic description is built. In such cases, what is passed to the message generation module is a UFO whose value for semantics is a variable. The majority of templates, however, specify that a semantic description is to be build. The specification is given as the type of description the belief module is required to build. The four types of description are: `object`, `object_in_ctx`, `object_with_ctx` and `object_and_ctx` (see section ??). For example, given a UFO whose dact value is *open_request* and whose view value is *view34* and matching rule specifying a semantic description type *object_in_ctx*, the message planning module requests the belief module to build an semantics *object_in_ctx* description of *view34*. The belief module then returns a description of this view as, for example, shown in (16).

(16)

$$\left[\begin{array}{ll} \mathit{id} : \mathit{time}4 & \\ \mathit{type} : \mathit{s\_time} & \\ \mathit{thedesc} : & \left[\begin{array}{ll} \mathit{id} : \mathit{arrive}1 & \\ \mathit{type} : \mathit{arrive} & \\ \mathit{thetheme} : & \left[\begin{array}{ll} \mathit{id} : o46 & \\ \mathit{type} : \mathit{individual} & \\ \mathit{thediscourserole} : & \left[\begin{array}{l} \mathit{id} : o47 \\ \mathit{type} : \mathit{discourserole} \\ \mathit{value} : \mathit{listener} \end{array}\right] \end{array}\right] \\ \mathit{thetime} : \left[\ \mathit{id} : \mathit{time}4\ \right] & \end{array}\right] \end{array}\right]$$

This description is then merged into the template specified by the rule. This template may specify additional semantic information pertinent to the UFO. In this example, the template specifies *modus:[def:wh]* is to be merged with the semantic description thereby ensuring that a UFO with the dialogue act *open_request* is realized as a wh-question such as *what time do you arrive?*. Finally, this description replaces the value of semantics in the UFO. Once all atomic UFOs have undergone this process, the UFO is passed to the message generation module for realization.

One complication is that the format of semantic descriptions differs depending on whether the message generation module is rule-based or template-based. With rule-based generation, semantic descriptions are given by the belief module in sil as shown in the preceding example. With template-based generation, however, a simplified sil description, restricted to the specification of task parameters, is required from the belief module. Accordingly, rather than provide the description in (16) the belief module produces the description given in (17)

(17)

$$\left[\textit{goaltime} : \_ \right]$$

which the template generator would realize as *what time do you want to arrive?*. The complete set of parameters for flight enquiry, train enquiry and reservations tasks is given in figure **??**.

dbflight	**dbtrain**	**dbreservation**
flightid	trainid	journeyid
sourcecity	sourcecity	date
sourceairport	sourcestation	passengername
sourceterminal	sourceplatform	passengernumber
goalcity	goalcity	class
goalairport	goalstation	meal
goalterminal	goalplatform	
sourcetime	sourcetime	
goaltime	goaltime	
date	date	
carrierid	carrierid	
vehicle	viacity	
class	changecity	

Figure 5.2: Task types and parameters

This difference in semantic description can be justified in terms of the eliminating duplication of effort in the generation cycle. Semantic information in the dialogue manager is represented in sil. The template generator, however, operates on the basis of task parameters. Rather than the message planner deliver sil representations from which the generator must extract the task parameters, the message planner directly delivers the task parameters. What facilates this is that the correspondences between sil and the task parameters is an integral part of the belief representation in dialogue manager (see chapter on belief module for further details on this point).

5.3.1.4 Using Syntactic Descriptions

One advantages of template-based generation over rule-based generation is the ease and speed with which 'stock phrases', such as *british airways flight information* and *can you repeat that?*, as well as semantically complex utterances, such as *you want to know the arrival time of the next flight from Paris?* can be generated. To redress the balance, rules in the message planning module have been augmented so as to specify syntactic as well as semantic information in their templates[2].

Syntactic information is specified in accordance with the definition in (18).

(18)

$$syntax : \begin{bmatrix} category : \_ \\ string : \_ \\ semantics : \_ \\ subtrees :< [syntax*] > \end{bmatrix}$$

Of these dimensions, **category**, **string** and **semantics** provide information about the prioritized expression, while **subtrees** provides the same information about governed expressions in the utterance. The value of the **category** attribute is a syntactic category, such as noun, verb or sentence augmented with morpho-syntactic feature value pairs such as mood. The value of the **string** attribute is a string representation of the phase. The value of the **semantics** is an index co-referential with the index of a sil object in the semantic field. The value of the **subtrees** attribute is a list which consists of syntactic representations of the expressions governed by the root expressions.

Two types of templates incorporate syntactic information: fixed templates and partial template. Fixed template fully specify the string attribute of the syntactic field and thereby do not require a semantic description to be built. (19) gives a rule with a fixed template

[2]If planning rules use syntactic template, then the message planning module will no longer be language-independent.

(19)
```
rule(open_task0,eng) :::=
conds([dact:open_task, semarg:dbreservation,
                       semarg:dbflight])
:=>
template([syntax:
              [string:[what,flight,do,you,want,to,reserve]]])
:& prosody(_) :& ellipsis(_).
```

If a UFO matches this rule, i.e. the language flag is set to 'eng', and UFO has a dact value of *open_task* and the semantic roles *dbreservation* and *dbflight* (as (14) does), then no semantic description is build and the syntactic template can simply be merged with the UFO to give (20)

(20)

$$\begin{bmatrix} id : ufo36 \\ dialogue : \begin{bmatrix} dact : open\_task \end{bmatrix} \\ semantics : \begin{bmatrix} dbreservation : \_ \\ dbflight : \_ \end{bmatrix} \\ syntax : \begin{bmatrix} string : \begin{bmatrix} what, flight, do, you, want, to, reserve \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Partial templates, on the other hand, provide only a partial syntactic specification and do require semantic descriptions to be built. For example, with the rule in (21)

(21)
```
rule(req_spec0,eng) :::=
conds([dact:req_spec,reintroduce:1])
:=>
template([[_:A],
          [syntax:
               [string:[and,can,you,provide,more,
                             details,about,(#1)],
           subtrees:
               [string:(#1),semantics:[id:A]]]]]) :&
    description(object_in_ctx) :&
    prosody(_) :&
    ellipsis(_).
```

The string in the template is partially specified. The gap, indicated by (#1), is co-indexed to the string value of the subtrees attribute which contains, as its semantic specification, a co-referential index. What this index points to is the value of the semantic attribute in the UFO. When this rule is matched by a UFO, the message planning module requests an 'object_in_ctx' description of the semantic value in the UFO and when the description is returned it is instantiated as the semantic value in subtrees. The instantiated template is then merged with the UFO yielding (22) which provides the rule-based generator with a partial syntactic description of the system message and thus is only required to construct a linguistic description for the subtree in the syntax.

(22)

$$\left[\begin{array}{l}
id : ufo87 \\
dialogue : \left[\begin{array}{l} dact : req\_spec \\ reintroduce : 1 \end{array}\right] \\
semantics : \left[\begin{array}{l} id : time45 \\ type : s\_time \\ thedesc : \left[\begin{array}{l} id : depart3 \\ type : depart \\ thetime : \left[\; id : time45 \;\right] \end{array}\right] \end{array}\right] \\
syntax : \left[\begin{array}{l} string : \left[\; and, can, you, provide, more, details, about, (\#1) \;\right] \\ subtrees : \left[\begin{array}{l} string : (\#1) \\ semantics : \left[\; id : time45 \;\right] \end{array}\right] \end{array}\right]
\end{array}\right]$$

This will then be realized as *and can you provide more details about the departure time?* or *and can you provide more details about the time of departure?*

5.3.2 Realizing System Utterances

The message planning module sends the message generation module addition information to guide the linguistic realization and synthesis of the UFOs. In particular, it sends an information profile and the current linguistic history. The information profile consists of accessibility, prosodic and ellipsis information: accessibility information describes the accessibility of indices in the UFOs within the belief model, the prosody a schematic prosodic contour and the ellipsis information whether ellipsis is permitted or not. Accessiblity information is returned by the belief module with semantic descriptions; ellipsis and prosodic information is (currently) contained in the planning rules. The linguistic history is derived from the linguistic interface module.

5.3.3 Updating the Dialogue Manager

When the message generation module has realized and synthesised the system utterance, it returns a complete UFO description to the message planning module.

The message planning module then informs the dialogue module of the moves which have been generated and the linguistic interface module of the complete generated utterance.

5.4 Algorithms

Figure **??** presents an overview of the functionalities of the message planning module, including its interactions with the dialogue module, belief module, linguistic interface module and the message generation module.

Figure 5.3: Overview of functionalities

We now describe the major algorithms in the message planning module and indicate the corresponding prolog predicates.

5.4.1 Initialization: `init_mp`

When the dialogue module sends an ‘init’ message to the message planning module, the planning rules are compiled into a compact format and the stores are cleared.

Planning rules, whose unexpanded structure and content is described in section ??, are compiled into `rule/3`. The first argument is `conds/1` which specifies the conditions of the rule as a list of attribute value pairs for the dialogue act, the dialogue specifiers and the language. The second argument, `action/4`, specifies the action of the rule in terms of the template, the semantic description type, the prosody value and the ellipsis value. Both the template and the description type undergo expansion during compilation: the type of the template, fixed or partial is made explicit and the description is expanded, for example, from `object` to `request_description_object`. The third argument is the name of the rule. The following examples illustate rule compilation: (23) gives the uncompiled version and (24t)he compiled rule.

(23)
```
rule(confirm_def0,any) :::=
conds([dact:confirm_def,fail:1])
:=>
template([semantics:[id:_,type:understand,
                          modus:[pol:neg],
    theagent:[id:_, type:individual,
    thediscourserole:[id:_,type:discourserole,
                           value:speaker]],
    thetheme:[id:_A,type:utterance,
    thedesc:[id:_,type:say,
                  theagent:[id:_,type:individual,
    thediscourserole:[id:\_,type:discourserole,
                           value:hearer]],
    thetheme:[id:_A]]]]])
:&  prosody(_) :& ellipsis(_).
```

```
(24)  rule(conds([lang:any,reintroduce:0,
                 rep:0,dact:confirm_def,fail:1]),
          action(fixed([semantics:[id:A,
                         type:understand,modus:[pol:neg],
          theagent:[id:B,type:individual,
              thediscourserole:[id:C,type:discourserole,
                         value:speaker]],
          thetheme:[id:D,type:utterance,thedesc:
                        [id:E,type:say,
                        theagent:[id:F,type:individual,
                            thediscourserole:
                                [id:G,type:discourserole,
                                     value:hearer]],
          thetheme:[id:D]]]]]),none,prosody_def,no),
          confirm\_def0).
```

Initialization also results in the following stores being cleared

ufo_move/2	pairs of MoveId and UfoId
infoProfile/1	UfoId:InfoProfile pairs
card/1	card counter for building UFOs
ufo/1	ufo index counter for building UFOs

5.4.2 Transforming Moves to UFOs: `reduce/2` and `buildUfos/2`

The move list input from the dialogue module is tranformed into UFOs in two passes: in the first pass, the semantic and dialogue information in the moves is given a dag representation; and in the second, the dags are transformed into UFOs.

5.4.3 Building Sil Descriptions: `buildSilStructure/1`

Once moves have been transformed to an input UFO, this UFO is partitioned into an atomic UFO plus a context consisting of the remaining atomic UFOs in the input UFO. The atomic UFO is then processed as follows (`CheckRules/2`)

1. The UFO constraints are obtained: i.e. the dialogue act, dialogue modifier and semantic argument information is extracted and structured as a list of attribute value pairs (`getConds/2`)

2. The language type for generation is determined (stored in `language/1`). Together with the UFO constraints they are matched against the planning rules (`getAction/2`). That is, a rule is sought which not only satisfies the constraints contained in the UFO, but also the constraint on which language is to be generated. The action part of the matching rule is subsequently used in the processing of the UFO.

3. The decision as to whether a semantic description needs to be built by the belief module is taken (`bmArgs/3`). No semantic description is built if the rule specifies a fixed template or the semantic attribute of the UFO is not instantiated. Otherwise, the views (or attribute values pairs) from which the semantic description is to be build are collected.

4. What happens next depends upon whether descriptions need to be built (`BuildDescription/4`). When there are no descriptions to build, then the template is merged with the UFO, the information profile is constructed using the ellipsis and prosody information from the rule and next atomic UFO is processed (`mergeTemplate/3`, `makeInfoProfile/4` and `nextUfo/2`). When descriptions are required from the belief module, the message planning module sends the appropriate requests to the belief module via postie and suspends itself with `buildingSilStructure/4` as the return predicate
(`send_to_mp(requestBmDescription/4)`)

5. The message planning module resumes when the belief module has returned the requested descriptions (`buildingSilStructure/4`). The descriptions are structured and merged with the UFO (`processBmInfo/3` and `mergeDescription/5`). In particular, the descriptions are successively merged with the template (which may contain additional information); for example, if there are two descriptions, as is the case with `object_and_ctx`, the first description is merged with an uninstantiated id in the template, the second with a different uninstantiated id and so on until there are no unmerged descriptions. Once the descriptions are merged, the information profile for the UFO is build and the next UFO is processed (`makeInfoProfile/4` and `nextUfo/2`).

6. When all atomic UFOs in the input UFO have been processed, the relevant entries in the linguistic history are requested from the linguistic interface module (currently the user's most recent utterance) and the message planning module suspends with `makePlan/2` as the return predicate (`send_to_mp( realize/1)`).

7. Once the linguistic interface returns the linguistic history entries, the message planning module resumes (`makePlan/2`). The information profiles for

the UFO are then recovered (`getInfoProfile/2`) and `realize(+UFOs, +InfoProfile, +LinguisticHistory, -ReturnUFOs)`, exported by the message generation module, is called. The UFOs are then linguistically realized, synthesised and returned to the message planning module as ReturnUFOs[3]. The MoveIds are then recovered from the ReturnUFOs (`dialogue_filter/2`). These are then send to the dialogue module thereby confirming that moves sent for generation have actually been generated. Finally, the ReturnUFO is sent to the linguistic interface module so the linguistic history is kept up to date.

5.4.4 Suspend and Resume: `suspend/5` and `resume/2`

One of the constraints imposed upon our implementation by the dialogue manager architecture is interactivity: i.e. it is assumed that modules are capable of interacting with other modules to, for example, obtain relevant information during processing. The approach to interaction adopted in the message planning module (and the task module) is to used a pair of suspend/resume predicates. The approach has the advantage that the overheads are cheap (cf. saving the goal stack, variable bindings, etc) and the disadvantage that processes which need to interact with other modules can't be recursive.

The idea behind the suspend/resume pair is very simple. `suspend/5` is called not only with the information about the external calling process, but also information about the process in the message planning module which is to be called when the external call succeeds. For example, in `suspend(bm, request_description_object, [view34], buildingSilStructure(_, Action, Ufo, UfoCtx), 1)` the first three arguments describe a request for the belief module to build a semantic description. Calling this predicate results in the following message being sent to postie: `send_to_pm(bm, mp, request_description_object(mp1, view34))`. Prior to this message being sent, however, the message planning module saves a state which specifies the message index (mp1), the return predicate (`buildingSilStructure/4`), the argument position in the return predicate to be instantiated with the result of the request (1) and an argument to hold intermediate results in cases where more than one call is necessary (`saved_state/4`). When the belief module returns the description — `send_to_pm(mp, bm, return_description_object(mp1, [id:city1, type:city, value:paris]))` — the resume predicate is called: `resume(mp1, [id:city1, type:city, value:paris])`. This predicate retrieves the appro-

[3]With template-based generation, the UFOs are returned unchanged. With rule-based generation, they are augmented syntactically and semantically.

priate saved state on the basis of the message index and, if there is only a single index, instantiates the appropriate argument in the return predicate with the description and calls the predicate: i.e. `buildingSilStructure([id:city1, type:city, value:paris],Action, Ufo, UfoCtx)`. In this way, the message planning module is capable of interaction with both the belief module and the linguistic interface module.

5.5 Knowledge Bases

The only knowledge base in the message planning module is a set of customizable planning rule. The rules determine whether or belief module is required to build a semantic description as well as specifying default prosodic information and ellipsis information and, optionally, syntactic information. If the UFO is to be realized as a fixed template, then no description building takes place. If it is to be realized as a partial semantic template or without a template ('raw' sil), then the belief module is requested to build the appropriate descriptions. When this happens, then the rules also determine the type of description required.

In their uncompiled form the rules have the following format

```
  rule(Name,Lang) ::= conds(Dact, Rep, Fail | ArgLabels) =>
template(Template) :& description(Type) :& prosody(Value) :&
                        ellipsis(Value).
```

`rule/2` takes two arguments. `Name` gives the name of the rule. `Lang` specifies the type of output generated by the rule. Template-based generation modules use rules whose `Lang` value is 'temp'. The remaining values are for rule-based generation modules. If the rule does not specify a syntactic description, then the value is 'any': output is given in standard sil and is accordingly language-independent. With rules which do specify syntactic description, the value indicates the language – 'eng', 'fr','ger' or 'ital' – and output is language-dependent.

`conds/4` describes the satisfaction conditions for the rule. `Dact` is the value of the dact attribute. `Rep` is the value of the dact rep (for repetition); if no value is given, then the default (0) is assumed. `Fail` is the value of the dact fail (for failure); if no value is given, then the default (0) is assumed. `ArgLabels` is a set of argument labels such as 'dbreservation'. If no ArgLabels are specified in the rule, then satisfaction of the rule does not depend upon the presence of argument labels (rather than actually require no argument labels to be present in the UFO).

Given a rule is satisfied, it supplies the information after the `::=`.

`template` describes the template used in planning the utterance. There are three possible values

1. ‘none’ (which is the default)
2. a partial template of the form `[IndexList,PartialTemplate]` : where `IndexList` is a list of indices specifying where the semantic description from the belief module is to be placed within the template. For example,

   ```
   partial([_:A],[syntax:[string:  [can,you,tell,me,(#1)]
           subtrees:[string:(#1) semantics:[id:A]]]])
   ```

   is a partial template used to generate phrase such as *can you tell me the time of departure.*
3. a fixed template – i.e. a template without subtrees. For example,

   ```
       rule(open_task0,eng) ::= conds(open_task,0,0,none) =>
   template([syntax:[string:[can,i,help,you]]]) :& prosody(_) :&
                            ellipsis(no).
   ```

No belief module descriptions are required with this type of template.

`Type` determines the type of index description sought from the belief module. The possible values are

object a simple description of the object

object_in_ctx a description of the object in terms of its context. For example,

$$\begin{bmatrix} \mathit{id} : \mathit{time}4 \\ \mathit{type} : \mathit{time} \\ \mathit{thedesc} : \begin{bmatrix} \mathit{id} : \mathit{flight}34 \\ \mathit{type} : \mathit{single\_journey} \\ \ldots \\ \mathit{thedeparture} : \begin{bmatrix} \mathit{id} : \mathit{depart}78 \\ \mathit{type} : \mathit{depart} \\ \mathit{thetime} : \begin{bmatrix} \mathit{id} : \mathit{time}4 \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

object_and_ctx as above, but the object and context are returned as separate descriptions

object_with_ctx as object_in_ctx but given as part of integrated description

Note that the same description type applies to all views in the UFO: it is not possible to described one view in one way and another view in an alternative way.

`Prosody` specifies the type of prosody to be assigned to the ufo. If no value is given, the default is assumed ('default_prosody').

`Ellipsis` specifies whether or not ellipsis is possible in the ufo. The two possible values are 'yes' and 'no'. If no value is given, the default is assumed ('no').

These rules are compiled into `rule/2` as described above.

5.6 Customization

Two aspects of the message planning module can be customized: the language type and the rule set.

The language type determine what type of rules are tried in building the description; for example, 'any' will allow only language independent rules to be tried and 'eng' will use those with the type 'eng' or 'any', but not 'ger', 'fr', etc. The type is set by the flag `language/1` where the argument is one of the permissible language values.

The rule set can be customized by simply adding new rules or deleting old ones. It is possible, however, to build up a large set of planning rules which allow the generation of utterance in different language and/or using different types of generators and selecting the required configuration via the language flag. Rules specific to other languages and/or generators will simply not be satisfied when the rules are matched against the UFO.

5.7 Implementation

Since the algorithmic description of the message planning module referenced the prolog predicates used in implementation, here we present an overview of the interfaces and the software structure.

Figure **??** gives an overview of message planning interface. 'Type' refers to whether or not the predicate requires suspension of the module.

interface	**predicate**	**type**
dm → mp	init	non-interactive
dm → mp	generate/1	non-interactive
mp → dm	generated/1	non-interactive
mp → bm	request_description_object/2	interactive
mp → bm	request_description_object_in_ctx/2	interactive
mp → bm	request_description_object_and_ctx/2	interactive
mp → bm	request_description_object_with_ctx/2	interactive
bm → mp	return_description_object/2	non-interactive
bm → mp	return_description_object_in_ctx/2	non-interactive
bm → mp	return_description_object_and_ctx/2	non-interactive
bm → mp	return_description_object_with_ctx/2	non-interactive
mp → li	request_linguistic_history/2	interactive
li → mp	return_linguistic_history/2	non-interactive
mp → li	generated_utterance/1	non-interactive
mp → mg	realize/4	non-interactive

Figure 5.4: Interfaces of the Message Planning module

Figure **??** summarizes the current software structure, indicating the entry points and main predicates in each software module.

5.8 Conclusions

In this chapter, we have presented a description of the message planning module. Three types of extensions are currently envisaged. Firstly, the module will be extended to 'optimize' the structure of the descriptions built by the belief module. For example, the belief module currently builds verbose solutions which can be realized as *ba 245 departs from London at 5pm and ba 245 arrives in Paris at 6pm.* Optimization would simplify this to *ba 245 departs from London at 5pm and arrives in Paris at 6pm.* Secondly, when interactive versions of the message generation module become available, the message planner will provide information about system utterances as and when it is required. Thirdly, the responsibility for dialogic planning will be moved from the dialogue module to the message planning module. One consequence of this move is that the planner is sent a set of dialogic goals which it must determine how, and when, to realize. The dialogue module is then informed of this decision. Another consequence is that predictions would be built after planning has taken place. Prediction build-

module	**entry points**	**key predicates**
mp.pl	receive_from_pm_mp/2 makePlan/2	language/1 send_to_mp/1
rplan.pl	reduce/2 buildUfos/2	
splan.pl	buildSilStructure/1 buildingSilStructure/4	checkRules/2 bmArgs/3 buildDescription/4 mergeDescriptions/5 mergeTemplate/3 makeInfoProfile/4 nextUfo/2
interaction.pl	suspend/5 resume/2	
mp_rules.pl	compile_rules/0	rule/3
mp_tools.pl	various ancillary	predicate
mg.pl	realize/4	

Figure 5.5: software modules with entry points and key predicate

ing could then take into account not only dialogic structure of the next system utterance, but also its semantic structure.

Chapter 6

The Linguistic Interface

6.1 Overview

The Linguistic Interface (LI) is the Dialogue Manager module responsible for managing the interface between the Dialogue Manager and the Linguistic Processor (LP). It maintains a local database, carries out structural transformations, and superintends communications between the Dialogue Manager and the LP.

6.2 Scope

The LI has a number of basic functions. These can be grouped under the following headings:

- linguistic history management
- prediction building
- interface management

6.2.1 Linguistic history management

A record of everything that has been said so far is kept in the Linguistic History (LH). More accurately, an Interface Language (IL) representation of everything that has been said so far is kept in the LH. This is currently used for two main purposes. Firstly, it is used in order to add detail to next-utterance predictions.

The Dialogue Module (DM) predicts the dialogue acts and task/lexical types and objects which are expected to occur in the next utterance. The LI examines the LH to see if any of the objects have been mentioned before and, if so, it retrieves the referring expressions used to identify them and builds these into the prediction. Secondly, the LH is used in order to retrieve linguistic forms for reuse by the Message Planner/Message Generator.

Typically, newly added LH entries contain variables (serving as placeholders for object ids). The Belief Module (BM) may send the LI object/variable bindings. The LI uses these to update the LH.

Entries in the LH must contain more than IL structures. They must record the identity of the speaker and the temporal course of the dialogue.

It is to be expected that new ways of using the LH — for example, in recovering from errors by reanalysing earlier utterances — will be added over time.

6.2.2 Prediction building

At present, the DM sends the LI a set of predictions consisting of one or more structures of the form:

```
[move:       ID,
 prediction: [context: _,
              acts:    [preferred:    _,
                        dispreferred: _]]]
```

The LP expects a corresponding list of one or more predictions, each one having structure of the form:

```
[ID,
 context(Task_Type, Lexical_Type_List, Word_list),
 Dialogue_Act_List]
```

The LI is also capable of discarding all of the structure in predictions and passing a single flat list of the form:

```
[ID,
 Dialogue_Act_List,
 Task_Type_List,
 Lexical_Type_List,
 Word_List]
```

6.2.3 Interface management

Some LP versions (France and UK) are implemented — at the top level at least — in Quintus Prolog. This allows the LI/LP interface to be implemented by means of simple Prolog predicate calls. The German LP version is implemented in Lisp. LI-LP interactions take place via the Quintus Prolog foreign language interface to an intermediate C program. In order to provide a maximally accessible interface, the LI also supplies the option of writing its output to and reading its input from text files. These files are accessed either linearly by reading the first available term at the start of a file or non-linearly by way of menu selection. Thus, part of the LI's functionality is concerned with the management of different kinds of interface.

6.3 Principles

The LI currently serves as a fairly transparent interface between the DM and the LP. It is expected that its role will increase in the future so that it plays a more active part in the construction of predictions.

A software design principle which is extensively used in the LI is to check flag settings in a defaults file before carrying out any of the major actions of the module. Thus, by changing the settings of a small number of customization flags in this file, the operation of the whole module can be changed.

6.4 Algorithms

The LI does not undertake any significant reasoning. The algorithms it uses to build predictions, manage the LH, and manage the interface are all simple, and should be obvious from the descriptions in Section **??**.

6.5 Knowledge bases

The LI does not make use of any special knowledge bases. It is supplied with all of the knowledge it needs by other modules. This knowledge does not have to be specially requested.

Of course, the LI does build a knowledge base as processing moves through a dialogue. This knowledge base (the LH) consists of a collection of clauses, one for each utterance in the dialogue so far. Implementation details of the LH can be found in Section ?? below.

6.6 Customization

The LI is very easy to customize. Currently, it can be customized in respect of six features relating to predictions and interface types. The behaviour of the system can be altered by changing the arguments of the following predicates (defined in the file `defaults.pl`, where the possible argument (i.e. flag) values are shown in square brackets.

Predicate	Possible Arguments
`do_you_want_predictions/1`	`[yes, no]`
`which_prediction_format/1`	`[flat, structured]`
`what_interface_type/1`	`[prolog_interface, text_interface, menu_interface, explorer_interface, file_interface]`
`which_prolog_predicate/1`	Any predicate
`which_text_bracket/1`	`[round, square]`
`which_text_file_in/1`	Any file
`which_text_file_out/1`	Any file
`which_dag_format/1`	`[attr_val, flat_list]`
`which_menu_language/1`	`[english, french, german]`
`file_input_file/1`	Any file
`trace_li/1`	`[off, on]`

The predicate named in `which_prolog_predicate/1` must be defined in the LP and explicitly imported into the LI by means of a `use_module/2` declaration at the start of the `li.pl` file.

It is expected that extra functionalities will be added to allow the format of predictions to be customized. These will be introduced as the need for experimentation with new formats arises.

These flags can be set in three different ways:

1. Prior to loading the LI program, the `defaults.pl` file can be edited.
2. After loading, settings may be changed dynamically by means of the `toggle/2` predicate. Calls take the form `toggle(Predicate,New_Setting)`. All of the active settings can be printed out by calling the `toggles/0` predicate.

3. After loading, the user may invoke a customization routine to step through a flag-setting menu. The routine is invoked by calling the `customize_li/0` predicate. Initially, the customization routine prompts the user (in English, French, or German) to select a language. This response is used to set the `which_menu_language/1` flag and all subsequent LI messages to the user are expressed in the language of choice. Subsequent prompt-response iterations are used to set the remaining customization flags. The relevance of certain flags is dependent upon the settings of other flags (e.g. `which_prolog_predicate/1` is only relevant if `which_interface_type/1` is set to `prolog_interface`). The `customize_li/0` routine only prompts for settings for flags which have not been rendered irrelevant since the start of the current invocation of the routine.

6.7 Implementation

The Quintus Prolog LI implementation makes use of three files, whose purpose is described in the following table.

FILENAME	PURPOSE
`li.pl`	Manages interactions with other modules (via postie). All major LI functionalities are located in this file, except those directly related to formatting I/O to fit the chosen communication channel.
`interfaces.pl`	This file contains predicates which manage the formatting and passing of communications at the DM/LP interface.
`defaults.pl`	This file contains default settings for the flags which control the behaviour of the LI. In principle, this is the only file which needs to be modified during customization.

6.7.1 Major functionalities: `li.pl`

The LI is defined as a Quintus Prolog module whose only exported predicate (other than any required at the DM/LP interface) is `receive_from_pm_li/2`. Six different definitions for this predicate can be found in `li.pl`. These are described below.

1. `receive_from_pm_li(_,init)`
   This is the initialization message. It is used to reset the linguistic history, i.e. to empty it. No constraints are placed on the identity of the module which sent the message, although it is currently assumed to be the DM. The only predicate called is `initialize_lh/0`.

2. `receive_from_pm_li(dm,prediction(Partial_Prediction))`
   The source of this message is the DM. Partial_Prediction is either a DAG whose top level has the form [move:_, prediction:_], or a list of such DAGs. The LI converts a Partial_Prediction into a Full_Prediction. This involves reorganizing the information in the Partial_Prediction and augmenting it with predicted words gleaned from the LH. The Full_Prediction is sent to the LP and processing suspends. Processing restarts when the LP returns an Utterance, i.e. an utterance field object (UFO). The built-in predicate `numbervars/3` is used to freeze any variables in the IL structure. The most recent entry in the LH is checked to find the last `numbervars/3` counter value. The call to `numbervars/3` is initialized with the value of the next integer. Once variables have been frozen, the LH is extended by asserting a clause of the form `linguistic_history(user,Utterance,V)`, where V is the last `numbervar/3` counter value. The semantic part of the UFO is extracted and sent to the DM via the postie by means of the postie's exported predicate `send_to_pm/3`.

   The following predicates are involved in this call to `receive_from_pm_li/2`.

```
build_prediction/2
  do_you_want_predictions/1
  bp/3
    extend_prediction/2
      find_acts/3
        lists:rev/2
        basics:member/2
      ordsets:list_to_ord_set/2
      find_task_types/2            % not fully implemented
      find_lexical_types/2         % not fully implemented
      find_words/3                 % not fully implemented
    ordsets:ord_union/3
make_prediction/2
  what_interface_type/1
  li_input:get_input_from_user/1   % EB's LP emulation menu
var_number/1
  linguistic_history/3
extend_lh/3
extract_semantics/2
```

```
  toolbag:get_path_val/3
pm:send_to_pm/3
```

3. `receive_from_pm_li(bm,bindings(Bindings))`
   The BM sends the LI a set of bindings. Bindings is a closed list containing one or more A/B correspondences, where A is an object (i.e. a unique identifier), and B is a numbervared variable. These correspondences are used to update the LH, so that all occurences of B are replaced by A. This can be done very easily by the predicate `toolbag:melt/3`. This predicate requires the bindings list to be open. At the moment it is assumed that only variables in user utterances need to be bound. The `selective_melt/4` predicate ignores system utterances in the LH and only melts/binds user utterances.

   The following predicates are involved in this call to `receive_from_pm_li/2`.

```
toolbag:make_open_tail/2
update_bindings/2
  selective_melt/4
    toolbag:melt/3
```

4. `receive_from_pm_li(mp,request_describe_object(Object))`
   This is used to allow the MP to request the set of wordings which have been used so far to identify Object. This functionality is not yet implemented.

5. `receive_from_pm_li(mp,generated_utterance(Utterance))`
   The MP sends the LI the UFO of the most recent system-generated utterance. This is added to the LH. The following predicates are involved in this call to `receive_from_pm_li/2`.

```
var_number/1
  linguistic_history/3
extend_lh/3
```

6. `receive_from_pm_li(Source,Query_mess))`
   This is an error trap. In the event of this instance of the predicate being called, an error message is generated of the form:

```
ERROR in LI: message not recognized
*** receive_from_pm_li(Source,Query_mess)
```

6.7.2 Interfaces: `interfaces.pl`

Currently two different kinds of interface are envisaged: a Prolog interface and a text interface. In the Prolog interface, the LI calls a predicate in the LP, for example a predicate of the form `li_lp_message(Prediction,Utterance)`. In the text interface, the LI writes messages to the LP in an ASCII file and reads messages from the LP from an ASCII file. There are two basic predicates.

1. `prolog_interface(Prediction,Utterance)`
   This consults the `defaults.pl` file to see what the LP's exported Prolog predicate is called. Suppose it is called `li_lp_message`. The LI then makes a call of the form `li_lp_message(Prediction,Utterance)`.

   The definition of `prolog_interface/2` is given below.

   ```
   prolog_interface(Prediction,Utterance) :-
       which_prolog_predicate(Prolog_Predicate),
       Interface_Call =.. [Prolog_Predicate, Prediction, Utterance],
       call(Interface_Call).
   ```

2. `text_interface(Prediction,Utterance)`
   The text interface is not yet fully defined. In principle, it works by checking the `defaults.pl` file to find out what kind of brackets to use in the Prediction (round or square) and what is the name of the file to write to and read from. Once it has converted the Prediction into a character string (rather than a Prolog structure) it writes the string to the named file. It is not yet clear how to get input from the LP when using a text interface.

   The current best definition of `text_interface/2` is given below.

   ```
   text_interface(Prediction,Utterance) :-
       which_text_bracket(Bracket_Type),
       textify(Prediction,Bracket_Type,Chars),
       which_text_file_out(Filename),
       tell(Filename),
       li_put(Chars),
       told.
   ```

6.8 Conclusion

A basic LI is in place. It achieves the modest range of functionalities specified for it in Sundial WP6 D1 and adds some extra functionalities in respect of customizability and LP simulation.

Chapter 7

The Postie

7.1 Overview

The modules of the Dialogue Manager communicate solely by passing messages from one to another. The 'Postie' (called after the colloquial name for a "post person") is a very simple module which administers the message passing. Because the Postie is at the hub of the Dialogue Manager and because all messages between modules pass through it, it is also the place for additional code to help with tracing, debugging and testing the Manager as a whole.

7.2 Scope

The Postie is intentionally very simple. The temptation to load it with any functionality beyond passing on the messages it receives has been resisted, in order not to compromise the modularity of the architecture.

In principle, it ought to be possible to distribute modules between processors in a distributed implementation, with the Postie merely supervising communications between processes. This would take seriously the implication in the Functional Specification that communications between modules should be asynchronous. In practice, the Dialogue Manager has not been tested on such an architecture and there are likely to be unacknowledged timing dependencies between messages which would cause operational difficulties.

7.3 Principles

The principle of the Postie is very simple: it receives messages from the modules and places them into a first–in, first–out (FIFO) buffer. Messages are taken from the front of this buffer and despatched to their destination modules. This process continues until the buffer is empty. The Postie then does nothing until another message is received.

The Postie starts off the Dialogue Manager's operation by clearing its message buffer and sending an 'init' message to one of the modules (arbitrarily chosen to be the dialogue module). This module processes the message and generates further messages destined for other modules. Thus the endless processing of messages is begun, to continue forever (or until one of the modules sends a 'closedown' message to the Postie).

7.4 Algorithms

As noted in Section **??**, the algorithm used by the Postie is a simple FIFO buffer.

The computation is slightly complicated by facilities to trace messages and to run test files.

7.4.1 Tracing

A switch can be set to enable tracing. When this switch is set to on, every message received by the Postie is displayed on the console, and is displayed again when it is despatched to the recipient module.

7.4.2 Testing

To assist in testing the Dialogue Manager and to verify that changes to the code have not introduced inadvertent errors, the Postie includes a facility to help with 'regression testing'.

In one mode (set by an internal switch), all messages passing through the Postie are also recorded to a file. In another mode, a particular module is nominated and the Postie reads messages from the test file, passing them to the module under test. It also collects the messages sent from that module and compares them with the expected messages recorded in the test file. Any differences are signalled as errors. Thus, with suitable test file, individual modules can be tested to ensure that their external behaviour with respect to other modules remains constant despite internal changes.

7.5 Knowledge Bases

There are no knowledge bases associated with the Postie.

7.6 Customization

There is no customization required for the Postie.

7.7 Implementation

The Postie code is contained in one file `postie.pl`. There are two user-settable switches which control tracing (see Section ??) and testing (see Section ??).

To toggle trace mode on or off, evaluate:

```
:- tracing_pm
```

To set the testing code to collect messages in a file, evaluate:

```
:- testing(collect)
```

The name of the file will be requested.

To test a module against the messages in this file, evaluate:

```
:- testing(xx)
```

where '`xx`' is the name of the module (one of `dm, bm, tm, mp, li`).

To revert to normal mode, evaluate:

```
:-testing(off)
```

Appendix A

An example of running the Dialogue Manager

This appendix includes a trace of the activity of the Dialogue Manager 'conversing' with a user. It is included here to illustrate the working of the modules and the messages which flow between them that has been described in the previous chapters. The dialogue consists of a simple enquiry about the time of a flight, typical of the UK application. Similar dialogues have been developed for the French flight reservation domain and the German train enquiry domain.

A.1 How the trace was generated

The trace is an annotated version of the trace of messages passing through the 'Postie' module described in Chapter 6. It was generated using the testing facility built into the Postie module. Expressions in SIL corresponding to the 'user' utterances shown in **??** were fed from a file into the Linguistic Interface. The system generated system utterances suitable for the Message Generator and also copied the messages passing between modules into a file. This latter file is the origin of the transcript shown in **??**. The raw transcript has been modified in several ways to make it more readable:

1. It has been formatted to break the text into lines and to insert indentation to make the complex structures easier to follow.
2. Comments (between `/*` and `*/`, and after `%`) have been inserted.
3. The output intended for the message generator has been replaced by English text.

A.2 The dialogue

The transcript shows the messages flowing through the Postie for the following dialogue:

```
S1:     hello. Can I help you?

U1:     I want to go from London to Paris.

S2:     London Paris. What date do you want to travel?

U2:     June the 20th.

S3:     June the 20th. What time do you want to leave?

U3:     17 15.

S4:     17 15.

U4:     yes.

S5:     Flight BA123 leaves london heathrow at 17 15 and
        arrives at paris charles de gaulle at 18 39.
        Do you have another enquiry?

U5:     no.

S6:     Good bye.

U6:     Good bye.
```

A.2.1 The transcript

```
/*********************************************************
*
*       INITIALIZATION
*
*********************************************************/
```

```
msg(dm,pm,init).
msg(tm,dm,init).
msg(mp,dm,init).
msg(bm,dm,init).
msg(li,dm,init).

/*********************************************************
*
*       OPENING
*

*********************************************************

Generation of system's first utterance and interpretation
of user's first utterance

S1:     hello. Can I help you?

U1:     I want to go from London to Paris.

*********************************************************/

msg(dm,tm,request_start_task([dbflight])).

msg(dm,pm,what_to_do).

msg(mp,dm,generate(
             [[move(1,main,system,initiative,greeting,B,2,[],[])],
              [move(2,main,system,initiative,open_task,[dbflight],3,[],[])
              ]])).

msg(li,mp,request_linguistic_history(mp1,B)).

msg(mp,li,return_linguistic_history(mp1,[lx:[]])).

msg(li,mp,generated_utterance([hello,can,i,help,you])).

msg(dm,mp,generated([1,2])).

msg(li,dm,prediction(
       [[move:1,
         prediction:[context:ctx(dummy,dialogic),
                     acts:[[act:demrep,pref:dp],
                             [act:demreppar,pref:dp],
                             [act:open_request,pref:dp],
                             [act:posi_check,pref:dp],
                             [act:request_confirm,pref:dp],
                             [act:greeting,pref:p],
                             [act:epsilon,pref:dp]]]],
         [move:2,
          prediction:[context:[ctx(_28690,[dbflight])],
          acts:[[act:demrep,pref:dp],
                [act:demreppar,pref:dp],
                [act:open_request,pref:dp],
                [act:posi_check,pref:dp],
                [act:request_confirm,pref:dp],
                [act:forpb,pref:p],
                [act:wh_answer,pref:p]]]]]])).


msg(dm,li,input_utterance(
    unknown,
    [id:_3657,
            type:want,

            theagent:[id:A,
                      type:individual,

                      thediscourserole:[id:_,
                                        type:discourserole,
                                        value:speaker]],
            thetheme:[id:_,
                       type:go,

                      thetheme:[id:A,
                                type:individual,

                                thediscourserole:[id:_,
                                                  type:discourserole,
                                                  value:speaker]],

                      thegoal:[id:_3787,
                           type:location,

                       thecity:[id:_3848,
                                type:city,
                                value:paris]],

                  thesource:[id:_3939,
                             type:location,

                         thecity:[id:_4002,
                                  type:city,
                                  value:london]]]]])).

msg(bm,dm,request_stripped(1,
            [id:_3657,
            type:want,

            theagent:[id:A,
                      type:individual,


                      thediscourserole:[id:_,
                                        type:discourserole,
                                        value:speaker]],
            thetheme:[id:_,
                       type:go,

                      thetheme:[id:A,
                                type:individual,

                                thediscourserole:[id:_,
                                                  type:discourserole,
                                                  value:speaker]],

                      thegoal:[id:_3787,
                           type:location,

                       thecity:[id:_3848,
                                type:city,
                                value:paris]],

                  thesource:[id:_3939,
                             type:location,

                         thecity:[id:_4002,
                                  type:city,
                                  value:london]]]]])).
```

```
msg(dm,bm,return_stripped(1,
       [id:_3657,
            type:want,

            theagent:[id:A,
                      type:individual,

                      thediscourserole:[id:_,
                                        type:discourserole,
                                        value:speaker]],
            thetheme:[id:B]],
           task,
           [id:B,
            type:go,

            thetheme:[id:A,
                      type:individual,

                      thediscourserole:[id:_,
                                        type:discourserole,
                                        value:speaker]],

             thegoal:[id:_3787,
                      type:location,

                   thecity:[id:_3848,
                            type:city,
                            value:paris]],

              thesource:[id:_3939,
                         type:location,

                     thecity:[id:_4002,
                              type:city,
                              value:london]]])).


msg(bm,dm,request_anchor_task(2,
    [dbflight],
    [id:B,
         type:go,
         thetheme:[id:A,
                   type:individual,

                   thediscourserole:[id:_,
                                     type:discourserole,
                                     value:speaker]],

          thegoal:[id:_3787,
               type:location,


           thecity:[id:_3848,
                    type:city,
                    value:paris]],

      thesource:[id:_3939,
                 type:location,

             thecity:[id:_4002,
                      type:city,
                      value:london]]])).

msg(li,bm,bindings([_])).

msg(dm,bm,return_anchor_task(2,
    direct,
    [id:dbflight1,type:dbflight,
     sourcecity:london,
     goalcity:paris])).

msg(tm,dm,start_task(
    [id:dbflight1,type:dbflight,
     sourcecity:london,
     goalcity:paris])).

/**********************************************************
*
*       DEPARTURE DATE CYCLE
*
*********************************************************

Generation of system's confirmation of departure and arrival cities,
request for departure date and interpretation of user's reply
```

```
S2:     London Paris. What date do you want to travel?

U2:     June the 20th.

********************************************************/

msg(bm,tm,request_query_view(tmmsg1,dbflight1,[date])).

msg(tm,bm,return_query_view(tmmsg1,dbflight1,qview1)).

msg(dm,tm,request_parameter(dbflight1,[qview1,[date:_]])).

msg(dm,pm,what_to_do).

msg(mp,dm,generate(
    [
    [move(4,main,system,evaluation,confirm,
                ctx(dbflight1,[sourcecity])=london,3,[],3),
         move(5,main,system,evaluation,confirm,
        ctx(dbflight1,[goalcity])=paris,3,[],3)],
    [move(7,main,system,initiative,open_request,qview1,5,[],[])]
        ])).

% get confirmation description

msg(bm,mp,request_description_object(mp2,know(shared,
    avs(dbflight1,[sourcecity:london,goalcity:paris]),_95403))).

msg(mp,bm,return_description_object(mp2,
          [id:go1,
           type:go,

           thesource:[id:A,
                      type: city,
                      value: london],

           thegoal:[id:B,
                      type: city,
                      value: paris]],
          [go1:accessible])).

msg(bm,mp,request_description_object_in_ctx(mp3,know(shared,qview1,
    caller))).


msg(mp,bm,return_description_object_in_ctx(mp3,
           [id:s_date1,
            type:s_date,

            thedesc:[id:go1,
                     type:go,

                     thesourcetime:[id:time_point1,
                                    type:time_point,

                    thedate:[id:s_date1]],

                     thetheme:[id:individual1,
                               type:individual,
                               thediscourserole:[id:disrole1,
                                                 type:discourserole,
                                                 value:hearer]]]],

           [s_date1:[owner:B,accessibility:qview1:contrastive],
            go1:[owner:[caller],acessibility:acessible]])).

msg(li,mp,request_linguistic_history(mp4,B)).

msg(mp,li,return_linguistic_history(mp4,[lx:[]])).


msg(li,mp,generated_utterance([london,paris,what,date,do,you,want,
    to,travel])).


msg(dm,mp,generated([[4,5],7])).

msg(li,dm,prediction(
    [[move:6,
      prediction:[
        context:(ctx(dbflight1,[sourcecity])=london),
             acts:[[act:phackyes,pref:p],
                   [act:epsilon,pref:p],
                   [act:echo,pref:p],
                   [act:correction,pref:p],
                 [act:correction_strong,pref:dp]]]],
      [move:7,
       prediction:[
         context: (ctx(dbflight1,[goalcity])=paris),
              acts:[[act:phackyes,pref:p],
                     [act:epsilon,pref:p],
                     [act:echo,pref:p],
                     [act:correction,pref:p],
                     [act:correction_strong,pref:dp]]]],

      [move:6-7,
       prediction:[
              context:[ctx(dbflight1,[sourcecity])=london,
                         ctx(dbflight1,[goalcity])=paris],
              acts:[[act:phackyes,pref:p],
                     [act:epsilon,pref:p],
                     [act:echo,pref:p],
                     [act:correction,pref:p],
                     [act:correction_strong,pref:dp]]]],

       [move:9,
       prediction:[
              context:ctx(dbflight1,[date]),
              acts:[[act:demrep,pref:dp],
                     [act:demreppar,pref:dp],
                     [act:open_request,pref:dp],
                     [act:posi_check,pref:dp],
                     [act:request_confirm,pref:dp],
                     [act:wh_answer,pref:p],
                     [act:wh_answer_weak,pref:p]]]]]])).

msg(dm,li,input_utterance(unknown,
              [id:_260298,type:time_point,
              thedate:[id:_,type:s_date,
              theday:[id:_255393,type:day,value:20],
              themonth:[id:_259478,type:month,value:jun]]])).

msg(bm,dm,request_stripped(3,
              [id:_260298,type:time_point,
              thedate:[id:_,type:s_date,
              theday:[id:_255393,type:day,value:20],
              themonth:[id:_259478,type:month,value:jun]]])).
```

```
msg(dm,bm,return_stripped(3,
    [],
    unknown,
            [id:_260298,type:time_point,
           thedate:[id:_,type:s_date,
           theday:[id:_255393,type:day,value:20],
           themonth:[id:_259478,type:month,value:jun]]])).

msg(bm,dm,request_anchor(4,
    [ctx(dbflight1,[sourcecity]),
     ctx(dbflight1,[goalcity]),
     qview1],
                 [id:_260298,type:time_point,
           thedate:[id:_,type:s_date,
           theday:[id:_255393,type:day,value:20],
           themonth:[id:_259478,type:month,value:jun]]])).

msg(li,bm,bindings([_])).


msg(dm,bm,return_anchor(4,[qview1],direct,[date:2006])).

msg(tm,dm,reply_parameter(dbflight1,[qview1,[date:2006]])).


/*********************************************************
*
*       DEPARTURE TIME CYCLE
*
********************************************************

Generation of system's confirmation of departure date, request for
departure time and interpretation of user's reply

S3:     June the 20th. What time do you want to leave?

U3:     17 15.

*********************************************************/

msg(bm,tm,request_query_view(tmmsg2,dbflight1,[sourcetime])).

msg(tm,bm,return_query_view(tmmsg2,dbflight1,qview2)).
msg(dm,tm,request_parameter(dbflight1,[qview2, [sourcetime:_]])).

msg(dm,pm,what_to_do).

msg(mp,dm,generate(
   [
    [move(8,main,system,evaluation,confirm,
        ctx(dbflight1,[date])=2006,6,[],[])],
    [move(9,main,system,initiative,open_request,qview2,6,[],[])]
   ])).


% get confirmation description

msg(bm,mp,request_description_object(mp5,know(shared,
    avs(dbflight1,[date:2006]),_95403))).

msg(mp,bm,return_description_object(mp5,
           [id:time_point1,
            type: time_point,

        thedate:[id:s_date1,
                 type:s_date,

                 theday:[id:A,
                         type: day,
                         value: 20],

                 themonth:[id:B,
                           type: month,
                           value: june]]],
            [time_point1:access])).


msg(bm,mp,request_description_object_in_ctx(mp6,know(shared,qview2,
    caller))).

msg(mp,bm,return_description_object_in_ctx(mp6,
    [id:s_time1,
         type:s_time,

         thedesc:[id:depart1,
                  type: depart,

              thetime:[id:time_point1,
                                type: time_point,

                                thetime:[id:s_time1]],

                  thetheme:[id:individual1,
                            type:individual,
                            thediscourserole:[id:disrole1,
                                              type:discourserole,
                                              value:hearer]]]],
        [s_time1:[owner:B,acessibility:qview2:contrastive],
         time_point1:owner:[caller],acessibility:accessible])).

msg(li,mp,request_linguistic_history(mp7,B)).

msg(mp,li,return_linguistic_history(mp7,[lx:[]])).

msg(li,mp,generated_utterance([june,the,twentieth,what,time,do,you,want,
    to,leave])).

msg(dm,mp,generated([8,9])).

msg(li,dm,prediction(
      [
    [move:11,
         prediction:[context:(ctx(dbflight1,[date])=2006),
         acts:[[act:demrep,pref:dp],
           [act:demreppar,pref:dp],
           [act:open_request,pref:dp],
           [act:posi_check,pref:dp],
           [act:request_confirm,pref:dp],
           [act:phackyes,pref:p],
           [act:epsilon,pref:p],
           [act:echo,pref:p],
           [act:correction,pref:p],
           [act:correction_strong,pref:dp]]]],
    [move:13,
     prediction:[context:ctx(dbflight1,[sourcetime]),
     acts:[[act:demrep,pref:dp],
           [act:demreppar,pref:dp],
           [act:open_request,pref:dp],
           [act:posi_check,pref:dp],
           [act:request_confirm,pref:dp],
           [act:wh_answer,pref:p],
           [act:wh_answer_weak,pref:p]]]]])).

msg(dm,li,input_utterance(unknown,
     [id:_3832,type:time_point,
        thetime:[id:_,type:s_time,
          thehour:[id:_3895,type:hour,value:17],
          theminutes:[id:_3982,type:minutes,value:15]]])).

msg(bm,dm,request_stripped(5,
         [id:_3832,type:time_point,
        thetime:[id:_,type:s_time,
          thehour:[id:_3895,type:hour,value:17],
          theminutes:[id:_3982,type:minutes,value:15]]])).

msg(dm,bm,return_stripped(5,
    [],
    unknown,
    [id:_3832,type:time_point,
        thetime:[id:_,type:s_time,
          thehour:[id:_3895,type:hour,value:17],
          theminutes:[id:_3982,type:minutes,value:15]]])).

msg(bm,dm,request_anchor(6,
    [ qview1,
         qview2],
    [id:_3832,type:time_point,
              thetime:[id:_3832,
                      type:s_time,
              thehour:[id:_3895,type:hour,value:17],
              theminutes:[id:_3982,type:minutes,value:15]]])).

msg(li,bm,bindings([_])).

msg(dm,bm,return_anchor(6,
    [qview2],
    direct,
    [sourcetime:1715])).

msg(tm,dm,reply_parameter(dbflight1,[qview2,[sourcetime:1715]])).
```

```
/*******************************************************
*
*       TM FINDS FLIGHT
*
********************************************************

The assumption here is that the tm uses a mask to determine which
parameters are relevant in the task dbflight.  The default mask for
dbflight (i.e.  the user hasn't specified which parameter is required)
is [flightid, sourcecity, sourceairport, goalcity, goalairport,
sourcetime, goaltime].  This parallels the information used with the
task dbreserve.

*******************************************************/

msg(bm,tm,request_alternative_solution_view(tmmsg3,
    dbflight1,
    [[flightid:ba123,
          sourcecity:london,
      sourceairport:heathrow,
      goalcity:paris,
          goalairport:charles_de_gaulle,
      sourcetime:1715,
          goaltime:1839
    ]])).

msg(tm,bm,return_alternative_solution_view(tmmsg3,dbflight1,[aview1])).

msg(dm,tm,reply_solutions(dbflight1,[aview1])).

msg(dm,tm,request_confirm_end_task(dbflight1)).

msg(dm,pm,what_to_do).

/*******************************************************
*
*       CONFIRM DEPARTURE TIME CYCLE
*
********************************************************

Generation of system's confirmation of departure time and
interpretation of user's reply
```

```
S4:     17 15.

U4:     yes.

*********************************************************/

msg(mp,dm,generate(
    [
            [move(15,main,system,evaluation,confirm,
        ctx(dbflight1,[sourcetime])=1715,9,[],14)]
        ])).


% get confirmation description

msg(bm,mp,request_description_object(mp8,know(shared,
    avs(dbflight1,[sourcetime:1715]),_95403))).

msg(mp,bm,return_description_object(mp8,
           [id:time_point1,
            type:time_point,

            thetime:[id:s_time1,
                     type:s_time1,

                     thehour:[id:A,
                             type: hour,
                             value: 17],

                     theminutes:[id:B,
                                 type: minutes,
                                 value: 15]]],
            [time_point1:accessible])).

msg(li,mp,request_linguistic_history(mp9,B)).

msg(mp,li,return_linguistic_history(mp9,[lx:[]])).

msg(li,mp,generated_utterance([17,15])).

msg(dm,mp,generated([10])).

msg(li,dm,prediction(
     [
    [move:15,
         prediction:[context:(ctx(dbflight1,[sourcetime])=1715),
         acts:[[act:demrep,pref:dp],[act:demreppar,pref:dp],
           [act:open_request,pref:dp],[act:posi_check,pref:dp],
           [act:request_confirm,pref:dp],
               [act:phackyes,pref:p],[act:epsilon,pref:p],
           [act:echo,pref:p],[act:correction,pref:p],
           [act:correction_strong,pref:dp]]]]
     ])).

msg(dm,li,input_utterance(unknown,[id:_,type:dm_marker,value:yes])).

msg(bm,dm,request_stripped(7,[id:_,type:dm_marker,value:yes])).

msg(dm,bm,return_stripped(7,[id:_,type:dm_marker,value:yes],[],[])).

msg(dm,pm,what_to_do).

/********************************************************
*
*       DELIVERING THE SOLUTION
*
********************************************************

Generation of solution, request for further enquiry and
interpretation of user's reply

S5: Flight BA123 leaves london heathrow at 17 15 and
        arrives at paris charles de gaulle at 18 39.
        Do you have another enquiry?
U5: no.

********************************************************/

msg(mp,dm,generate(
     [
    [move(11,subarg,system,initiative,renup,[aview1],10,[],[])],
    [move(12,main,system,initiative,rensat,aview1,7,[],11)]
    ])).

msg(bm,mp,request_description_object_with_ctx(mp10,
    know(shared,[aview1],carrier))).
```

```
msg(mp,bm,return_description_object_with_ctx(mp10,
    [id:and1,
         type: and,

     first:[id:depart1,
            type:depart,


              thetheme:[id:carrier1,
                        type:carrier,

                        thecarrier_id:[id:carrier_id1,
                                       type:carrier_id,
                                       value:ba],

                        thecarrier_number:[id:number1,
                                           type:number,
                                           value:123]],


               theplace:[id:location1,
                         type:location,

                         theairport:[id:airport1,
                                     type:airport,
                                     value:heathrow],

                         thecity:[id:city1,
                                  type:city,
                                  value:london]],



               thetime:[id:time_point1,
                        type:time_point,

                        thetime:[id:s_time1,
                                 type:s_time,

                                 thehour:[id:hour1,
                                          type:hour,
                                          value:17],
                                  theminutes:[id:minutes1,
                                              type:minutes,
                                              value:15]]]],

          rest:[id:arrive1,
                type:arrive,

              thetheme:[id:carrier1,
                        type:carrier,

                        thecarrier_id:[id:carrier_id1,

                                       type:carrier_id,
                                       value:ba],

                        thecarrier_number:[id:number1,
                                           type:number,
                                           value:123]],


               theplace:[id:location2,
                         type:location,

                         theairport:[id:airport2,
                                     type:airport,
                                     value:charles_de_gaulle],

                         thecity:[id:city2,
                                  type:city,
                                  value:paris]],


               thetime:[id:time_point2,
                        type:time_point,

                        thetime:[id:s_time2,
                                 type:s_time,

                                 thehour:[id:hour2,
                                          type:hour,
                                          value:18],
                                     theminutes:[id:minutes2,
                                                 type:minutes,
                                                 value:39]]]]],

             [and1:access])).



msg(li,mp,request_linguistic_history(mp11,B)).

msg(mp,li,return_linguistic_history(mp11,[lx:[]])).

msg(li,mp,generated_utterance(
          [flight,ba123,leaves,london,heathrow,at,17,15,and,
           arrives,at,paris,charles_de_gaulle,at,eighteen,thiry,nine,
        do,you,have,another,enquiry])).

msg(dm,mp,generated([11,12])).

msg(li,dm,prediction(
     [
    [move:19,
         prediction:[context:[aview1],
     acts:[[act:demrep,pref:dp],[act:demreppar,pref:dp],
          [act:open_request,pref:dp],[act:posi_check,pref:dp],
          [act:request_confirm,pref:dp],[act:alt_answer,pref:p],
          [act:correction,pref:p],[act:epsilon,pref:dp]]]],
        [move:21,
     prediction:[
         context:ctx(dbflight1,[dbflight1]),
         acts:[[act:posi,pref:p],
         [act:nega,pref:p]]]]
     ])).

msg(dm,li,input_utterance(unknown,[id:_,type:dm_marker,value:no])).

msg(bm,dm,request_stripped(8,[id:_,type:dm_marker,value:no])).

msg(dm,bm,return_stripped(8,[id:_,type:dm_marker,value:no],[],[])).

msg(dm,pm,what_to_do).

/*****************************************************
```

```
*
*       CLOSING
*
*******************************************************

Generation of system good bye and interpretation of user's reply

S6:     Good bye.

U6: Good bye.

*******************************************************/

msg(mp,dm,generate(
     [  [move(23,main,system,initiative,ph_closure,
         _,13,[],[])]
     ])).

msg(li,mp,request_linguistic_history(mp12,B)).

msg(mp,li,return_linguistic_history(mp12,[lx:[]])).

msg(li,mp,generated_utterance([good,bye])).

msg(dm,mp,generated([13])).

msg(li,dm,prediction(
     [
    [move:23,
         prediction:[context:ctx(dummy,dialogic),
     acts:[[act:demrep,pref:dp],[act:demreppar,pref:dp],
           [act:open_request,pref:dp],[act:posi_check,pref:dp],
           [act:request_confirm,pref:dp],
               [act:ph_closure,pref:p],[act:hangoff,pref:dp],
           [act:epsilon,pref:p]]]]
     ])).

msg(dm,li,input_utterance(unknown,[id:_,type:dm_marker,value:goodbye])).

msg(bm,dm,request_stripped(9,[id:_,type:dm_marker,value:goodbye])).

msg(dm,bm,return_stripped(9,[id:_,type:dm_marker,value:goodbye],[],[])).
msg(pm,dm,[closedown]).
```