Deliverable D3.5

Draft Proposal on Best Practice Methods and Procedures in Dialogue Management

August 1999

Esprit Long-Term Research Concerted Action No. 24823

Spoken Language Dialogue Systems and Components: Best practice in development and evaluation.

DISC

TITLE	**D3.5 Draft proposal on best practice methods and procedures in dialogue management.**
PROJECT	DISC (Esprit Long-Term Research Concerted Action No. 24823)
EDITORS	
AUTHORS	Niels Ole Bernsen, Laila Dybkjær (NIS)
ISSUE DATE	27 August 1999
DOCUMENT ID	wp3d5
VERSION	0.2
STATUS	Final
NO OF PAGES	65
WP NUMBER	3
LOCATION	
KEYWORDS	WP3, dialogue management, best practice

Document Evolution

Version	Date	Status	Notes
0.1	06/98	Draft	First draft published for review from partners.
0.2	08/98	Final	Final version.

Draft Proposal on Best Practice Methods and Procedures in Dialogue Management

Abstract

This paper presents a first draft methodology for the development and evaluation of dialogue management for spoken dialogue systems (SLDSs). Dialogue management is arguably the core functionality of SLDSs because the dialogue manager tends to have the controlling function of the system as a whole and maintains a model of the evolving dialogue context. The paper is based on DISC Working Paper D1.5 on current practice in the development and evaluation of dialogue management for SLDSs [Bernsen et al. 1998b]. D1.5 presented the results of in-depth analyses of the following dialogue manager exemplars: the Daimler-Benz Dialogue Manager [Heisterkamp and McGlashan 1996], the dialogue managers in the Danish Dialogue System [Bernsen et al. 1998a], in Railtel/ARISE [Lamel et al. 1995, den Os et al. 1999], in Verbmobil [Bub and Schwinn 1996, Alexandersson et al. 1997, http://www.dfki.uni-sb.de/verbmobil/], and in Waxholm [Bertenstam et al. 1995, Carlson 1996, http://www.speech.kth.se/waxholm/waxholm.html].

Compared to D1.5, the present paper provides a draft best practice proposal for testing by DISC colleagues, DISC Advisory Panel members and other intended users in industry and academia. In fact, the paper will serve as a stepping stone towards packaging the draft best practice methodology in hypertext/hypermedia form on the new DISC-2 website [http://www.disc2.dk]. It is the website version of the methodology which will be put out for testing.

The paper consists of two main chapters, the first of which presents a best practice model of a dialogue manager for SLDSs. The second main chapter presents a best practice life-cycle model for dialogue manager development. Dialogue manager evaluation has been presented in a separate DISC Working Paper [Bernsen 1999]. The dialogue manager model is presented using a common DISC format of issues, options, and pros and cons proposed by DISC collaborators Jan van Kuppevelt and Ulrich Heid [van Kuppevelt and Heid 1999]. Jan van Kuppevelt and Ulrich Heid also contributed to the current practice analysis of dialogue managers in D1.5.

The strength of the present best practice methodology for dialogue management resides in its grounding in the analysis of a large variety of dialogue managers. The authors hope that this has resulted in views on how to design, develop and evaluate appropriate dialogue managers, that are more general than those which are currently available. Current views on dialogue management tend to be heavily biased towards a particular SLDS, a particular interactive task, and a particular dialogue management solution for that task. The reason is that few, if any, dialogue manager developers have extensive hands-on experience from developing different dialogue managers for a wider range of SLDSs and interactive tasks. Hands-on experience remains a major source of knowledge about dialogue management, given the fact that the generalisations that are required in the field cannot yet be found in the literature. If considered as general truths, those views may induce sub-optimal solutions to the development of SLDSs and dialogue managers which, by their nature, require a different approach.

The intended readership of the paper is specialists in SLDSs in general and in dialogue management in particular. Their comments and criticisms are cordially invited.

Contents

1. Introduction

Dialogue management is arguably the core functionality of spoken language dialogue systems (SLDSs). Figure 1 helps explain why this is the case. The figure shows the logical architecture of SLDSs as organised in a series of layers called performance, speech, language, control and context, respectively. A user interacting with an SLDS produces speech input (speech layer) and receives speech output (speech layer) from the system. In addition, the regular user might figure out more abstract properties of the system's behaviour, such as how cooperative it is during dialogue, the distribution of user and system dialogue initiative, or how the system attempts to influence the user's dialogue behaviour through its choice of words and in other ways (performance layer). The rest of the system's workings are hidden from inspection by the user. Figure 1 classifies these workings in terms of a series of headers, such as 'user utterances' or 'domain model', and elements subsumed by each header. The elements are high-level references to the functionality that may be present in today's SLDSs. Most of today's SLDSs do not have all of the functionalities referred to in Figure 1, but all SLDSs have some of the functionalities.

Dialogue management is primarily located in the control and context layers in Figure 1. When the user inputs an utterance to the system, the dialogue manager may receive a representation of the meaning of what the user said from the speech and language layers. It is the task of the dialogue manager to handle that representation properly, eventually producing an appropriate output utterance to the user. Depending on the complexity of the task which the system helps the user solve, the dialogue manager may have to make use of most or all of the elements listed in the control and context layers in Figure 1. For instance, to determine if the user has provided a new piece of information which the system needs in order to complete the task, the dialogue manager may have to check the history of sub-tasks performed so far (context layer: task history). Or, to decide if shortcuts in the dialogue can be made because this particular user does not need a certain piece of information or advice, the dialogue manager would check if the user belongs to the class of experienced users (context layer: user group).

When addressing dialogue management best practice, it is sometimes difficult to distinguish clearly between issues to do with dialogue management *functionality* and dialogue management *usability*. Representing the core functionality of SLDSs, dialogue management cannot be designed and developed without keeping the intended users in mind. Still, the present paper primarily addresses the issues of functionality to be considered by the dialogue management developer. For the usability issues, the reader is referred to [Failenschmid et al. 1999].

This paper only addresses best practice issues to do with the development of dialogue managers for *task-oriented* SLDSs. These SLDSs are being developed to help the user solve a particular task, or several tasks, and they standardly do so in a dialogue with the users in which speech input and output play a major part. The alternative to task-oriented SLDSs is *conversational* SLDSs which aim to perform intelligent conversation with the user more generally, for instance in order to pass some Turing test. Existing experimental systems of this kind are not being considered in what follows.

The issue of dialogue management can be addressed at several different levels of abstraction. In this paper, we shall mostly ignore low-level implementation issues such as programming languages, hardware platforms, software platforms, generic software architecture, database formats, query languages, data structures which can be used in the different system modules, etc. Generic software architectures for dialogue management are still at an early stage, and low-level implementation issues can be dealt with in different ways with little to distinguish between these in terms of efficiency, adequacy etc. Good development tools would appear to be of more relevance at this point. A survey of existing dialogue management tools is provided in [Luz 1999].

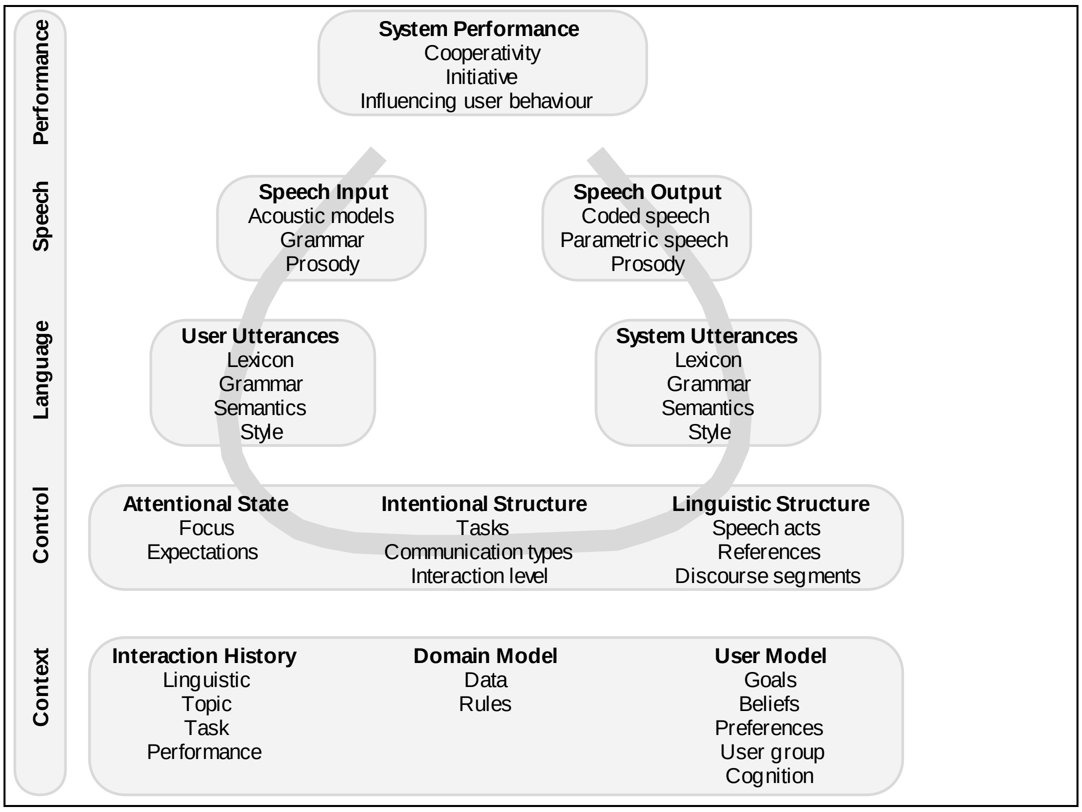

Figure 1. Elements of an interactive speech theory. The grey band and grey boxes reflect the logical architecture of SLDSs. From [Bernsen et al. 1998a].

Dialogue management is but one of the six different aspects of SLDSs investigated in DISC. The five other aspects are: speech recognition, speech generation, language understanding and generation, human factors and systems integration. Each of these aspects are being investigated from two different perspectives, called the 'grid' and the 'life-cycle', respectively. Accordingly, we speak of 'grid properties' and 'life-cycle properties' of SLDSs and their components. *Grid properties* are factual properties of SLDSs and components as well as interrelationships among such factual properties. *Life-cycle properties* are characteristics of the development process of an SLDS or component.

This paper presents a draft proposal for best practice in dialogue management development and evaluation. The grid properties are structured in terms of issues, options, and pros and cons. An *issue* is a question of functionality to be considered during specification: should the dialogue manager being specified have this functionality or not? For each issue there may be one or more *options,* i.e., ways of adding the functionality, and each option may have pros and/or cons, i.e. considerations speaking in favour, or against, choosing that option in the context of the development project at hand. As regards the dialogue manager life-cycle, a separate report presents an in-depth study of dialogue manager evaluation [Bernsen 1999]. For this reason, dialogue manager evaluation will only be briefly mentioned in the present report.

Chapters 2 (grid) and 3 (life-cycle) present the DISC draft proposal for best practice in dialogue management development and evaluation which has evolved from the current practice overview presented in [Bernsen et al. 1998b] and the work on tools in Work Package 2. Dialogue management theory is still relatively uncharted territory. The material presented and synthesised in this paper, if properly analysed, might form the basis for sketching a first general theory of dialogue management. Chapter 4 concludes the paper.

2. Designing an Appropriate Dialogue Manager

A dialogue manager should be designed based on careful consideration of its task(s) and users. What actually happens is that the designer (or developer) designs a system based on both *design-time* and *run-time* considerations. Some design decisions only affect design-time work whereas many others affect run-time operation as well. The system's task, for instance, is fixed at design-time and cannot be modified at run-time. On the other hand, what the user actually inputs at some point, cannot be fully predicted at design-time and must be handled at run-time. The distinction between design-time and run-time considerations is inherent to the following account and we trust that the reader will be able to decide when a particular comment applies to design-time only or to run-time as well.

This chapter provides a draft best practice guide to dialogue manager design. For ease of use, the guide has been broken down into 24 issues. These issues have been structured under the following headings as follows:

Goal of the Dialogue Manager

System Varieties

Input Speech and Language

Getting the User's Meaning

Communication

History, Users, Implementation

This structure derives from the review of a dialogue manager's main tasks and their ordering, as presented under Issue 24. Some issues have sub-issues. Some issues do not give rise to alternative options. Pros and cons are often embedded in the text.

Goal of the Dialogue Manager

2.1 Issue: Efficient task performance

The central goal of an SLDS is to enable the interactive task to be done efficiently and with a maximum of usability. As we will not be considering usability aspects here (see [Failenschmid et al. 1999]), from the point of view of the present paper this goal reduces to efficient task performance. Efficient task performance means that the system always performs appropriate processing of the user's input and always provides appropriate output at a given stage during the dialogue. This is the - admittedly ideal - goal that SLDS developers should strive for, and this goal subsumes a number of sub-goals which are common in discussions of SLDSs and their components, such as robustness, flexibility, naturalness etc. (see Chapter 3).

System Varieties

Everyone's standard idea of an SLDS is a unimodal speech input/speech output system which conducts a dialogue about a single task in a single language with one user at a time. This section looks at the complementary possibilities. One of these might be just what is needed for a particular application.

2.2 Issue: Multimodal systems including speech

Appropriate SLDS output is not always *spoken* output.

Option 2.2.1: Connecting to a human being and other output actions

For some tasks, the system's output may be *system actions* which the user can perceive, such as connecting the user with the person the user asked to speak to. Obviously, SLDSs may also produce other non-speech output actions in response to the user's input. Providing operator fall-back is one such action (see below). Other actions are often due to the fact that the system provides output in modalities other than speech (see next option).

Option 2.2.2: Using other modalities than speech

Some tasks require the system to output information which is most effectively presented in, e.g., textual or pictorial form. Long lists of flight connections, for instance, are much more efficiently conveyed by text, and it is well-known that the contents of many pictures are virtually impossible to render through verbal means. A bicycle repair guide, for instance, is not much worth to the novice without ample pictorial illustration. In such cases, the SLDS needs to be enhanced through other modalities of (output) information presentation, such as text or images presented on a screen. This is the case in the Waxholm system which also uses an animated graphical speaking face whose lip movements help the user disambiguate the system's synthetic speech and whose eye movements help the user focus on newly added information on the screen. Through its user speaks/interrupts input button, Waxholm also illustrates the fact that SLDSs do not need to only take speech as input.

For many other tasks as well, an appropriate solution may be to combine spoken input with other input modalities, such as pointing gesture. So, more generally speaking, there is an important question here about when to use speech input and/or speech output for a particular application, when not to use speech, and when to use speech in combination with other particular input/output modalities for information representation and exchange. The solution to this question about the functionalities, or roles, of modalities in particular cases not only depends on the task but on many other parameters as well. A web-based tool called SMALTO has been developed in DISC to support the decision on when (not) to use speech for particular applications [Bernsen and Luz 1999]. A first test version of the tool is available at http://disc.nis.sdu.dk/smalto/.

2.3 Issue: Multilingual systems

Appropriate SLDS output is not always spoken output *in the same language* as the spoken input provided by the user. In addition, the SLDS may take input in several different languages.

Option 2.3.1: Translation from one language into another

Spoken translation systems, for instance, such as Verbmobil, translate spoken input in one language into spoken output in a different language. These systems are basically different from standard SLDSs in that they do not conduct any dialogue with the user about the application domain: rather, they mediate (task-oriented) dialogue between users who speak different languages. At most, spoken translation systems conduct meta-communication dialogues with their users (see 2.15). Still, spoken translation systems share many issues of dialogue management with standard SLDSs.

Option 2.3.2: Offering the user to choose among several languages

An SLDS may accept spoken input in different languages for the same task(s) and respond in the language in which it is being addressed.

2.4 Issue: One or several tasks and users?

In what follows, we focus on the design of an SLDS for a single task. Already today, systems are being built to handle several more or less independent tasks, such as consulting one's diary and answering email over the phone. These applications increase an already existing need for task and domain independent dialogue managers. Task and domain independent dialogue managers would facilitate the rapid prototyping of SLDSs for new tasks and new domains. We will return to this issue later (Section 2.24).

In what follows, we also focus on single caller (or user) – single system dialogue. An obvious next step is for several callers to conduct dialogue with the system to solve one and the same task. There are several real issues involved in creating smooth multi-user spoken dialogue with machines, such as that of handling simultaneous input from several users. These issues will not be discussed further in this report.

Input Speech and Language

2.5 Issue: Are the speech and language layers OK?

Suppose that we have identified a certain task, T1, for which we consider building an SLDS. Suppose, in addition, that T1 is already known not to generate problems which cannot be solved by following best practice in developing the speech and language layers of the application (cf. Figure 1). In other words, our speech recogniser can be expected to come to possess the necessary robustness for the application, the vocabulary will not be infeasibly large, we are able to develop the grammar and parser required, language generation and speech generation of sufficient quality can be developed, etc.

2.6 Issue: Do the speech and language layers need support from the dialogue manager?

Things in this world often do not come for free. It may be that conditions have to be imposed on the dialogue manager in order to guarantee feasibility in the speech and language layers. The feasibility conditions may derive from, e.g., the need for real-time performance (see 2.7), task complexity (see 2.8), or achieving a sufficient recognition rate in a large-vocabulary system. The dialogue manager may thus have to actively support the processing done in the speech and language layers. It is worth noting, however, that many existing systems use no dialogue manager support for the speech and language layers. In the future, we will see many more forms of dialogue manager support for the speech and language layers than those listed below. We distinguish between dialogue manager support for *input prediction, input language processing* and *output generation.*

Option 2.6.1: Input prediction support

Input prediction support may be possible, i.a., if the task has some structure to it (see 2.8 and 2.11). Conditions could be that the dialogue manager must:

- constrain the search space of the speech recogniser by constraining the set of words which are likely to occur in the next user utterance (sub-vocabulary prediction);

- constrain the search space of the keyword or phrase spotting component by delimiting its search space to the most probable keywords or phrases (keyword prediction);

- constrain the search space of the syntactic analysis component by narrowing the set of applicable grammar rules to a specific and probable subgrammar (subgrammar prediction);

- constrain the search space of the parser by narrowing the set of semantically meaningful units it should be working with (semantic prediction).

Pros

Successful input prediction techniques can significantly reduce search spaces and thus increase speed and recognition/understanding accuracy.

Cons

The risk is that users turn out to behave differently from what was predicted, i.e. that the scope of the prediction was too narrow. This risk must be minimised. Failed input predictions impose severe demands on the system's meta-communication abilities. A first step in minimising the risk of failed input prediction is to take a careful look at the task complexity (see 2.8).

Option 2.6.2: Input language processing support

Input language processing support will become increasingly important as the user's input utterances grow in length as well as in lexical, grammatical and linguistic context-determined complexity. Conditions could be that the dialogue manager must:

- constrain the search space of the semantic analysis component by, e.g., delimiting the set of possible antecedents in cross-sentence anaphora resolution.

Option 2.6.3: Output generation support

Output generation support is an important means of controlling the user's dialogue behaviour (see 2.9). Conditions could be that the dialogue manager must:

- control the prosody of the spoken output based on control layer information about the message to be produced, such as speech act information;

- control the lexical variation of dialogue expressions to be produced by the language generation component;

- control the grammar of the dialogue utterances to be produced by the language generation component;

- control the style of the dialogue utterances to be produced by the language generation component.

2.7 Issue: Real-time requirements

It may be assumed that most or all SLDSs need to work in real-time or close-to-real-time, for such is the nature of spoken dialogue. In some cases of spoken human-human dialogue, we actually do tolerate waiting for a response from our dialogue partner, especially when we know that the partner has to search for information or do complex calculations or inferences before responding. Users may be expected to tolerate the same from machines but, generally speaking, close-to-real-time operation is a very important goal in SLDS design.

Getting the User's Meaning

2.8 Issue: Task complexity

Ideally, we would like to develop SLDSs which simply let the users speak their minds and then do whatever is necessary to complete the task. For some tasks, this is clearly feasible. Suppose, for instance, that our SLDS has to ask if the user wants to accept a collect call and, depending on the user's answer, simply routes or does not route the call to the user. In this case, the risk of letting the users speak their minds probably is as small as it ever gets. However, suppose that the task is about ordering VIP dinner arrangements at a large restaurant including date, time, duration, choice of rooms, many-course meals, special diets, wines, flowers, timing for speeches, entertainment, seating arrangements, payment arrangements etc. It is easy to imagine that some users will have quite a bit to say about that. They may have so much to say, in fact, that even the dialogue manager support to the speech and language layers described in 2.6 above will not be sufficient to guarantee that the system will always be able to provide appropriate output. Task complexity, therefore, probably is the single most important factor to consider for the dialogue manager developer.

Unfortunately, at this point, task complexity cannot be measured in any objective way. The reason is that task complexity is a function of several factors, including:

- the number of pieces of information which have to be communicated between the user and the system, such as departure airport, arrival airport, departure time, arrival time etc.;
- the number of optional pieces of information which could be, but do not have to be, communicated between the user and the system, such as whether or not the caller wants to book a return flight;
- whether or not the task itself has some structure in terms of a full or partial ordering of its component sub-tasks. For instance, train schedules are often different on weekdays and Sundays, which means that the system must know the date before it can offer particular departure times;
- whether or not the sub-tasks constituting the task are independent. For instance, a user may want to trade departure time for a cheap fare. In that case, the sub-task of fixing the departure time is not independent of the sub-task of determining if the caller has an interest in discount;
- whether the flow of information is one-way or two-way. One-way information flow is the simpler, such that, for instance, the system asks all the questions and the user provides all the answers;
- whether or not negotiation is involved;
- whether or not substantial economic commitments are being made during dialogue, such as the purchase of a flight ticket or the ordering of a bank transfer;
- whether or not real-time updates are important, such as information about delayed flights or impending strikes;
- etc.

In a maximum-complexity scenario, the task involves exchange of many necessary pieces of information, several optional loops in the dialogue structure, little or no structure, several sub-task inter-dependencies, two-way flow of information, negotiation, security measures for safe-guarding the transactions made, and frequent real-time updates.

Based on factors such as the above, the following guidelines may be useful.

Sub-issue 1: Volume of information

How many pieces of information must be exchanged between the user and the system to complete the task? In many cases, one or two pieces of information will suffice. The collect call SLDS above is a case in point: the system just needs one piece of information to know whether or not to transfer the call. The Operetta telephone directory assistance system is another example: all it needs is a person's name [Fraser et al. 1996]. In other cases, exchange of something in the order of 4-6 pieces of information suffice to complete the task, as in the RailTel/ARISE train time-table information systems. In these systems, the user has to provide the system with just enough information about departure station, arrival station, date, and possibly time of day, that the system can compute the missing information to be offered to the user. It is feasible today with such systems to let the users speak their minds, have the system realise that one or two data points may be missing from the user's input, ask for these, and then compute the answer to the user's query. The Danish Dialogue System, on the other hand, requires so much information from the user that letting the users speak their minds may be counter-productive. The likelihood that the system fails to recognise and understand what the users say becomes too great. The same is true of the dinner arrangement system mentioned above. In such cases of relatively large task complexity, it may be necessary to adopt strategies for controlling the user's *input* so that it may have a good chance of being recognised and understood by the system. Such strategies will be reviewed below (2.9).

Information optionality is another complicating factor. Callers asking about train time-tables get just that. But if the system can also book the train tickets, it must have the capability to exchange additional pieces of information, some of which are obligatory whereas others are optional: how many will travel; their age categories; will they need return tickets; will they all need return tickets or only some of them; will some need round-trip tickets; will they need information on the notions of different kinds of discount departures; etc. In such cases, the option of letting users speak their minds may be impossible and the challenge for the dialogue designer becomes that of eliciting all the pieces of information needed in as elegant a way as possible.

Sub-issue 2: Ill-structured tasks

What is the task structure, if any? Some tasks are ill-structured or have no structure at all. Consider, for instance, an SLDS whose database contains all existing information on flight travel conditions and regulations for a certain airline company. This information tends to be quite comprehensive both because of the many practicalities involved in handling differently composed groups of travellers and their luggage, but also because of the legal ramifications of travelling which may surface if, e.g., the flight crashes. Some users may want many different individual pieces of information from the database whereas others may want only one piece. Which piece(s) of information a particular user wants is completely unpredictable. We call such user tasks *ill-structured tasks:* the user may want one, two or several pieces of information from the database, and the order in which the user may want the information is completely arbitrary as seen from the system's point of view. The system must be prepared for everything from the user all the time.

Option 2.8.1: Using domain structure

One way to try to reduce the complexity of large ill-structured tasks is to use, or invent, *domain structure.* That is, the domain may be decomposable into a number of sectors which themselves may be hierarchically decomposed, etc. So the system asks, for instance: "Do you want to know about travelling with infants, travelling with pets, travelling for the elderly and the handicapped, hand luggage, or luggage for storage?" And if the user selects 'hand luggage', the system asks: "Do you want to know about volume of permitted luggage, electronic equipment, fragile items, or prohibited luggage?" Etc.

Pros

This structuring allows the system to handle the dialogue.

Cons

However, few users will accept to navigate through many hierarchical levels prompted by the system in order to find a single piece of information at the bottom of some deep domain hierarchy. Making the hierarchy more shallow will often make matters even worse. No user will accept to have to go through a shallow but broad hierarchy prompted by the system in order to find a single piece of information at the end of it. Just imagine a Danish train time table inquiry system which asks the user: "Do you want to go from Aabenrå, Aalborg, Aarhus ...", mentioning that small country's 650 or so train stations in alphabetical order and in chunks of, say, ten at a time.

Option 2.8.2: Electronic yellow pages

Another important problem about ill-structured tasks is that their vocabularies tend to differ considerably from one user to another. Contrary to, e.g., train time-table inquiries which can start from the "official" city names, or used car sales catalogues which can start from the car brands and production years, there is no "official" vocabulary, used by all or most users, for inquiring about pet transportation, high-volume luggage or staff support for the elderly. We are all familiar with this problem from the (non-electronic) yellow pages whose entries are often labelled differently from how we would have labelled them ourselves. SLDSs in fact offer an elegant solution to this problem, i.e. an electronic dictionary of equivalents. The user speaks his or her mind about the entry of interest. The system only has to spot a keyword in the user's input corresponding to an entry in its electronic dictionary, in order to direct the user to the right entry in its hierarchy. And the user does not have to worry about knowing or remembering the headings used to structure the hierarchy itself.

Sub-issue 3: Well-structured tasks

Other tasks have some structure to them. Consider the flight ticket reservation task handled by the Danish Dialogue System. This task is a partially ordered one. It would normally make little sense, for instance, to ask for one of the morning departures until one has specified the date; and it normally makes little sense to expect the system to tell whether or not flights are fully booked on a certain date until one has indicated the itinerary. Moreover, ordinary users know that this is the case. Task structure is helpful if the task complexity makes it advisable to control the user's input (see 2.9).

It is important to note, however, that partial-order-only is what one is likely to find in most cases. Moreover, sub-task interdependencies may interfere with the designer's pre-conceived ideas about "the" task order. For instance, some users may want to know which departures are available at reduced fares before wanting to know about departure times, whereas others do not care about fare reductions at all. In this way, sub-task interdependencies may also introduce an element of negotiation into the dialogue. For instance, before accepting a certain departure, the caller may want to know if the subsequent departure is cheaper. This makes the dialogue a two-way exchange where the caller and the system take turns in asking questions of one another and answering those questions. Both elements, negotiation and two-way exchange, complicate task model design.

Moreover, even the partial task order that there is, may be by default-only. If somebody simply wants to leave town as soon as possible, the itinerary matters less than the departure time of the first outbound flight which has a free seat.

Sub-issue 4: Negotiation tasks

Does the task require substantial negotiation? Some tasks involve a relatively low volume of information and in addition have some structure to them. Still, they may be difficult to manage if they involve a considerable amount of negotiation. The Verbmobil meeting scheduling task is an example. Fixing a meeting simply requires fixing a date, a time or a time interval, and possibly a place, so the volume of information is relatively low. Unless one knows the date, it can be difficult to tell if one is free at a certain time, so the task has some structure to it. And for busy people, the date-time pair may matter more than the exact venue. The problem inherent to the Verbmobil task is that fixing meetings may require protracted, and ill-structured, negotiations of each sub-task. The outcome of a meeting date/time/venue negotiation is not a simple function of calendar availability and prior commitments but also depends on the importance of the meeting, the importance of the prior commitments, the possibilities of moving, cancelling or apologising other meetings, the professional and personal relationships between the interlocutors etc.

In addition, Verbmobil does nothing to impose structure on the (human-human) dialogue for which it provides translation support, allowing the dialogue to run freely wherever the interlocutors want it to go. In such cases, it makes little sense to judge task complexity in terms of the volume of information to be exchanged or in terms of the task structure that is present, because the real problem lies in negotiating and eventually agreeing to meet in view of the imponderables involved. Had Verbmobil been a *human-machine* dialogue system product, the SLDS representing the diary of someone absent to the conversation, state-of-the-art engineering practice would probably have dictated much stronger system control of the dialogue, effectively turning meeting date/time/venue negotiation into a booking exercise (see 2.9).

Note also that negotiation is a natural human propensity. It seems likely that most SLDSs can be seen to potentially involve an element of negotiation. Verbmobil just involves a considerable amount of it. In the case of the Danish Dialogue System, for instance, the system may propose a list of morning departures and ask if the user wants one of them. If not, the joint search for a suitable departure time continues. So the reason why negotiation is less obvious or prominent in, e.g., the dialogue conducted with the Danish Dialogue System when compared to the dialogue conducted with Verbmobil, is not that the task of the Danish system by itself excludes negotiation. Rather, the reason is that the system's dialogue behaviour has been designed to constrain the way in which negotiations are done.

Summarising, when developing SLDSs we want to let the users speak their minds. However, if the task is a complex one, the SLDS project either has to be given up as a commercial undertaking, turned into a research project, such as Verbmobil, or requires some or all of the following strategies: (a) input prediction, (b) input language processing control, (c) output control and (d) control of user input. To the extent that they are applicable, all of (a) through (d) are of course useful in any system but their importance grows with increasing task complexity. (d) subsumes (c) as shown in Section 2.9. Obviously, (a) through (d) can also be used for very simple tasks, making dialogue engineering more easy to do for these tasks than if the users were permitted to speak their minds in a totally unconstrained fashion.

2.9 Issue: Controlling user input

The dialogue manager can do many things to control the user's input in order to keep it within the system's technical capabilities of recognition and understanding. Among task-oriented SLDSs, extreme lack of user input control may be illustrated by a system which simply tells the user about the task it can help solve and then invites the user to go ahead. Lack of user input control allows the user to produce unconstrained input. Clear signs of unconstrained user input are: very long input sentences; topics are being raised, or sub-tasks addressed, in any order; and any number of topics is being addressed in a single user utterance. Even for comparatively simple tasks, the handling of unconstrained user input is a difficult challenge which most SLDS projects cannot afford to try to meet.

What user input control does is to impose, through a variety of explicit and implicit means, more or less strong constraints on what would otherwise have been unconstrained user input. Some of the user input control mechanisms available to the SLDS developer are:

Option 2.9.1: Information on system capabilities

This section is about explicit information to users on what the system can and cannot do. For all but the simplest SLDSs, and for all but very special user populations, it is advisable that the system, up-front, clearly and succinctly tells the user things like what is its domain, what is its task(s) etc. This helps "tailoring" the user's expectations, and hence the user's input, to the knowledge the system actually has, thereby reducing the task of the dialogue manager. System capability information does not have to be given to users through speech, of course. Waxholm, for instance, provides this information as text on the screen as well as in synthetic speech. If the information is given through speech, it is virtually always important that it is expressed briefly and clearly because users tend to loose attention very quickly when they have to listen to speech-only information.

It is often an empirical issue how much users need to be told about the system's capabilities in order that their mental model of the system's capabilities roughly match what the system actually can and cannot do. If the story is longer than a few facts, it is advisable to make it possible for regular users to skip the story. Also, those parts of the story which only relate to some optional loop in the dialogue, might be better presented at the start of that loop than up-front.

Brief on-line information on the system's capabilities cannot be made redundant by any amount of paper information about the system.

Option 2.9.2: Instructions on how to address the system

This section is about explicit instructions to users on how to address the system. By issuing appropriate instructions, the SLDS may strongly increase its control of the way it is being used. For instance, the Danish Dialogue System tells its users that it will not be able to understand them unless they answer the system's questions briefly and one at a time. If used at all, such operating instructions should be both very brief and eminently memorable. Otherwise, they will not work because too many users will forget the instructions immediately. Indications are that the quoted instruction from the Danish Dialogue System worked quite well. However, as part of the operating instructions for the Danish system, users were also instructed to use particular keywords when they wanted to initiate meta-communication (see 2.15). This worked less well because too many users forgot the keywords they were supposed to use.

Option 2.9.3: Feedback on what the system understood

Feedback on what the system has understood from what the user just said, so that, throughout the dialogue, the user is left in no doubt as to what the system has understood (see also 2.19). All SLDSs need to provide this form of (information) feedback. A user who is in doubt as to whether the system really did understand what the user just said, is liable to produce unwanted input.

Option 2.9.4: Processing feedback

Processing feedback is about what the system is in the process of doing. When the system processes the information received from the user and hence may not be speaking for a while, processing feedback keeps the user informed on what is going on (see also 2.19). Most SLDSs can benefit from this form of (processing) feedback. A user who is uncertain about what is going on inside the system, if anything, is liable to produce unwanted input.

Option 2.9.5: Output control

The aim of output control (cf. 2.6) is to "prime" the user through the vocabulary, grammar and style adopted by the system. There is ample evidence that this works very well for SLDSs. Humans are extremely good at (automatically, unconsciously) adapting their vocabulary, grammar and style to those of their partners in dialogue or conversation. It is therefore useful to make the system produce output which only uses the vocabulary and grammar which the system itself can recognise, parse and understand, and to make the system use a style of dialogue which induces the user to provide input which is terse and to the point. The unconscious adaptation performed by the users ensures that they still feel that they can speak their minds without feeling hampered by the system's requirements for recognisable vocabulary, simple grammar, and terse style.

A particular point to be aware of in this connection is that if the system's output to the user includes, e.g., typed text on the screen, then the textual output should be subjected to the same priming strategy as has been adopted for the system's spoken output. It is not helpful to carefully prime the user through the system's output speech and then undercut the purpose of the priming operation through a flourishing style of expression in the textual output.

Pros

A very effective method for reducing user input complexity. Should be used in all SLDSs.

Cons

None.

Option 2.9.6: Focused output and system initiative

Focused output combined with system initiative (cf. 2.10). If the system determines the course of the dialogue by having the initiative all or most of the time, for instance through asking questions of the user or providing the user with instructions which the user has to carry out, a strong form of user input control becomes possible. The system can phrase its questions or instructions in a focused way so that, for each question, instruction etc., the user has to choose between a limited number of response options. If, for instance, the system asks a question which should be answered by a 'yes' or 'no', or by a name drawn from a limited set of proper names (of persons, weekdays, airports, train stations, streets, car brand names etc.), then it exerts a strong influence on the user's input. Dialogue managers using this approach may be able to handle even very-large-complexity tasks in terms of information volume (2.8).

Note that, in itself, system initiative is far from sufficient for this kind of input control to take place. If, for instance, the system says "ADAP Travels, can I help you?", it does, in principle, take the initiative by asking a question. However, the question is not focused at all and therefore does not restrict the user's input. An open or unfocused system question may therefore be viewed as a way of handing over the dialogue initiative to the user.

Pros

A very effective method for reducing user input complexity and increasing user input predictability (cf. 2.6).

Cons

The method is obviously better suited for some tasks than for others, in particular for tasks consisting of a series of independent pieces of information to be provided to the system by the user. Beyond those tasks, the strong form of input control involved will tend to give the dialogue a less-than-natural and rather mechanical quality. Moreover, some tasks do not lend themselves to system initiative-only, and large unstructured tasks cannot be handled through focused output combined with system initiative in a way which is acceptable to the users.

Option 2.9.7: Textual material

The term 'textual material' designates information about the system in typed or hand-written form and presented graphically, such as on screen or on paper, or haptically, such as using Braille. Typically, this information tells users what the system can and cannot do and instructs them on how to interact with the system. For particular purposes, such as when the users are professionals and will be using the SLDS extensively in their work, strong user input control can be exerted through textual material which the user is expected to read when, or before, using the system. For obvious reasons, text-based system knowledge is difficult to rely on for walk-up-and-use systems unless the system, like Waxholm, includes a screen or some other text-displaying device.

Pros

In principle, the textual material could present all or some of the following: complete answers to all or most questions the users might have about task, domain and operations, exemplify typical dialogues, highlight what to do and what to avoid, present the steps to take, block false assumptions, provide background information etc.

Cons

In general, users are not likely to have textual material on paper at hand when using the system: it is somewhere else, it has disappeared, or they never received it in the first place.

Option 2.9.8: Barge-in

Barge-in means that users can speak to, and expect to be recognised and understood by, the system whenever they so wish, such as when the system itself is speaking or is processing recent input. Barge-in is not, in fact, an input control mechanism. Rather, it is something which comes in handy because full input control is impossible. In particular, it is impossible to prevent enough users from speaking when the system is not listening, no matter what means are being adopted for this purpose.

The likelihood of users speaking their minds "out of order" varies from one application and user group to another and this point should be considered by the dialogue manager developer(s). In some applications, it may even be desirable that the users can speak freely among themselves whilst the system is processing the spoken input.

Still, barge-in technology is advisable for very many SLDSs, so that the system is able to recognise and process user input even if it arrives when the system is busy doing something other than just waiting for it. In Waxholm, the system does not listen when it speaks. However, the user may barge in by pressing a button to interrupt the system's speech. This is useful when, for instance, the user already feels sufficiently informed to get on with the task. For instance, users who are experts in using the application can use the button-based barge-in to skip the system's introduction. The Danish Dialogue System does not allow barge-in when the system speaks. This turned out to cause several transaction failures, i.e. dialogues in which the user did not get the result asked for. A typical case is one in which the system's feedback to the user (see 2.19) shows that the system has misunderstood the user, for instance by mistaking "Saturday" for "Sunday". During the system's subsequent output utterance, the user says, e.g.: "No, Saturday". The system's lack of reaction to what the user said is easily interpreted by the user as if the system had received the error message, which of course it hasn't, and couldn't have. As a result, the user will later receive a flight ticket which is valid for the wrong day.

2.10 Issue: Who should have the initiative?

Dialogue in which the initiative lies solely with the system was discussed as an input control mechanism in Section 2.9.6. This section generalises the discussion of initiative initiated in Section 2.9.6.

Dialogue management design takes place between two extremes. From the point of view of technical simplicity, one might perhaps wish that all SLDSs could conduct their transactions with users as a series of questions to which the users would have to answer 'yes' or 'no' and nothing else. Simpler still, 'yes' or 'no' could be replaced by filled pauses ("grunts") and unfilled pauses (silence), respectively, between the system's questions, and speech recognition could be replaced by grunt detection. From the point of view of natural dialogue, on the other hand, users should always be able to say exactly what they want to say, in the way they want to say it, and when they want to say it, without any restrictions being imposed by the system. Both extremes are unrealistic, of course. If task complexity is low in terms of, among other things, information volume and negotiation potential, then it is technically feasible today to allow the users to say what they want whilst still using some of the input control mechanisms discussed in Section 2.9. As task complexity grows in terms of information volume, negotiation potential and the other factors discussed in Section 2.8, it really begins to matter who has the initiative during the dialogue.

We may roughly distinguish three interaction modes, i.e. *system directed dialogue, mixed initiative dialogue,* and *user directed dialogue.* This distinction is a rough one because initiative distribution among user and system is often a matter of degree depending upon how often which party is supposed to take the initiative. In some cases, it can even be difficult to classify an SLDS in terms of who has the initiative. If the system opens the dialogue by saying something like: "Welcome to service X, what do you want?" - it might be argued that the system has the initiative because the system is asking a question of the user. However, since the question asked by the system is a completely open one, one might as well say that the initiative is being handed over to the user. In other words, only focused questions clearly determine initiative (cf. 2.9.6). The same is true of other directive uses of language in which partner A tells partner B to do specific things, such as in instructional dialogues.

Option 2.10.1: System directed dialogue

As long as the task is one in which the system requires a series of specific pieces of information from the user, the task may safely be designed as one in which the system preserves the initiative throughout by asking focused questions of the user. This system directed approach would work even for very-large-complexity tasks in terms of information volume and whether or not the tasks are well-structured. Note that, if the sub-tasks are not mutually independent or if several of them are optional, then system initiative may be threatened (cf. 2.8).

From a dialogue engineering perspective, it may be tempting to claim that system directed dialogue is generally simpler to design and control than either user directed dialogue or mixed initiative dialogue, and therefore should be preferred whenever possible. This claim merits some words of caution. Firstly, it is not strictly *known* if the claim is true. It is quite possible that system directed dialogue, for all but a relatively small class of tasks, is *not* simpler to design and control than its alternatives because it needs ways of handling those users who do not let themselves be fully controlled but speak out of turn, initiate negotiation, ask unexpected questions etc. Secondly, products are already on the market which allow mixed initiative dialogue for relatively simple tasks, such as train time-table information. And it is quite likely that users generally tend to prefer such systems because they let the users speak their minds to some extent.

Pros

A very effective strategy for reducing user input complexity and increasing user input predictability (cf. 2.6). Relatively natural for some tasks.

Cons

The strategy is obviously better suited for some tasks than for others. Beyond those tasks, the strong form of input control involved will tend to give the dialogue a less-than-natural and rather mechanical quality which implies that users may tend to take the initiative in some situations.

Option 2.10.2: Mixed initiative dialogue

In mixed initiative dialogue, any one of the participants may have – or take - the initiative. A typical case in which a mixed initiative approach is desirable is one in which the task is large in terms of information volume and both the user and the system need information from one another. Whilst asking its questions, the system must, always or sometimes, be prepared that the user may ask a question in return instead of answering the system's own question. For instance, the user may want to know if a certain flight departure allows discount before deciding whether that departure is of any interest.

In practice, most SLDSs have to be mixed initiative systems in order to be able to handle user-initiated repair meta-communication. The user must have the possibility of telling the system, at any point during dialogue, that the system has misunderstood the user or that the user needs the system to repeat what it just said (see 2.15). Only systems which lack meta-communication altogether can avoid that. Conversely, even if the user has the initiative throughout, the system must be able to take the initiative to do repair or clarification of what the user has said. When thinking about, or characterising, SLDSs, it may be useful, therefore, to distinguish between two issues: (a) who has the initiative in domain communication? and (b) who has the initiative in meta-communication? (More in 2.15).

The need for mixed initiative dialogue in *domain communication* is a function of bi-directionality of the flow of information needed to complete the task, sub-task interdependencies, the potential for negotiation occurring during task completion, the wish for natural and unconstrained dialogue etc. (cf. 2.8).

Cons

Mixed initiative dialogue is harder to control and predict than system directed dialogue.

Option 2.10.3: User directed dialogue

In user directed dialogue, the user has the initiative. User directed dialogue is recommended, in particular, for ill-structured tasks in which there is no way for the system to anticipate which parts of the task space the user wants to address on a particular occasion. The flight conditions information task (2.8) is a case in point, as is the email operation task (2.4).

Pros

Natural for some tasks. Good for tasks with regular users who have the time to learn how to speak to the system to get the task done.

Cons

User directed dialogue is harder to control and predict than system directed dialogue. Not recommended for walk-up-and-use users except for very simple tasks.

2.11 Issue: Input prediction/prior focus

In order to support the system's speech recognition, language processing and dialogue management tasks, the dialogue manager developer should investigate if selective prediction of the user's input is possible at any stage during the dialogue (2.6). This may be possible if, e.g., the system asks a series of questions each requesting specific pieces of information from the user. If the task has some structure to it, it may even be possible to use the structure to predict when the user is likely to ask questions of the system, thus facilitating mixed initiative (2.10) within a largely system directed dialogue. I.a. the Daimler-Chrysler dialogue manager and the Danish Dialogue System use various forms of input prediction. Another way of describing input prediction is to say that the dialogue manager establishes a (selective) *focus of attention* prior to the next user utterance.

Useful as it can be, input prediction may fail because the user does not behave as predicted. In that case, the dialogue manager must be able to initiate appropriate meta-communication (see 2.15). This is not necessarily easy to do in case of failed predictions because the system may not be aware that the cause of failed recognition or understanding was its failure to predict what the user said. Unless it relaxes or cancels the prediction, the risk is that the dialogue enters an error loop in which the system continues to fail to understand the user.

Pros

Successful input prediction techniques can significantly reduce search spaces and thus increase speed and recognition/understanding accuracy.

Cons

The risk is that users turn out to behave differently from what was predicted, i.e. that the scope of the prediction was too narrow. This risk must be minimised. Failed input predictions impose severe demands on the system's meta-communication abilities. A first step in minimising the risk of failed input prediction is to take a careful look at the task complexity (see 2.8).

Input prediction can be achieved in many different ways. It may be useful to distinguish between the following two general approaches.

Option 2.11.1: Knowledge-based input prediction

In *knowledge-based input prediction,* the dialogue manager uses a priori knowledge of the context to predict characteristics of the user's next utterance. Note that the a priori nature of knowledge-based input prediction does not mean that implemented predictions should not be backed by data on actual user behaviour. It is always good practice to test the adopted knowledge-based input prediction strategy on user-system interaction data.

Option 2.11.2: Statistical input prediction

In *statistical input prediction,* the dialogue manager uses corpus-based information on what to expect from the user. Given a corpus of user-system dialogues about the task(s) at hand, it may be possible to observe and use regularities in the corpus, such as that the presence of certain words in the user's input makes it likely that the user is in the process of addressing a specific subset of the topics handled by the system, or that the presence of dialogue acts DA5 and DA19 in the immediate dialogue history makes it likely that the user is expressing DA25. Waxholm uses the former approach, Verbmobil the latter.

2.12 Issue: Sub-task identification

It is a useful exercise for the dialogue manager developer to consider the development task from the particular point of view of the dialogue manager. The dialogue manager is deeply embedded in the SLDS, is out of direct contact with the user, and has to do its job based on what the speech and language layers deliver. This happens in the context of the task, the target user group, and whatever output and input control the dialogue manager may have imposed.

Basically, what the speech and language layers can deliver to the dialogue manager is some form of meaning representation. Sometimes the dialogue manager does not receive any meaning representation from the speech and language layers even though one was expected. But even if a meaning representation arrives, there is no guarantee that this representation adequately represents the contents of the message that was actually conveyed to the system by the user because the speech and language layers may have gotten the user's expressed meaning wrong. Still, whatever happens, the dialogue manager must be able to produce appropriate output to the user.

Current SLDSs exhibit different approaches to the creation of a meaning representation in the speech and language layers as well as to the nature of the meaning representation itself. An important point is the following: strictly speaking, the fact that a meaning representation arrives with the dialogue manager is not sufficient for the dialogue manager to carry on with the task. *First, the dialogue manager must identify to which sub-task(s), or topics, if any, that incoming meaning representation provides a contribution. Only when it knows, or believes that it knows, that, can the dialogue manager proceed to sort out which contribution(s), if any, the incoming meaning representation provides to the sub-task (or those sub-tasks).* In other words, many task-oriented SLDSs require the dialogue manager to do sub-task identification or topic identification.

The task solved by most SLDSs can be viewed as consisting in one or several sub-tasks or topics to be addressed by user and system. One or several of these sub-tasks have to be solved in order that the user and the system succeed in solving the task. Other sub-tasks may be optional, i.e. their solution is sometimes, but not always, required. One example of optional sub-tasks is the meta-communication sub-tasks (see 2.15): if the dialogue proceeds smoothly, no meta-communication sub-tasks have to be solved. Another example is optional sub-task loops, i.e. domain sub-tasks which some, but not all, users need solved, such as asking the system for a particular piece of information in order to be able to appropriately answer a system question.

Basically, dialogue managers can be built so as to be in one of two different situations with respect to sub-task identification. In the first case, the dialogue manager has good reason to assume that the user is addressing a particular domain sub-task; in the second case, the dialogue manager does not know which domain sub-task the user is addressing. In both cases, the dialogue manager must start from the semantic representations that arrive from the speech and language layers, look at the semantically meaningful units, and seek to figure out which sub-task the user is addressing.

Option 2.12.1: Local focus

The dialogue manager may have good reason to assume that the user's utterance addresses a specific sub-task, such as that of providing the name of an employee in the organisation hosting the system. Depending on the task and the dialogue structure design, there can be many different reasons why the dialogue manager knows which sub-task the user is currently addressing: there may be only one sub-task, as in an extremely simple system; the task may be well-structured; the system just asked the user to provide input on that sub-task, etc. Generally speaking, this is a good situation for the dialogue manager to be in, as in the Danish Dialogue System. This system almost always knows, or has good reason to believe, that the user is either addressing a specific domain sub-task or has initiated meta-communication (2.15). Since it has good reason to believe which sub-task the user is addressing, the task of the dialogue manager reduces to that of finding out exactly what is the user's contribution to that sub-task (or one of those sub-tasks, if we count in the possibility that the user may have initiated meta-communication). In such cases, the system has a *local focus.*

The system may still be wrong, of course, and then it becomes the joint task of the system and the user to rectify the situation through meta-communication.

Option 2.12.2: Global focus

The dialogue manager does not know which of several possible sub-tasks the user is addressing. The main reason why the dialogue manager does not know which sub-task the user is currently addressing, is that the dialogue manager has given the user the initiative, for instance by asking an open question in response to which the user may have addressed any number of possible sub-tasks. Alternatively, the user unexpectedly took the initiative. In such situations, the dialogue manager has to do sub-task identification, or topic identification, before it can start processing the user's specific contribution to the sub-task.

Sub-task identification is crucial in systems such as RailTel/ARISE, Waxholm, and Verbmobil. Waxholm uses probabilistic rules linking semantic input features with topics. Given the rules and a particular set of semantic features in the input, Waxholm infers which topic the user is actually addressing. For sub-task identification, Verbmobil uses weighted default rules to map from input syntactic information, keywords, and contextual information about which dialogue acts are likely to occur, into one or several dialogue acts belonging to an elaborate taxonomy of approximately 54 speech acts (or dialogue acts). The mapping can be done in a 'shallow processing' mode as well as in a 'deep processing' mode.

An important additional help in sub-task identification is support from a *global focus,* for instance when the dialogue manager knows that the task history (see 2.21) contains a set of not-yet-solved sub-tasks. These tasks are more likely to come into local focus than those which have been solved already. Another form of global focus can be derived from observation of dialogue phases. Most task-oriented dialogues unfold through three main phases, the introduction phase with greetings, system introductions etc., the main task-solving phase, and the closing phase with closing remarks, greetings etc. Sometimes it may be possible to break down the main task-solving phase into several phases as well. If the system knows which phase the dialogue is in at the moment, this knowledge can be used for sub-task identification support. Knowing *that* can be a hard problem, however, and this is a topic for ongoing research. For instance, the joint absence of certain discourse particles called topic-shift markers and the presence of certain dialogue acts may suggest that the user has not changed dialogue phase.

Generally speaking, the more possible sub-tasks the user might be addressing in a certain input utterance, the harder the sub-task identification problem becomes for the dialogue manager. When doing sub-task identification, the dialogue manager may follow one of two strategies.

Option 2.12.3: Identify one sub-task only

The simpler strategy is to try to identify one sub-task only in the user's input even if the user may have been addressing several sub-tasks, and continue the dialogue from there as in Waxholm.

Option 2.12.4: Identify each single sub-task

The more demanding strategy is to try to identify each single sub-task addressed by the user as in RailTel/ARISE. The latter strategy is more likely to work when task complexity is low in terms of volume of information.

Depending on what arrives from the speech and language layers, and provided that the dialogue manager has solved its sub-task identification task, the dialogue manager must now determine the users' specific contribution(s) to the sub-task(s) they are addressing (see 2.13). Following that, the dialogue manager must do one of five things as far as communication with the user is concerned:

(a) advance the domain communication (see 2.14) including the provision of feedback (2.19),

(b) initiate meta-communication with the user (see 2.15 and 2.18),

(c) initiate other forms of communication (see 2.16),

(d) switch to a fall-back human operator, or

(e) end the dialogue (see 2.20).

Advancing the domain communication means getting on with the task. Initiating meta-communication means starting a sub-dialogue (or, in this case, a meta-dialogue) with the user in order to get the user's meaning right before advancing the domain communication any further. No SLDS probably can do without capabilities for (a) and (b). If (b) fails repeatedly, some systems have the possibility of referring the user to a human operator (d). Otherwise, calling off the dialogue is the only possibility left (e).

In parallel with taking action vis-à-vis the user, the dialogue manager may at this stage take a smaller or larger series of internal processing steps which can be summarised as:

(f) updating the context representation (see 2.21) and

(g) providing support for the speech and language layers to assist their interpretation of the next user utterance (2.6).

2.13 Issue: Advanced linguistic processing

The determination of the user's contribution to a sub-task requires more than, to mention just one example, the processing of semantic feature structures. Processing of feature structures often implies value assignment to slots in a semantic frame even though these values cannot straightforwardly be derived from the user's input in all cases.

If the system has to decide on *every* contribution of a user to a sub-task, something which few systems do at present, advanced linguistic processing is needed. It may involve, among other things, cross-sentence co-reference resolution, ellipsis processing, the processing of discontinuous user input involving large gaps in the expected sequence of dialogue acts, and the processing of indirect dialogue acts. In nearly all systems, the processing of most of these phenomena is controlled and carried out by one of the natural language components - by the parser in the Daimler-Chrysler dialogue manager, by the semantic evaluation component in the Verbmobil speech translation system - but never without support from the dialogue manager.

Option 2.13.1: Co-reference and ellipsis processing

In case of cross-sentence co-reference and ellipsis processing, the natural language system component in charge is supported by the dialogue manager which provides a representation of contextual information for the purpose of constraining the relevant search space. The contextual information is part of the dialogue history (see 2.21). Dialogue history information consists in one or several data structures that are being built up incrementally to represent one or more aspects of the preceding part of the dialogue. In principle, the more aspects of the preceding dialogue are being represented in the dialogue history, the more contextual information is available for supporting the processing done in the language layer, and the better performance can be expected from that layer. Still, co-reference resolution and ellipsis processing remain hard problems.

Option 2.13.2: Discontinuous user input

Discontinuous user input (‘input gaps’), where the user unexpectedly “jumps” ahead or backwards in the dialogue model, represents a problem for most systems. The size of the problem depends on gap size and, more basically, on whether the user’s input is controlled in such a way that the occurrence of the phenomenon is avoided as much as possible. If the occurrence of discontinuous input is not controlled by the system, relatively large input gaps may create problems. In Verbmobil, for instance, dialogue act prediction by the keyword spotting component fills in missing dialogue acts in case of small input gaps. However, the technique used by this component is insufficient for gaps that are larger than two dialogue acts.

Option 2.13.3: Processing of indirect dialogue acts

Finally, advanced linguistic processing also includes the processing of indirect dialogue acts. In this case, the central problem for the system is to identify the “real” dialogue act performed by the user and disguised as a dialogue act of a different type. In contrast to direct dialogue acts, indirect dialogue acts cannot be determined on the basis of their surface form, which makes the frequently used keyword spotting techniques used for the identification of direct dialogue acts almost useless in such cases. Clearly, the processing of indirect dialogue acts calls for less surface oriented processing methods involving semantic and pragmatic information associated with input sequences. This is a hard problem.

Communication

2.14 Issue: Domain communication

The primary task of the dialogue manager is to advance the domain communication based on a representation of the meaning-in-task-context of the user’s input (cf. 2.12 and 2.13). Let us assume that the dialogue manager has arrived at an interpretation of the user’s most recent input and decided that the input actually did provide a contribution to the task. This means that the dialogue manager can now take steps towards advancing the domain communication with the user. Obviously, what to do in a particular case depends on the task and the sub-task context. The limiting case is that the dialogue manager simply decides that it has understood what the user said and takes overt action accordingly, such as connecting the caller to a user who has been understood to want to accept a collect call, replaying an email message, or displaying a map on the screen. Some other cases are:

Option 2.14.1: More information needed

The dialogue manager inserts the user’s input meaning into a slot in the task model, discovers that more information is needed from the user, and proceeds to elicit that information.

Option 2.14.2: Database look-up

The dialogue manager looks up the answer to the user's question in the database containing the system's domain knowledge and sends the answer to the language and speech generation components (or to the screen, etc.).

Option 2.14.3: Producing an answer

The dialogue manager inserts the user's input meaning into a slot in the task model, verifies that it has all the information needed to answer the user's query, and sends the answer to the language and speech generation components (or to the screen, etc.).

Option 2.14.4: Making an inference

The dialogue manager makes an inference based on the user's input meaning, inserts the result into a slot in a database and proceeds with the next question. In Waxholm, for instance, the system completes the user's 'on Thursday' by inferring the appropriate date, and replaces qualitative time expressions, such as 'this morning', by well-defined time windows, such as '6 AM - 12 AM'. Verbmobil does inferencing over short sequences of user input, such as that a counterproposal (for a date, say) implies the rejection of a previous proposal; a new proposal (for a date, say) implies the acceptance of a (not incompatible but less specific) previous proposal; and a change of dialogue phase implies the acceptance of a previous proposal (for a time, say).

It is important to note that such domain-based inferences abound in human-human conversation. Without thinking about it, human speakers expect their interlocutors to make those inferences. The dialogue manager has no way of replicating the sophistication of human inferencing during conversation and dialogue. Most current systems are able to process only relatively simple inferences. The dialogue manager developer should focus on enabling all and only those inferences that are strictly necessary for the application to work successfully in the large majority of exchanges with users. Even that can be a hard task. Field data may be needed to decide whether a particular type of inference is needed or can be omitted.

Furthermore, it is sometimes unclear which inferences are really necessary. For instance, should the system be able to perform addition of small numbers or not? Simple as this may appear, it would add a whole new chapter to the vocabulary, grammar and rules of inference that the system would have to master. In travel booking applications, for instance, some users would naturally say things like "two adults and two children"; or, in travel information applications, some users may want to know about the 'previous' or the 'following' departure given what the system has already told them. The developer has to decide how important the system's capability of understanding such phrases is to the successful working of the application in real life. In many cases, making an informed decision will require empirical investigation of actual user behaviour.

Finally, through control of user input (see 2.9), the developer must try to prevent the user from requiring the system to do inferences that are not strictly needed for the application or which are too complex to implement.

Option 2.14.5: More constraints needed

The dialogue manager discovers that the user's input meaning is likely to make the system produce too much output information and produces a request to the user to provide further constraints on the desired output.

Option 2.14.6: Inconsistent input

The dialogue manager discovers that the user's input meaning is inconsistent with the database information, infeasible given the database, inconsistent with the task history, etc. The system may reply, e.g., "There is no train from Munich to Frankfurt at 3.10 PM ....", or "The 9 o-clock flight is fully booked already ....".

Option 2.14.7: Language translation

The dialogue manager translates the user's input meaning into another language.

Among the options above, the first four ones and the last one illustrate straightforward progression with the task. The two penultimate options illustrate domain sub-dialogues. Meta-communication sub-dialogues are described in Section 2.15.

Quite often, the system will, in fact, do something more than just advancing the domain communication as exemplified above. As part of advancing the domain communication, the system may provide feedback to the user to enable the user make sure that what the user just said has been understood correctly (see 2.19).

2.15 Issue: Meta-communication

Meta-communication, although secondary to domain communication, is crucial to proper dialogue management. Meta-communication is often complex and potentially difficult to design. In meta-communication design, it is useful to think in terms of distinctions between (a) *system-initiated* and *user-initiated* meta-communication and (b) *repair* and *clarification* meta-communication. These are all rather different from each other in terms of the issues they raise, and distinction between them gives a convenient breakdown of what otherwise tends to become a tangled subject. In general, one of the partners in the dialogue initiates meta-communication because that partner has the impression that something went wrong and has to be corrected.

Note that we do not include system feedback under meta-communication. Some authors do that and there does not seem to be any deep issue involved here one way or the other. We treat feedback as a separate form of system-to-user communication in 2.19. Primarily for the user, feedback (from the system) is the most important means for discovering that something went wrong and has to be corrected.

Option 2.15.1: System-initiated repair meta-communication

System-initiated repair meta-communication is needed whenever the system has reason to believe that it did not understand the user's meaning. Such cases include, i.a.:

- nothing arrived for the dialogue manager to process although meaning input was expected from the user. In order to provide appropriate output in such cases, the dialogue manager must get the user to input the meaning once more. It is worth noting that this can be done in many different ways, from simply saying, e.g., "Sorry, I did not understand" or "Please repeat" to asking the user to speak louder or more distinctly. The more the system knows about the probable cause of its failing to understand the user, the more precise its repair meta-communication can be. Any such gain in precision increases the likelihood that the system will understand the user the next time around and thus avoid error loops (see 2.18);

- something arrived for the dialogue manager to process but what arrived was meaningless in the task context. For instance, the user may be perceived as responding 'London' to a question about departure date. In order to provide appropriate output in such cases, the dialogue manager may have to ask the user to input the meaning once more. However, as the system actually did receive some meaning representation, it should preferably tell the user what it did receive and that this was not appropriate in the task context. This is done by, e.g., Waxholm and the Danish Dialogue System. For instance, if the Danish Dialogue System has understood the user to want to fly from Aalborg to Karup, it will tell the user that there are no flight connections between these two airports. Another approach is taken in RailTel/ARISE. These systems would take the user's 'London' to indicate a change to the point of departure or arrival (see below, this section). Verbmobil uses a statistical technique to perform a form of constraint relaxation in case of a contextually inconsistent user input dialogue act (cf. 2.13).

A core problem in repair meta-communication design is that the user input that elicits system-initiated repair may have many different causes. The dialogue manager often has difficulty diagnosing the actual cause. The closer the dialogue manager can get to correctly inferring the cause, the more informative repair meta-communication it can produce, and the more likely it becomes that the user will provide comprehensible and relevant input in the next turn.

Option 2.15.2: System-initiated clarification meta-communication

System-initiated clarification meta-communication is needed whenever the system has reason to believe that it actually did understand the user's meaning which, however, left some kind of uncertainty as to what the system should produce in response. Such cases include, i.a.:

- a representation of the user's meaning arrived with a note from the speech and/or language processing layers that they did not have any strong confidence in the correctness of what was passed on to the dialogue manager. The best approach for the dialogue manager to take in such cases probably is to get the user to input the meaning once more rather than to continue the dialogue on the basis of dubious information, which may easily lead to a need for more substantial meta-communication later on. Alternatively, as the system actually did receive some meaning representation, it might instead tell the user what it received and ask for the user's confirmation;

- a representation of the user's meaning arrived which was either inherently inconsistent or inconsistent with previous user input. In cases of inherent inconsistency which the system judges on the basis of its own domain representation, the system could make the possibilities clear to the user and ask which possibility the user prefers, for instance by pointing out that 'Thursday 9th' may be either 'Thursday 8th' or 'Friday 9th'. Cases of inconsistency with previous user input are much more diverse, and different response strategies may have to be used depending on the circumstances;

- a representation of the user's meaning arrived which was (semantically) ambiguous or underspecified. For instance, the user asks to be connected to Mr. Jack Jones and two gentlemen with that name happen to work in the organisation; or the user wants to depart at "ten o-clock", which could be either AM or PM. In such cases, the system must ask the user for information that can help resolve the ambiguity. The more precisely this can be done, the better. For instance, if the system believes that the user said either 'Hamburg' or 'Hanover', it should tell the user just that instead of broadly asking the user to repeat. Experience indicates that it is dangerous for the system to try to resolve ambiguities on its own by selecting what the system (i.e. the designer) feels is generally the most likely interpretation. The designer may think, for instance, that people are more likely to take an airplane at ten AM than at ten PM and may therefore assign the default interpretation "ten AM" to users' "ten o-clock". If this approach of interpretation by default is followed, it is strongly advised to explicitly ask the user for verification through a "yes/no" feedback question (see 2.19).

Whilst user meaning inconsistency and (semantic) ambiguity are probably the most common and currently relevant illustrations of the need for system clarification meta-communication, others are possible, such as when the user provides the system with an irresolvable anaphor. In this case, the system should make the possible referents clear to the user and ask which of them the user has in mind.

As the above examples illustrate, system-initiated clarification meta-communication is often a 'must' in dialogue manager design.

In general, the design of system clarification meta-communication tends to be difficult, and the developer should be prepared to spend considerable effort on reducing the amount of system clarification meta-communication needed in the application. This is done by controlling the user's input and by providing cooperative system output. However, as so often is the case in systems design, this piece of advice should be counter-balanced by another. Speaking generally, users tend to loose attention very quickly when the system speaks. It is therefore no solution to let the system instruct the user at length on what the system really means, or wants, on every occasion where there is a risk that the user might go on to say something which is ambiguous or (contextually) inconsistent. In other words, careful prevention of user behaviour which requires system-initiated clarification meta-communication should be complemented by careful system clarification meta-communication design. One point worth noting is that, for a large class of system-initiated clarifications, yes/no questions can be used.

Option 2.15.3: User-initiated repair meta-communication

User-initiated repair meta-communication is needed whenever the system has demonstrated to the user that is has misunderstood the user's intended meaning. It also sometimes happens that users change their minds during dialogue, whereupon they have to go through the same procedures as when they have been misunderstood by the system. In such cases, the user must make clear to the system what the right input is. Finally, users sometimes fail to get what the system just said. In this case, they have to ask the system to repeat just as when the system fails to get what the user just said. These three (or two) kinds of user repair meta-communication are mandatory in many systems. Examples are Waxholm, the Daimler-Chrysler dialogue manager and the Danish Dialogue System.

User-initiated repair meta-communication can be designed in several different ways.

(a) *Uncontrolled repair input.* Ideally, we would like the users to just speak their minds whenever they have been misunderstood by the system, changed their minds with respect to what to ask of, or tell, the system, or failed to get what the system just said. Some systems do that, such as Waxholm, but with varying success, the problem being that users may initiate repair in very many different ways, from "No, Sandhamn!" to "Wait a minute. I didn't say that. I said Sandhamn!"

(b) *Repair keywords.* Other systems require the user to use specifically designed keywords for this purpose, again with varying success. In the Danish Dialogue System, users are asked to use the keyword 'change' whenever they have been misunderstood by the system or changed their minds, and to use the keyword 'repeat' whenever they failed to get what the system just said. Keywords are simpler for the system to handle than unrestricted user speech. The problem is that users sometimes fail to remember the keywords they are supposed to use. The more keywords users have to remember, the higher the risk that they forget them. For walk-up-and-use systems, something in the order of two to three keywords would seem to be the maximum users can be expected to remember. Even that may be demanding too much of the user if the keywords must be used in situations in which keywords are unnatural and the user is already cognitively overloaded.

(c) *Erasing.* A third approach is used in RailTel/ARISE. This approach is similar to using an eraser: one erases what was there and writes something new in its place. For instance, if the system gets 'Frankfurt to Hanover' instead of 'Frankfurt to Hamburg', the user simply has to repeat 'Frankfurt to Hamburg' until the system has received the message. No specific repair meta-communication keywords or meta-dialogues are needed. The system is continuously prepared to revise its representation of the user's input based on the user's latest utterance. Whilst this solution may work well for low-complexity tasks, it will not work for tasks involving selective input prediction (see 2.11) and may be difficult to keep track of in high-complexity tasks.

Option 2.15.4: User-initiated clarification meta-communication

User -initiated clarification meta-communication is probably the most difficult challenge for the meta-communication designer. Just like the user, the system may output, or appear to the user to output, inconsistent or ambiguous utterances, or use terms which the user is not familiar with. In human-human conversation, these problems are easily addressed by asking questions such as: "What do you mean by <u>green departure</u>?" or "Do you mean <u>scheduled</u> arrival time or <u>expected</u> arrival time?" Unfortunately, most current SLDSs are not being designed to handle such questions at all. The reasons are (a) that this is difficult to do and, often more importantly, (b) that the system developers have not discovered such potential problems in the first place. If they had, they might have tried to avoid them in their design of the system's dialogue behaviour, i.e. through user input control. Thus, they would have made the system explain the notion of a green departure before the user was likely to ask what it is, and they would have made the system explicitly announce when it was speaking about scheduled arrivals and when is was speaking about expected arrivals. In general, this is one possible strategy to follow by the dialogue manager developer: to remove in advance all possible ambiguities, inconsistencies and terms unknown to users, rather than to try to make the system handle questions from users about these things. A tool is being developed in DISC to support the design of co-operative system dialogue [Dybkjær 1999]. Part of the purpose of this tool is to avoid situations in which users feel compelled to initiate clarification meta-communication.

There is an obvious alternative to the strategy recommended above of generally trying to prevent the occurrence of user-initiated clarification meta-communication. The alternative is to strengthen the system's ability to handle user-initiated clarification meta-communication. The nature of the task is an important factor in determining which strategy to follow or emphasise. Consider, for instance, users inquiring about some sort of yellow pages commodity, such as electric guitars or used cars. Both domains are relatively complex. In addition, the inquiring users are likely to differ widely in their knowledge about electric guitars and cars. A flight ticket reservation system may be able to address its domain *almost* without using terms which are unknown to its users, whoever these may be. Not so with a used cars information system. As soon as the system mentions ABS brakes, racing tyres or split back seats, some users will be wondering what the system is talking about. In other words, there seems to be a large class of potential SLDSs which can hardly start talking before they appear to speak gibberish to some of their intended users. In such cases, the dialogue manager developers have better prepare for significant user-initiated clarification meta-communication. It is no practical option for the system to explain all the domain terms it is using as it goes along. This would be intolerable for users who are knowledgeable about the domain in question.

To summarise, the dialogue manager is several steps removed from direct contact with the user. As a result, the dialogue manager may fail to get the user's meaning or it may get it wrong. Therefore, both the system and the user need to be able to initiate repair meta-communication. Even at low levels of task complexity, users are able to express themselves in ways that are inconsistent or ambiguous. The system needs clarification meta-communication to handle those user utterances. In some tasks, user clarification meta-communication should be prevented rather than allowed. In other tasks, user clarification meta-communication plays a large role in the communication between user and system.

2.16 Issue: Other forms of communication

Domain communication including feedback (see 2.19) and meta-communication are not the only forms of communication that may take place between an SLDS and its users. Thus, the domain-independent opening of the dialogue by some form of greeting is neither domain communication nor meta-communication. The same applies to the closing of the dialogue (see 2.20). These formalities may also be used in the opening and closing of sub-dialogues. Another example is system time-out questions, such as "Are you still there?", which may be used when the user has not provided input within a certain time limit.

Option 2.16.1: Handling out-of-domain terms

If the SLDSs task-delimitation is not entirely natural and intuitive to users, users are likely to sometimes step outside the system's unexpectedly limited conception of the task. By the system's definition, the communication then ceases to be domain communication. For some tasks, users' out-of-domain communication may happen too often for comfort for the dialogue manager developer who may therefore want to do something about it. Thus, Waxholm is sometimes able to discover that the user's input meaning is outside the domain handled by the system. This is a relatively sophisticated thing to do because the system must be able to understand out-of-domain terms. Still, this approach may be worth considering in cases where users may have reason to expect that the system is able to handle certain sub-tasks which the system is actually unable to deal with. When the users address those sub-tasks, the system will tell them that, unfortunately, it cannot help them. In the Verbmobil meeting scheduling task, users are prone to produce reasons for their unavailability on certain dates or times. Verbmobil, however, whilst being unable to understand such reasons, nevertheless classifies them and represents them in the topic history (see 2.21).

2.17 Issue: Expression of meaning

Once the system has decided what to say to the user, this meaning representation must be turned into an appropriately expressed output utterance. In many cases, this is being done directly by the dialogue manager. Having done its internal processing jobs, the dialogue manager may take one of the following approaches, among others:

Option 2.17.1: Pre-recorded utterances

The dialogue manager selects a stored audio utterance and causes it to be played to the user by sending a message to the player.

Option 2.17.2: Concatenation of pre-recorded words and phrases

The dialogue manager concatenates the output utterance from stored audio expressions or phrases and causes it to be played to the user by sending a message to the player.

Option 2.17.3: Filling in a template used by a synthesiser

The dialogue manager selects or fills an output sentence template and causes it to be synthesised to the user.

Option 2.17.4: Producing meaning

A more sophisticated approach is to have the dialogue manager produce the *what,* or the meaning, of the intended output and then have the output language layer determine the *how,* or the form of words to use, in the output. In this approach, the how is often co-determined by accompanying constraints from the dialogue manager's control and context layers, such as that the output should be a question marked by rising pitch at the end of the spoken utterance.

The first two options are closely related and are both used in, e.g., the Danish Dialogue System. Waxholm and the Daimler-Chrysler dialogue manager use the third option. This option is compatible with relatively advanced use of control layer information for, e.g., determining the prosody of the spoken output. This can also be done in the first approach but is difficult to do in the second approach because of the difficulty of controlling intonation in concatenated pre-recorded speech.

2.18 Issue: Error loops and graceful degradation

An important issue for consideration by the dialogue management developer is the possibility that the user simply repeats the utterance which caused the system to initiate repair meta-communication. The system may have already asked the user to speak louder or to speak more distinctly but, in many such cases, the system will be in exactly the same uncomprehending situation as before. The system may try once more to get out of this potentially infinite loop but, evidently, this cannot go on forever. In such cases, the system might either choose to fall back on a human operator or close the dialogue. To avoid that, a better strategy is in many cases for the system to attempt to carry on by changing the level of interaction into a simpler one, thereby creating a 'graceful degradation' of the (domain or meta-) communication with the user. Depending on the problem at hand and the sophistication of the dialogue manager, this can be done in many different ways:

Option 2.18.1: Focused questions

The user may be asked focused questions one at a time instead of being allowed to continue to provide one-shot input which may be too lengthy or otherwise too complex for the system to understand. For instance, the system goes from saying "Which information do you need?" to saying "From where do you want to travel?" This is a common approach used in, i.a., Waxholm and the Daimler-Chrysler dialogue manager.

Option 2.18.2: Asking for re-phrasing

The user may be asked to re-phrase the input or to express it more briefly, for instance when the user's answer to a focused question is still not being understood.

Option 2.18.3: Asking for a complete sentence

The user may be asked to produce a complete sentence rather than grammatically incomplete input, as in Waxholm.

Option 2.18.4: Yes/no questions

The user may be asked to answer a crucial question by 'yes' or 'no'.

Option 2.18.5: Spelling

The user may be asked to spell a crucial word, such as a person name or a destination.

It is important to note that the levels of interaction/graceful degradation approach can be used not only in the attempt to get out of error loops but also in combination with system-initiated clarification meta-communication. So, generalising, whenever the system is uncertain about the user's meaning, graceful degradation may be considered. The Daimler-Chrysler dialogue manager standardly accepts three repetitions of a failed user turn before applying the graceful degradation approach. This seems reasonable.

2.19 Issue: Feedback

System feedback to users is essential to successful dialogue management. In order to be clear about what system feedback involves, it is convenient to distinguish between two kinds of feedback, information feedback and process feedback.

Note that we do not include system feedback under meta-communication (see 2.15). Some authors do that and there does not seem to be any deep issue involved here one way or the other. We treat system feedback as a separate form of system-to-user communication.

Option 2.19.1: Information feedback

The user must have the opportunity to verify that the system has understood the user's input correctly. In general, the user should receive feedback on each piece of information which has been input to the system. The feedback needs not be communicated through speech. Waxholm, for instance, provides a textual representation on the screen of what the system has recognised as well as of the system's output response. The important thing is that the user can perceive the feedback and verify if what the system did was what the user intended the system to do by providing a certain input. So the system's feedback may consist in presenting a particular map on the screen, or a table packed with information of some kind or other, or in playing a certain voice mail which it believes that the user has asked for. In many cases, however, the feedback will be speech produced by the system. Imagine the following dialogue:

Dialogue 1

S1: "ADAP Travels, can I help you?"

U1: “When is the first morning train from Frankfurt to Hamburg tomorrow morning?”

S2: “5.35 AM”.

U2: “Many thanks. Goodbye.”

S3: “Goodbye”.

The conversation looks OK, doesn’t it? However, the problem is that the system misrecognised ‘Hamburg’ as ‘Hanover’! Dialogue 1, therefore, is a transaction failure (see Chapter 3). Moreover, the transaction failure is in this case a result of sloppy dialogue design. A better solution would be:

Dialogue 2

S1: “ADAP Travels, can I help you?”

U1: “When is the first morning train from Frankfurt to Hamburg tomorrow morning?”

S2: “The first train from Frankfurt to Hanover on 3rd May 1999 leaves at 5.35 AM”.

U2: [Initiates repair meta-communication.]

S3: ....

The system feedback in the example above is relatively straightforward to design. One thing to consider is whether the system should repeat *twice* some or all of its core information to the user, such as in saying “... 5.35 AM, 5.35 AM”. However, the feedback strategy in Dialogue 2 cannot always be trusted. What may happen is the following transaction failure:

Dialogue 3

S1: “ADAP Travels, can I help you?”

U1: “When is the first morning train from Frankfurt to Hamburg tomorrow morning?”

S2: “The first train from Frankfurt to Hanover on 3rd May 1998 leaves at 5.35 AM”.

U2: “Thank you.”

S3: “Do you need any other information?”

U3: ....

In Dialogue 3, the user simply ignores the system’s feedback and happily walks away with what is likely to be false information, just as in Dialogue 1. If the users turn out to be too prone to make the error illustrated in Dialogue 3, the more burdensome strategy shown in Dialogue 4 might be considered:

Dialogue 4

S1: “ADAP Travels, can I help you?”

U1: “When is the first morning train from Frankfurt to Hamburg tomorrow morning?”

S2: “You want to go from Frankfurt to Hanover?”

U2: [Initiates repair meta-communication.]

S3: ....

Experience has shown that the strategy illustrated in Dialogue 4 is more robust than the strategy illustrated in Dialogue 3 [Sturm et al. 1999]. The price to be paid for adopting the Dialogue 4 strategy is that the user and the system have to spend more dialogue turns to solve the task. The Dialogue 4 strategy can be considered an early step in graceful degradation (2.18).

The amount and nature of the feedback the system should give to the user also depends on factors such as the cost and risk involved in the user-system transaction. Feedback on important bank transfers or costly travels are obviously more critical than feedback on which email the system should now be reading to the user. Current opinion probably is that the dialogue manager developer should prefer the safer among the two most relevant feedback options. Even travel information, if the user gets it wrong, can have serious consequences for that user.

For important transactions, an additional safeguard is to give the user a full summary of the agreed transaction at the end of the dialogue, preceded by a request that the user listens to it carefully. If this request is not there, the user who has already ignored crucial feedback once, may do so again. The additional challenge for the dialogue designer in this case is of course to decide what the system should do if the user discovers the error only when listening to the summarising feedback. One solution is that the system goes through the core information item by item asking yes/no questions of the user until the error(s) have been found and corrected, followed by summarising feedback once again.

Option 2.19.2: Process feedback

SLDS dialogue manager developers may also consider to provide process feedback.

Process feedback is meant to keep the user informed that the system is "still in the loop", i.e. that it has not gone down but is busy processing information. Otherwise, the user may, e.g., believe that the system has crashed and decide to hang up, wonder what is going on and start asking questions, or believe that the system is waiting to receive information and start inputting information which the system does not want to have. All of these user initiatives are, or can be, serious for the smooth proceeding of the dialogue.

Process feedback in SLDSs is still at an early stage. It is quite possible for today's dialogue manager designers to come up with new, ingenious ways of providing the needed feedback on what the system is up to when it does not speak to the user and is not waiting for the user to speak. The best process feedback need not be spoken words or phrases but might be grunts or ehm's, tones, melodies, or appropriate earcons. Waxholm tells the user, for instance, "I am looking for boats to Sandhamn.", thereby combining information feedback and process feedback. In addition, Waxholm uses its screen to tell the user to wait whilst the system is working.

2.20 Issue: Closing the dialogue

Depending on the task, the system's closing of the dialogue may be either a trivial matter, an unpleasant necessity, or a stage to gain increased efficiency of user-system interaction.

Option 2.20.1: Closing (only) when finished

Closing the dialogue by saying, e.g. "Thank you. Good bye." is a trivial matter when the task has been solved and the user does not need to continue the interaction with the system. Users often hang up without waiting for the system's farewell.

In some cases, however, when the user has solved a task, the dialogue manager should be prepared for the possibility that the user may want to solve another task without interrupting the dialogue. This may warrant asking the user if the user wants to solve another task. Only if the user answers in the negative should the system close the dialogue.

Option 2.20.2: Closing when failing

Closing the dialogue is a dire necessity when the system has spent its bag of tricks to overcome repeated error loops (2.18) and failed, or when the system hands over the dialogue to a human operator. In the former case, the system might ask the user to try again.

History, Users, Implementation

2.21 Issue: Histories

As soon as task complexity in terms of information volume exceeds one piece of information, the dialogue manager may have to keep track of the history of the interaction. 'Dialogue history' is a term which covers a number of different types of dialogue records which share the function of incrementally building a dialogue context for the dialogue manager to use or put at the disposal of the language and speech layers (2.6, 2.13). Note that a 'dialogue history' is *not* a log file of the interaction but a dedicated representation serving some dialogue management purpose. Note also that a 'dialogue history' may be a record of some aspect of the entire (past) dialogue or it may be a record only of part of the dialogue, such as a record which only preserves the two most recent dialogue turns. In principle, a dialogue history may even be a record of several dialogues whether or not separated by hang-ups. This may be useful for building performance histories (see below) and might be useful for other purposes as well.

Option 2.21.1: Task history

Most applications need a *task history,* i.e. a record of which parts of the task have been completed so far. The task history enables the system to:

- focus its output to the user on the sub-tasks which remain to be completed;
- avoid redundant interaction;
- have a global focus (2.12); and
- enable the user to selectively correct what the system has misunderstood without having to start all over with the task.

The task history does not have to preserve any other information about the preceding dialogue, such as how the user expressed certain things, or in which order the sub-tasks were resolved. If an output screen is available, the task history may be displayed to the user, as in Waxholm. If the user modifies some task parameter, such as altering the departure time, it becomes necessary to remove all dependent constraints from the task history.

Option 2.21.2: Topic history

A *topic history* is in principle more complex than a task history. It is a record of the topics which have come up so far during the dialogue and possibly in which order they have come up. Even low-complexity systems can benefit from partial or full topic histories, for instance for detecting when a miscommunication loop has occurred (2.18) which requires the system to change its dialogue strategy, or for allowing users to do input repair arbitrarily far back into the preceding dialogue. Another thing which a topic history does is to build a context representation during dialogue, which can be much more detailed than the context built through the task history. In spoken translation systems, such as Verbmobil, the context representation provided by the topic history is necessary to constrain the system's translation task.

Option 2.21.3: Linguistic history

A *linguistic history* builds yet another kind of context for what is currently happening in the dialogue. The linguistic history preserves the actual linguistic utterances themselves (the surface language) and their order, and is used for advanced linguistic processing purposes (2.13). Preserving the linguistic history helps the system interpret certain expressions in the user's current input, such as co-references. For instance, if the user says: "I cannot come for a meeting on the Monday", then the system may have to go back through one or more of the user's previous utterances to find out which date 'the Monday' is. The Daimler-Chrysler dialogue manager and Verbmobil, for instance, use linguistic history for co-reference resolution. Compared to the task history and the topic history, a linguistic history is a relatively sophisticated thing to include into one's dialogue manager at present.

Option 2.21.4: Performance history

A *performance history* is rather different from any of the above histories. The system would build a performance history in order to keep track of, or spot relevant phenomena in, the users' behaviour during dialogue. So a performance history is not about the task itself but about how the user handles the task in dialogue with the system. For instance, if the system has already had to resolve several miscommunication loops during dialogue with a particular user, it might be advisable to connect that user with a human operator rather than continue the agony. One way or another, performance histories contribute to building models of users, whether during a single dialogue or during a series of dialogues with a particular user.

Future systems solving high-complexity tasks are likely to include both a task history, a topic history and a linguistic history, as is the case in Verbmobil. For increased usability, they may need a performance history as well.

2.22 Issue: Novice and expert users, user groups

The discussion above has focused on the central importance of the task to the dialogue manager developer. However, developers also have to take a close look at the intended users of the application as part of designing the dialogue manager.

An important issue common to many different dialogue management tasks is the difference between novice and expert users. In most cases, this is a *difference continuum,* of course, rather than an either/or matter. Furthermore, it may sometimes be important to the dialogue manager developer that there are, in fact, two different distinctions between novice and expert users. In particular, someone may be an expert in the domain of the application but a novice in using the system itself. Depending on for which of four user groups (system expert/domain expert, system expert/domain novice, system novice/domain expert, system novice/domain novice) the system is to be developed, the dialogue manager may have to be designed in different ways.

Option 2.22.1: Domain and system experts

If the target user group is domain and system experts only, the developer may be able to impose strict task performance order, a relatively large number of mandatory command keywords, etc., and support use of the system through written instructions, all of which makes the dialogue manager design much easier to do.

Option 2.22.2: System novices

If the target group is walk-up-and-use users who can be expected to be novices in using the system, a much more user-tailored design is required.

Option 2.22.3: Domain and system novices

The need for elaborate, user-tailored design increases even further if the system novices are also domain novices, so that any domain technicality either has to be removed or explained at an appropriate point during dialogue. For instance, even though virtually every user comes close to being a domain expert in travel time-table information, many users do not know what a ‘green departure’ is and therefore have to be told.

Option 2.22.4: Other user groups

Depending on the task and the domain, the dialogue manager developer(s) may have to consider user groups other than novices and experts, such as the visually impaired, users speaking different languages, or users whose dialects or accents create particular problems of recognition and understanding. In the case of dialects or accents, performance history information might suggest that the dialogue manager makes use of the graceful degradation approach (2.18).

Cons

The ‘downside’ of doing elaborate dialogue design for walk-up-and-use users can be that (system) expert users rightly experience that their interaction with the system becomes less efficient than it might have been had the system included special shortcuts for expert interaction.

Given the relative simplicity of current SLDSs, users may quickly become (system) experts, which means that the short-cut issue is a very real one for the dialogue manager developer to consider. The Danish Dialogue System, for instance, allows (system) expert users to by-pass the system's introduction and avoid getting definitions of green departures and the like. Waxholm allows its users to achieve the same thing through its barge-in button. The acceptance of unrestricted user input in RailTel/ARISE means that experienced users are able to succinctly provide all the necessary information in one utterance. Novice users who may be less familiar with the system's exact information requirements, may provide some of the information needed and be guided by the system in order to provide the remaining information.

2.23 Issue: Other relevant user properties

If the task to be solved by the dialogue manager is above a certain (low) level of complexity, the dialogue manager designer is likely to need real data from user interactions with a real or simulated system in order to get the design right at design-time. Important phenomena to look for in this data include the following options.

Option 2.23.1: Standard goals

What are the users' *standard goal(s)* in the task domain? If the users tend to have specific standard goals they want to achieve in dialogue with the system, there is a problem if the system is only being designed to help achieve some, but not all, of these goals. Strict user input control (2.9) may be a solution - but do not count on it to work in all possible conditions! Deep-seated user goals can be difficult or impossible to control. Of course, another solution is to increase the task and domain coverage of the system.

Option 2.23.2: User beliefs

Do the users tend to demonstrate that they have specific *beliefs* about the task and the domain, which may create communication problems? It does not matter whether these user beliefs are true or false. If they tend to be significantly present, they must be addressed in the way the dialogue is being designed.

Option 2.23.3: User preferences

Do the users tend to have specific *preferences* which should be taken into account when designing the dialogue? These may be preferences with respect to, for instance, dialogue sub-task order.

Option 2.23.4: Cognitive loads

Will the dialogue, as designed, tend to impose any strong *cognitive loads* on users during task performance? If this is the case, the design may have to be changed lest the cognitive load makes the users behave in undesirable ways during dialogue. One way to increase users' cognitive load is to ask them to remember to use specific keywords in their interaction with the system; another, to use a feedback strategy which is not sufficiently heavy-handed so that users need to concentrate harder than they are used to doing in order not to risk ignoring the feedback.

Option 2.23.5: Response packages

Other cognitive properties of users that the dialogue manager developer should be aware of include the *response package* phenomenon. For instance, users seem to store some pieces of information together, such as "from A to B". Asking them *from where* they want to go therefore tends to elicit the entire response package. If this is the case, then the dialogue manager should make sure that the user input prediction enables the speech and language layers to process the entire response package.

2.24 Issue: Implementation issues

The issue of dialogue management can be addressed at several different levels of abstraction. In this article, we mostly ignore low-level implementation issues such as programming languages, hardware platforms, software platforms, generic software architecture, database formats, query languages, data structures which can be used in the different system modules, etc. Generic software architectures for dialogue management are still at an early stage. Speaking generally, low-level implementation issues can be dealt with in different ways with little to distinguish between these in terms of efficiency, adequacy etc. Good development tools would appear to be more relevant at this point. A survey of existing dialogue management tools is provided in [Luz 1999].

Sub-issue 1: Architecture and modularity

There is no standard architecture for dialogue managers, their modularity, or the information flow between modules. Current dialogue manager architectures differ, among many other things, with respect to their degree of domain and task independence. The functionality reviewed above may be implemented in any number of modules, the modularity may be completely domain and task dependent or relatively domain and task independent, and the dialogue manager may be directly communicating with virtually every other module in an SLDS or it may itself be a module which communicates only indirectly with other modules through a central processing module. As to the individual modules, Finite State Machines may be used for dialogue interaction modelling, and semantic frames may be used for task modelling. However, other approaches are possible and the ones just mentioned are among the simplest approaches.

Sub-issue 2: Main task of the dialogue manager

If there is a central task which characterises the dialogue manager as a *manager,* it is the task of deciding how to produce appropriate output to the user in view of the dialogue context and the user's most recent input as received from the speech and language layers.

Basically, what the dialogue manager does in order to interpret user input and produce appropriate output to the user is to:

- use the knowledge of the current dialogue context and local and global focus of attention it may possess to:
- *map from the semantically significant units in the user's most recent input (if any), as conveyed by the speech and language layers, onto the sub-task(s) (if any) addressed by the user;*
- *analyse the user's specific sub-task contribution(s) (if any);*
- use the user's sub-task contribution to:
- *execute a series of preparatory actions (consistency checking, input verification, input completion, history checking, database retrieval, etc.) usually leading to:*
- *the generation of output to the user, either by the dialogue manager itself or through output language and speech layers.*

The dialogue management activities just described were discussed in Sections 2.12 - 2.20 and 2.22 - 2.23 above. The analysis of the user's specific sub-task contribution is sometimes called 'dialogue parsing' and may involve constraints from most of the elements in the speech input, language input, context and control layers in Figure 1. In order to execute one or more actions that will eventually lead to the generation of output to the user, the dialogue manager may use, e.g., an AI dynamic planning approach as in Verbmobil, a Finite State Machine for dialogue parsing as in Verbmobil, an Augmented Transition Network as in Waxholm, or, rather similarly, decide to follow a particular branch in a node-and-arc dialogue graph as in the Danish Dialogue System.

As the dialogue manager generates its output to the user, it must also:

- *change or update its representation of the current dialogue context; and*
- *generate whatever constraint-based support it may provide to the speech and language layers.*

These dialogue management activities were described in Sections 2.6, 2.9, 2.11, and 2.21 above.

At a high level of abstraction, what the dialogue manager has to do thus is to apply sets of decision - action rules, possibly complemented by statistical techniques, to get from (context + user input) to (preparatory actions + output generation + context revision + speech and language layer support). For simple tasks, this may reduce to the execution of a transformation from, e.g. (user input keywords from the speech layer) to (minimum preparatory actions + dialogue manager-based output generation including repair meta-communication) without the use of (input or output) language layers, context representations and speech and language layer support. Different dialogue managers might be represented in terms of which increased-complexity properties they add to this simple model of a dialogue manager. Waxholm, for instance, adds semantic input parsing; a statistical topic spotter ranging over input keywords; out-of-domain input spotting; two-level dialogue segmentation into topics and their individual sub-structures; preparatory actions, such as dialogue parsing including consultation of the topic history, and database operations including temporal inferencing; user-initiated repair meta-communication; a three-phased dialogue structure of introduction, main phase, and closing; an output speech layer; advanced multimodal output; and topic history updating.

Sub-issue 3: Order of output to the user

As for the generation of output to the user, a plausible default priority ordering could be:

(i) if system-initiated repair or clarification meta-communication is needed, then the system should take the initiative and produce it as a matter of priority;

(ii) even if (i) is not the case, the user may have initiated meta-communication. If so, the system should respond to it;

(iii) if neither (i) nor (ii) is the case, the system should respond to any contribution to domain communication that the user may have made; and

(iv) then the system should take the domain initiative.

In other words, in communication with the user, meta-communication has priority over domain communication; system-initiated meta-communication has priority over user-initiated meta-communication; user domain contributions have priority over the system's taking the next step in domain communication. Note that feedback from the system may be involved at all levels (2.19). Note also that the above default priority ordering (i) through (iv) is *not* an ordering of the system states involved. As far as efficient processing by the dialogue manager is concerned, the most efficient ordering seems to be to start from the default assumption that the user has made a contribution to the domain communication. Only if this is not the case should (i), (ii) and (iv) above be considered.

Sub-issue 4: Task and domain independence

The fact that dialogue management, as considered above, is task-oriented, does not preclude the development of (relatively) task independent and domain independent dialogue managers. Task and domain independence is always independence in some *respect* or other, and it is important to specify that respect (or those respects) in order to state a precise claim. Even then, the domain or task independence is likely to be limited or relative. For instance, a dialogue manager may be task independent with respect to some, possibly large, class of information retrieval tasks but may not be easily adapted to all kinds of information retrieval tasks, or to negotiation tasks. Modular-architecture, domain and task independent dialogue managers are highly desirable, for several reasons (cf. 2.4). For instance, such dialogue managers may integrate a selection of the dialogue management techniques described above whilst keeping the task model description and the dialogue model description as separate modules. These dialogue managers are likely to work for all tasks and domains for which this particular combination of dialogue management techniques is appropriate. The Daimler-Chrysler dialogue manager and the RailTel/ARISE dialogue manager are cases in point.

Much more could be done, however, to build increasingly general dialogue managers. To mention just a few examples, it would be extremely useful to have access to a generalised meta-communication dialogue manager component, or to a domain independent typology of dialogue acts.

3. Dialogue Management Life-Cycle Model

This chapter introduces the DISC life-cycle model for eliciting information on current practice in dialogue manager development and evaluation.

At a high level of abstraction, any standard software engineering life cycle model applies to the development and evaluation of dialogue managers. However, as such models are aimed at describing software development processes in general, they do not specialise to the development and evaluation processes which are specific to particular classes of systems or system components, such as SLDSs or dialogue managers. In addition, general software engineering life cycle models do not include advise on the methods and tools to be used when developing, e.g., dialogue managers.

Figure 1 (from [Bernsen et al. 1998a]) shows a general software engineering life cycle model which has been slightly specialised to the development and evaluation of SLDSs and their components. The figure shows an overall framework for the development and evaluation process through to installation at the user's site. After this point there may be maintenance of the software, the software may be ported to other platforms, or it may be adapted to new applications in which case a new life cycle starts.

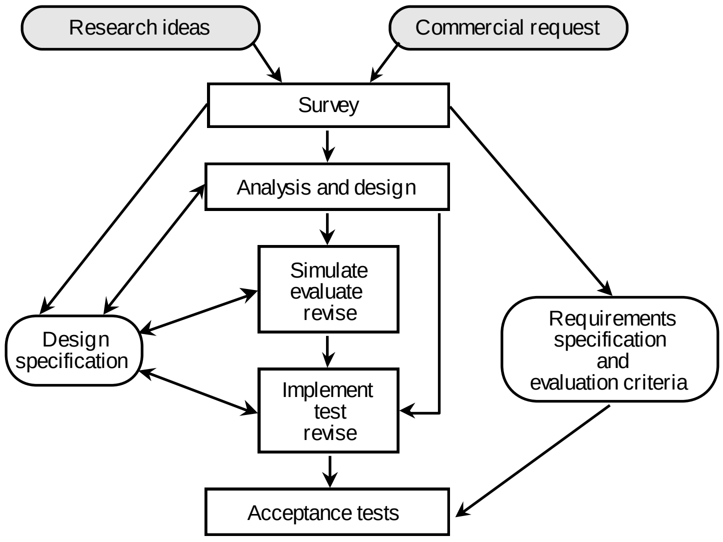

Figure 1. A software engineering life cycle model for the development and evaluation of SLDSs and components. Rectangular boxes show process phases. The development and evaluation process is iterative within each phase and across phases. Arrows linking phases indicate the overall course of the process. The requirements specifications and evaluation criteria, and the design specification (rounded white boxes) are used throughout the development process. Rounded grey boxes indicate that the system being developed may be either a research system or a commercial system.

The DISC dialogue engineering life cycle model draws on a general software engineering life cycle model but is specialised to capturing the process of developing and evaluating SLDSs and components. The DISC life cycle model asks a series of fairly detailed questions which are all related to the model in Figure 1, go beyond that model into, e.g., maintenance, and are specialised to dialogue engineering.

The DISC life cycle model is intended to be used no matter if one describes an entire SLDS or a single component. This is possible because each piece of software has a life cycle and most life cycle issues are relevant independently of the specific nature of the software. The precise contents of the questions to ask on each issue may, however, depend on the specific nature of the software. For example, the issue of evaluation is highly relevant to both speech recognisers and dialogue managers. However, the precise tests to carry out are very different. In the following we address the DISC life cycle issues with special emphasis on dialogue management.

3.1 Overall design goal(s)

What is the general purpose(s) of the design process?

Dialogue managers have so far typically been designed as part of a particular (commercial or research prototype) SLDS. Probably for this reason focus has rarely been on dialogue management for its own purpose but rather on dialogue management as one of several components which have to work together to form an SLDS. In some cases focus has actually not been on dialogue management at all. The dialogue manager was built only because it is a necessary part of an SLDS and hence a necessary evil if one wants to study, e.g., speech recognition in an SLDS environment. However, there are now ongoing efforts world-wide to develop dialogue managers which are more general-purpose and which may fairly easily be adapted to a new task domain, such as the Daimler-Chrysler dialogue manager, and the dialogue manager embedded in the Philips SpeechMania system [Aust et al. 1995]. In addition to the overall purpose of having a dialogue manager, there may be several different design goals for the dialogue manager, such as to explore a particular approach to dialogue management, study certain aspects of the dialogue management component, ensure multilinguality or focus on modularity in the case of distributed development.

3.2 Hardware constraints

Were there any a priori constraints on the hardware to be used in the design process?

Typically there are no a priori hardware constraints on the dialogue manager. The hardware is more likely to be determined by, e.g., what is available at the sites involved, economical considerations as regards the procurer of the SLDS in question, performance demands on the system (e.g. real-time), or platform decisions.

3.3 Software constraints

Were there any a priori constraints on the software to be used in the design process?

Since dialogue managers so far typically have been developed as one-of-a-kind, software constraints have not been dictated by already existing pieces of software which should function as (part of) the dialogue manager. However, platform decisions or already existing modules to be included in an SLDS may impose constraints/desiderata on which software to use for dialogue manager development. If a dialogue manager already exists and will be adapted to a given application, the only constraint is that the programming language in which it is written must be able to run on the platform on which the dialogue manager will be adapted or further developed.

3.4 Customer constraints

Which constraints does the customer (if any) impose on the system/component? Note that customer constraints may overlap with some of the other constraints. In that case, they should only be inserted once, i.e. under one type of constraint.

Many different customer constraints may be imposed on the development of a dialogue manager. Typical constraints relate to resources, type of interface, software, protocols, existing hardware, existing databases, other systems already in use, and ordinary business processes. Currently, there is a strong trend towards PC-based solutions. Customer constraints must be collected by talking to relevant people from the customer organisation.

For research projects there are typically no real customer. At best the research group has contact to an organisation which potentially could be a real procurer or end-user and which is willing to give relevant information and feedback to the project. “Customer constraints” must in such cases be deduced to the extent possible from the input provided by the contact organisation.

The dialogue manager must be able to handle the task(s) specified and agreed on with the customer and thus needs access to relevant task domain information. Moreover, it must possess the rules relevant for handling the information. For commercial projects this sometimes causes problems. In some cases, companies are not interested in revealing internal data that is basic to their offer, and the system developers may thus not have access to the entire task structure. This means that part of the task model and the inferences involved have to be made external to the dialogue manager.

3.5 Other constraints

Were there any other constraints on the design process?

Several constraints other than hardware, software and customer constraints may influence the dialogue manager life cycle. Precisely what to mention under other constraints depends on what has already been mentioned under the above constraint categories. For example, cost and development time will often be customer constraints. Other examples of constraints are personpower, development phases, standards conformation, and knowledge in the developer team. Other constraints may be imposed either indirectly by the customer organisation (if imposed directly it would be customer constraints), or they may be imposed by the organisation/department developing the dialogue manager.

3.6 Design ideas

Did the designers have any particular design ideas which they would try to realise in the design process?

The design ideas underlying the dialogue manager design are typically related to the overall design goal. Thus, if the focus is *not* on dialogue management, the idea may simply be to create a dialogue manager which works sufficiently well to serve as part of an entire SLDS, allowing investigation of issues not directly related to dialogue management. If dialogue management is a focal issue, examples of design ideas could be to create a generic dialogue manager, to explore modularity, to investigate different ways in which the dialogue manager can support the language layers, to explore co-operativity, to experiment with different ways in which to control dialogue, or to explore ways in which to exploit contextual information to improve the dialogue.

3.7 Designer preferences

Did the designers impose any constraints on the design which were not dictated from elsewhere?

Designer preferences may influence the development process by introducing new constraints. Such preferences may relate to many different issues, such as preferred development platform, programming language, or positive experience in doing things in a certain way and following certain development methodologies. For example, there may be a preference for using rapid prototyping in developing the dialogue manager in order to produce early demonstrations.

3.8 Design process type

What is the nature of the design process?

The dialogue manager design process type has so far mainly been exploratory research, either with a focus on dialogue management or on some other SLDS component embedded in a system of which the dialogue manager forms part. For dialogue managers which form part of commercial systems, the design process type has mostly been that of one-of-a-kind product development. In the cases where an already existing dialogue manager has been used in a new application, the design process type has been, e.g., adaptation, optimisation or redesign. For the emerging generic dialogue managers, the design process will typically reduce to focusing on the task and domain specific parts, i.e. those parts which are not general but depends on the concrete task and therefore have to be tailored to each particular application.

3.9 Development process type

How was the dialogue manager developed?

Dialogue managers, in particular in research projects, have often been developed through Wizard of Oz (WOZ) prototyping. WOZ is a relatively costly development method. On the other hand, by producing data on the interaction between a (fully or partially) simulated system and its users, WOZ provides the basis for early tests of, i.a., the dialogue model and its feasibility, as well as of the coverage and adequacy of requirements prior to implementation. It is fairly costly to develop dialogue managers, and in research projects dialogue managers are typically rather complex and meant to explore new areas, thus running a high risk that the first design is relatively far from what is technologically feasible or desirable from a user's point of view. In such cases it may be advisable to use WOZ. WOZ may also be used to test whether there is a market for a certain application before money is spent on developing it [Basson et al. 1996].

For simple dialogues and in most industrial settings, WOZ is normally replaced by an implement-test-and-revise approach based on emerging development platforms. Whether or not WOZ is preferable to implement-test-and-revise depends on several factors, such as the novelty of the development objectives relative to the skills, methods and tools available to the developers, the complexity of the interaction model to be designed, the task domain, and the risk and cost of implementation failure. Low complexity speaks in favour of directly implementing the interaction model without interposing a simulation phase, especially if the developers have built similar systems before. High complexity, on the other hand, may advocate iterative simulations.

Very slowly, tools are emerging which has WOZ support as an integrated part of the development environment [Dybkjær and Failenschmid 1999]. This means that the parts developed for WOZ experiments can be immediately re-used for systems development, thus saving effort and resources.

3.10 Requirements and design specification documentation

Is one or both of these specifications documented? Describe the specifications.

Identified goals and constraints are represented in a *requirements specification* document. The purpose of this document is to eventually list all the agreed requirements which the envisaged dialogue manager should meet. As the dialogue manager is usually part of an SLDS, the requirements specification is normally made for the SLDS as a whole and not separately for the dialogue manager but of course it will contain requirements specifically related to the dialogue manager.

The purpose of the *design specification* is to describe and eventually make operational how to build a dialogue manager which will satisfy the requirements specification and meet the evaluation criteria. The design specification, therefore, will be clearly related to, and may simply include, the requirements specification.

Befitting their exploratory purpose, the framework for research systems development is often loosely defined compared to that of commercial systems. The requirements specification does not serve contractual purposes. This means that requirements can be more easily modified later in the development process because there is no procurer who has to approve of the changes made.

The design specification must be sufficiently detailed to serve as a basis for implementation and, in contrast to the requirements specification, is often expressed in a formal or semi-formal language understood by the systems developers and serving its operational purpose.

To serve its purpose, the design specification must be constantly updated to include the most recent additions and revisions. If this is to be done systematically and coherently, an explicit representation of the changing and accumulating design decisions is needed for keeping track of the development process. This is good engineering practice but difficult to do. If it is not done, it becomes hard or even impossible to (i) keep track of the design decisions that have been made and why they were made, (ii) explain to new developers joining the team what is going on, and (iii) carry out informed maintenance and re-engineering once the project has been completed.

For more information, see e.g. [Bernsen et al. 1998a].

3.11 Development process representation

Has the development process itself been explicitly represented in some way? How?

Dialogue manager development processes differ widely with respect to the extent to which a development process representation has been made and how much of it is available in any systematic fashion. What is typically publicly available are bits and pieces found in scientific papers. Internal reports and working papers may contain more detailed representations of the dialogue manager development process, meeting protocols and minutes, early drafts, sketches, diagrams, scenarios, transcriptions, task analyses, protocols from experiments and other internal documents may exist as well. Technical reports and working papers may be publicly available from research projects whereas they are usually hard to obtain from commercial projects. However, internal papers and documents are usually inaccessible no matter if they are from research projects or from commercial projects. This means that it may be hard to draw on the experiences of other external groups as regards development process representation. However, it is strongly recommended to explicitly and thoroughly document the development process. The advantages of explicit development process representations are that these can be re-used, possibly in revised form, in new projects and with new developers coming on the team, and can support re-design and maintenance. The main disadvantage is that creating them represents an additional cost. However, this additional cost may easily be regained in terms of saved effort of new people joining the development team or saved effort in other projects through re-use.

3.12 Realism criteria

Will the dialogue manager meet real user needs, will it meet them better, in some sense to be explained (cheaper, more efficiently, faster, other), than known alternatives, is the dialogue manager "just" meant for exploring specific possibilities (explain), other (explain)?

Dialogue managers are typically viewed as part of particular SLDSs. Most SLDSs have something to do with real user needs. However, to appropriately address real user needs, the development process often needs to include extended end-user contact, extensive work on domain delimitation, clear up-front evaluation criteria, extended quantitative and qualitative evaluation throughout the development process, an explicit development methodology, etc.

In research projects, focus is typically on exploration and realism criteria are not being considered hard constraints, that is, one may deviate from the realism criteria to the extent needed to achieve other, more important project goals, such as exploring how the dialogue manager can best support other parts of the SLDS, trying out a new programming language specially meant for dialogue management development, or experimenting with the dialogue manager using various kinds of histories to improve its performance, cf. 2.21.

For dialogue managers which form part of commercial applications, the realism criteria may be related to, for example, cost-savings, making the performance of a task faster and more efficient, increased quality of service, or making sure that the interface is in accordance with some system(s) already in use to ensure as little learning overhead as possible.

3.13 Functionality criteria

Which functionalities should the dialogue manager have (this entry expands the overall design goals)?

The overall function of a dialogue manager is typically to control the dialogue performed by an SLDS. In addition, several other functionalities may be in focus. These criteria may address needs or requirements or desiderata of the procurer and the end-users as well as of the developers. Examples of functionality criteria are that the dialogue manager should be language independent, be modular, be able to understand and initiate meta-communication, be able to handle user initiative, have a sufficient coverage of the selected task domain, and, within this domain, be able to conduct a dialogue and perform the selected task(s) with its users. This includes being able to assign an interpretation to the input it receives, being able to decide how the dialogue may best continue, and being able to plan an adequate system utterance whenever it is the system's turn to speak.

3.14 Usability criteria

What are the aims in terms of usability?

Since dialogue managers are usually embedded in SLDSs, there are two usability aspects to consider. One aspect is the usability criteria as seen from the *system developer's* point of view. For the system developer, usability criteria typically relate to issues such as how easy it is to embed the dialogue manager component into the SLDS, how easy it is to modify the component, and whether it will be able to run in different environments. For generic dialogue managers, issues such as ease of adaptation to new task domains and ease of portability are highly important.

The second aspect is the *users'* point of view. Users will not experience the dialogue manager as a stand-alone component but only as part of an entire system. To a large extent, however, it is the dialogue manager which is responsible for how easy the SLDS is to use, whether it is able to perform a dialogue perceived by the users to be natural, flexible and robust within its domain, whether sufficient meta-communicative facilities are available, whether the delimitation of the domain has been done in a reasonable and understandable way, whether the selected domain is sufficiently covered so that users can get the information they need, etc.

3.15 Organisational aspects

Will the dialogue manager have to fit into some organisation or other, how?

Only if the dialogue manager is developed as a (commercial) "stand-alone" component may organisational aspects be relevant. In this case, the dialogue manager must be useful to the organisation and fit its business programme.

For research projects and projects focusing on entire systems rather than on the dialogue manager, organisational aspects are either not applicable or must be seen in relation to the entire SLDS. The purpose of an SLDS will often be to support, partially replace or improve a service which has traditionally been carried out by humans. Given the core role of the dialogue manager in SLDSs, organisational aspects may thus easily affect its design. For example, the new SLDS may have to be able to work together with some already existing system, it may need to draw on an existing database, or there may be certain requirements on the interface, i.e. on the system-user communication, determined by organisational standards or desiderata.

3.16 Customer(s)

Who is the customer for the system/component (if any)?

So far, dialogue managers mostly have been components that were developed for specific SLDSs. For research prototypes there is usually no customer. At best there are contacts to potential customers who can provide input and feedback on the design. For industrial SLDSs, typical customers are telecoms, railway companies, banks and insurance companies, and companies that want, e.g., an automatic switchboard system. This situation is rapidly changing, however, towards a much larger variety of customers who, for instance, may want SLDSs for their Internet pages. Also, car companies are increasingly playing a central role in the exploitation of SLDSs.

If a generic and stand-alone dialogue manager has been developed, the customer will be the organisation which makes use of the dialogue manager for new applications. The dialogue manager may be part of a generic system which is only sold as a whole, i.e. the dialogue manager is not sold as a stand-alone module. This is the case, i.a., for the Philips SpeechMania system. In other cases, individual components may be developed by the same company and sometimes exploited together, in-house or externally, and sometimes sold as individual components to, e.g., other companies who may want to use different off-the-shelf components in the SLDSs they are building. This is the approach taken by e.g. Daimler-Chrysler.

3.17 Users

Who are the intended users of the dialogue manager?

Direct users of dialogue managers are system developers using dialogue managers for SLDS applications. These users may be familiar as is the case in SLDS development project collaborations in which a one-of-a-kind dialogue manager for that particular SLDS is being developed. For generic dialogue managers which are being sold to external organisations, the users are not known in advance but the intended user group is still systems developers.

Indirect users (in the sense that they use the entire SLDS and not only the dialogue manager) are often the broad public. For example, the person calling a switchboard system, a telephone service or a train timetable information system may be anybody. Thus many SLDSs are walk-up-and-use systems requiring no particular skills in advance. The main restriction often is the language as SLDSs are not language independent but require the user to speak, e.g., French. The dialogue manager viewed as a separate component may be language independent but not the SLDS.

3.18 Developers

How many people took significant part in the development? What were their backgrounds?

Often, several developers are involved in dialogue manager development simply because of the complexity of the issues involved. The backgrounds of the developers may vary considerably, including engineering, computer science, linguistics and psychology. The reason for this may be found in the fact that dialogue management development can hardly be called a well-established discipline since the area is fairly new. Moreover, it draws on many different resources and skills, including, e.g., knowledge of user behaviour, linguistic knowledge and implementational skills. The team developing the dialogue manager needs these different skills. Thus it may be quite practical to have people with different backgrounds on the development team. However, current skills are more important than educational background. For example, a computer scientist who has gained the needed linguistic knowledge may easily make it up for a linguist in the team.

3.19 Development time

When was the dialogue manager developed? What was the actual development time for the component (estimated in person/months)? Was that more or less than planned? Why?

Development of anything but the simplest dialogue manager is fairly time consuming. In many research projects a dialogue manager is being developed over a three year period which is the standard duration of a project. However, much more than three person/years of effort is spent on the development. So far, developers of dialogue managers for advanced SLDSs have had little previous experience to draw upon because the area is so new and there are no guidelines on how to do it efficiently. This means that the development of a dialogue manager includes much trial-and-error experimentation and comparatively little routine-based work.

Dialogue managers for simple systems which are system-driven and based on single word recognition, such as some telephone service systems and switchboard systems, are of course much easier to construct and thus much less time consuming than those for advanced SLDS taking spontaneous spoken input.

In fact there are now tools, such as the CSLU Toolkit [http://cslu.cse.ogi.edu], which enable fast and easy design of primitive dialogue managers. More advanced dialogues can be specified using the Philips SpeechMania [Aust et al. 1995]. SpeechMania includes a generic dialogue manager. It has been claimed that if one has attended a course it should be possible to design a dialogue in a week's time. However, if one wants to develop a new dialogue manager (and not only a new dialogue drawing on existing tools and environments) it is much more time consuming as mentioned above.

3.20 Requirements and design specification evaluation

Were the requirements and/or design specifications themselves subjected to evaluation in some way, prior to dialogue manager implementation? If so, how?

There is often no systematic and explicit evaluation of the requirements and design specifications. In research projects these documents often only exist in some rudimentary form but even in industrial projects where such documents do exist, evaluation is typically not being done.

The dialogue manager has to be able to manage the appropriate dialogue structures and otherwise satisfy the specified constraints. However difficult this may be to do in any formal way, it is essential to good development practice to carry out a systematic and explicit evaluation of whether the goals and constraints are reasonable, feasible and non-contradictory. It is much easier and much cheaper to make changes, or even decide to cancel a project, at this early stage than it is when implementation has already started.

3.21 Evaluation criteria

Which quantitative and qualitative performance measures should the dialogue manager satisfy?

In addressing the issue of dialogue manager evaluation, it is important to keep two different situations in mind. The first one concerns evaluation of the SLDS of which the dialogue manager forms a part. The second concerns evaluation of the dialogue manager itself. Dialogue manager evaluation takes place in both situations. However, the first situation obviously makes it harder to exactly distinguish between those aspects of the SLDS's performance which are due to the dialogue manager and which aspects are due to the performance of other system components. Poor speech recognition, for instance, can only to a certain extent be counter-balanced by good dialogue manager design. If the speech recognition is too poor, the users will walk away even if they are dealing with a brilliant dialogue manager. The second situation may be one in which the dialogue manager is being evaluated as part of SLDS development or it may be one in which the dialogue manager itself is the sole development target. In both cases, the evaluation criteria involved are likely to be much more dialogue manager-specific than those involved in evaluating the SLDS as a whole. The reader is encouraged to keep these different possibilities in mind during what follows.

The evaluation criteria that will be used in evaluating the final dialogue manager should in principle be specified along with the requirements specification. The evaluation criteria state the parameters that should be measured or otherwise evaluated and the results that should be achieved for the final dialogue manager to be acceptable. For instance, if the requirements specification includes a quantitative requirement on dialogue transaction success as achieved under certain specified circumstances, then dialogue transaction success should be included among the evaluation criteria and will be among the parameters measured during evaluation of the final dialogue manager.

Only a few years ago, the field of dialogue management was still so new that evaluation criteria did not exist at all, i.e., nobody had really thought about what to test and no experience from previous development efforts was available. Evaluation criteria were invented ad hoc when the dialogue manager and the SLDS in which it was embedded came up for evaluation. This way of doing things it is still quite common, in particular in research projects, but cannot be recommended because it means that developers have little support during development in terms of criteria that the dialogue manager should fulfil in order to be considered satisfactory and acceptable. The definition, from early on in the development process, of clear, relevant and appropriate evaluation criteria, and the continuous and methodologically sound evaluation of progress with reference to those criteria, should be main characteristics of dialogue manager development and evaluation.

There is still no well-defined set of evaluation criteria to draw upon. Most criteria in current use are purely quantitative. These may be relatively easy to apply but provide insufficient information on quality. Qualitative measures, on the other hand, are difficult to apply when, e.g., they are related to subjective opinions. However, in some cases performance parameters which can be measured quantitatively are also able to express quality. For example, the number of dialogue interaction problems can be measured quantitatively but at the same time the resulting figure expresses something important about the quality of the dialogue manager. Qualitative objective evaluation, however, requires experts and there are not many experts in the field of dialogue management.

For industrial projects, the main evaluation criteria may be related primarily to cost. The really interesting figures are, e.g., the cost-per-ticket-sold, the cost-per-contract-signed, or the cost-per-customer-informed. These quantities can be measured for both the human-human and the human-machine version of the dialogue. They capture all aspects of the dialogue because an inadequate system will only get considerably less tickets sold, contracts signed, customers informed or whatever is the task of the system, meaning less customer acceptance. However, these criteria are difficult or impossible to reliably apply before the system has been deployed in the field.

In DISC we have worked out a series of evaluation criteria that are relevant to dialogue manager evaluation. Whilst not all of them are applicable to all systems, each criterion may help throw light on the adequacy and performance of some dialogue managers. They need not all be included in the "official" list of evaluation criteria stated in the requirements specification. Even if not included, they may still be relevant for evaluating how good or bad the dialogue manager is and what progress is being made during its development. A first list of 39 dialogue manager evaluation criteria was proposed in [Bernsen et al. 1998b]. These criteria have been divided into six groups and thoroughly explained by using the same evaluation description template on each criterion in [Bernsen 1999]. The six groups are:

1. Use of knowledge of the current dialogue context and local and global focus of attention.

    1.1 Dialogue segmentation adequacy.

    1.2 Relevance and success of predictions.

    1.3 Sufficiency of dialogue histories (linguistic, topic, task, performance) and their use.

2. Map from the semantically significant units in the user's most recent input (if any), as conveyed by the speech and language layers, onto the sub-task(s) (if any) addressed by the user.

    2.1 Adequacy of dialogue manager support for dedicated processing of ellipsis.

    2.2 Adequacy of dialogue manager support for co-reference interpretation.

    2.3 Adequacy of dialogue manager support for indirect speech act interpretation.

    2.4 Adequacy of dialogue manager support for multimodal input fusion, i.e. for combining more or less simultaneous input messages expressed in different modalities into a single semantic representation or sub-task contribution.

    2.5 Sub-task or topic identification success.

3. Analyse the user's specific sub-task contribution(s) (if any) through the execution of a series of preparatory actions (consistency checking, input verification, input completion, history checking, database retrieval, etc.).

    3.1 Adequacy of domain inferences.

    3.2 Database information sufficiency.

    3.3 Adequacy of strategy for identifying and responding to out-of-task and/or out-of-domain input.

    3.4 Robustness - wrt. unexpected (user) deviations from the dialogue plan.

4. Generation of output to the user, either by the dialogue manager itself or through output language and/or speech layers and/or other output modalities.

    4.1 Adequacy of on-line information to users on how to interact with the system.

    4.2 Adequacy of on-line information to users on the system's domain and task coverage.

    4.3 Feedback strategy sufficiency: information feedback.

    4.4 Sufficiency of meta-communication facilities: system-initiated repair, system-initiated clarification, user-initiated repair, user-initiated clarification.

    4.5 Robustness - wrt. error loops and graceful degradation.

    4.6 Adequacy of operator fallback strategy.

    4.7 Conformance of system phrases to the cooperativity guidelines (individually as well as in context) / dialogue interaction problems caused by flawed system utterance design.

    4.8 User model adequacy (adequacy of (non-) distinction between novice and expert users, etc.).

    4.9 Adequacy of initiative distribution among user and system relative to the task.

    4.10 Adequacy of output distribution over speech and other output modalities in multimodal SLDSs.

5. Global issues of dialogue management evaluation.

    5.1 Complexity of the interaction model expressed, e.g., in terms of number of nodes if a graph representation is used.

5.2 Task and domain model coverage: are these sufficient? Are they delineated in a principled and intuitive way?

5.3 Degree of utilisation of the knowledge sources available to the dialogue manager.

5.4 Feedback strategy sufficiency: processing feedback.

5.5 Ease of maintenance/modification of the dialogue manager and/or of its individual modules.

5.6 Portability (re-usability) of the dialogue manager and/or of its individual modules.

6. Global issues of dialogue system evaluation

6.1 Average time for task completion as compared to other ways (not using an SLDS) of solving the same task.

6.2 Average cost per transaction as compared to other ways of solving the same task (not using an SLDS).

6.3 Average number of turns to complete a task compared to human-human interaction.

6.4 Real-time performance.

6.5 Transaction success rate.

6.6 Translation success rate of spoken language translation (support) systems.

6.7 User satisfaction and other subjective parameters (explain).

For more details on the dialogue manager evaluation criteria the reader is referred to [Bernsen 1999, Bernsen and Dybkjær 1999]. It should be added that the valuation template mentioned above has been subsequently revised by Bernsen and Dybkjær. A presentation and first application of the revised template is presented in [Failenschmid et al. 1999].

The list of dialogue management evaluation criteria provides a good illustration of the complexity of dialogue manager evaluation, reflecting the core "hidden" role of the dialogue manager in SLDSs. Few of the evaluation criteria are straightforwardly quantitative, at least for the time being. And some of the criteria which are quantitative, cannot be applied to the SLDSs performance as a whole but must be applied diagnostically, for instance by inspecting interaction logfiles to see whether, e.g., the dialogue manager adequately supports the performance of correct co-reference resolution or correct ellipsis processing. Very many of the criteria are qualitative. Some criteria, and not the least important ones, are subjective and must be measured through interviews, questionnaires and other contacts with the users to elicit those among their subjective impressions from interacting with the system that may be attributed to the workings of the dialogue manager.

The reason for the existence of so many qualitative dialogue management evaluation criteria is the highly contextual nature of most dialogue managers. A good meta-communication strategy, or a good feedback strategy, for instance, may solve problems arising from the insufficiency of other elements of the dialogue manager or of elements in the speech or language layers as well. If, for instance, the system lacks barge-in, the dialogue manager may have to be designed differently from its design for a similar SLDS which does have barge-in. Similarly, criteria involving terms such as 'adequacy' and 'sufficiency' often conceal one or several implicit conditions, such as 'relative to the task' or 'relative to the intended users'.

Furthermore, the list is incomplete in at least one important respect: it does not include the, often quantitative, criteria which may derive from particular constraints on a given SLDS, such as that the average user utterance length should be, say, four words maximum. To include such constraints would make the list unwieldly and overly specific. In developing one's dialogue manager, one may measure many different things which it would make no sense to measure in connection with another dialogue manager development project which is subject to very different constraints.

As it stands, the list may be used as a checklist by dialogue manager developers in four iterations, so to speak, as follows:

- in the *first iteration,* the developer selects the criteria which are relevant to the particular dialogue manager to be developed;

- in the *second iteration,* the developer makes the selected criteria more specific and applicable by making explicit the implicit conditions that apply to the development task at hand;

- in the *third iteration,* the developer plans when to apply the specified criteria during the development process;

- finally, in the *fourth iteration,* the criteria are applied in a methodologically sound manner as planned.

Note that the above four-stage model can be used for 'meta-evaluation' of dialogue manager development processes. Central questions to ask during meta-evaluation are:

- did the developers select all the right evaluation criteria?

- did they make these criteria sufficiently specific to their development task?

- did they apply the criteria correctly at all the development stages at which they should have been applied?

- what were the results?

- did the developers take adequate action in view of the results?

The next section will pick up on this point.

3.22 Evaluation

At which stages during design and development was the dialogue manager subjected to testing/evaluation? How (type of test, number of users, etc.)? Describe the results. How was data collection? Was data annotation done? How? Which information has been extracted from the data? What were the results? Is test material/data/test suites (and/or a description of the test conditions) available? Can the test be replicated? Can anybody perform the tests? Is anything stated about comparability of the test(result)s with those of other systems/components of similar scope?

Evaluation is very important and is/should be tightly interwoven with systems development. Evaluation is constantly needed throughout development to measure progress towards the goals which the dialogue manager has to meet. However, evaluation of dialogue managers as well as of entire SLDSs is today as much of an art and a craft as it is an exact science with established standards and procedures for good engineering practice. There is not even consensus on terminology in the field.

Distinction may be made among three types of evaluation which, although clearly not orthogonal, seem to cover the relevant aspects of evaluation and subsume the scopes of other commonly used terms and distinctions. Each of these three types of evaluation may be used at any stage during dialogue manager development:

- *performance evaluation*, i.e. measurements of the performance of the dialogue manager and its components in terms of a set of quantitative parameters;
- *diagnostic evaluation*, i.e. detection and diagnosis of design and implementation errors;
- *adequacy evaluation*, i.e., how well do the dialogue manager and its components fit their purpose and meet actual user needs and expectations.

Other common terms are 'blackbox' and 'glassbox' tests, and 'progress evaluation'. Blackbox and glassbox tests may be considered forms of diagnostic evaluation but these tests are carried out on implemented components or systems only (se further below). *Progress* *evaluation* compares two iterations of the same system during development and is a kind of performance evaluation. Progress evaluation is typically used to compare, e.g., Wizard of Oz iterations of a dialogue manager.

Performance, diagnostic and adequacy evaluation should be performed as integral parts of the development process to measure progress towards satisfaction of the requirements specification, evaluation criteria and design specification.

Performance evaluation is made throughout the development process with more or less the same emphasis from one iteration to another.

Diagnostic evaluation is of central importance in the early development process but should require less effort in the final phase by which time most errors should have been removed. During debugging of the implemented dialogue manager, two typical types of test to carry out are the glassbox tests and the blackbox tests. There is no general agreement on the definitions of glassbox and blackbox tests. By a *glassbox test* we understand a test in which the internal system representation can be inspected. The test should ensure that reasonable test suites, i.e. data sets, can be constructed that will activate all loops and conditions of the program being tested.

In a *blackbox test* only input to and output from the program are available to the evaluator. How the program works internally is made invisible. Test suites are constructed on the basis of the requirements specification and along with a specification of the expected output. Expected and actual output are compared when the test is performed and deviations must be explained. Either there is a bug in the program or the expected output was incorrect. Bugs must be corrected and the test run again. The test suites should include fully acceptable as well as borderline cases to test if the program reacts reasonably and does not break down in case of errors in the input. Ideally, and in contrast to the glassbox test suites, the blackbox test suites should not be constructed by the system programmer who implemented the system since s/he may have difficulties in viewing the program as a black box.

Adequacy evaluation of SLDSs typically includes a few general key performance measurements as well as measurement of user satisfaction. If we are talking about adequacy evaluation of a dialogue manager *per se,* there is a need for much more specific evaluation criteria. Adequacy evaluation is used mostly in the later phases of development. This is because a number of adequacy aspects cannot be tested in a sensible way until an implemented and debugged dialogue manager is available for the purpose. For instance, it does not make sense to measure real-time performance on a simulated system.

Another useful distinction is the distinction between objective evaluation and subjective evaluation. *Objective evaluation* addresses objectively measurable performance parameters. *Subjective evaluation* addresses the opinions which users have formed after using the dialogue manager as imbedded into an SLDS. Performance evaluation and diagnostic evaluation are forms of objective evaluation whereas adequacy evaluation includes both objective and subjective evaluation.

In addition to distinctions between different types of evaluation such as the above, distinction may be made between different types of tests. Test types refer to aspects of the context of the evaluation, such as the users involved, whether scenarios are being used or not, or whether the system being tested is an implemented one or a simulated one. The tests to be mentioned below are controlled tests, field tests and acceptance tests. Roughly speaking, controlled tests are performed during simulation and often also after implementation; field tests are performed after implementation and towards the end of systems development; and the acceptance test is the final test of a system. This notwithstanding, a test may be carried out as a controlled test or as a field test no matter if the system is being simulated or not. Each test typically includes all of the three evaluation types mentioned above (i.e. performance, diagnostic and adequacy evaluation).

In a *controlled test,* the users need not be those who will actually use the final system. However, it is recommended to select the test subjects from the target user group to ensure that they have a relevant background. In the controlled test, the tasks to be carried out (the scenarios) should not be selected by the participants. To ensure reasonable representativity of scenarios with respect to system functionality and task domain coverage, and to bring the controlled test as close to benchmarking as possible, scenario selection should ideally be done by an independent panel according to guidelines on, i.a., who should select the scenarios, their coverage of system functionality and task domain, the number of scenarios per user and the number of users. The panel should include end-users as well as system developers. A *field distribution problem* attaches to all results of controlled tests. The frequency of different tasks across the domain of application may be different in real life from that imposed in the controlled test. This may significantly affect the frequency of the interaction problems encountered in the test.

In a *field test,* the dialogue manager is being tested by real end-users in their appropriate environments. This means that the tasks carried out will be real-life tasks which, however, may not necessarily be representative of the full range of system functionality unless the duration of the field test is very long. The field test option will not always be available for research systems due to the missing customer. It may be preferable to carry out a controlled test before a field test because the controlled test will allow an evaluation which is close to benchmarking.

The *acceptance test* is the final test of the dialogue manager and perhaps also of the system in which it is embedded, before it is accepted for operational use. The test aims to demonstrate that the requirements in the contract (the requirements specification) have been satisfied and the evaluation criteria met. Often the dialogue manager is tested with data supplied by the procurer or in a set-up specified by the procurer. Detected errors must be corrected immediately. In case of larger disagreements with, or omissions in, the requirements specification, developer and procurer must discuss what to do. In the worst case the procurer may turn down the product if the dialogue manager or system does not meet the requirements agreed upon. However, it is not always solely the system developer's fault that the dialogue manager does not exhibit the performance and functionality anticipated by the procurer. In such cases, procurer and developer must negotiate what to do in order to reach an agreement.

This section on evaluation will be updated as part of the evaluation template to be used in DISC-2.

3.23 Mastery of the development and evaluation process

Of which parts of the process did the team have sufficient mastery in advance? Of which parts didn't it have such mastery?

There are two aspects to be aware of. One is the general software engineering related aspect. The other concerns the issues specifically related to dialogue management. As regards the general software engineering issues, the team may or may not know the programming language and other general development and evaluation techniques in advance and they may be more or less experienced in software development and evaluation. With respect to the dialogue management related issues, developers have, in many projects, little mastery of dialogue management because the field is fairly new and the project itself is meant to serve as a competence-building exercise. However, this does not prevent a team from having good skills in software engineering and other general issues relevant to dialogue management, including, e.g., human factors and linguistics. Other teams have worked extensively on dialogue management and have obtained a good deal of experience in the field.

3.24 Problems during development and evaluation

Were there any major problems during development and evaluation? Describe these.

Problems during development and evaluation may be of many different kinds and both human and technical. Examples are that more than one site is involved in dialogue manager development, which may cause problems in collaboration or technical problems in making the parts fit together; the developers have little experience in dialogue management; one or more developers left the team, resulting in delays; the requirements specification was not properly evaluated to begin with and turns out to be (partially) infeasible, resulting in re-specification and redesign of the dialogue manager; the interfacing to other parts of the SLDS was not properly specified; other parts of the SLDS turned out to be efficient or powerful than planned thus causing problems for the dialogue manager.

3.25 Development and evaluation process sketch

Please summarise in a couple of pages key points of development and evaluation of the dialogue manager. To be done by the developers.

This entry is intended to provide a brief overview of the dialogue manager development and evaluation process. The sketch may include information on how requirements were established, how a dialogue model was developed and evaluated, implementation issues, and tests done on the dialogue manager. On the issue of evaluation, the sketch could draw extensively on the list of criteria presented in Section 3.21 and in [Bernsen 1999].

3.26 Component selection/design

Describe the dialogue manager and its origin.

Off-the-shelf dialogue managers only exist to a limited extent and often only as part of a larger environment. If an off-the-shelf component is chosen, it must be able to fit the purposes of and the requirements to the dialogue management needed for the particular application to be built. Most dialogue managers are still built in-house for particular applications and as part of an SLDS.

3.27 Maintenance

How easy is the dialogue manager to maintain, cost estimates, etc.

Maintenance costs for dialogue managers are usually low. When research projects are finished, time may still be spent on keeping the SLDS running for demonstration purposes, but that is all. Also for commercial dialogue managers, maintenance costs seem to be at a minimum. Rather, resources may be spent on modifications, additions and customisation, cf. below.

3.28 Portability

How easily can the dialogue manager be ported?

Basically, a dialogue manager only requires that the language in which it is being developed can run on the platform to which it is being ported. However, the SLDSs of which a dialogue manager is typically a component, may impose much more severe restrictions on portability deriving from its other components and their platform requirements.

3.29 Modifications

What is required if the dialogue manager is to be modified?

This question may be difficult to answer in general since what is required depends on the type of modification. Small modifications compatible with the lines along which the dialogue manager has been developed already, may be fairly easy to implement, such as slight modifications of some sub-task. Larger modification, on the other hand, can be very difficult to do, such as the addition of a new task. Modification difficulty depends on how generic the dialogue manager is relative to the modifications proposed.

3.30 Additions, customisation

Has customisation of the dialogue manager been attempted or carried out? Is there a strategy for resource updates? Is there a tool to enforce that the optimal sequence of update steps is followed?

Most dialogue managers are built as one-of-a-kind to fit into a specific SLDS. This means that the possibility of additions and customisation has not necessarily been considered and is rarely documented.

However, dialogue managers do exist which are intended for adaptation to new tasks. As a minimum, when being adapted to a new service, the dialogue manager will need a model of the new task(s), a model of the domain and a model of the new dialogue. If the dialogue manager is language independent (which it preferably should be), it should be relatively easy to switch to another language. Moreover, if the knowledge bases are kept separate it will also be fairly easy to adapt the dialogue manager to using the new ones. For dialogue managers which are being traded or made available as free generic software there will usually be a manual or some kind of guide on what to do to build your own application.

3.31 Property rights

Describe the property rights situation for the system/component.

The property rights to the dialogue manager typically belong to the site(s) where the component was developed but, of course, other arrangements may exist. If an off-the-shelf dialogue manager is being used the situation may be more complex. One usually buys a license to use, e.g., a speech recogniser or a dialogue manager from a company which holds the property rights. However, the organisation which develops an application which uses and integrates the off-the-shelf component, will normally hold the property rights to this application and the new software.

3.32 References to additional dialogue manager documentation

Please include references to any background material in which information on the dialogue manager can be found.

References may be to conference papers, technical reports, user manuals, course material or any other material which contains information on the dialogue manager.

4. Conclusions and Future Work

The present paper is a working paper. Its descriptive apparatus has proved sufficient for capturing all of the available details on properties, development and evaluation of five rather diverse state-of-the-art dialogue managers, cf. [Bernsen et al. 1998b]. The current practice draft on dialogue management presented in [Bernsen et al. 1998b] has been expanded into a first draft best practice model of dialogue management development and evaluation. The next step will be to package the final DISC best practice methodology for dialogue management and evaluation for optimised usability by developers by making it available on the web in a version well-suited for hypertext/hypermedia access. We will then test the best practice draft on novel, and different, exemplars from industry and research as well as on the developers who represent the target audience for the DISC results. This will be done as part of DISC-2.

5. Acknowledgements

The work presented was done in the Esprit Long-Term Concerted Action No. 24823, DISC (1997-1998): Spoken Language Dialogue Systems and Components: Best practice in development and evaluation. The support is gratefully acknowledged. We would like to thank the DISC partners Nils Dahlbäck, Patrik Elmberg, Uli Heid, Arne Jönsson and Jan van Kuppevelt who also analysed the exemplar dialogue managers, cf. [Bernsen et al. 1998b]. We also want to warmly thank the DISC partners who have demonstrated their systems and provided detailed information for the grid and life cycle analyses.

6. References

[Alexandersson et al. 1997] Alexandersson, J., Reithinger, N. and Maier, E.: Insights into the Dialogue Processing of Verbmobil. Proceedings of the Fifth Conference on Applied Natural Language Processing, ANLP '97, Washington, DC 1997, 33-40.

[Aust et al. 1995] Aust, H., Oerder, M., Seide, F. and Steinbiss, V.: The Philips automatic train timetable information system. Speech Communication 17, 1995, 249-262.

[Basson et al. 1996] Basson, S., Springer, S., Fong, C., Leung, H., Man, E., Olson, M., Pitrelli, M., Singh, R. and Wong, S.: User participation and compliance in speech automated telecommunications applications. In *Proceedings of ICSLP'96*, Philadelphia, 1996, 1680-1683.

[Bernsen 1999] Bernsen, N. O.: Working Paper on Dialogue Management Evaluation. DISC Deliverable D3.10, April 1999.

[Bernsen and Dybkjær 1999] Bernsen, N. O. and Dybkjær L.: Evaluation of Spoken Language Dialogue Systems. In Luperfoy, S. (Ed.): Automatic Spoken Dialogue Systems. MIT Press, 1999.

[Bernsen and Luz 1999] Bernsen, N. O. and Luz, S.: SMALTO: Speech Functionality Advisory Tool. DISC Deliverable D2.9, 1999.

[Bernsen et al. 1998a] Bernsen, N. O., Dybkjær, H. and Dybkjær, L.: *Designing Interactive Speech Systems. From First Ideas to User Testing.* Springer Verlag 1998.

[Bernsen et al. 1998b] Bernsen, N. O., Dybkjær, L., Heid, U., van Kuppevelt, J., Dahlbäck, N., Elmberg, P. and Jönsson, A.: Working Paper on Dialogue Management Current Practice. DISC Deliverable D1.5, May 1998.

[Bertenstam et al. 1995] Bertenstam, J. Blomberg, M., Carlson, R., Elenius, K, Granström, B., Gustafson, J., Hunnicutt, S., Högberg, J., Lindell, R., Neovius, L., de Serpa-Leitao, A., Nord, L. and Ström, N.: The Waxholm system - a progress report. Proceedings of ESCA Workshop on Spoken Dialogue Systems, Vigsø, 1995, 81-84.

[Bub and Schwinn 1996] Bub, T. and Schwinn, J.: Verbmobil: The Evolution of a Complex Large Speech-to-Speech Translation System. DFKI GmbH Kaiserslautern. Proceedings of ICSLP'96, Philadelphia 1996, 2371-2374.

[Carlson 1996] Carlson, R.: The Dialog Component in the Waxholm System. Proceedings of the Twente Workshop on Language Technology (TWLT11) Dialogue Management in Natural Language Systems, University of Twente, the Netherlands, 1996, 209-218.

[den Os et al. 1999] den Os, E., Boves, L., Lamel. L. and Baggia, P.: Overview of the ARISE project. Proceedings of Eurospeech 1999.

[Dybkjær 1999] Dybkjær, L.: CODIAL, a Tool in Support of Cooperative Dialogue Design. DISC Deliverable D2.8, April 1999.

[Dybkjær and Failenschmid 1999] Failenschmid, K. and Dybkjær, L.: State-of-the-Art Survey of Existing Human Factors Tools. DISC Deliverable D2.7b, February 1999.

[Failenschmid et al. 1999] Failenschmid, K., Williams, D., Dybkjær, L. and Bernsen, N. O.: *Draft Proposal on Best Practice Methods and Procedures in Human Factors*. DISC Deliverable D3.6., April 1999.

[Fraser et al. 1996] Fraser, N. M., Salmon, B. and Thomas, T.: Call Routing by Name Recognition: Field Trial Results for the Operetta(TM) System. IVTTA'96, NJ, USA, 1996.

[Heisterkamp and McGlashan 1996] Heisterkamp, P. and McGlashan, S.: Units of Dialogue Management: an example. Proceedings of ICSLP'96, Philadelphia, 1996, 200-203.

[Lamel et al. 1995] Lamel, L., Bennacef, S., Bonneau-Maynard, H., Rosset, S. and Gauvain, J. L.: Recent Developments in Spoken Language Systems for Information Retrieval. In Proceedings of the ESCA Workshop on Spoken Dialogue Systems, Vigsø, Denmark, 1995, 17-20.

[Luz 1999] Luz, S.: State-of-the-Art Survey of Existing Dialogue Management Tools. DISC Deliverable D2.7a, February 1999.

[Sturm et al. 1999] Sturm, J., den Os, E. and Boves, L.: Issues in Spoken Dialogue Systems: Experiences with the Dutch ARISE System. Proceedings of ESCA Workshop on Interactive Dialogue in Multi-Modal Systems. Kloster Irsee, Germany, 1999, 1-4.

[van Kuppevelt and Heid 1999] van Kuppevelt, J. and Heid, U.: From a Description of Spoken Language Dialogue Systems to their Evaluation and Best Practice". DISC Deliverable D3.8b, August 1999.