WP6D3—Dialogue Manager Design Evaluation

Morena Danielli, Wieland Eckert,
Norman Fraser, Nigel Gilbert,
Marc Guyomard,
Paul Heisterkamp,
Maloud Kharoune,
Jean-Yves Magadur,
Scott McGlashan, David Sadek,
Jacques Siroux and Nick Youd

July 1992

TITLE	:	WP6D3—Dialogue Manager Design Evaluation
PROJECT	:	P2218 - SUNDIAL
AUTHOR	:	Morena Danielli, Wieland Eckert, Norman Fraser, Nigel Gilbert, Marc Guyomard, Paul Heisterkamp, Maloud Kharoune, Jean-Yves Magadur, Scott McGlashan, David Sadek, Jacques Siroux and Nick Youd
ESTABLISHMENT	:	Cap Gemini Innovation, CNET, CSELT, Daimler-Benz Forschunginstitut, IRISA, Logica Cambridge Ltd, University of Erlangen, University of Surrey
ISSUE DATE	:	July 1992
DOCUMENT ID	:	
VERSION	:	4
STATUS	:	Final
WP NUMBER	:	6
LOCATION	:	Mikhail
KEYWORDS	:	Deliverable, Dialogue, Evaluation, Documentation

Abstract

This report is the third deliverable from Work Package 6: Dialogue Management.

It consists of an evaluation of the functionality of the intermediate Dialogue Manager which was demonstrated to the CEC in October 1991, together with proposals for the design and implementation of the final version of the Dialogue Manager.

Contents

Chapter 1

Introduction

This document constitutes the third deliverable from Work Package 6, the work package which is concerned with dialogue management in the Sundial demonstrators. The first deliverable, released in July 1990, was a functional specification of the 'Dialogue Manager', a module which was intended to handle all the dialogue management functions of the demonstrator systems. This specification (referred to in the rest of this document as WP6D1) was used as a guide for the development of the first implementation of the Dialogue Manager module. This formed the major part of the second deliverable, WP6D2, released in July 1991. The deliverable also included a volume of program documentation. The Dialogue Manager was integrated with other modules (Front-End Processor, Linguistic Processor, and Message Generator) for a demonstration of the so-called 'intermediate system' to the CEC in October 1991, at Logica's Cambridge office.

Because this initial implementation was the first attempt to translate the specification into code, it had a number of weaknesses. These included lengthy processing times, some inelegant and redundant code resulting in difficulties in extending the module to cover further functionality, and a number of 'bugs'. These problems were to be expected in a first version of a complex and experimental system and were anticipated when the project was initially devised. The plan for development of the Dialogue Manager set out in the project's Technical Annexe envisages that the initial version would be evaluated to ascertain whether it had the specified functionality and that it would be rebuilt to take advantage of lessons learnt. The rebuilt version would form part of the final system to be evaluated with users towards the end of the project.

The present report documents in some detail the problems encountered in the initial version and proposes how they will be overcome in the final version. While this evaluation report was being prepared the Dialogue Manager was also being developed further and so the majority of the proposals listed in the report have already been implemented. The remainder will be incorporated in versions of the Dialogue Manager due for release in July, September and November 1992.

As noted in previous deliverables, the Dialogue Manager is divided into several largely independent 'modules' communicating by means of a message protocol through a central message buffer. The merits of this architecture are reviewed in the next chapter of this report. One advantage which is not mentioned there, but which has practical import for the development of the Dialogue Manager implementation, is that each module may be independently re-implemented without affecting the rest of the system, provided that its message-passing interface is maintained and the messages it sends and receives are not changed. This feature has been valuable: for example, the dialogue module was entirely recoded, using quite different internal algorithms, while making only minor changes to the rest of the system. The internal workings of the other modules have not changed so drastically from the October 1991 version, but there have been significant changes and at the time of writing the Dialogue Manager is very much more robust, quicker, and has greater functionality as compared with the initial version.

Another advantage of the modular design of the Dialogue Manager is that it is thereby easier to design and implement on a European scale. Each module is the responsibility of one or two 'implementors'; these implementors are located in Germany, France and the UK. The Italian partners were originally not involved in Work Package 6 and made no contribution to the Dialogue Manager. The WP6 team is very pleased that they have now decided that the final version of their national demonstrator will use the Dialogue Manager, and the last chapter of this report summarises some of the issues which are being considered in order to link it with the Italian Linguistic Processor.

It should be emphasised that the Dialogue Manager will now be common to all the four Sundial national demonstrators. This means that all the demonstrators will be running exactly the same code for dialogue management. What will vary from country to country, and application to application, are the contents of Prolog databases which specify, for example, domain-specific task knowledge, and language-specific semantics. In addition, each demonstrator will have a purpose-built interface to link the generic Task Module to the application database (for example, the Italian train timetable database).

Following this introductory chapter, the report begins by considering the architecture of the Dialogue Manager and its strengths and weaknesses. The following five chapters review each of the Dialogue Manager's modules in turn, considering the problems encountered with the implementation of the module that was used for the October 1991 demonstration, and indicating the improvements that will or have already been put into place for the final version. The last chapter reports on progress with connecting the Dialogue Manager to the Italian demonstrator. This uses a special interface between the Dialogue Manager and the Linguistic Processor. No major difficulties have yet been identified, and there is no doubt that integration with the Italian system will shortly be achieved, within the time scale originally planned. This work is interesting as a test of the flexibility of the Dialogue Manager's design and its potential for use in systems other than those for which it has been constructed.

This report has been written jointly by those responsible for implementing the Dialogue Manager modules. Each chapter was first drafted by the implementors of the module and then commented on by the other authors. Implementation of the modules since the October 1991 review has been carried out as shown in table 1.

Module	Implementors
Dialogue	Jean-Yves Magadur, David Sadek
Task	Jacques Siroux, Marc Guyomard,
	Maloud Kharoune, Wieland Eckert
Belief	Paul Heisterkamp, Nick Youd
Message Planner	Scott McGlashan
Linguistic Interface	Nick Youd
'Postie'	Norman Fraser
Integration and version control	Scott McGlashan
Direction	Nigel Gilbert

Chapter 2

The Dialogue Manager Architecture

2.1 Overview of the architecture

The Dialogue Manager has a modular structure accounting for a distributed architecture. It is composed of five basic modules: the Dialogue Module, the Belief Module, the Task Module, the Linguistic Interface and the Message Planner. These modules interact via a message exchange synchronisation module called "Postie". Each module has its own processing mechanisms and knowledge base, and is responsible for a part of the processing achieved by the Dialogue Manager.

At the heart of the Dialogue Manager is the Dialogue Module (DM). Its role is to interpret the user's utterances in terms of dialogue act realisations, and, then, to determine the relevant system reactions in the same terms. In its current design, the DM depends on a dialogue grammar and on a set of adjacency pairs of dialogue acts, expressing dialogue regularities and resulting from corpus analyses. The dialogue grammar allows the prediction of the next possible dialogue acts of the user by fitting the dialogue history within a dialogue structure; the DM interprets (the representation of) a user's utterance by comparing it with the predictions. As regards the adjacency pairs, they are used to select the right reaction(s) corresponding to the user's dialogue act(s).

The role of the Belief Module (BM) is to extract the semantic (or propositional) content of a user's utterance and to isolate for the DM the utterance part which normally realises the communicative function(s) (i.e., the dialogue act type(s)). For the semantic content to be unambiguous, the BM maintains a "world" model and current focus spaces in order to determine the referents of descriptions, and, thus, to solve ellipses and anaphora.

It is the Task Module (TM) which provides the system with task-oriented goals, and, in this respect, motivates the task-oriented system “initiatives”. Roughly, for a given task, these goals are the collection of legal values of the database attributes which have to be known for the system to be able to query the database and satisfy the user’s request. Moreover, the TM is endowed with mechanisms which trigger cooperative behavior of the system: the inference of parameter values allows the abbreviation of the parameter collecting dialogue phase; constraint relaxation triggers the generation of cooperative (e.g., corrective/suggestive) answers; and the database query results are ordered, reduced or summarized so that they are better suited for presentation to the user.

As indicated by its name, the Linguistic Interface (LI) is one of the two modules — the other one being the Message Planner — which bridge the gap between the linguistic-dependent and independent aspects of the dialogue system. By maintaining a linguistic history which records the surface forms of everything that has been said so far, its role is twofold: firstly, it builds part of the linguistic form of the DM predictions by retrieving referring expressions which have been used previously in the dialogue to identify objects mentioned in the predictions; secondly, it retrieves the possible linguistic forms which can be reused in the next system utterance.

The role of the Message Planner (MP) is to prepare the linguistic realisation of the system dialogue acts by providing a linguistic-like description of the set of dialogue acts produced by the DM. In its current design, the MP does not perform either global semantic planning (i.e. the construction of a semantic representation of each system move), which is carried out by the BM, nor local planning (i.e. deciding whether and how objects are to be given a linguistic realisation), which is done by the Message Generator.

2.2 Strengths and weaknesses of the Dialogue Manager architecture

Both the strengths and the weaknesses of the Dialogue Manager are numerous. Only a few of them are addressed here. For the sake of improving the system, and because the practical strengths will clearly appear through the detailed description of the modules in the following chapters, in this chapter the strengths are slightly less developed than the weaknesses.

2.2.1 Strengths

The strengths of the Dialogue Manager architecture are mainly due to the basic principles which motivates it. These principles ensure that the system is generic along three different analysis dimensions. They can be stated as follows:

(1) *Separate the task level from the dialogue level:* This principle guarantees that the system is transportable from one application to another. It engenders the separation between the Task Module and the Dialogue Module.

(2) *Separate the semantic level from the pragmatic level:* This principle is justified by the necessity of distinguishing object-related properties from interaction principles. It gives rise to the Belief Module.

(3) *Separate the language-dependent level from the language-independent level:* This principle is intended to allow the system to be customised as regards the communication language. It leads to the "input/output" modules, that is, the Linguistic Interface and the Message Planner.

2.2.2 Weaknesses

Some of the weaknesses from which the Dialogue Manager suffers are due to the type of modularisation adopted; others are inherent in the approach underlying this modularisation. Only the former are addressed here; the limitations and the problems related to the structural approach of the dialogue grammar model are not examined. Some of the latter and their corresponding solutions are discussed in the chapter on the Dialogue Module.

(1) *Redundancy of information between modules:* Since the approach to modularisation aimed at ensuring that the modules work autonomously, each module has its own knowledge base and processing mechanisms. This leads not only to explicit duplication of certain information, as in the linguistic history (three copies of which are respectively used by the LI, the MP and the BM), but also implicit duplication, since there is no common knowledge representation policy used within the modules (except the interface language, which is used to normalise the information exchanged between the modules). Thus, information may be represented in two (or more) different ways in two (or more) different modules.

(2) *Complexity of certain phases of the synchronisation protocol:* The Dialogue Manager architecture being totally distributed, a module is "unaware" of

the activity of another module unless the latter explicitly informs it about that activity. The control strategy used is a "suspend/resume" protocol between the modules via "Postie". A consequence of these features is the complexity of the message exchanges needed to achieve certain treatments. As an example, here is a problem concerning the synchronisation between the DM and the MP, encountered during the development of the system.

When the DM produces a dialogue act, say X, it sends the message "generate(X)" to the MP and the message "build_prediction(Y)" to the LI. These two messages are asserted in Postie memory. Then the DM hands over to Postie which extracts the first message (i.e., "generate(X)") and sends it to the MP. Now the MP runs and may need information about some objects involved in X. In this case, it sends the message "describe(X)" to the BM. Again, this message is asserted in Postie memory. Now Postie runs. It extracts the next message from the FIFO structure, i.e., the message "build_prediction(Y)", and sends it to the LI which runs, sends the computed predictions to the Linguistic Processor (LP) and waits for a new input from the user through the LP. However, this will never arrive since the user is still expecting that the system takes its speech turn. Hence, the LI never hands over, the BM never receives the MP message, and the system never achieves the dialogue act generated by the DM: deadlock!

There are at least two solutions to this problem:

1. Before sending the predictions to the LP, the LI has to wait for a "generated_utterance(X)" message from the MP.
2. Before sending the predictions to the LI, the DM has to wait for a "generated_utterance(X)" message from the MP.

It is the solution (2) that is implemented.

(3) *The difficulty of evaluating the DM and extending the scope of the system to cater for more general "problem solving" functionalities:* The architecture adopted derives from an approach in which dialogue phenomena are not treated as resulting from basic "cognitive" processes but, rather, as primitive phenomena per se. In addition, the modules implement operationally- and not logically-motivated components. A consequence of this is that functionalities such as inference capacities, consistency maintenance or co-reference handling cannot be identified and then analysed as generic properties of the system.

The architecture does not rest on a formal account of the specifications. Therefore, practical tests of the system's abilities cannot be strengthened by theoretical expectations. Thus, for example, it cannot be guaranteed that the system is able to produce ellipses, but only that, in certain dialogue situations (isomorphic to test configurations), the dialogue system will generate an ellipsis. This is because the mechanisms which generate these phenomena cannot be viewed as a clearly circumscribed part of the dialogue system.

Chapter 3

The Dialogue Module

3.1 The current dialogue module

The previous release of the dialogue module, demonstrated during the October 1991 review in Cambridge, was based on the use of dialogue structures. For the generation of system dialogue acts and the recognition of user acts, a two step algorithm was used. For generation, these two steps perform the following actions :

- step 1: Dialogue allowances are computed on the basis of dialogue rules which look for characteristic patterns in the dialogue structure,
- step 2: These dialogue allowances and the current goals of the system (mainly task goals and dialogue goals) are used to decide which system dialogue acts must be produced. The dialogue structure is updated accordingly.

For the interpretation of user's utterances, the first step is the same. In the second step, one uses the dialogue allowances plus the semantic representation of the user's input to decide how to update the dialogue structure.

This two steps algorithm is quite complex to implement and makes the code difficult to understand and to maintain. The current algorithm is the result of the attempt to merge steps 1 and 2 in the previous algorithm.

The main idea is to use a richer dialogue structure, which we shall call the dynamic dialogue structure, which contains not only a trace of the dialogue acts produced so far in the dialogue, but also the current predictions of the system about the next possible dialogue acts. These predictions are computed, as before, on the basis of a list of the pairs which are the most frequently observed in the Wizard of Oz corpus. There are not only (system_act, user_act) pairs, but also (user_act, system_act) pairs. This means that, after having recognized the dialogue acts produced by the user, the system makes predictions about its own next acts.

Some modifications have also been done to the format of the dialogue structure. The idea is to use dialogue structure which can be produced by a simpler grammar than before, but without any loss of information. We shall first examine this new format of dialogue structures before describing the new algorithm.

Let us consider a dialogue fragment like:

S_1	*On what date do you want to leave ?*
U_1	*June the 6th.*
S_2	S_2^1 *June the 6th.*
	S_2^2 *At what time ?*
U_2	*8 p.m.*

The dialogue structure which was associated with this fragment in the previous version of the dialogue module was:

$$D\left[\begin{array}{ll} E & \left[\begin{array}{ll} S_1 & \textit{On what date do you want to leave}? \\ U_1 & \textit{June the } 6\textit{th.} \\ S_2^1 & \textit{June the } 6\textit{th.}\end{array}\right] \\ E & \left[\begin{array}{ll} S_2^2 & \textit{At what time}? \\ U_2 & \textit{At } 8 \textit{ pm.}\end{array}\right]\end{array}\right]$$

In this representation, the evaluation of the first exchange is just an intervention made of one dialogue act.

But for a dialogue fragment like:

S_1	*On what date do you want to leave?*
U_1	*June the 6th.*
S_2	*June the 6 th ?*
U_2	*Yes, that's right, june the 6th.*

the associated dialogue structure is as follow:

In that case, the evaluation is not reduced to an intervention, but is an ex-

$$D\left[\begin{array}{ll} E & \left[\begin{array}{ll} S_1 & \textit{On what date do you want to leave?}\\ U_1 & \textit{June the 6th.}\\ E & \left[\begin{array}{ll} S_2 & \textit{June the 6th?}\\ U_2 & \textit{Yes, that's right, june the 6th.}\end{array}\right.\end{array}\right.\end{array}\right.$$

change. In order to have a more unitary representation, we have chosen to consider that the evaluation of an exchange is always an exchange. With this convention, the dialogue structure of the first fragment must be:

$$D\left[\begin{array}{ll} E & \left[\begin{array}{ll} S_1 & \textit{On what date do you want to leave?}\\ U_1 & \textit{June the 6th.}\\ E & \left[\begin{array}{ll} S_2 & \textit{June the 6th.}\end{array}\right.\end{array}\right.\\ E & \left[\begin{array}{ll} S_2^2 & \textit{At what time?}\\ U_2 & \textit{At 8 pm.}\end{array}\right.\end{array}\right.$$

Another convention which is used in the new release is that a reaction is always included in an exchange. These two conventions lead to the following description of the way exchanges are embedded in a dialogue structure:

An exchange is made up of an intervention (the initiative) followed by an arbitrary number of exchanges (optional incident exchanges, the reaction, and the evaluation).

So, the possible shapes of a dialogue structure are described by the following grammar, in which $\mathcal{E}$ is a non-terminal which generates a list of exchanges:

- $\mathcal{E} \to \begin{array}{l} E \left[\begin{array}{l} act\\ \mathcal{E}\end{array}\right. \\ \mathcal{E}\end{array}$
- $\mathcal{E} \to \varepsilon$

where ε is the empty dialogue structure. Of course, this grammar is just intended to describe how exchanges are embedded in a dialogue structure. To get a grammar representing the valid dialogue structures, we parameterize the non-terminal $\mathcal{E}$ by an illocutionary function (initiative, reaction or evaluation) and by a list of predicted dialogue acts. This leads to the following grammar:

- $\mathcal{E}(\text{i}, L_i) \rightarrow \begin{array}{l} E\left[\begin{array}{l} \textit{act} \\ \mathcal{E}(\text{r, L'}) \end{array}\right. \\ \\ \mathcal{E}(\text{i}, L_i) \end{array}$

- $\mathcal{E}(\text{r, L}) \rightarrow \begin{array}{l} E\left[\begin{array}{l} \textit{act} \\ \mathcal{E}(\varepsilon) \end{array}\right. \\ \\ \mathcal{E}(\text{e, L'}) \end{array}$

- $\mathcal{E}(\text{e, L}) \rightarrow \begin{array}{l} E\left[\begin{array}{l} \textit{act} \\ \mathcal{E}(\text{r, L'}) \end{array}\right. \\ \\ \mathcal{E}(\varepsilon) \end{array}$

- $\mathcal{E}(\varepsilon) \rightarrow \varepsilon$

In these rules, L_i is the list of all possible initiatives (for both system and user), ε denotes the empty dialogue structure, *act* is the dialogue act uttered by one of the speakers and L' is the list of dialogue acts which are paired with *act*. i, r and e denotes the three illocutionary functions.

A dynamic dialogue structure is a production of the above grammar which contains terminal symbols (like D, E, or a dialogue act label), and non-terminal symbols ($\mathcal{E}$ parameterized by a list of predicted acts, a context and an illocutionary function). Therefore, a dynamic dialogue structure contains all the information contained in regular dialogue structures (this is the information carried by terminal symbols), plus information on the possible continuations of the dialogue (carried by the non-terminal symbols). The main advantage of dynamic dialogue structures is that they are updated by a simple recursive spanning: each time a non-terminal symbol is encountered, the system checks if a predicted act attached to this non-terminal can be selected. If so, the act is selected, and the non-terminal is re-written, using the grammar rules given above.

The following is an example illustrating how dynamic dialogue structures are used and how they are updated at each speech turn. Before the beginning of the dialogue, the dynamic dialogue structure is:

$D\left[\mathcal{E}(\text{i}, L_i)\right.$

This means that a dialogue is about to begin and that this dialogue will be made up of exchanges. Then, the DM receives a request from the task module which ask for the departure date. This request is stored in the DM memory and, in generation cycle, selects the act *open_request*. This act being paired with *wh_answer*, the dialogue structure becomes:

$$D\left[\begin{array}{ll} E & \left[\begin{array}{l} \textit{On what date do you want to leave?} \\ \mathcal{E}(\text{r}, (\ldots, \textit{wh\_answer}, \ldots)) \end{array}\right. \\ \mathcal{E}(\text{i}, L_i) & \end{array}\right.$$

When the user answers, the semantic representation of the user's input is sent to the belief module for anchoring. The anchoring context is departure_date and the surface form of the user's utterance is compatible with a *wh_answer* act. Therefore, *wh_answer* is selected and, being paired with *confirm*, the dialogue structure becomes:

$$D\left[\begin{array}{ll} E & \left[\begin{array}{l} \textit{On what date do you want to leave?} \\ E \left[\begin{array}{l} \textit{June the 6th.} \\ \mathcal{E}(\varepsilon) \end{array}\right. \\ \mathcal{E}(\text{e}, (\ldots, \textit{confirm}, \ldots)) \end{array}\right. \\ \mathcal{E}(\text{i}, L_i) & \end{array}\right.$$

The departure date is sent to the task module, which in return sends a new request_parameter to the dialogue module, asking for the departure time.

When recursively spanning the dialogue structure, the first non-terminal encountered is $\mathcal{E}$(e, (..., *confirm*, ...)). The act *confirm* has no preconditions (apart from the implicit precondition which is to have been predicted). So *confirm* is produced. Then, the next non-terminal encountered is the prediction on initiatives. It can be re-written since an *open_request* about the departure time is selected. Hence, the resulting dialogue structure is as follow:

$$D\left[\begin{array}{ll} E & \left[\begin{array}{l} \textit{On what date do you want to leave?} \\ E \left[\begin{array}{l} \textit{June the 6th.} \end{array}\right. \\ E \left[\begin{array}{l} \textit{June the 6th.} \\ \mathcal{E}(\text{r}, (\ldots)) \end{array}\right. \\ \mathcal{E}(\varepsilon) \end{array}\right. \\ E & \left[\begin{array}{l} \textit{At what time ?} \\ \mathcal{E}(\text{r}, (\ldots, \textit{inform}, \ldots)) \end{array}\right. \\ \mathcal{E}(\text{i}, L_i) & \end{array}\right.$$

It is important to notice that the information carried by the terminal symbols is never used in this algorithm. We just use the non-terminal symbols, i.e. the current predictions. This means that it is possible to use a reduced structure which contains only the current predictions at any time. This possibility has been implemented in the dm, which can handle both kind of structures: complete dynamic dialogue structures (terminal symbols plus non-terminal symbols) and reduced dynamic dialogue structures (simply made of non-terminal symbols, i.e. predictions).

3.1.1 Strategy switching

The system must be able to support several strategies. It is currently able to do so. Three confirmation strategies are available:

- confirmation alone for a bunch of parameters

  This strategy is illustrated in the following example.

S_1	*Hello. Can I help you ?*
U_1	*I want to reserve a Paris-London.*
S_2	S_2^1 *Paris-London.*
	S_2^2 *On what date ?*

- confirmation for a bunch of parameters plus initiative

  as in:

S_1	*Hello. Can I help you ?*
U_1	*I want to reserve a Paris-London.*
S_2	*You want to travel from Paris to London ?*
U_2	*Yes, that's correct.*
S_3	*On what date do you want to leave?*

- confirmation parameter by parameter, and then initiative

  as in:

S_1	*Hello. Can I help you?*
U_1	*I'd like to reserve a Paris-London.*
S_2	*You want to leave from Paris ?*
U_2	*Yes.*
S_3	*You want to go to London ?*
U_3	*Yes.*
S_4	*On what date do you want to leave?*

The behaviour of the system when detecting repeated communication failures also defines a strategy. We shall see in next section that several strategies are available in that case.

In the current version of the dialogue module, a declarative knowledge database describes all user and system dialogue acts (their labels, preconditions, effects, etc). Each particular strategy is associated with such a database. This means that changing the strategy consists of changing the file which declares dialogue acts. Of course, this implies quite a lot of redundancy between these files. We shall see in section 2 what could be done to obtain a more elegant solution.

3.1.2 Robustness issues

Some work has been done in order to provide a robust DM. To deal with communication failures, two approaches must be developed concurrently.

- First, if the parser or the recognition module fails to produce data, or if the data is corrupt, the dialogue manager must handle this failure by presenting an appropriate behaviour.
- Second, the DM must also try to prevent failure. If it seems that communication is difficult, the dialogue must be progressively degraded toward a more constrained and rigid behaviour.

The DM tries to meet these two requirements.

When the parser is unable to parse the input lattice, a message is sent to the LI to signal the failure. This message is then passed to the DM, which produces a request for repetition.

The DM also decides when to trigger more rigid sub-dialogues if repeated communication failures occur. If a task parameter (or several) is disconfirmed several times by the user, as in the following dialogue:

S_1	*From where do you want to leave ?*
U_1	*From Paris.*
S_2	S_2^1 *From Paris.*
	S_2^2 *On what date ?*
U_2	*No, from Bari.*
S_3	*From Bari ?*
U_3	*No, Perros*
S_3	*Could you spell the departure city, please ?*

The DM decides to enter a special mode, in order to solve the problem. In the example above, the special mode is a spelling mode. The spelling mode is handled by a module (which may be located in the recognizer) which is not a part of the dialogue manager and which does not have any dialogue capabilities of its own. The two most obvious special modes are a spelling mode (for names), and a mode which asks the user to provide a number digit by digit.

3.2 Objectives for the final system

By carefully examining the algorithms which are currently used in the dialogue module, we have found that the module should be improved in several directions. Two major improvements deserve priority:

- to suppress the use of dynamic dialogue structure, by enriching the description of dialogue acts,
- to separate clearly the declaration of dialogue acts and the rules specifying how these acts are used.

In addition to their theoretical interest, these changes will lead to a module easier to understand and to maintain. These changes are described in more details in the next two sections.

The last section describes the list of FEP messages that will have to be handled by the dialogue module in the final version of the system.

3.2.1 Algorithms

It has been seen that the current algorithm can only use non-terminal symbols i.e. predictions, since during the recursive spanning of the dialogue structure, one uses only these non-terminal symbols. To be selected, a dialogue act must:

- be present in at least one of the non-terminal symbols contained in the dynamic dialogue structure,
- have all its preconditions fulfilled.

This means that, in the current algorithm, an implicit precondition for all dialogue acts consists of being predicted. This leads to the idea that, if this implicit precondition is made explicit, the predictions contained in the dynamic dialogue structure are no longer needed. It is therefore possible to use an algorithm producing exactly the same behaviour as the current one which will make no use of dialogue structures of any kind. To do so, one must add one precondition per act in the declaration of dialogue acts, and remove the part of the code which manages the dynamic dialogue structure and its recursive spanning.

The final version of the dialogue module will be designed in this way. The next section, about strategy switching, will lead to another specification.

3.2.2 Strategy switching

We have seen that the dialogue module is currently able to support several strategies (3 confirmation strategies plus strategies to trigger special modes). At present, each strategy corresponds to a declarative knowledge base to describe dialogue acts. This leads to quite a lot of redundancy in the code.

In the next version, dialogue acts will be described in a single declarative knowledge base, whatever the strategy. Each particular strategy will be described by a set of inference rules. At the end of each generation cycle or interpretation cycle, the inference rules are triggered. They modify the dialogue module memory in order to permit or forbid the production of certain dialogue acts in the next cycle. That way, they implement the different strategies in a much more compact way that it is done now and provide a fairly high level of genericity and accuracy.

3.2.3 Robustness issues

Several messages can or are planned to be sent by the recognizer. These messages are not currently handled by the DM. They will be taken into account in the next version. These messages are:

- too_loud.

  In this case, the appropriate behaviour is to answer: “Could you repeat that a little more quietly please?”

- too_soft.

  In this case, the appropriate behaviour is to answer: “Could you repeat that a little louder please?”

- timeout

This is when a prespecified timeout has elapsed from the start of the recording and no speech has been detected. Currently this is set to 1 second in the UK system, but it can be anything we like. If such a case occurs, the system should answer something like: "I'm sorry. I didn't hear anything. Please speak after the tone."

- outside_window

  this is when the speaker is talking either over the start of the recording or over the end of the recording (or both!). It would be nice if this could be broken down into two messages (such as 'too_soon' and 'too_late') but a single message is all that we are being offered at the moment. An appropriate response could be: "Sorry. I tend to miss what you say if you speak before the tone or if your answer is too long. Please try again."

- hangup

  This is not yet implemented by the FEP. The system must be reset.

All these messages will be taken into account in the final release of the DM.

Chapter 4

The Belief Module

The Belief Module is the sub-module of the dialogue manager that handles dialogue semantics in the broadest sense (i.e. it interprets the parts of utterances that can not be handled by linguistic semantics alone) it manages the different states of knowledge of the system and the user and it translates user's references to objects into descriptions the application system interface (Task Module) can deal with. It also builds descriptions of the semantic objects that the system has to talk about.

This chapter has three subparts. In the first, we outline the Belief Module system as it was demonstrated in the review of October, 1991 (in the following, referred to as the 'current system'). We then give an evaluation of the current system with respect to the following criteria:

- core functionality
- additional functionality
- computational behaviour
- code and architecture

Finally, we give a brief overview of the conclusions reached.

4.1 The current system

In the following, we give an overview of the current system. We will first describe the tasks of the belief module in the dialogue manager, then the functionality of the current module that fulfills the most important of these tasks, and some implementation details of this.

4.1.1 Tasks of the Belief Module in Dialogue Management

The core task of the Belief Module in the Dialogue Manager is the creation, maintenance and management of semantic descriptions of the objects that are being referred to in the dialogue by both partners in coherent belief states[1] for both dialogue partners (whence the name). We call this task area *Dialogue Semantics*.

Semantic descriptions of objects in user utterances that come from the Linguistic Processor are treated as instantiating parts of the object hierarchy in the Belief Module. As these descriptions are hypothetical, these instantiations have to be grouped into coherent (i.e. non-contradictory) wholes, or belief states (see above), internally called ‘views’, that represent alternative possibilities of the current world of objects that the dialogue is about. The object instances are also checked to see whether they immediately represent relevant material for the Task Module, or whether such material can be deduced from one coherent group of objects. In either case, a task-oriented description of the material is built. Thus, the Belief Module fulfills the task of translating surface-oriented object descriptions into task parameters.

Likewise, the Belief Module is an active part in the interpretation of user utterances. The user’s semantic descriptions can be underspecified. This can be the case with diectic (temporal and spatial) (e.g. ‘the last flight to Paris’, ‘a flight from there to Berlin’), anaphoric (e.g. ‘does it call at Dusseldorf’) and elliptic (e.g. ‘to Rome’) expressions, as they are particularly common in spoken language in general as well as in our application domains. In the course of the instantiation the Belief Module tries to construct a fully specified description not only using the world knowledge and the knowledge of the belief states it has itself, but also by referring to the Linguistic History maintained and managed by the Linguistic Interface Module. Where the construction of full descriptions is not possible, as in the case of temporal diectic expressions, because the knowledge needed to this is only available via a query to the application system, the Belief Module passes on this part of the semantic description of the user utterance as uninterpreted. It is one of the main features of the SIL language that it allows a mixture of

[1]We do not refer to these states as *knowledge* states, because due to the hypothetical nature of the semantic description of the user utterance (in turn due to the hypothetical nature of the word recognition of the input signal), the system cannot be sure of the user’s state of knowledge, i.e. it can not know, but only believe. Accordingly, the system’s reaction, based on this hypothetical understanding, manifests the system’s belief of what its knowledge state should be like.

different layers of interpretation to be built into one and the same expression, thus allowing the different modules and sub-modules of the SUNDIAL system pass on information until it can be fully interpreted.

In terms of the interaction of the Belief Module with the other sub-modules inside the Dialogue Manager, this task area is described by the following external functions:

- The Belief Module receives via the Linguistic Interface and the Dialogue Module the semantic description of the user utterance
- It separates the parts of this description according to propositional content(s) and dialogue function(s)
- It updates its knowledge base according to the propositional content and sends back to the Dialogue Module the dialogue function markers, the resulting state markers of the knowledge base (i.e. it indicates whether the update resulted in an extension of the current view, the creation of a contradictory view etc.) and (if any) the task parameters contained in or deducible from the knowledge base after the update
- It constructs SIL descriptions of task objects which are sent to the Message Planner and used in the generation of system utterances.

We now look at the implementation of the Belief Module and give an overview of how the functionality described above is realized in the current system.

4.1.2 Implementation of the current system

The three major requirements of the implementation environment are:

- Portability across the sites of all of the work groups in the project, as the fundamental prerequisite for the realization of distributed software development
- Easy understandability and changeability of the resulting code
- Rapid prototyping capability.

Also, as the Dialogue Manager as a whole has to be task (i.e. application) independent as well as language independent, every one of its sub-modules also has to be independent in both these directions. In the Belief Module this independence is achieved by initializing it with customized knowledge descriptions and definitions.

The core of the Belief Module is the knowledge base, built in the object oriented library package NOOP, also developed in SUNDIAL. The object hierarchy that is the initial content of the knowledge base is built using a SIL definitions file that is in turn constructed automatically from core SIL definitions plus a customization definitions file, which contains the information and path equations needed for each different application of the overall system. At initialization, the Belief Module constructs a world of objects consisting of this hierarchy and some rules tied to the objects by daemon mechanisms.

In the interpretation phase, the Belief Module takes sub-parts of the SIL description of the user's utterance and searches through the hierarchy for possible 'anchoring' places for these parts, i.e. the terminal or non-terminal nodes of the world graph that allow an instantiation with the value of the Sil description. Then, it sees whether such an instantiation at this point already exists and it creates the respective instances indexed with the respective view identifier.

At the creation of the instances, the demon-activated rules fire and create the path links from the user oriented semantic part of the hierarchy to the task oriented part (if any). If there is no current task, i.e. user's informational goal of the dialogue, the Belief Module tries to identify this using the constellation and nature of the objects mentioned in the user utterance. These transition or translation processes are the core interpretative task of the Belief Module. As the result of this anchoring stage, it returns the task type, the task parameters and their values, and the type of instance it created for the respective object, e.g. addition, repetition, contradiction etc.

As the Task Module finds either resolutions for the hitherto uninterpreted Sil or is able to get from the application system information that could be found or deduced using the user provided parameters, it passes information to the Belief Module. The Belief Module in turn extends the world model by instantiating the appropriate objects with the values it gets from the Task Module, typing the respective system belief views accordingly, e.g. received from application system (the normal case), inferred, constraint-relaxed etc.

In the generation phase, the Message Planner requests from the Belief Module SIL descriptions of the objects the Dialogue Module wants the system to talk about to the user. The Belief Module then searches for these objects and, going upwards in the SIL hierarchy, it builds the SIL description of these objects, which are then sent to the Message Generator.

So, the Belief Module implementation realizes the core functionality as a store and retrieve mechanism of objects in the world of discourse, their properties, their state of reliability and their relation to other objects and their explicit semantic description.

4.2 Evaluation of the current system

The Belief Module demonstrated at the October 1991 review implemented the basic functionality of this module. This means that it was capable of executing the core tasks of the Belief Module that are required for relatively simple information dialogues in the context of the overall architectural distribution of tasks in the Dialogue Manager. We will first evaluate how these task are performed. Some additional functionality has been built into the current system to explore and exemplify the possibilities of the current internal Belief Module architecture. We give an evaluation of this functionality in the next section. We then proceed to give an evaluative account of the computational behaviour of the current system, the strengths and weaknesses of the code, and, in connection with this, we show what light is thrown on the general architecture of the Belief Module by the experiences we have gathered with it to date.

4.2.1 Evaluation of core functionality

The Belief Module meets the requirements of the implementation environment. It is portable across all sites and, thanks to its customizability, it is language and task independent.

The core tasks of the Belief Module are all fulfilled by the current system. For a range of test dialogues, the anchoring and interpretation phase, the task-connected phase and the generation phase have been tested in conjunction with the integrated Dialogue Manager and have shown consistently correct behaviour of the Belief Module. For admissible SIL input, it finds the appropriate interpretation, it handles correctly the information it gets from the Task Module, and outputs the appropriate SIL descriptions of the objects to the Message Planner.

4.2.2 Evaluation of computational behaviour

The core functionality of the Belief Module makes extensive use of the object management facilities provided by NOOP. NOOP is written in Prolog, and makes considerable use of the ‘assert’ and ‘retract’ predicates that permanently modify the Prolog database. These predicates are notoriously slow, because they have to update Prolog's internal dynamic memory management.

The search the Belief Module has to perform through the semantic description graph across all possible world states has to be by necessity exhaustive, at least when is has to prove the non-existence of certain objects, and thus is failure driven, leading to a considerable amount of backtracking in Prolog.

The present lack of efficiency in the BM interpretation routines can be traced to suboptimal search. Currently structure-building—creating new objects, building new links, and pursuing the (forward chaining) inferential implications—is inextricably interleaved with search. The following reimplementation steps are therefore necessary:

1. Postpone 'anchoring' (binding of SIL identifiers to nodes in the BM memory) as long as possible;
2. Likewise, postpone role assignment and inference actions (these in fact are logically dependent on prior anchoring);
3. Postpone decisions to assume modified views as long as possible;
4. Make use of more heuristics in determining anchorings; one such heuristic, based on the notion of *rigid designators*, has already been implemented.
5. Drop the explicit representation of queries in the BM.

The reimplementation will be a partial one, affecting only the interpretation component, and not the book-keeping or description functions. The additional step of moving from forward chaining inference to another form may or may not be necessary. A reasonable hunch is that by postponing structure building (and hence inference) until after most 'read-only' search is complete, there will be little need to backtrack through inference steps.

This reimplementation is particularly necessary, in view of the added inferential demands that will appear when the knowledge definition language SIL is upgraded to handle temporal inferences. A detailed design of the revised algorithm is already and in place; it is hoped that implementation will be completed by the end of July 1992.

4.2.2.1 Evaluation of code and architecture

The Prolog code of the Belief Module makes use of the modularization facility of Quintus Prolog. The Belief Module is realized as a number of files, each of which represents a Prolog module. This technique avoids name clashes between predicates used in different modules and allows a neat separation of functionality. It is also important for the use of the rapid prototyping programming style. The modular construction clearly reflects the internal architecture of the Belief Module. It can easily be seen which module is responsible for what. This should allow a straightforward implementation of the additional interpretative functionality.

The code of the Belief Module is very clear and it is relatively easy to understand. Consequently, it is also relatively easy to change and, as the single module files can be reconsulted, changes in the Belief Module proper can be easily tested. Some debugging and display facilities have been built to support the use of the NOOP object management module. There are facilities which will allow the user to:

1. display an arbitrary part of the semantic network, in a given view, either pretty printed or in Latex
2. display the current inheritance tree for views, with the currently active view marked
3. display the current state of the accessibility history.

It would of course be beneficial to have more debugging and browsing facilities. The main advantage of the object oriented approach is the clear separation between processing and the knowledge that is processed. This makes it possible to initialized the Belief Module with different world descriptions for different languages and applications.

These properties of the code, taken together, make the current implementation of the Belief Module a very flexible piece of software. It is an excellent basis for the enlargement and enhancement of its functionality.

4.3 Conclusion

We have shown the core tasks of the Belief Module and the way the functionality fulfilling these tasks is realized in the current system. We have evaluated different aspects of this implementation. The implementation is good, meets the need defined for the current system and is a sound basis for the ongoing and future work on the Belief Module. The only major problem area is the speed of the system. We proposed two straighforward ways to overcome this problem, though possibly none of these can be realized within the remaining duration of the SUNDIAL project.

Chapter 5

The Task Module

5.1 Overview

This section describes the objectives, the new functionalities and architecture of the task module (TM) that are to be implemented in the final SUNDIAL system prototype.

It is based on an analysis of the current version of TM. The specifications provided in the deliverable WP6D1 and the description presented in the WP6D2 and the documents from WP6 team (Description of the Erlangen TM (Nov. 91), analysis of the TM (Jan. 92)) have been considered.

The weaknesses and omissions of the current version of the TM will be discussed and specifications for the final prototype will be stated. The set of functionalities described in WP6D1 will not be discussed in this section.

5.2 TM in the initial implementation

WP6D2 provides a detailed description of the implementation of the TM of the first, October 1991 prototype. It describes how a large number of the functionalities detailed in WP6D1 have been implemented. A discussion follows of the weaknesses and omissions of the TM that will be overcome in the design of the final version.

Most of the following remarks arose from the constraints of the realization of the first prototype: the need for a quick response, robustness, preserving the generic aspects of the TM and "toy" applications. These remarks are classified under three headings: the weaknesses, the incomplete points and the points not addressed.

5.2.1 Weaknesses

1. Knowledge
   The TM's functionalities require an extensive knowledge of the application: knowledge of both the data (input/output parameters) and of the application tasks to be performed. This knowledge has different forms; for example, details about data type and structure as well as the relationships between tasks, need to be known by the TM.

   The declaration of knowledge about data is made using a large number of predicates and low level facts which are distributed in several parts of the TM. Although it seems very easy in principle to customize the TM for a new application, some redundancy and coherency problems arise very quickly. The same type of problem is met in the exploitation of the knowledge. The possibilities for declaring data types (especially the domains) are very limited.

2. The TM-DM and TM-BM interfaces
   Many low level messages between the TM and its environment are declared. Some of these messages appear to be redundant or superfluous. On the other hand, messages for announcing failures are not present.

5.2.2 Incomplete aspects

Some of the implemented functionalities appear to be incomplete or too restricted.

1. Parameter relaxation
   Parameter relaxation is used when there is no response in the database to the initial user request. Parameter values are transformed in order to try to obtain an non-empty response from the database. The current algorithms deal with only one parameter at a time and cannot deal with parameters which have discrete values.

2. Declaring knowledge about data
   It seems that the declaration of knowledge about structured data (for example, a date structured into day, month and year) is necessary in the system but, at present, this is not possible.

3. Task formulation (e.g. to find the answer to the query "What is the price of a ticket from London to Paris?")
   The scope of task formulation messages is still under-developed, being currently restricted to simple queries, and needs to be extended.

4. Subtasks and task parallelism are catered for, but they are not found in the corpora.

5.2.3 Points not addressed

1. Application
   Some points in relation to the application have not been dealt with and have been postponed.

   - Applications handled
     For the first prototype, the application has been considered as a "toy" data base with the following characteristics:
     - only a few data items are included, each represented by a collection of PROLOG facts
     - only one relation (universal relation) is present.
   - points not handled
     The following problems arising from real applications have not been considered:

     the access to the application may take a lot of time, so the number of accesses must be optimized.

     interfacing with a variety of databases (and the subsequent genericity problem for the TM)

     real applications which contain more than one relation

     exploitation of all the interesting information contained in the application

2. Summarization
   The case where the application provides a large number of responses has not yet been considered: the functionality to summarize a large set of responses has not been implemented.

5.3 Specifications for the final version

5.3.1 Goals and constraints

The main goal of the final version is to overcome the weaknesses and omissions of the first version. Some constraints need to be taken into account. This new version must preserve the generic character of the TM and ensure good performance. The TM must be adapted to the functionalities of the other modules and submodules of the system. In particular, the information provided by the TM needs to influence the dialogue and also needs to be used by the other modules.

5.3.2 New architecture and implementation

1. Data and task description
   Declaration of knowledge about data and task will be done using a minimal set of predicates. The "daemon" concept will be used to automatically trigger some processes on data (instead of the designer stating explicitly what needs activating). For example, declaring a default value for a data item in the global description of the application would be sufficient for the TM to manage the installation of the default value (to await user confirmation) as soon as the data is evoked[1].
   The application may contain one relation, or several relations and structured data.

2. Interface between the TM and the rest of the system
   The number of messages will be reduced by having a better definition of both their content and their meaning.

3. Interfacing the application
   In order to use the TM in several real applications and to take into account the performance constraints, it is necessary to design a special interface module (the Database Interface, DBI) between the TM and the application (fig. 5.1).

   The role of the DBI is to take into account the requests from the TM, to transform them, to manage the communication with the application (open the session, send if necessary a DB query, accept the response from the application) and transmit the results to the TM. The DBI has to take into account performance considerations, for example, limiting the access to the database.

[1]The current version does possess this specification.

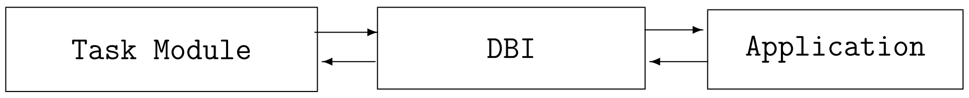

Figure 5.1: Interfacing the TM and the application

The communication between TM and DBI will use a minimal set of messages:

1) tm–>DBI

> **accessdb(+open)**
> **accessdb(+close)**
> **accessdb(+enquire, +ParamValpairs)**
> **accessdb(+update, +ParamValpairs)**
> **accessdb(+next/solution)**
> **accessdb(+prev/solution)**
>
> ParamValpairs could be a list, example:
> [depplace: Paris, destplace: Lannion, time: [0900,1200]]

2) DBI– > TM

> **resultdb(+open,+Result)**
> **resultdb(+close,+Result)**
> **resultdb(+enquire, +Results, +Comment)**
> **resultdb(+next/prev, +Results, +Comment)**
> **resultdb(+update, +Comment)**
>
> +Results is a list (possibly empty) of solutions. The parameters of each solution could be different from those of the ParamValPairs list, so the TM has to interpret these parameters.
> If the DB finds too many solutions (on which the summarization algorithm cannot apply), no solution is sent and +Comment states this fact. + Comment could also encode some extra messages from DB such as "no flight earlier" from the OAG database.

Some other messages will be added to this minimal set following the specificity of the applications. Each national demonstrator will be in charge of its proper DBI.

5.3.3 Expanding functionalities

1. Constraints relaxation
   The main goal is to provide a larger set of algorithms for relaxing constraints. We have to take into account two facts: (1) the application in some cases can provide some interesting information concerning the relaxation ; (2) the application in other cases (for example the German application) automatically carries out some relaxation.

   The new algorithms will:

   - relax the temporal parameters in different ways. However some applications automatically relax (without control or explanation) the parameter *hour*.
   - relax some parameters which have discrete values (for example, place: town, airport or train station). Knowledge for this type of relaxation will be contained either in the TM or in the application. In the latter case, interface messages between the TM and the DBI will be added: other knowledge (for example, the relative weights of two values) may be needed.
   - relax two parameters at the same time. When relaxation of a single parameter is insufficient, it may be preferable from an ergonomic and dialogue point of view to negotiate two parameters at the same time rather than in sequence. Thus a new algorithm dealing with this type of relaxation will be proposed.

2. Summarization
   Summarization is needed because providing the user with a very long list of items is not ergonomic and leads the user to behave in an undesirable manner. Several methods can be used to solve the problem of too many responses: some of them are concerned with dialogue strategies, others with the TM. We will propose a set of heuristics dealing with the summarization problem. The following algorithms will be put into the TM:

   - to choose the first or the last response in the list (following the specification of WP3D1 chap. 4). In this case, it would be also necessary to provide the user with some explanation about the choice made by the system.
   - to find a splitting criteria (for example the carrier, AF or BA, could be one of these criteria) to build up subsets. Then the DM will have to propose a choice to the user.

- to compute, if it exists, some regularity within the set of responses and to announce it to the user (for example *there is a flight each hour from 8am*).

Chapter 6

The Linguistic Interface

6.1 Overview of functionality

The linguistic interface module manages a bidirectional interface with the parser; it is customisable to deal with particular parser input. It transmits predictions generated by the dialogue module, and stores the results of parsing in the linguistic history, at the same time passing on the semantics for further interpretation. The linguistic history also receives copies of analysis trees produced by the generator; it thus serves as a database of all linguistic forms produced during the course of the dialogue.

6.2 Limitations of previous version

6.2.1 The parser interface

In previous versions, the parser interface has been limited to transmitting predictions and receiving input. It has not been capable of conveying information about the state of the dialogue, to the parser and recognition components. This is important, (i) when recognition is repeatedly unsuccessful, and a new strategy (using lower-level units such as single words or spellings) becomes necessary; (ii) at those times when it is necessary to start, stop, or suspend the conversation. Moreover, for increased sensitivity of dialogue management, it is important for the reason for any failure, when it is known, to be signalled. Thus, when recognition has failed because the caller spoke too softly, the dialogue manager can be informed; it can then request that the caller speaks louder.

6.2.2 The linguistic history

The previous form of the linguistic history does not allow for easy access of subtrees by semantic features. This is because each entry has its semantics recorded only once, at the root. Moreover, the structure of the analysis tree is different from that used internally by the generator, so that a certain amount of time-consuming rewriting is required.

6.2.3 The form of predictions

The form of predictions previously defined is based on a somewhat coarse granularity, where semantic objects of type *date, time, flight_number* are predicted. This does not permit interactive dialogue about objects at a finer level, for example: *month* or *day* in the case of *date.* The coarseness of representation is not primarily the responsibility of the LI; for this change to come about, the definition of task-oriented SIL needs to be refined.

The parser has not been provided with a linguistic context, which would enable it to detect whether a previous form was being re-used, or even to use the existence of a previous echoed form during its search.

6.3 Modifications carried out in current LI

6.3.1 Interfaces

The parser interface has been adapted:

1. Initialisation values of parser flags are now exported when the parser is loaded;
2. to send start and stop signals to the parser and speech recognition components;
3. to return a status flag;
4. to handle *special modes*, such as spelling or single-word mode;
5. to include information from the linguistic history with predictions

The definition of the prolog interface with the parser broadly follows that proposed in Thornton (1992). It can be described as follows:

$$\begin{array}{lcl} Interface & \equiv & dm\_lp\_dm(CommandLabel, Out, In, Status) \\ CommandLabel & \equiv & wait\_for\_call|end\_call|run\_parser|special\_mode \end{array}$$

$$\begin{array}{llcl} wait\_for\_call, end\_call: & Out, In & = & \text{UNDEF} \\ & Status & = & success|failure \\ run\_parser: & Out & \equiv & Predictions \times Atrees \\ & In & \equiv & UFO \\ & Status & \equiv & success|\langle parser\_error\rangle|\langle recogniser\_error\rangle \\ special\_mode & Out & \equiv & Sm\_command \times Sem\_type \\ & In & \equiv & UFO \end{array}$$

$$\begin{array}{lcl} Predictions & \equiv & \mathrm{seq}\, Prediction \\ Prediction & \equiv & \\ Sm\_command & \equiv & spelling|word|\ldots \\ Sem\_type & \equiv & city|hour|minutes|number|\ldots \end{array}$$

UFO (utterance field object) and *Atrees* (analysis trees) are described in Section 6.3.2.

6.3.2 Linguistic history

The datastructure passed from the parser to the LI has changed in the following ways:

- A complex ufo (corresponding to contiguous phrasal solutions which are unrelated grammatically, but may be related discoursally) takes the form of a list of simple ufos, in order of mention.
- a simple ufo is structured as follows:

$$UFO \equiv \begin{bmatrix} id: \ldots \\ dialogue: \ldots \\ string: String \\ atree: Atree \end{bmatrix}$$

  Currently the values of *String* and *Atree* are returned by the parser; *id* is assigned dynamically by the LI;

- An analysis tree is based on the definition of *Sign*:

$$
\begin{array}{lcl}
Sign & \equiv & \left[\begin{array}{l} mor : Mor \\ syn : Syn \\ sem : Sem \\ order : Order \\ score : Score \end{array}\right] \\
Mor & \equiv & \left[\begin{array}{l} form : MorForm \\ root : MorRoot \end{array}\right] \\
Syn & \equiv & \left[\begin{array}{l} head : SynHead \\ args : Args \end{array}\right] \\
SynHead & \equiv & \left[\begin{array}{l} major : (v|n|prep|det|\ldots) \\ \ldots \end{array}\right] \\
Args & \equiv & \mathrm{seq}\, Sign \\
Order & \equiv & \left[\begin{array}{l} dir : (pre|post) \\ adj : (next|notnext) \end{array}\right]
\end{array}
$$

  $MorForm$ is the inflected form of the word (or phrase, in the case of a phrasal entry) introduced in the lexicon. *MorRoot*, if present, is the uninflected lexical headword. *SynHead* contains at least the major category of the node; it may optionally contain other (language-specific) head features. *Args* corresponds to the feature indicating valency or subcategorisation in UCG/HPSG. Here it serves the dual purpose of pointing to subtrees in the analysis tree. Argument positions which are not *saturated* by the presence of some subtree corresponding to a non-null substring of the utterance, but which are required as placeholders, may be marked [*null* : +]. This is currently the case with analysis trees produced by the generator, where the valency of nodes of type *sign* is dictated exactly by the corresponding lexical entries. It may not be the case for parser analysis trees.

  Heads in analysis trees are coalesced with the corresponding phrasal nodes, thus achieving economy of representation. The resulting structure is similar to that generated by a dependency grammar. Substrings corresponding to nodes in the tree may be read out, by taking those *Args* marked [*order* : *dir* : *pre*] first, then the (inflected) head, then the *Args* marked [*order* : *dir* : *post*]. Order must also respect the *adj* constraint. *Ceteribus paribus*, it defaults to the order of constituents in the analysis tree. There is redundancy between the string so generated for a complete analysis tree, and the *string* attribute of the enclosing ufo. This is because currently only linguistic generators are capable of returning analysis trees. Template generators return strings; their *Atree* is empty.

Figures 6.1 and 6.2 compare the old and new forms of simple ufo. For downward compatibility, the binary valued flag **old_analysis_tree** is available. The

$$
\left[\begin{array}{ll}
\mathit{syntax}: & \left[\begin{array}{l}
\mathit{id}: \mathit{want}1 \\
\mathit{category}: v \\
\mathit{string}: \mathit{i\ want\ to\ travel\ to\ london\ on\ the\ twentieth\ of\ june} \\
\mathit{subtrees}: \left\langle \begin{array}{l}
\left[\begin{array}{l} \mathit{id}: \mathit{caller} \\ \mathit{category}: n \\ \mathit{string}: i \end{array}\right] \\
\left[\begin{array}{l} \mathit{id}: \mathit{want}1 \\ \mathit{category}: v \\ \ldots \end{array}\right] \\
\left[\begin{array}{l} \mathit{id}: \mathit{go}1 \\ \mathit{category}: v \\ \mathit{string}: \mathit{to} \end{array}\right] \\
\left[\begin{array}{l} \mathit{id}: \mathit{go}1 \\ \mathit{category}: v \\ \mathit{string}: \mathit{travel\ to\ london\ on\ the\ twentieth\ of\ june} \\ \mathit{subtrees}: \left\langle \begin{array}{l} \mathit{to\ london} :: \mathit{prep} :: \mathbf{location} \\ \mathit{on\ the\ twentieth\ of\ june} :: \mathit{prep} :: \mathbf{s\_date} \end{array}\right\rangle \end{array}\right]
\end{array}\right\rangle
\end{array}\right] \\
\mathit{semantics}: & \left[\begin{array}{l}
\mathit{type}: \mathbf{want} \\
\mathit{theagent}: \left[\begin{array}{l} \mathit{type}: \mathbf{individual} \\ \mathit{id}: \mathit{caller} \end{array}\right] \\
\mathit{thetheme}: \left[\begin{array}{l} \mathit{type}: \mathbf{go} \\ \mathit{id}: \mathit{go}1 \\ \ldots \end{array}\right] \\
\mathit{id}: \mathit{want}1
\end{array}\right]
\end{array}\right]
$$

Figure 6.1: Old form of analysis tree encoding

default is **off**. The notation $String :: Cat :: \mathbf{Id}$ is shorthand for

$$
\left[\begin{array}{l} \mathit{string}: \mathit{String} \\ \mathit{category}: \mathit{Cat} \\ \mathit{id}: \mathit{Id} \\ \mathit{subtrees}: \ldots \end{array}\right]
$$

The linguistic history can be accessed by other modules as a database. The access command can stipulate a range of entries to be accessed, together with featural templates indicating the syntactic and semantic properties of the subtree required. In addition, it is possible to specify whether all matching candidates, or only the first, are returned:

$$
\begin{array}{lcl}
\mathit{Access\_Command} & \equiv & \mathit{NumEntries} \times \mathit{SignTemplate} \times \mathit{Qualifier} \\
\mathit{Qualifier} & \equiv & \mathit{single}|\mathit{all}
\end{array}
$$

$$\left[\begin{array}{ll}
id : \ldots & \\
dialogue : \ldots & \\
\multicolumn{2}{l}{string : i\ want\ to\ travel\ to\ london\ on\ the\ twentieth\ of\ june} \\
atree : & \left[\begin{array}{ll}
syn : & \left[\begin{array}{ll}
head : & \left[\ major : v\ \right] \\
args : & \left\langle \begin{array}{l}
\left[\begin{array}{l}
mor : [form : i] \\
syn : [head : n(nom)] \\
sem : [id : caller, type : \textbf{individual} \\
order : [dir : pre]
\end{array}\right] \\
\left[\begin{array}{l}
mor : [form : to] \\
syn : [head : v] \\
order : [dir : post]
\end{array}\right] \\
\left[\begin{array}{l}
mor : [form : go] \\
syn : [head : v, args :< to\ london \ldots, on\ the\ 20th \ldots >] \\
sem : [id : go1, type : \textbf{go} \ldots] \\
order : [dir : post]
\end{array}\right]
\end{array} \right\rangle
\end{array}\right] \\
mor : & \left[\begin{array}{l}
root : want \\
form : want
\end{array}\right] \\
sem : & \left[\begin{array}{l}
id : want1 \\
type : \textbf{want}
\end{array}\right]
\end{array}\right]
\end{array}\right]$$

Figure 6.2: New forms of analysis tree encoding

Solutions are sequences of analysis trees. They are always returned by recency; *NumEntries* gives the number of entries (since the most recent) to be accessed. *SignTemplate* is a partially specified sign (for example, just the semantic type). If it is null, complete entries are returned. So with the access command

$$(3, \_, all)$$

The most recent three entries (complete) are returned.

6.4 Timetable and Plans

A major speed-up for the whole system can be achieved by moving over entirely from dag-encoded to term-encoded structures. This already happens in the case of the UK and French parsers, and is in the process of being carried out for the UK linguistic generator. When this is done, the linguistic history can be kept in the form of terms. Only the semantics of the roots of the atree component need to be interpreted. Predicates are already implemented for accessing and updating term-encoded structures. These can be unfolded for special calls, so that access and update is often simply prolog unification.

Month	Version	New Features
July	60.0	New form of analysis trees
		control messages to parser
		previous atrees to parser
		status received from parser
		initialisation flags to parser
		message planner requests fully implemented
September	70.0	term encoded analysis trees
		more generic access predicates
		internal logging, and dump commands

Chapter 7

The Message Planner

7.1 Introduction

In this chapter, we evaluate the functionality and implementation of the message planning module as described in WP6D2 (referred to as the 'current' message planning module). In this section, we provide an overview of its current functionality and implementation, before evaluating these in sections 7.2 and 7.3 respectively.

The message planning module interfaces the dialogue manager component with the message generation component in the Sundial system. The message planning module is initiated by the dialogue module which sends it a set of schematic utterance descriptions. These are transformed into the IL format used by the message generators. If a rule-based generator is being used for output, then the semantic part of the schematic description, given in task-oriented SIL, is sent to the belief module for conversion into a contextually relevant, linguistically-oriented SIL description, and this latter SIL description is sent to the message generator. With template-based generators, no conversion takes place since they operate directly on the basis of task-oriented SIL. Once the message generation component has generated the system utterance, it returns a full IL description of the system utterance to the message planner. The message planner then sends this description to the linguistic interface module, so as to update the linguistic history, and informs the dialogue module that the system utterance has been generated.

The implementation of the message planning module is based upon the distinction between algorithms and declarative knowledge. This has facilitated customization of the message planner for different different languages and generators. Declarative knowledge, such as the type of linguistically-oriented semantic description required from the belief module, are implemented as condition action rules and stored in separate rule files. The algorithms have been implemented so that the behaviour is the module can be controlled through system flags. For example, when the `mp_interface` flag is set to `template`, indicating that a template generator is used, no semantic descriptions are sent to the belief module for conversion. When this flag is sent to `rule` (for rule-based generators), the appropriate rule files are loaded, and the semantic descriptions are sent to the belief module for conversion. In addition, the message planner is capable of using language-specific stock phrases for some system utterances. If the `language` system flag is set to `english`, then the file containing English-specific stock phrases is loaded. When a schematic utterance from the dialogue module meets the conditions for one of these phrases — for example, the phrase *can you repeat that please?* is triggered by the dialogue act `demrep` — semantic conversion is by-passed and the linguistic representation of this phrase is sent directly to the generator.

7.2 Evaluation of Functionality

The message planning module displays the functionality described in WP6D2. However, it became apparent when testing and evaluating this module that additional functionalities are required. In particular, extra functionality is required to produce semantic descriptions for rule-based generators. These generators take as input a linguistically-oriented SIL descriptions and, starting with the root object, recursively build a syntactic description of the system utterance. This description is then augmented with a morphological and prosodic description. However, the semantic descriptions produced by the message planner are limited in a number of ways.

7.2.1 Propositional Attitude Information

The belief module converts a task-oriented semantic description to a linguistically-oriented description by finding a semantic type which is realized by a entry in the lexicon. It then recursively constructs a semantic description using this type which includes all the task-oriented information. For example, the message planning module may require a linguistically-oriented description of:

(7.1)

$$\begin{bmatrix} id : dbflight1 \\ type : dbflight \\ date : \_ \end{bmatrix}$$

i.e. a request for the date of the user's flight. The belief module would then produce the following linguistically-oriented description where the semantic type DEPART is realized by the lexical entry *leave*:

(7.2)

$$\begin{bmatrix} id : s\_time1 \\ type : s\_time \\ thedesc : \begin{bmatrix} id : depart1 \\ type : depart \\ thetheme : \begin{bmatrix} id : individual1 \\ type : individual \\ thediscourserole : \begin{bmatrix} id : discourserole1 \\ type : discourserole \\ value : hearer \end{bmatrix} \end{bmatrix} \\ thetime : \begin{bmatrix} id : time\_point1 \\ type : time\_point \\ thetime : \begin{bmatrix} id : s\_time1 \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

This semantic description would then be realized by the message generator as *what date do you leave?*. The problem with this utterance is that it lacks the politeness observed in dialogue corpora studies; typically, an enquiry agent would say *what date do you <u>want to</u> leave?*. That is, the propositional attitude verb *want* indicates that agent will understand the information furnished by the user — say *12th October* — as a requirement of the user rather than as definite statement of fact.

One obvious solution — augment the description routines in the belief module to include propositional attitude information — is not possible since the belief module only interprets and describe object-level rather than meta-level semantic descriptions. For example, in interpreting a user's turn such as *I want to leave next Tuesday*, the belief module strips the 'want' part from the input and only interprets *I leave next Tuesday.* Rather we have adopted the solution of adding a functionality to the message planner which augments the object-level semantic descriptions from the belief module with meta-level knowledge. In the above case, the message planning module augments the description in (7.2) by embedding the DEPART object in a WANT object as shown in (7.3):

(7.3)

$$\left[\begin{array}{ll}
\mathit{id} : \mathit{s\_time}1 & \\
\mathit{type} : \mathit{s\_time} & \\
\mathit{thedesc} : & \left[\begin{array}{ll}
\mathit{id} : \mathit{want}1 & \\
\mathit{type} : \mathit{want} & \\
\mathit{theagent} : \left[\ \mathit{id} : \mathit{individual}1\ \right] & \\
\mathit{thetheme} : & \left[\begin{array}{ll}
\mathit{id} : \mathit{depart}1 & \\
\mathit{type} : \mathit{depart} & \\
\mathit{thetheme} : & \left[\begin{array}{ll}
\mathit{id} : \mathit{individual}1 & \\
\mathit{type} : \mathit{individual} & \\
\mathit{thediscourserole} : & \left[\begin{array}{l}
\mathit{id} : \mathit{discourserole}1 \\
\mathit{type} : \mathit{discourserole} \\
\mathit{value} : \mathit{hearer}
\end{array}\right]
\end{array}\right] \\
\mathit{thetime} : & \left[\begin{array}{l}
\mathit{id} : \mathit{time\_point}1 \\
\mathit{type} : \mathit{time\_point} \\
\mathit{thetime} : \left[\ \mathit{id} : \mathit{s\_time}1\ \right]
\end{array}\right]
\end{array}\right]
\end{array}\right]
\end{array}\right]$$

A similar approach has been adopted towards existential descriptions. When the belief module introduces a single new object into the dialogue, the message planner embeds it within a existential phrase. Consequently, rather than say *a flight BA123 leaves at 4pm on the 12th October*, the system says *there is a flight BA123 which leaves at 4pm on the 12th October.*

7.2.2 Information for Prosodic Assignment and Re-Cycling Algorithms

The rule-based message generator assigns a prosodic description to an utterance on the basis of its dialogue, syntactic, and semantic descriptions. Using just this information, however, provides only a 'default' prosody for an utterance: it does not take into account the status of the semantic objects referenced in the belief model. In cases of mis-understandings, for example, the status of the arrival city object may change from being an object in a 'base' view to one in a 'modified' view:

(7.4) S: You want to go to Montreal?
U: No, to Melbourne
S: to Melbourne?
U: Yes

If this change of status information is conveyed from the belief module to the message generator, then prosodic assignment can be sensitive to the status of these object — *Melbourne* may be assigned a 'crystal clear' intonation. This information is conveyed through the information profile. Originally, the information profile contained default information supplied by the message planning rules. This has now been augmented so that the contents of the information profile are supplied by the belief module when it constructs a linguistically-oriented description for the message planner. For example, the change of status with the arrival city above can be indicated by the pair `ifocus: modf_view`.

The information profile has also been augmented to specify whether a particular semantic object has a linguistic description which can be re-cycled during message generation. In the original approach, the re-cycling algorithm operated in a 'blind' manner: the linguistic history was always searched for a matching description. In many cases, the search was not fruitful. However, since the belief module interprets user utterances and constructs semantic descriptions of the system utterances, this knowledge can be used to indicate which semantic objects in a system utterance have re-cyclicable linguistic description. For example, in constructing the semantics for the second system utterance in (7.4), the belief module can indicate to the message planner that the object *city2* has an existing linguistic description in its most recent entry (i.e. the record for *No, to Melbourne*). See the section on The linguistic Generator in WP7D3 for further details.

7.2.3 Style of Solutions

In the current dialogue manager, the style of the solution given by the system is not sensitive to the specific query made by users as well as the parameters they provide. For example, the solution given in (7.14) would be generated if the user made one of the queries in (7.5) to (7.13):

(7.5) When does BA123 leave for Paris?

(7.6) When does BA123 arrive at Paris?

(7.7) Can you confirm that BA123 arrives in Paris at 16 30?

(7.8) Does BA123 arrive in Paris at 17 30?

(7.9) Which Heathrow terminal does BA123 leave from?

(7.10)Can you check that BA123 leaves Heathrow from terminal 4?

(7.11)Can you confirm that BA123 leaves Heathrow at 15 30 from terminal 4?

(7.12)Does BA123 leave Heathrow from terminal 2?

(7.13)Can you check the departure time of BA123? I think it leaves from terminal 4.

(7.14)BA123 departs from terminal 4 London Heathrow at 15 30 and arrives at Paris charles de gaulle at 16 30.

The same solution is generated at present because all the parameters of a flight or train solution are sent from the task module to the belief module and when the belief module constructs a semantic description it uses all the parameters in the solution view.

A more appropriate response to the user's query can be generated if the functionality of the message planning modules is augmented. At present, the message planner keeps no record of what parameters have been confirmed, nor of the type of the user's query (i.e. whether it was a request for information, or for confirmation of information). If this information were retained, then the message planner could request a description of the solution from the belief module which was sensitive to the user's query. Consider for example, the following dialogue:

(7.15)U1: Can you confirm that BA123 arrives in Paris at 16 30?
S1: BA123 from London to Paris?
U2: yes
S2: you want to confirm an arrival time of 16 30?
U3: yes
S3: Yes, BA123 does arrive in Paris at 16 30

In generating S1, the message planner could retain the flight number, the departure and arrival cities as *identifier parameters* (they identify the flight in question). With S2, the arrival time would be retained as a *requested parameter* and 'confirm' would be the *query type.* Note that if the user had said *can you tell me when BA123 arrives in Paris?*, the query type would be 'know'. Prior to presentation of the solution, the following structure would be created:

identifier parameters	[goalcity:paris,sourcecity:london]
requested parameters	[goaltime:1630]
query type	confirm

When the solution is sent to the message planner for generation, this structure is used by a set of rhetorical planning rules to determine which parts of the solution view is to be described in the system utterance. In this instance, only the flightid, goalcity and goaltime are described. The returned semantic description would then be augmented with the *yes*, and the information profile augmented with the fact that this a request for confirmation of information.

In addition to the extension of the message planner, the belief module will also need to be extended to return the status of the parameters when describing them for the message planner. For example, in constructing semantic descriptions for the S1 and S2 in (7.15) above, the belief model would return the following information in the information profile of its description:

S1	ips:[goalcity,sourcecity], qt: confirm
S2	rps:[goaltime]

where the value of ips is a list of identifier parameters, and the value of rps is a list of requested parameters. The value of qt is the type of query — here a confirm of the arrival time. Since this information is currently available in the query view of the belief model, this extension should be straightforward.

7.3 Evaluation of Implementation

Since the implementation of the message planner supports the required functionality, and does so in a relatively fast and easily customizable manner, there has been no major changes in the implementation. In fact, the only changes have been to support the additional functionality described in section 7.2, and in the interface formats described below.

7.3.1 Dialogue Module Interface

With the re-implementation of the dialogue module, we have taken the opportunity of simplifying its interface with the message planner.

Previously, the dialogue module specified moves using `move/9`. This included dialogue information, such as the dialogue exchange identifiers, which is redundant in generation. Consequently, the dialogue information has been simplified to include one necessary property — the dialogue act — plus three optional properties: *rep* (repetition) whose value is the number of times the move has been repeated, *reintroduce* whose value is the number of times the moves have been re-introduced in the dialogue, and *fail* for the number of times the user failed to understand the move. Each of these optional parameters defaults to 0.

The description of semantic information has also been simplified. Whereas the previous implementation used a mixture of object identifiers, such *dbflight1*, and view identifiers, such as *view1*, to reference objects in the belief module, the former are now consistently employed.

7.3.2 Belief Module Interface

The interface with the belief module has also been simplified. Whereas the previous implementation distinguished the type of semantic description to be constructed by the belief module in the name of the predicate — for example, `request_description_object(MessageId, SemanticDescription)` and `request_description_object_in_ctx(MessageId, SemanticDescription)` — the type of description is now indicated in the second argument of the predicate. Thus all description requests are indicated the predicate `request_description(MessageId, DescriptionType, SemanticDescription, Controls)` where Controls is a list of additional constraints on the description process. Likewise, the descriptions are now all returned in the predicate `return_description(MessageId, SemanticDescription, InformationProfile)`.

7.3.3 Message Generator Interface

The only changes in this interface concern the `init` and `reset` messages. These will now take a list of arguments indicating, for example, whether a term-encoded or dag-encoded lexicon is to be used in generation.

7.4 Conclusion

Evaluation of the message planning module shows that it satisfies the functional and implementation requirements imposed by the current dialogue manager. Of course, while it might have been desirable to extend it in the manner suggested in WP6D2 — including taking responsibility for dialogic planning — these changes would involve re-design of the whole dialogue manager and, given the available resources, such a re-design would not have paid off in terms of either increased functionality or efficiency. Rather, the only major extension of the message planner will be the addition of functionality for making system solutions appropriate to user requests. This will be complete by the release in September 1992.

Chapter 8

Integrating the Dialogue Manager with the Italian demonstrator

This section describes the work which has been done by the Italian partners within WP6. At present, most of the effort has been devoted to the integration of the Dialogue Manager with the rest of the Italian Sundial system in order to meet the requirements of WP6 interface standards.

8.1 Connecting the DM with the Italian SUNDIAL Demonstrator

The problem of the physical connection of the DM with the rest of the Italian system has been approached.

The Dialogue Manager and the FEP/parser modules run on different machines, since the FEP machine does not have Prolog. We are implementing a socket based access mechanism to a server process, where the speech understanding system is the client and the server is the Dialogue Manager. At present, the basic socket interface is running and we have to address the problem of synchronizing the Prolog process and the C one.

8.2 Adapting the DM to the Italian domain

Some work was done in order to tailor the Dialogue Manager to the Italian application domain. In particular, by adapting the local SIL train definitions of the German partners, we implemented the kernel of the generator of the system answers for the Italian language.

Most of the work done on the present release of the system concerned the extension of the module that interfaces the Dialogue Manager with the Message Generator in the Sundial system, i.e. the Message Planner. A list of Prolog facts to simulate data base access results has been placed within the Task Module environment. The customization of the Message Planner concerned the language type and the rule set.

To customize the language type we simply allowed the Message Planner to try the Italian rules in building its description. The customization of the rule set consisted in adding a new set of rules for Italian.

8.3 Test Dialogues

Another activity was to create twenty dialogue test files to be sent to the Surrey central archive in order to test the overall Dialogue Manager.

These dialogues were created by collecting single sentences which are part of our domain corpus. Each dialogue was submitted to some Italian native speakers, in order to get a judgment on their naturalness. Previously these sentences were uttered by real speakers and the lattices were processed by the speech understanding system. The outputs of the understanding system were processed by the Prolog program which translates the deep semantic representations of the Italian parser into SIL representations.

The typical structure of the dialogues is constituted by an initial request by the user about the train timetables to go from a city A to a city B. This initial question may contain the specification of the hour. Also the specification of the station may or may not be present. Of course the values of these parameters may be supplied by the user after being explicitly requested by the system. When the system finds a train which satisfies the user's request, the user may ask about the particular services which are available on that train; these questions concern the presence of services such as a restaurant compartment, a sleeping compartment or the necessity of reserving the place.

8.4 Future Plans

Future plans are to complete the integration, to work on template generation and to test Version 50.1 with the Italian test dialogues. The integration has to be completed in order to implement heuristics to deal with the lack of linguistic history, since the SIL representations produced by the post-processing of the output of the Italian parser do not contain syntactic information. Finally, a further issue is the integration of the Dialogue Manager with the Italian train timetable data base. The Task Module output to the data base will be dealt with by a C program that will transform it into a query accepted by our socket based data base interface. The same program will be responsible for transmitting the result of the data base query to the Task Module.