Spoken Language Dialogue Systems

Report 9a, January 1996

Test of the Danish Spoken Language Dialogue System

CPK - Center for PersonKommunikation, Aalborg University
CCS - Centre for Cognitive Science, Roskilde University
CST - Centre for Language Technology, Copenhagen

Printed at Roskilde University

January 1996

ISBN 87-7349-300-7

The present report is part of the documentation series from the strategic research programme *Spoken Language Dialogue Systems*. The programme is sponsored by the Danish Technical Research Council. The project partners are the Center for PersonKommunikation (CPK) (coordinating partner) at Aalborg University, the Centre for Cognitive Science (CCS) at Roskilde University, and the Centre for Language Technology (CST), Copenhagen.

Authors:

Laila Dybkjær, CCS

Niels Ole Bernsen, CCS

Tom Brøndsted, CPK

Anders Baekgaard, CPK

Hans Dybkjær, CCS

Lars Bo Larsen, CPK

Børge Lindberg, CPK

Bente Maegaard, CST

Bradley Music, CST

Claus Povlsen, CST

The project partners can be contacted at:

Center for PersonKommunikation (CPK):

Paul Dalsgaard
Aalborg University
Frederik Bajers Vej 7
DK-9220 Aalborg Ø, Denmark
Phone: +45 98 15 85 22. Fax: +45 98 15 15 83
Email: pd@cpk.auc.dk

Centre for Cognitive Science (CCS):

Niels Ole Bernsen
Roskilde University
P.O.Box 260
DK-4000 Roskilde, Denmark
Phone: +45 46 75 77 11. Fax: +45 46 75 45 02
Email: nob@cog.ruc.dk

Centre for Language Technology (CST):

Bente Maegaard
Njalsgade 80
DK-2300 Copenhagen S, Denmark
Phone: +45 35 32 90 90. Fax: +45 35 32 90 89
Email: bente@cst.ku.dk

Preface

This report is the ninth in the documentation series from the research programme *Spoken Language Dialogue Systems.* The objective of the programme is through the development of prototypes to gain new knowledge in the research fields of speech technology, natural language processing and human-computer interaction (cognitive engineering) and especially in the combination of these fields. The programme is scheduled to run for a four year period (primo 1991 - primo 1995) and is divided into the two sub-projects P1 and P2, each scheduled to produce a running prototype in the domain of flight ticket reservation and information. Both prototypes have been implemented and tested.

Report 9 has three parts, 9a, 9b and 9c. The present report (9a) focuses on the test of the implemented modules of P1 and P2, P2 being an improved version of P1. Report 9b presents the user test of the final system, called the Danish dialogue system, or sometimes P2 for short, with a simulated speech recogniser. Report 9c builds on the user test material from report 9b and describes the sufficiency of the linguistic coverage and the performance of the parser for input outside the linguistic coverage.

Chapter 1 provides an introduction to the test strategies and techniques used for testing the implemented components of P1 and P2. Chapter 2 describes in detail for each system component how it was tested. Chapter 3 concludes the report. Appendix A provides a set of design rationale (DR) frames used for the representation and discussion of dialogue problems detected during the test. Appendix B gives a brief description of a program constructed to facilitate the indication of test input to the dialogue part and to the application database of P1 and P2.

Keywords

Spoken language dialogue systems, test, glassbox, blackbox.

Danish Summary

Denne rapport er den niende i dokumentationsserien fra forskningsprojektet *Behandling af talt sprog i dialogstyrede applikationer.* Projektets formål er gennem udvikling af prototyper at erhverve ny viden inden for forskningsområderne taleteknologi, natursprogsbehandling og menneske-maskine interaktion (kognitionsforskning) og specielt i kombinationen af disse discipliner. Programmet løber over en fireårig periode (fra primo 1991 til primo 1995) og er opdelt i to dele, P1 og P2, der hver skal resultere i en kørende prototype inden for domænet flybilletbestilling og -information. Begge prototyper er på nuværende tidspunkt færdig-implementeret og afprøvet.

Rapport 9 består af tre dele, 9a, 9b og 9c. Nærværende rapport (9a) fokuserer på afprøvningen af de implementerede P1- og P2-moduler. P2 er en forbedret version af P1. Rapport 9b præsenterer brugertesten af det endelige system, kaldet det danske dialog system eller kort og godt P2, med en simuleret talegenkender. Rapport 9c bygger på brugertestmaterialet fra rapport 9b og beskriver tilstrækkeligheden af den lingvistiske dækningsgrad og parserens performans ved input uden for den lingvistiske dækningsgrad.

Kapitel 1 giver en introduktion til de afprøvningsstrategier og -teknikker, der er brugt ved afprøvningen af P1 og P2. Kapitel 2 beskriver i detaljer, hvordan hver systemkomponent er blevet afprøvet. Endelig afrunder kapitel 3 rapporten. Appendiks A viser et sæt design

rationale (DR) rammer, der er blevet brugt til at repræsentere dialogproblemer, der er opdaget under afprøvningen, og til at diskutere mulige løsninger. Appendix B giver en kort beskrivelse af et program, der blev konstrueret for at lette angivelsen af testinput til dialogdelen og til databasen i P1 og P2.

Contents

1 Introduction

The present report focuses on the test of the implemented components of the two prototypes, P1 and P2, developed in the Danish dialogue project. P2 is a revised and improved version of P1 and is also called the Danish dialogue system.

Program testing is an important part of systems development. It is the process of making the system behave as intended [Lauesen 1979]. Testing serves to detect errors in the implemented program and to make a diagnosis of what is wrong in each case so that errors can be corrected. Basically, there are two strategies for testing an implemented system: it may be tested bottom-up or top-down. In *bottom-up* testing, each system module is tested separately by embedding it in artificial test surroundings and providing it with input on the form requested by the module in question. By contrast, *top-down* testing tests the system as a whole. Missing parts are replaced by dummies simulating the effect of the absent parts. System input in a top-down test corresponds to input to the final system.

The advantage of bottom-up testing is that system components developed at different sites and/or not finished at the same time can be tested separately and independently of the existence of other components. The drawbacks of bottom-up testing are that artificial test surroundings must be built which may be costly, and that disagreements on formats in the communication exchange between the modules are not necessarily revealed.

Top-down testing requires an (almost) final system and the construction of dummies if there are unfinished parts, but will reveal disagreements on formats and is necessary to make sure that all the modules behave together as intended.

Testing typically includes three types of test: a glassbox test, a blackbox test and a user test, cf. Figure 1.1.

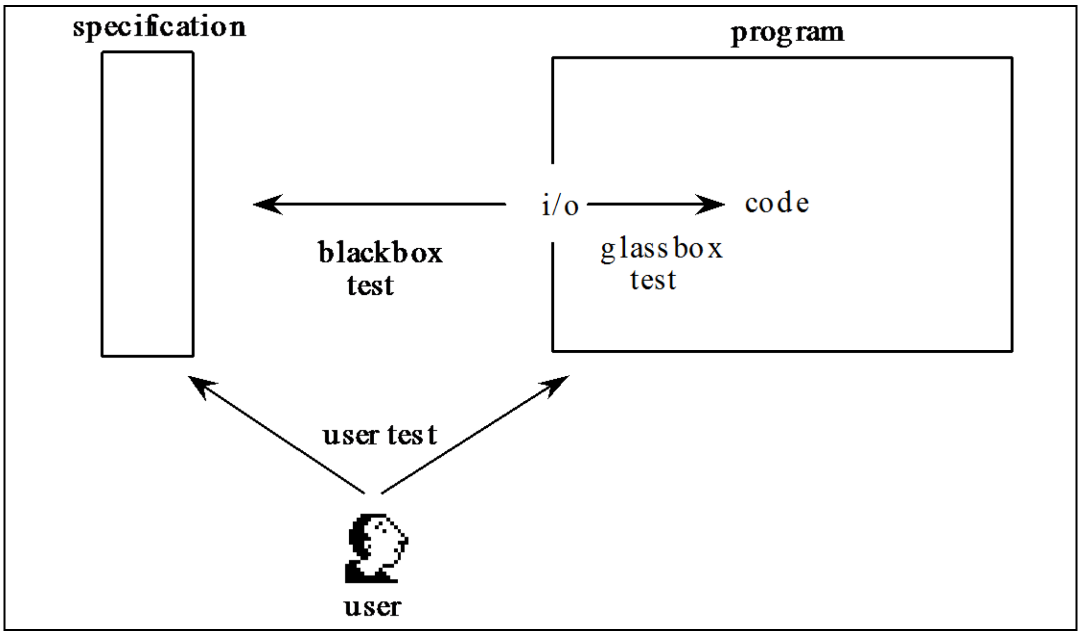

Figure 1.1. Typically three different kinds of test are used to test a program. The glassbox test serves to ensure that all parts of the code can be activated and make a reasonable contribution to system behaviour as described in the specification. The purpose of the blackbox test is to ensure that the program behaves in accordance with the specification and reacts robustly and rationally on input

outside the specification. Finally, the user test is meant to test if the specification, and hence the program, is sound and complete in relation to users' expectations as to which tasks they can perform with the system.

In a *glassbox test* the internal representation may be inspected. The test should ensure that reasonable data sets can be constructed that will activate all loops and conditions of the program being tested. The relevant test data are constructed by the system programmer(s) along with an indication of which program parts the data are supposed to activate. Via test print-outs in all loops and conditions it is possible to check which ones were actually activated.

In a *blackbox test* the program is viewed as a black box. Only input to and output from the program are available to the evaluator whereas how the program works internally is invisible. Test data are constructed along with an indication of expected output which is compared to the actual output when the test is being performed. Differences between expected and actual output must be explained. Either there must be a bug in the program being tested or the indicated expected output was not correct. Bugs must be corrected and the test run again. The test data 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 input. Ideally, and in contrast to the glassbox test data, the blackbox test data should not be constructed by the system programmer(s) who may have difficulties in viewing the program as a black box.

The final test to perform is the *user test* which measures overall system performance in a number of respects and provides information on the usability of the system and the success of the specification. Test data may be constructed by system designers if the purpose is to test a specific part of the system while avoiding, e.g., known shortcomings. In other cases the decision on test data is left to the users. Typically, this solution is chosen when a system is considered almost ready for being used in practice. The user test of the Danish dialogue system is reported in [Report 9b].

Bottom-up and the top-down test strategies have been used for P1 and P2, although bottom-up testing was mainly used for P1. Glassbox and blackbox test types have been used in connection with the tests of P1 as well as P2. The tests are described in detail in the following chapter.

2 Test of system platform and system components

This chapter describes the test of the P1/P2 system platform and system components both separately and as an entire system. Since P1 and P2 are rather similar, the applied testing methods will in many cases be approximately the same for P1 and P2. In such cases we will refer to them as P1/P2 without distinguishing in detail between the test of the two prototypes.

The P1/P2 systems consist of several components based on the DDL/ICM architecture as outlined in Figure 2.1. All components have been subject to bottom-up test as well as top-down test. Glassbox test was only used during bottom-up testing whereas blackbox test was used along with both strategies. The bottom-up tests were performed at the sites where each module was developed, i.e. the speech recogniser, the player (part of the reproductive speech module), the telephone line interface, and the text recogniser have been tested at CPK, the linguistic analysis module at CST, and the dialogue description and the database at CCS. The prerecorded phrases (part of the reproductive speech module) were also tested at CCS but only top-down, and the text recogniser was tested in practice at CCS during the user tests. The main part of the architecture has been tested at CPK but DDL and DDL-Tool have been tested at CCS as well along with the implementation of the dialogue description.

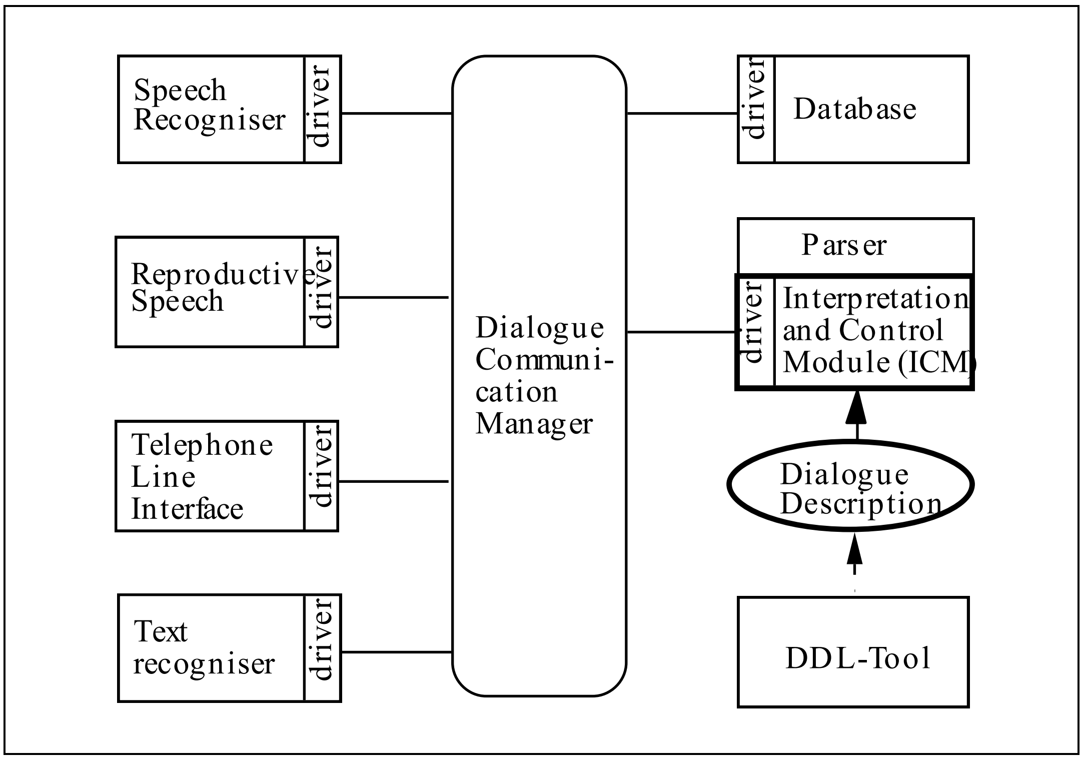

Figure 2.1. Overall system architecture of P2.

2.1 System platform

The P1 dialogue system is built on top of the dialogue system platform described in [Report 10, Baekgaard 1995]. The architecture of the platform is designed to be modular and open such that it can be easily expanded, and all modules operating in the dialogue system adhere to a well-defined protocol. This allows for efficient adaptation and integration of new operating environments and new special purpose devices. [Report 2, Report 10]. During design and implementation of the architecture and the main components, generality, flexibility and re-usability were major concerns. Further, a design criterion was to aim at a high degree of formalisation.

The main components of the platform are the DDL dialogue description formalism, the DDL-Tool, the ICM generic dialogue manager and the communication system. Parts of the main components of the system platform were originally developed in the SUNSTAR project, and extensive tests were then conducted. Further tests have been performed in the current project. Most tests were conducted manually either by observing the behaviour of a module when given certain input (blackbox tests), or by inspecting output like dumps of data structures, compiled output, and traces (glassbox tests). A number of tests suites were developed that allowed test results to be reproduced. However, most of the tests carried out during development were conducted manually and cannot be reproduced automatically.

2.1.1 The DDL-Tool

The DDL-Tool has been tested by independent bottom-up tests of the major functions:

1. The user interface (menus, buttons, drawing facilities, editing on the textual, frame and graphical levels, the lexica, etc.). Blackbox tests have been conducted by observing that each element of the user interface behave correctly.

2. Saving and retrieving files (DDL descriptions). Tests have been conducted by observing that each retrieved file is equivalent to the saved one, and by comparing files retrieved and then saved.

3. The compiler. Tests have been conducted by inspecting the compiler output, and by transferring the output to the ICM for analysis.

4. The verifier. Tests have been conducted that valid dialogue descriptions are recognised and that errors are detected.

5. The debugger. Tests have been conducted by observing that each function of the debugger works.

2.1.2 The ICM dialogue manager

The ICM has been tested by independent tests of the major functions. There are test suites that allow tests to be repeated. The ICM is a complex module as it contains many components each of which is complex and which must interact. The major functions are:

1. Parsing of dialogue descriptions and creation of the internal representation of the dialogue description;

2. Optimisation of the representation;

3. Parsing of events according to DDL grammars;

4. Interfacing to the NJAL parser;

5. Maintenance of dialogue context (subgrammars, word sets);

6. Interpretation of rules;

7. Procedure activations;

8. Control of handlers;

9. Protocol conversions;

10. Timing;

11. Multi-threading.

These functions have been tested by a number of small DDL test programs that each were designed to test a specific function. In addition, test programs were designed for testing the entire system.

2.1.3 The communication system

For the communication system the following tests have been performed:

1. Verification of configuration files being parsed correctly. This is done by inspecting that dumps of the internal structures correspond to the input.

2. Verification that the commands are handled correctly. This is done by inspecting the response when commands are sent to the communication system. Since responses depend on the state of the communication system, a large number of tests have been conducted.

2.1.4 Test modules

Several replacement modules have been developed that allow tests of specific modules or functions in modules. The following replacement modules have been developed:

1. A reproductive speech output device for Sun work stations that replaces the DSP based device described in section 2.3.3.1. The two devices have the same interface and functionality seen from the point of view of the dialogue manager.

2. A text input device that allows text entered at a keyboard to simulate the recogniser. This device allows unrestricted text input as opposed to the text recogniser described in section 2.3.1. This device is useful for testing a dialogue system where the speech recogniser is simulated.

3. A pseudo parser that allows semantic representations to be entered from the keyboard (or taken from a file). The pseudo parser replaces the speech recogniser and the NJAL parser, and is useful for testing the dialogue model.

When all errors found during the tests had been fixed, the platform was distributed to partners. During the development of P1 and P2 several bugs were discovered and fixed.

2.2 The speech recognition module

The speech recognition module mainly consists of an acoustic recogniser, acoustic models, language models, and tools for training or generating these models. The training software for generating acoustic models is not described in this report, because it has not been developed within the dialogue project, but see [Jacobsen 1991] and [Young 1992].

2.2.1 Test of the speech recogniser

The speech recogniser is a continuous speech recogniser based on Continuous Density Hidden Markov Models (CDHMM) and the token passing Viterbi decoding algorithm [Young et al. 1991].

The CDHMM model structure has been chosen because of the apparently impressive performance achieved on systems based on CDHMMs, mainly in the US, during the late eighties and early nineties. CDHMMs are powerful in speech pattern processing for several reasons. CDHMMs are parametric models and offer a compact representation of stochastic signals such as speech patterns. CDHMMs are trainable and methods such as the Baum-Welch re-estimation algorithm exist for the estimation of CDHMM parameters. Finally, efficient methods, such as the Viterbi-algorithm, exist for conducting pattern recognition based on CDHMMs. This provides an attractive basis for the implementation of low-cost real-time recognisers based on CDHMMs.

The Viterbi token passing algorithm has been chosen because it dynamically generates the required data structures for representing potential word boundaries and string candidates during the optimal Viterbi search.

The speech recogniser uses language models for constraining the decoding of the acoustic signal. In modern speech technology, this is an established method for increasing the recognition rate. The most commonly used language models are grammars equivalent to the "type 3" in the Chomsky hierarchy (regular grammars, finite state automata, cf. [Chomsky 1959]). The speech recogniser supports both deterministic and probabilistic language finite state language models. However, for reasons discussed in section 2.2.2 we have chosen only to use deterministic models. Please refer to section 2.2.2 also for a description of the test of tools for generation of finite state language models.

The speech recogniser is an extended version of the SUNCAR batch recogniser implementation originating from the SUNSTAR project, cf. [Report 8]. As the demand of real-time speech recognition was mandatory within the present project, a further development of the SUNCAR batch recogniser was conducted, resulting in the recogniser described in the present report (the SUNCAR Real-time Recogniser). The SUNCAR Real-time Recogniser is implemented to run on a PC equipped with an AT&T DSP32C signal processor based board making use of the 3R Software [Lindberg 1995] for the acoustic processing such as feature processing, density computation and Viterbi decoding at CDHMM model level.

The SUNCAR Real-time Recogniser is able to execute in real time on a PC running either DOS or Linux and equipped with an AT&T DSP32C based board. Real time capacity, however, is limited to approximately 120, 10 state, single mixture CDHMM models, and grammar finite state networks of less than 450 arcs and 75 states. Real time execution has been obtained by distributing the Viterbi decoding between the DSP and the host as described in [Report 8]. If the limits for real-time execution are exceeded, the recogniser switches to non-real time execution in which the DSP is used for speech signal acquisition and feature processing, only.

The recogniser acts as a device in the dialogue system as presented in section 2.1. It has therefore been verified that the recogniser, as a device, comply with the system communication protocol specification as presented in [Report 8]. This has been achieved partly by simulating commands from the ICM (blackbox), and partly by inspecting contents of events from the driver during processing (glassbox). However, verification of the driver functionality has been complicated in the present system, as three of the present drivers communicate with the same DSP32C based PC board, running the 3R software [Lindberg 1995].

The speech recogniser draws advantage of many of the components originating from the 3R software which was originally designed for isolated word recognition. Within the ESPRIT SAM project [SAM 1992], test control drivers have been developed for this isolated word recogniser, and extensive testing has been conducted and reported on that recogniser according to the standard protocols within the SAM project.

Several formal automated tests of the 3R Software have thus been conducted although test control drivers for the present (continuous) speech recogniser have not been implemented.

Although the individual modules of the original batch SUNCAR recogniser have been significantly extended and modified, comparisons on a recognition result basis were still possible. This has been utilised to verify reliability of recogniser output. This reliability verification has included real time execution, as results obtained during processing of the microphone signal were verified against those subsequently obtained from batch testing on the corresponding speech signal file. Similar blackbox verification procedures have been applied for verifying the feature processing part of the recogniser.

In conclusion the recogniser has been verified at the acoustic analysis, acoustic matching and acoustic decoding levels to obtain identical performance compared to the training and testing software, cf. [Report 5, Report 5a]. Please refer to section 2.2.3 for a description of the training and test of acoustic models. Unfortunately, this is no valid prediction of the ultimate recogniser performance observed within a real-life application such as the present P2-application. This is due to several factors which all will degrade the performance. Examples of such factors are the fact that the speech signal within an application will be noise-degraded and the fact that there is no optimal level-adjustment. These examples are most apparent when using external telephone lines.

2.2.2 Test of tools for generation of language models

In the present system the speech recogniser is constrained by deterministic (non-probabilistic) finite state language models. Note that the classification of the language models as finite state ("Type 3") grammars within the Chomsky hierarchy implies an important glassbox evaluation, as the restrictions imposed by each grammar class in the hierarchy are well-described in the literature, cf. [Chomsky 1959]. Examples of syntactic structures which cannot be described in finite state models are given in [Report 5].

Deterministic language models are based on binary indicator functions. Such models can separate grammatical sequences of words from ungrammatical ones, however they do not assign probabilities to the sequences. Formally, probabilistic models are more "expressive" than deterministic models, because they generate a likelihood for each sequence of words. On the other hand, probabilities must be based on observation sequences of finite length, typically one (unigrams), two (bigrams) or three word sequences (trigrams).

The main reason for using deterministic instead of probabilistic language models in the present system is the fact that probabilistic models must be trained on large amounts of transcribed speech data. Such data have not been available within the project because of limited resources. The recorded and simulated (Wizard generated) speech data used for sublanguage definition and dialogue modelling, cf. [Report 3, Report 4] are quantitatively insufficient for training of N-grams. Further, if language modelling is done with pure data-driven techniques, there is no guarantee that the resulting models will observe the external restrictions imposed by the recogniser as regards number of active words, number of nodes, and number of arcs (cf. section 2.2.1). For further discussion, see [Brøndsted 1994].

The deterministic language models used by the speech recogniser are generated automatically from the APSG sublanguage definitions described in section 2.4. The software package that converts the APSG subgrammars into finite state grammar models to be accessed by the recogniser is described and documented in [Report 5, Report 5b]. The software package includes three grammar converters:

1) A program apsg2rtn that converts APSGs into fully equivalent non-augmented (single feature based) recursive transition network grammars;

2) A program rtn2wp that converts the RTN-output from apsg2rtn into word pair grammars to be accessed by the speech recogniser;

3) A program rtn2fsn that expands the RTN-output from apsg2rtn into approximately equivalent finite state network grammars. This converter is equivalent to the built-in RTN expander of the speech recogniser.

The multiple converter concept has been chosen in order to enable flexible and alternative experiments with approximation methods. Further, the intermediate RTN grammar format generated by the apsg2rtn converter has functioned also as input to the software which generates training databases (sentence generation and selection). Finally, the software package has served as a tool for debugging and refining the APSG grammars. The built-in debugging facilities of the software have been described in [Report 5b].

The converters have been blackbox tested with a number of tests sessions, where output from one module has been used as input to another. The following example describes the separate steps of a typical test session which also includes the sentence generation and reference file checking software (sgen and checkref, cf. section 2.2.3.):

a. Conversion of APSG subgrammars to RTN subgrammars with apsg2rtn.

b. Conversion of the RTN subgrammars to word pair subgrammars with rtn2wp.

c. Conversion of the RTN subgrammars to FSN subgrammars with rtn2fsn.

d. Generation of a random set of sentences from the RTN, word pair, and FSN subgrammars (with sgen).

e. Checking how each set of sentences is covered by the APSGs, RTNs, FSNs and word pair grammars (with the checkref sentence parser equivalent to the built-in NLP-parser of the ICM [Report 5b]).

In addition, the converter software has been compiled by various C++ compilers under various operative systems using different available debugging tools, cf. [Report 5b]. This has been an important part of the verification and ensures a high degree of code standards and portability.

The glassbox tests of the converting software has concentrated on the aspg2rtn converter, because it is a widespread (and well-founded) assumption that unification grammars normally imply an enormous reduction of rules compared to label-based coding in standard context-free grammar formats (like RTNs). Consequently, the danger of converting APSGs into strongly equivalent RTNs is that the RTN output can grow into huge sizes (e.g. in terms of number of rules, kilo bytes etc.). In general, the conversion algorithm described in [Report 5] which during the expansion of compound feature based rules attempts to eliminate impossible instantiations, has proven to be very strong and fast. However, theoretically it is possible to construct certain APSGs which bring the converter and the platform on which it runs to its knees.

The converter software has been designed mainly to support two approximation methods:

1) Generation of word pair language models;

2) Generation of fullgram language models.

A word pair language model represents an approximation method which gives priority to small-size internal representations. This reduces the computational load of recognition at the expense of perplexity (recognition rate). A fullgram finite state network gives priority to low perplexity, however, it tends to exceed the size limits imposed by the DSP version of the recogniser.

As the development of the system and the APSG subgrammars proceeded it soon became clear that fullgram language models could only be used for off line tests with the batch version of the recogniser. The DSP version was not capable of loading fullgram subgrammar sets. Especially, when fullgram language models were augmented with garbage transitions using the phrase-spotting or word-rejection facilities of the grammar converters, cf. [Report 5], the number of states and arcs grew to a size which was not feasible for the real time implementation of the recogniser. On the other hand, preliminary user tests (non-systematic tests where the system designers posed as users) made it clear, that pure word pair grammars would be insufficient to obtain an acceptable transaction success rate with non-trained ("naive") users. The system simply generated too many false answers when configurated with word pair models.

The non-systematic pseudo user tests indicated that the best right-answer rate was achieved with language models mixing word pair structures with fullgrams. Only the semantically significant parts of the APSGs, i.e. the syntactic rules that build structures actually used by semantic mapping, were converted into fully equivalent finite state language structures. The rest of the APSG rules was modelled as word pair structures. For instance, considering the sentence "jeg har kundenummer et hundrede tre og halvfjerds" ("I have customer number one hundred seventy three"), the APSGs of course apply a syntactic analysis to the entire sequence. However, the mapping rules only take the ending phrase structure denoting the number into account ("et hundrede tre og halvfjerds"). When mixing word pair with fullgram structures, the semantic mapping rules are used to determine whether a certain structure building rule is semantically significant and must be fully expanded or if it is insignificant and can be reduced to word pair structures.

Originally, the grammar converting software package was not designed to generate mixed word pair and fullgram language models based on semantic significance. However, as the converters allow the system designer to specify axioms and to manipulate parts of APSGs separately (e.g. NP structures, PP structures, etc.), such language models could be generated easily with only a minimum of manipulation by hand. In general, the converters have proven to be a very powerful and flexible tool during the development of the system.

2.2.3 Test of databases and acoustic models

As described in the section 2.2.1, Continuous Density Hidden Markov Models (CDHMMs) are used to model the acoustic units in the recognition process. The speech recogniser can use whole- or subword models, as well as a mixture of both. Therefore, the generation of the training database has aimed at taking a sufficient coverage of both whole words and subwords into account. As subword units, left-right context dependent phoneme models, denoted triphones, were chosen.

In section 2.2.1 the derivation of the language model for the speech recogniser is described. This language model was in turn used to automatically generate the training database. To ensure that the desired coverage was obtained (both for whole words and triphones), an iterative process was employed, in which an alternative set of sentences was generated, and the optimal subset with respect to word and triphone occurrences was chosen. The main module of the sentence database generation software is the sentence generator sgen, cf. [Report 5, Report 5b]. Test of this program is described in the previous section.

The result of the generation process was a set of 702 sentences, the key figures of which are shown in Figure 2.2.3.1, cf. [Report 5, p. 55]. A total of 23 persons (12 males and 11 females) were recorded. This figure is too low for speaker independent recognition, but had to be restricted due to limited resources.

CHARACTERISTICS OF THE TRAINING CORPUS:	
Total number of sentences in the corpus	702
Total number of words in the corpus	3.921
Number of different words	510
Total number of triphones in the corpus	18.695
Number of different triphones	1.370
Percentage of words rep. more than twice	63
Percentage of triphones rep. more than twice	95
Percentage of word boundary triphones	55

Figure 2.2.3.1. Key figures for the Training database. All figures are per speaker.

The total vocabulary for the dialogue project is 510 words. Of these, only 210 words are used in the demonstrator described in the present report. However, as the training tokens (words) are embedded in sentences, all models for the total vocabulary had to be trained simultaneously.

The training database has been verified by listening through all recordings. A number of errors were found and corrected. These were mainly caused by repetitions or deletions of words. Furthermore, about 5% of the utterances contained saturation errors, introduced in the downsampling process. The test results shown in the figures below were obtained before this correction. Later results have shown that the recognition rate increased approximately 1 - 2% after the corrections, cf. [Report 5a].

In order to verify the quality of the acoustic models (and the recogniser) a test database was recorded. The recording conditions were the same as for the training database, i.e a microphone was used in a laboratory with no noise and simulated telephone bandwidth [Report 5]. Whereas the criteria for the training database were mainly connected with linguistic and acoustic coverage, this was not the case for the test database. The test database was, for practical reasons chosen to be considerably smaller than the training database. The objective was to achieve an even division between male and female voices, and between voices that had previously appeared in the training corpus, and new voices. It therefore only includes 11 speakers. As the language model is subdivided into 10 subgrammars, the test database reflects this. In most cases, between 30 and 40 sentences per speaker were recorded for each subgrammar. The test database was generated partly manually and partly with the sentence generator sgen mentioned in the previous section. The key figures for the test database are shown in Figure 2.2.3.2, cf. [Report 5, p. 61]. Note that each set is divided into an "inside" and an “outside” part. The “inside” sentences are within the linguistic coverage, whereas the “outside” are not. The classification of sentences into “inside”/”outside” was conducted with the reference file checker (NLP parser) checkref mentioned in the previous section (for a further description of checkref, see [Report 5, Report 5b].

SUBGRAMMAR	“Inside”	“Outside”	Vocabulary
COMMAND	3	5	3
DATE	37	10	74
DELIVERY	9	5	31
DISCOUNT	11	5	19
HOUR	29	10	58
NUMBER	35	10	68
PERSONS	35	10	52
ROUTE	5	5	37
START	30	10	69
YESNO	11	5	26
TOTAL	205	75	211

Figure 2.2.3.2. Key figures for the test database. All figures refer to sentences.

This section reports only a very brief summary of the baseline test results obtained on the test database. For a more thorough discussion and presentation of results, see [Report 5]. These results are, of course, also part of the blackbox test of the speech recogniser. All results are obtained using word- or phrase spotting techniques, with a number of garbage- and silence models. As the present version of the dialogue system only uses whole word models, all results are for word models. Language models are of the word pair grammar type. Three tables are shown. Figure 2.2.3.3 shows the results for each subgrammar. Figure 2.2.3.4 shows the average recognition rates, when all subgrammars are combined and used in parallel, and Figure 2.2.3.5 shows a subdivision into male/female and known/unknown speakers.

SUBGRAMMAR	Word Error Rate	Sentence Error Rate
COMMAND	3	3
DATE	23	41
DELIVERY	34	55
DISCOUNT	32	53
HOUR	25	40
NUMBER	26	54
PERSONS	18	25

ROUTE	15	36
START	13	46
YESNO	23	56
AVERAGE	22	43

Figure 2.2.3.3. Recognition rates for each subgrammar.

As shown in Figure 2.2.3.2, each subgrammar differs with respect to vocabulary size. This is reflected in the recognition rates for the respective subgrammars.

GRAMMAR	Word Error	Sentence Error
COMMAND	42	30
DATE	28	42
DELIVERY	34	55
DISCOUNT	46	62
HOUR	35	46
NUMBER	28	58
PERSONS	26	34
ROUTE	29	73
START	13	47
YESNO	29	62

Figure 2.2.3.4. Average error rates for all subgrammars combined.

When all subgrammars are combined, the average recognition rate drops when compared with the figures for the individual subgrammars. This is caused by the enlarged search space, and hence harder recognition task.

As was expected from the low number of speakers represented in the training database, there is a significant (5 %) drop in performance for unknown speakers.

The results clearly show that improvements of the acoustic decoding are required. No clear conclusion has appeared, however, as to whether the principle of automatically generating training sentences is applicable. It ensures that the specified coverage is fulfilled, but on the other hand the auto-generated sentences are typically semantically deviant and difficult to pronounce naturally.

SPEAKER CATEGORY	Word Error	Sentence Error
KNOWN	19	40
UNKNOWN	24	46

MALE	21	43
FEMALE	22	43
AVERAGE	22	43

Figure 2.2.3.5. Recognition error rates for male/female and known/unknown speakers.

2.3 Other devices

This section discusses the remaining devices of the system architecture as illustrated in Figure 2.1.

2.3.1 The text recogniser

The program txtrec is a text recogniser based on grammar constrained pattern matching similar to the techniques used in modern continuous speech recognition technology [Brøndsted 1995]. The pattern matching algorithm is based on dynamic time warping where local distances between characters are accumulated to global scores. The syntactic decoding is a token-passing Viterbi-algorithm similar to the decoding scheme of the acoustic speech recogniser, cf. section 2.2. The text recogniser uses the same grammar format (e.g. word pairs, fullgrams) as the speech recogniser. The text recogniser has been designed mainly to support WOZ experiments (bottom-up tests) where the Wizard types input from the user directly on to the system, bypassing the acoustic recogniser. In future, the text recogniser may be improved to simulate acoustic word and sentence recognition rates in order to support predictive assessment of dialogues under changing recognition performances.

2.3.1.1 Test of the text recogniser

As the text recogniser was designed it was debugged and tested with different input forms. The tests were conducted as pseudo user tests, where the designer typed the user input types which can be expected in a real life WOZ experiment:

1) Possible sentences (covered by the grammars) possibly with minor spelling mistakes.

2) "Naive" input where the user has unrealistic expectations to the system and makes a request which by no means can be answered correctly.

3) Uncooperative input, e.g. nonsense utterances, where the user has absolutely no intention of using the system for its purpose.

The input-output examples below were generated with the text recogniser configurated with the Command, Hour, and Route word pair subgrammars. Garbage text patterns (marked with [*]) were inserted as loops at the start states and the end states of the grammars corresponding to the standard setup of the acoustic speech recogniser in the dialogue system.

1)

I: jeg vil gerne til odense [I would like to go to odense]

O: grammar Route, score 0.000000: jeg vil gerne til odense [I would like to go to odense]

I: jeg vil gerne til Odense [I would like to go to odense]

O: grammar Route, score -20.000000: jeg vil gerne til odense [I would like to go to odense]

I: jegvgerntlodense [Iwoudliketogtoodense]

O: grammar Route, score -52.000000: jeg vil gerne til odense [I would like to go to 0dense]

2)

I: kan jeg have min hund med til odense [can I bring my dog with me to odense]

O: grammar Route, score -200.000000: [*] til odense [* to odense]

I: kan der medbringes husdyr paa odense flyet [can you bring pets on the odense flight]

O: grammar Route, score -279.000000: [*] fra odense [*] [* from odense *]

3)

I: asdlkjs [asdlkjs]

O: grammar Hour, score -66.000000: ja syv [*] [yes seven *]

The user test reported in [Report 9b] has been conducted with the text recogniser.

2.3.2 The TLI Telephone Line Interface device

The task of the TLI device is to enable the dialogue system to connect to an ordinary telephone connection. This is achieved via the DSP-board which has a special circuit for this purpose [Ensigma 1990]. Similar to the player- and speech recogniser devices, the TLI consists of two parts, a DSP-part and an SDD-part, running on the PC-host. The DSP-part is documented in [Lindberg 1995], and will not be described in further detail here, except for the functionality it provides.

The basic functionality of the TLI includes: Detection of incoming (user) calls, detection of user hang-ups, and detection and identification of touch tones (DTMFs). The TLI reacts to input as described below:

- Ring detection: The DSP is switched to the telephone circuit, and the circuit is polled for an incoming call.
- Hangup detection: When the connection is established, the incoming signal is continuously checked for tone input. If a tone is detected, it is verified whether it is a "telephone busy" signal. If this is true, a hangup has been detected.
- DTMF detection: When a tone signal is detected, it is checked whether it corresponds to one of the DTMF tone-pair values.

Ring detection is performed every 1 second, and DTMF and hangup detection 6 times per second.

2.3.2.1 Tests performed on the TLI

Glass box evaluation. The TLI code has been debugged, and the results inspected to verify that the desired actions take place correctly.

Black box evaluation. A test dialogue has been constructed to verify that the TLI reacts correctly both with regard to commands from the ICM and hardware manager, and with regard to detection of calls, hangup and DTMF input. This is the case.

The DTMF-detection algorithm has been tested with internal and external connections, and also when speech occurs concurrently with the tones. In all cases, the identified frequences were within the required band of +/-1.5% from the reference values [AT&T Application Note

1989]. No tests have been conducted to verify whether the specifications for the twist (difference in level between the two tones), and duration of the tones are met.

The functionality and tests are described in closer detail in [Larsen 1995].

2.3.3 The reproductive speech driver

In the present system speech output is generated by using reproductive (prerecorded) speech (cf. Section 2.7) rather than a text to speech synthesiser.

2.3.3.1 Test of the reproductive speech driver

The reproductive speech driver (player) acts as a device in the dialogue system as presented in section 2.1. The implementation of the player was based on the SUNSTAR implementation and only minor changes have been made: The device driver has been changed to run under the Linux OS (from the original Venix OS) making use of the 3R Software [Lindberg, 95]. Within the SUNSTAR project extensive testing was conducted to verify correct queuing of messages to be replayed, and to verify the compliance with the system communication protocol specification.

These tests have been replicated in the present project, in part only, by simulating commands from the ICM (blackbox), and by inspecting contents of events from the driver during processing (glassbox). However, as was the case with the telephony interface driver and the recogniser driver, verification of the driver functionality has been complicated in the present system, as three of the present drivers are communicating with the same DSP32C based PC board, running the 3R software.

2.4 The linguistic module

This section will focus on how the NLP subsystem of the overall spoken language dialogue system was tested and evaluated during development.

The NLP module consists of a static linguistic description expressed in APSG grammar rules and the lexicon, and a program (the parser) which applies the linguistic information. For a more thorough description of the design of the NLP Module see [Report 7]. As a consequence of the separation of the static declarative information and the dynamic parser algorithm, this section has the following structure:

Subsection 2.4.1 presents the evaluation scenario for the NLP Module, in particular the linguistic information expressed in the grammar and the lexicon.

Subsection 2.4.2 contains a description of different kinds of test data used for testing and evaluating NLP systems, which throughout the section will function as a frame of reference for describing how different sorts of data have been used for locating deficiencies, and for subsequently adjusting them in the lingware part of the overall system.

In addition to outlining the design principles for modelling the specialised sublanguage, subsection 2.4.3 focuses on how collected data were used as a basis for defining and specifying the domain-specific sublanguage.

Subsection 2.4.4 describes how the test suite was generated and how it was applied, to ensure that the specified linguistic coverage corresponded to the coverage expressed in the lexicon and the implemented grammars of the system, and also that the semantic interpretation rules of the system generated the correct semantic representation.

Finally, the last part of this section describes the evaluation of the dynamic part, i.e. the parser of the NLP Module, cf. [Report 7]. In subsection 2.4.5.1 a brief account of the incremental implementation of the two parser-algorithms is given. Thereafter in subsection 2.4.5.2 a description of the two comparative performance tests is given.

2.4.1 The evaluation scenario

Only recently standard methodologies for the evaluation of NLP systems and components have begun to emerge, cf. [Thompson 1992, EAGLES 1994, Galliers and Sparck Jones 1993]. This emerging methodology distinguishes i.a. the purposes of the evaluation in the following way: progress and diagnostic evaluation are used during the development of a system/component, whereas adequacy evaluation is used to test a system's performance with respect to the users' needs and would usually be carried out only on completed systems.

The present report deals with the testing and evaluation performed during system development only, i.e. progress and diagnostic evaluation. The purpose of this type of evaluation is to test the system's conformity to the specifications.

The specifications of the lingware part of the NLP Module state the intended linguistic coverage in terms of grammatical and lexical coverage. For systems with a limited complexity, grammatical coverage may be tested by using systematically produced test data (test suites) which more or less exhaustively describe the domain (in larger NLP systems with a broader coverage, the length of sentences is normally not restricted, and therefore a potentially unlimited number of sentences are possible, [Chomsky 1971]), cf. subsection 2.4.2.). Lexical coverage is tested by checking that the lexical items specified are present, and by testing that their coding is correct. As the test suites can only partially describe the interaction between phenomena, tests of more complex systems are often performed additionally by using corpus data, if available. The present testing involved the use of systematically generated test data only.

The parser may be tested with the same input data as the grammar, as the parser needs to be able to handle at least all phenomena appearing in the grammar. As the functionality of the implemented parser in the present case exceeds the requirement for analysing the grammatical phenomena appearing in the grammar at present, the test data for the parser - in the initial phase of the implementation - consisted of constructed sample sentences being beyond the linguistic coverage of the implemented subsubgrammars of the system.[1]

2.4.2 Test data for NLP systems

Several EU-financed ongoing research projects have as their goal the definition of a general framework for a principled evaluation of NLP systems. The EAGLES initiative (Expert Advisory Group on Language Engineering Standards) has created a subgroup on evaluation of NLP products and projects [EAGLES 1994] which deals with the general framework for evaluation, whereas the TEMAA project (A Testbed Study of Evaluation Methodologies: Authoring Aids) deals with a concrete application of this framework. Other projects, e.g. TSNLP (Test Suites for Natural Language Processing) [Balkan et al. 1994], deal with the creation of test data for certain types of NLP applications.

[1] Using a more general grammar which was developed in order to generate sentences stored in a database used for training the word models of the system. For a description see [Report 5].

In the following paragraphs an overall description of different kinds of test data is given. This data categorisation is based on the one outlined in [Galliers and Sparck Jones 1993] and will function as a frame of reference for describing the types of data that have been used in the development of the NLP Module of the Spoken Language Dialogue System.

According to [Galliers and Sparck Jones 1993] test data, in broad terms can be divided into 1) *corpora* (written or spoken), 2) *test suites* and 3) *test collections*, the contents of which will be elaborated below.

In the present context, *corpora* will be understood as authentic linguistic material, such as running text or recorded human-human or human-machine dialogues. In the present project human-human, as well as human-machine corpora were collected, the former by the tape recordings made at a travel agency, cf. [Report 3], and the latter by Wizard of Oz (WOZ) experiments.

Depending on the size of the collected corpora and thus of its representativeness, this data type can be a useful source of information. The dialogue corpora mentioned were used as a basis for the definition of the coverage of the system, for instance word frequency lists and keyword in context concordances were made in order to help identifying and defining the domain-specific subset of natural language, cf. [Report 4]. The two types of collected data, however, were not on its own suitable for testing the system coverage for the following reasons. The coverage definition of the system was primarily based on the human-machine dialogues, so the recorded human-human dialogues - not reflecting the actual coverage of the system - were not applicable. The corpus examples from the WOZ experiments, although the basic source for defining the domain-specific sublanguage, only represented a subset of the system coverage (cf. subsection 2.4.3 below) and were therefore not well-suited as test corpus on its own.

Test suites are defined as artificially constructed material designed to fulfil a specific task, such as testing a specific component of an overall system to determine whether it performs according to its specifications. Test suites were used in the present project, an example of this is checking the agreement between designed and implemented linguistic coverage. Test suites as built by TSNLP [Balkan et al. 1994] test the grammatical coverage of a system in a quite exhaustive way, while using a very small vocabulary: In order to check that the system can handle verb valency correctly, the TSNLP test suites focus exclusively on verbs and their valence frames. The test suites used in the present project aim at testing grammatical as well as lexical coverage.

Unlike corpora and test suites, *test collections* include both input and output data, i.e. the test input data are associated with a corresponding set of expected outputs or required data. In other words, the test collection includes a specification of what the system ought to produce as output. Test collections can be used for automatic checking of conformity with expected output, but are otherwise no different from test suites.

As an example of a test collection, the data used to compare and evaluate the error recovery functionality of the two implemented parser algorithms in the overall spoken language dialogue system consisted of two parts; besides the reference material (what was actually uttered), the test collection included the expected semantic interpretation of the reference material recorded manually in advance cf. subsection 2.4.5.2 below (for a complete description of the testing of the two implemented parser algorithms, see [Report 7]).

Assessment of different kinds of test data is closely related to what kind of activity or purpose they are used for. According to Galliers and Sparck Jones [Galliers and Sparck Jones 1993], the data should, ideally, be assessed based on the following parameters: *representativeness*, *legitimacy* and *realism*. In our case, representative test data have to exhibit the characteristics of user utterances. Here the distinction between *coverage* and *distribution* corpora becomes relevant: coverage has been the main concern in the construction of test material, whereas distribution, i.e. the frequency of construction types in the utterances have had less influence. A necessary aspect of representativeness is also the size of the data set. *Legitimacy*, according to [Galliers and Sparck Jones 1993], has to do with the fact that data may be representative for the testing of some dialogue system, but not legitimate in other scenarios (op cit., p. 125). The question of legitimacy comes up in particular when reuse of data is considered, and hence is not a major concern in the present case. *Realism*, finally, is used to focus the attention on the fact that "data may be representative without being realistic for some task point of view" (op cit., p. 125). Consequently, [Galliers and Sparck Jones 1993] suggest that this point always be taken into account separately.

When providing the description of the different types of data applied for testing purposes during the development of the sublanguage model in the area of domestic flight ticket reservation, the data will be assessed according to these evaluation parameters. This applies to the corpus collected via WOZ experiments as well as the generated test suite. With respect to the application of test collections in the evaluation of the NLP module see subsection 2.4.5.

2.4.3 Corpus-based definition of the sublanguage model

The advantage of using test collections is i.a. that time can be saved by automating the testing of the correspondence between the intended (manually assigned) and automatically generated analysis results. This presupposes, however, that a fixed set of sample utterances is generated once and for all. As the test data were incrementally extended with extra sample utterances in parallel with the development of the specified linguistic coverage of the NLP module, test suites were considered more appropriate for the diagnostic and progress evaluation of the lingware part of the system. In addition, due to the relatively uncomplicated sample utterances parsed, the inspection of the automatically generated analysis results was easily conducted.

As the test suite was designed and generated in close correlation with the type of system it is designed to test (i.e. the NLP module of a spoken language dialogue system), we shall start by giving a brief description of the design criteria for modelling the domain-specific sublanguage and of how it was actually developed. For a further description of the design of the sublanguage, cf. [Report 4].

In order to meet the requirements of real-time performance of the overall system and the constraints set by current speech technology, the design constraints for modelling the sublanguage within the domain of domestic flight ticket reservation were as follows:

- In order to achieve the optimal speech recognition quality the structural grammars of the NLP-system should be characterised by low perplexity;
- The number of activated words or word models at a given wait-state should not exceed 100;
- The lexicon size should not exceed 500 words.

As mentioned above, these design criteria are motivated by the functionality of the system's speech recogniser. Besides being used for doing a linguistic analysis of the utterance, the syntactic APSG grammar of the NLP Module also serves as input to the generation of the linguistic knowledge which the speech recogniser has access to during the signal processing. If the linguistic knowledge expressed in the domain-specific grammar is too complex, the recognition performance will decrease drastically.

One measure to resolve this problem was to split the subgrammar covering flight ticket reservation up into smaller and less complex grammars, which, depending on the given event state, are activated by the Dialogue Handler of the system. Another means was - based on a collected domain-specific corpus - to apply a bottom-up approach for designing the sublanguage. Using this method brings about conspicuous drawbacks, as the grammars will be domain-specific and so not portable to other domains. However, the alternative - the top-down-approach - in which the detected linguistic phenomena are formalised and implemented in depth would result in high-perplexity grammars and an activation of a huge number of lexical entries violating more of the design principles stated above.

Following the corpus-based approach, the user utterances from the simulated human-machine dialogues thus provided a backdrop for the definition and specification of the domain-specific sublanguage. By using the corpus as both a coverage and distribution corpus - i.e. as a basis for identification of the domain-specific linguistic phenomena and for identification of the most frequently used syntactic constructions - the linguistic coverage of the domain-specific sublanguage was defined.

Though the corpus collected via WOZ-experiments per se can be said to be both realistic and legitimate, the small size of the corpus (the number of tokens is about 3800), and its consequent lack of representativeness, made it necessary to adjust the linguistic coverage based on intuition and about language in general. Especially within the sub-areas of *time* and *date*, an extension of the coverage was obviously necessary since all dates and hours should be possible, but of course not all would occur in a corpus, even if the corpus had had a more adequate size. Furthermore, in application-domains such as the current one, in which the overall usability of the system is to a large degree determined by the users' possibility of expressing times and dates, the coverage within these sub-areas was consequently extended to include conceivable expressions - under due consideration to the design criteria cf. above[2] (for a description of the procedure for defining the domain-specific sublanguage see [Report 4]).

During the development of the domain-specific subgrammar, the linguistic complexity in terms of size and recognition task perplexity was continually monitored to ensure that it did not exceed the limitations stated above. Based on the automatically converted FSN grammars and word pair grammars (cf. subsection 2.2.2 above), the syntactic subsubgrammars (in various phases of the development) were measured.

2.4.4 Testing the lingware part of the NLP module

As mentioned above, systematically generated test suites are primarily used for diagnostic purposes and applied by researchers and developers of prototypes in order to verify the actual coverage of their system during development. Before describing in detail the actual generation of the test suite for checking and debugging the lingware part of the NLP Module, the objectives of the test suite are given.

The ultimate goal of generating the test suite is to provide a tool for checking the correctness and precision of the lingware part of the NLP Module. This overall task can be broken down into the following sub-tasks:

- to ensure agreement between the defined linguistic coverage and the implemented coverage,
- to check consistency between the grammar rules and the lexical entries, and
- to check the accuracy of the instantiated semantic slots.

[2] Coordination and enumerations of times and dates, for instance, were not included in the coverage.

The method for generating the test suite can briefly be described as a gradual accumulation of sets of sentences designed to cover the phenomena defined in the specifications, in isolation and in combination.

The dialogue model of the prototype system was implemented as a recursive state transition network in which the subsubgrammars, as arcs, define the vocabulary and syntax available for the user at a given state. The splitting of the domain-specific subgrammar into less complex subsubgrammars made it possible - in the generation of the test suite - to reflect the fragmentation of the overall domain-specific subgrammar. Thus for each subsubgrammar developed, a test suite of sample sentences expressing the defined *linguistic* coverage was generated. These sample sentences were partly extracted from the domain-specific corpus and partly manually constructed based on domain and linguistic knowledge.

2.4.4.1 Description of the applied test procedure

The general method for testing the syntactic and lexical coverage of each of the subsubgrammars using test suites, can be outlined as follows. Based on the linguistic coverage definition of a given subsubgrammar, the various grammatical and area-specific phenomena plus the lexical entries within a given subsubgrammar were successively developed. After having formally expressed a given fragment of the domain-specific coverage, the EUROTRA-rule compiler was used to check for syntax errors in the implemented grammar and lexical rules. Thereafter, sub-domain sample sentences (as part of the incrementally generated test suite cf. above) covering the implemented, grammatical phenomenon and the coded lexical entries were constructed. The batch process version of the developed parser (NJAL)[3] was then used to generate the analysis of the sample sentences. The output data from NJAL, was then submitted to human inspection in order to check the agreement between the output representation and the specifications of the lingware part of the NLP module.

After having developed a subsubgrammar, its consistency was finally tested by using a sentence generator programme (SGEN) (for a more thorough description see [Report 5]. Based on the RTN version of the subsubgrammar (a product of the above mentioned converter program) and of the lexicon, the SGEN was used for testing the adequacy and precision of the implemented grammar and lexicon.

To make the method clear, consider the following area-specific expression for stating a date in Danish:

På mandag den toogtyvende i tredje

[lit: On Monday the twenty second in third]

The area-specific phenomenon expressed in this sentence falls in three parts, which can be formalised as follows:

```
Date_Phrase:- Day_of_week_Phrase,
              Ordinal_Phrase,
              Ordinal_Phrase
```

In order to avoid the acceptance of “incorrect” sentences, the activation of the Date_Phrase rule was constrained in the following way. The Day_of_week_Phrase is only valid for days of week in indefinite and singular form. The first Ordinal_Phrase only covers ordinals from første [first] to enogtredivte [thirty first]. The second one is restricted to ordinals from første [first]

[3] As the parser and the lingware part of the NLP module were developed in parallel, the powerful Eurotra parser (being capable of compiling a superset of the grammar rules defined in the present project) was applied in the initial phase of the grammar development.

to tolvte [twelfth]. Depending on the type of constraints, the various restrictions were either coded directly in the lexicon or computed by grammar rules.

After having expressed these restrictions in various ways, test sentences within the extended coverage of the implemented subsubgrammar were then generated. The test suite thus included sample sentences covering:

I) på {mandag .. søndag}	[on {Monday .. Sunday}]
II) den {første .. enogtredivte}	[the {first .. thirty first}]
II) i {første .. tolvte}	[in {first .. twelfth}]

The test was conducted stepwise; first the lexical coverage within each area-specific subconstituent was tested[4], i.e. in subconstituent (I), it was checked if Monday to Sunday (expressed in the sample sentences) gave the expected analysis results. Thereafter, the combination of subconstituents was tested. This was done by parsing one sample sentence having one representative element from each subconstituent, such as in

På *torsdag* den *femte* i *fjerde*

[lit: On Thursday the fifth in fourth]

After each step the analysis results was inspected and if any of the parsings had failed, the lexicon (step one) and the subsubgrammar (step two) were adjusted until the parser generated the expected results.

Thereafter, the SGEN was used to check the precision of the grammar rules and the lexical entries in a given subsubgrammar. If, for instance, the developed Date subsubgrammar accepted an incorrect sentence such as: "på mandag den femogtyvende i sekstende" [lit: on Monday the twenty fifth in sixteenth], the constraints obviously were too loosely expressed and hence had to be modified. In order to identify the lack of constraints in the linguistic module, the next step would be to parse incorrect sample sentences with NJAL, using the analysis result as a basis for detecting the error. The Boolean attribute value pair for distinguishing between ordinals that can refer to months and those which can't is *mth=yes;no*. This constraint is directly coded in the lexicon so in the present case the entry for seksten [sixteen] is either given a wrong value for *mth*, or, alternatively, the grammar rule for the `Ordinal_Phrase` in question lacks the attribute value; *mth=yes* and thus does not filter out the "incorrect" ordinals. After having adjusted the grammar or lexical rules the SGEN programme was run again and so forth until the generated sentences were "grammatically" correct.

As a consequence of the partitioning of the overall grammar of the system, testing of the overall grammar - in which side-effects of one implemented phenomenon on other parts of the grammar are identified - was not required. The procedure described for testing the syntactic and linguistic coverage of the system was applied in the development of all the ten syntactic subsubgrammars of the Danish dialogue system.

As described in [Report 7], the syntactic and semantic rules of the NLP-module are separate. This division - besides making it possible for the grammar writer to develop syntactic and semantic structures independently - also made it evident to separate the testing of the syntactic structures and semantic interpretation. As the above mentioned batch process version of the parser includes the option of turning off the automatic semantic analysis, it was easy to test the linguistic part of the NLP Module sequentially, i.e. to first test and check the agreement

[4] This sequential procedure seemed natural since elliptical user utterances with the scope of one single subconstituent, for instance, "på søndag" [on Sunday] covered by the `Day_of_Week Phrase` were part of the grammatical coverage.

between defined coverage and implemented coverage (including checking the consistency between grammar rules and the lexical entries) and then to check the accuracy of the instantiated semantic slots. The domain-specific test suites were reused in the testing of correctly assigned semantic values. Having formulated semantic interpretation rules covering a given subsubgrammar, the rules were thus tested on the sample sentences in the corresponding test suite. The syntactic and semantic analysis results achieved by NJAL were then manually checked for correct assignment of the semantic slots. To illustrate the types of error which had to be corrected, consider the semantic interpretation rule below which maps the structural analysis of “Den toogtyvende i tiende” [lit: The twenty second in tenth] into the following semantic object: sem={month={ones={number=10}}, day={ones={number=2}, tens={number=2}}}

```
date_p_map   =   {sem={month={ones={number=A}},   day={ones={number=C}}}   /
{cat=date_p}}
             [
                {cat=det},
                {cat=ord_p}
                  [
                     {cat=ord, scat=date, int=C}
                  ],
                {cat=p},
                {cat=ord_p}
                  [
                     {cat=ord, mth=yes, int=A}
                  ]
             ].
```

The semantics of the rule is the following; the right-hand side of the rule (marked by the ‘/’) represents the syntactic structure which must unify with the structural analysis of the input in order to activate the semantic role assignment expressed on the left-hand side of the rule. The inspection of the instantiated semantic slots often revealed, either that the semantic slots were not given any value at all or that a wrong value was assigned. In the former case which implied that the rule in question had not been used, the standard error was that the structural description expressed on the right-hand side of the rule was wrong or too restricted. In the latter case, that there was some kind of disagreement between the variables (in bold) in the semantic head and in the structural description of the rule. In any case, the semantic rules were adjusted until the parsing results showed that the semantic slots were filled correctly.

2.4.5 NLP software testing

The following subsections describe various evaluations of the NLP module software. The description falls into two parts. First a brief description of the NLP software and its functionality is given - including an outline of the diagnostic test procedure applied. Thereafter performance tests comparing the two implemented parsing algorithms based on various test data are described.

The NLP module consists of the NJAL parser together with an interface procedure and the lingware. The task of the module is to apply rules defining syntactic and semantic information in order to fill in relevant semantic objects, i.e. generate a semantic analysis of each input utterance. The NLP module takes a well-defined set of information as input, and outputs only the limited semantic information permissible within the interface data structures, (the so-called semantic objects). Natural language processing is thus not directly integrated with speech recognition, nor with dialogue management.

The functionality of the NLP module used in the prototype exceeds the requirements of the prototype design criteria cf. [Report 2], implementing a top-down and a bottom-up syntactic parsing algorithm applicable to longer input with more complexity than the short elliptical utterances foreseen for the initial prototype.

Given the application domain, some kind of *robust capability* for recovering from extra-grammatical utterances and misrecognised words in a relatively elegant fashion was necessary. The robustness facility will be touched upon in connection with the description of the performance tests given below (for a thorough description see [Report 7])

Other factors had a more direct influence at the implementation level. For instance, given the development environment, it had to be simple and quick to test new grammars, for example by allowing loading of the grammars and lexicon directly from text files into the parser. This was done by hard-coding the formalism compilation procedures into the parser itself. In addition, given the application as part of an interactive spoken dialogue system, real-time or close to real-time performance was another criterion, so that significant development resources were invested in optimisations.

2.4.5.1 The debugging of the NLP software

The development process proceeded incrementally, with subcomponents of the parser being implemented and tested, and then integrated into the parser. To give an example, after the grammar formalism compiler was developed, it was tested to check if the loading programme worked according to its specification. This testing was facilitated by the fact that the existing Eurotra rule compiler could parse a superset of the lingware formalism considered adequately expressive for the NLP in the Danish dialogue system. Throughout the development process the possibility of comparison helped detection of errors in the compiler procedure. This last factor proved to be quite useful, as the parser can be considered an experiment in parsing design and implementation: to our knowledge, no similar unification-based NL parsers implemented in the object-oriented C++ exist. Fundamental functions such as unification, which in Prolog is quite simple to implement, had to be developed from scratch. Having another system capable of parsing using the same formalism was an essential aid in this development context.

Once a prototype of the parser was ready, the test suites created for testing the linguistic coverage (cf. above) were used for testing and debugging the parser. The synchronic testing of both the prototype of the parser and the lingware part of the system using the same test suite increased the amount of possible sources of errors and thus complicated the debugging process. As with the grammar compilation procedure, parser debugging was facilitated by the existing Eurotra software. The possibility of running the developed subsubgrammars with the Eurotra parser (cf. above), thus, reduced the effort of debugging.

2.4.5.2 Testing of algorithm performance

Once the implementation of the two parsing algorithms had been debugged, a new development phase began wherein the performance of the parser was examined for weaknesses and optimisations implemented. Among these optimisations is the Left-Corner Dependency Tree, which is basically an indexing structure generated during loading of the grammars for quick look-up of rules based on the left-most daughter (for a description see [Report 7] pp. 8-9).

The availability of two algorithms operating with the same grammar formalism and data structures provided the opportunity for conducting various experiments with and accurate comparison of the parsing algorithms themselves and of their different strategies for robustness.

In order to evaluate the final performance of the algorithms and of the parser overall, purpose-specific *test suites* were created. These were mostly based on the existing test suites for grammatical coverage, but adjusted to ensure that certain rules were not over-represented when testing the performance. Especially long sentences (and thereby the rules covering this type of utterances) were under-represented in the existing test suite and therefore had to be constructed and included in the specific test suite. The narrow scope and legitimacy of this kind of generated test suite, makes it infeasible to reuse them in other testing scenarios. Note that this test was not intended to be a simulation of the real-life conditions under which the system would be operating, but an internal test of how the parser performs generally given all types of grammatical input (within the defined linguistic coverage), i.e. sets of input that require it to apply all the grammar rules and create a complete syntactic analysis of each input sentence (for a thorough description of the test and its results see [Report 7]).

Another test was run for determining the relative performance of the algorithms when faced with extra-grammatical (wrongly recognised) input. A *test collection* was created based on test results from a baseline test of the recognition component of the system, cf. [Report 5]. Each utterance was manually assigned the semantic slots, which a human could be expected to derive from it based on the limited domain. Using the manually assigned semantic objects (of each utterance) as a basic reference source, the different algorithms' analyses of the speech recognition results of the same sentences were automatically compared.

To make clear the test scenario and the testing method consider the following two examples from the test results (the test collection examples are based on the Hour subsubgrammar)

Ex. (1)

Ref.: Nej syv halvtreds

[No seven fifty]

Man.: sem={choice=0,hours={ones={number=7}}, minutes={ones={number=50}}}

Hyp.: Halv syv halvtreds

[lit: Half seven fifty]

Pars_td.: sem={hours={ones={number=7}}, minutes={tens={number=3,sign=-}}}

Pars_bu.: sem={choice=0,hours={ones={number=7}}, minutes={ones={number=50}}}

Ex. (2)

Ref.: Nej jeg vil godt have femten minutter i

[No I will like to have a quarter to]

Man.: sem={choice=0, minutes={ones={number=15, sign=-}}}

Hyp.: Nej otte femten minutter i ti

[lit: No eight fifteen minutes to ten]

Pars_td.:

sem={choice=0, hours={ones={number=8}}, minutes={ones={number=15}}}

Pars_bu.:

sem= {choice=0, hours={ones={number=10}}, minutes={ones={number=15, sign=-}}}

The various abbreviations used above, have the following meanings:

Ref = the reference text, i.e. what was actually uttered;

Man = the manually assigned semantic object;

Hyp = the recognition hypothesis, i.e. the recognition result of the reference text;

Pars_td = the automatically generated semantic object (based on the top-down algorithm);

Pars_bu = the automatically generated semantic object (based on the bottom-up algorithm).

Both the top-down and the bottom-up algorithms have a general recovery strategy implemented, the latter being inherently more robust (working bottom-up with left-corner rule invocation) which was reflected in the test results. In example (1) the semantic object generated by Pars_bu (in contrast to the one generated by Pars_td) is partly correct in that the correct time is found.

Based on the implemented hour-grammar two subconstituents can be found in the speech recognition hypothesis of example (1); halv syv [half past six] and syv halvtreds [seven fifty]. Since the top-down algorithm uses top-down rule invocation, the grammar rule covering the subconstituent syv halvtreds [seven fifty] is not predicted and can therefore not be part of the parse-results. The Pars_bu working bottom up finds (per definition) both the subconstituents and, as the right-most subconstituent (in the extraction procedure) is given precedence, this results in the partly correct meaning extraction of the reference text of example (1), namely syv halvtreds. As illustrated in example (2), the error recovery facility implemented in the bottom-up algorithm does not always give a successful result. The recognition hypothesis in example (2) is covered by three grammar rules covering n□, otte femten and femten minutter i ti, respectively. According to the description given, the Pars_td error recovers the two left-most subconstituents, while the Pars_bu extracts the left- most and the rightmost constituent. Compared to the manually assigned semantic object, both the automatic interpretations are wrong. Example (2) illustrates an overall problem for NLP subsystems in spoken dialogue systems - whatever error recovery strategy is chosen. If the wrong speech recognition result (randomly) lies within the coverage of a grammar rule, it will almost always lead to an incorrect semantic interpretation. and furthermore the possibility of rejecting the speech hypotheses as invalid is non-existent.

The test results of the two parsers' error recovery functionality [Report 7, Chapter 6] showed that the bottom-up algorithm's error recovery performance is the most robust one.

Basing the test on results collected from a speech recognition baseline test can be said to be an approximation of testing the NLP-software in realistic conditions. It was therefore considered to provide a solid basis for assessing which parsing algorithm should be chosen for the overall Spoken Language Dialogue System. Given the functionality of the speech recognition component and of the lingware module, the bottom-up parser algorithm proved to the best suited and consequently was chosen. Whether this decision was the right one will be examined further in the adequacy evaluation of the overall system.

2.4.6 Conclusion

This subsection has given a description of how the NLP module of the Danish dialogue system was tested during development. According to the defined terminology for evaluating NLP systems [Galliers and Sparck Jones 1993, EAGLES 1995], the focus has been on diagnostic (or glassbox) testing. In broad terms, the purpose of this kind of testing is to ensure correspondence between specifications and the actual implementation of a subsystem. In order to achieve this agreement, various test data were used.

The testing of the lingware part of the NLP module was exclusively based on constructed domain-specific test suites, representing the specified linguistic coverage in each subsubgrammar.

In the development phase of the NLP software, these test suites were reused in the debugging of the two implemented parsing algorithms. Thereafter, two comparative performance tests of the two parsing algorithms were carried out. Using the same lingware source, one test examined efficiency in terms of spent CPU-time. Although based on the already generated test data, the test suites had to be supplemented with a number of long sentences in order to ensure a more uniform application of the grammar rules. The other performance test extended the scope of the internal testing of the NLP-system in that the input data for the implemented parsing algorithms were base line speech recognition results. This comparative performance test was based on a test collection which made it possible to conduct the test automatically.

Determining whether the specifications of the NLP-subsystem fulfils users' needs, has not been treated in this section. In connection with the analysis of conducted user tests of the overall system, it will be evaluated whether the linguistic coverage and the parsing algorithm chosen will adequately correspond to real-life user demands.

2.5 The dialogue description

The dialogue handling module consists of the ICM and the dialogue description, cf. Figure 2.1. This section only describes the test of the dialogue description. The ICM is part of the platform the test of which is described in Section 2.1. The main issues to be tested as regards the dialogue description are:

- Does it behave as intended with respect to domain communication and is the behaviour reasonable?
- Does it handle meta-communication as intended and in a reasonable way?
- Does it permit reservations as intended and in an acceptable way?

The dialogue description was implemented and tested through a kind of prototyping. This is reflected in the division of the test into three phases:

In the first phase (Section 2.5.1) the programmer debugged the program until it functioned reasonably for basic input. In the beginning, a bottom-up strategy was used but as soon as possible a top-down approach was used instead and we never returned to the bottom-up strategy again.

In the second phase (Section 2.5.2) a blackbox test was performed. During this test ordinary bugs were corrected and a number of shortcomings were identified. These identified problems were analysed and represented in DR-frames, cf. Appendix A, and solutions were proposed. Selected solutions were implemented.

In the third phase (Section 2.5.3) a blackbox test using the same input as in the second phase was run on the improved dialogue description and identified bugs were corrected.

The dialogue description has not been subject to a glassbox test in its proper sense. DDL which is the programming language used for the dialogue description, contains a (textual level) print-out function meant for debugging. However, the contents of the test output is only to some degree automatically generated and must in many cases be written by the programmer. Furthermore, because of the rapidly changing code it would have been almost impossible to maintain data for a complete glassbox test. It would have been much too time consuming in relation to what we would gain and to the resources available. Only for final production programs a complete glassbox test may be required. So it was decided to concentrate on the blackbox test and on a user test [Report 9b].

2.5.1 The first test phase

In the beginning of the first test phase the only possible strategy was a bottom-up test. Resources for constructing artificial test surroundings for testing the dialogue description were reduced since an already existing program, called cio, from an earlier project at CPK could be used. cio simulates the interface between the module being tested and the databus (the Dialogue Communication Manager sometimes referred to as the hardware master manager), cf. Figure 2.1.

Input from the parser as well as from the database was simulated and had to be provided manually. In order to simulate input from the parser it was necessary to construct a dummy parser which would make the indicated assignments of values to fields in semantic objects. This was done at CPK.

The test data used in the first phase were all constructed by the programmer with the purpose of eliminating so many bugs that it was possible to perform a basic reservation without the system breaking down. Therefore this test can neither be called real glassbox nor real blackbox but is rather a kind of mixture of the two.

Three test files were constructed for the test in this phase. The first one included the minimum input needed for reservation of a single ticket. The second one was a basic reservation of a return ticket. The third file was a reservation of a return ticket in which each user utterance providing information was followed by user utterances asking for repetition and for correction of the input.

While using the bottom-up strategy, data were formulated as dialogue sequences written in the Sunstar Network Format (SNF). The example in Figure 2.5.1.1 shows the test sequence for reservation of a single ticket. The file contents are all written in Courier. Comments (which are not included in the file) explaining the meaning of each input line are given in brackets.

```
eve rec icm rsent "yesnoso choice: 1" ;
```

(the customer answers yes)

```
eve rec icm rsent "customerso number ones: 3" ;
```

(the user tells that his/her customer number is 3)

```
eve app icm APP_P1 $01;
```

(the database confirms the existence of customer number 3)

```
eve rec icm rsent "personsso number ones: 1" ;
```

(one person is going to travel)

```
eve rec icm rsent "idso number ones: 3" ;
```

(the id-number of this person is 3)

```
eve app icm APP_P2 ($01 "HD") ;
```

(the database confirms the existence of id-number 3; the initials of this person are HD))

```
eve rec icm rsent "routeso from: ko8benhavn";
eve rec icm rsent "routeso to: a5lborg";
```

(the desired route is Copenhagen—Aalborg)

```
eve app icm APP_P1 $01;
```

(the database confirms the existence of the route Copenhagen—Aalborg)

```
eve rec icm rsent "yesnoso choice: 0" ;
```

(the customer does not want a return ticket)

```
eve rec icm rsent "dayso day_of_week: mon dayso date day ones: 11  \
                                            dayso date month ones: 10" ;
```

(desired date of departure is Monday October 11)

```
eve app icm APP_P3 ($01 ((1993 $0A 11) $00)) ;
```

(the database confirms that Monday October 11 1993 is a valid date)

```
eve rec icm rsent  "hourso hour hours ones: 9  \
                    hourso hour minutes tens: 4  \
                    hourso hour minutes ones: 5" ;
```

(desired hour of departure is 9:45)

```
eve app icm APP_P4 ($01 3 4 ( ((9  45)  1))) ;
```

(the database confirms that 9:45 is an existing departure not sold out)

```
eve app icm APP_P6 ($01 142 680 4);
```

(the database confirms that the reservation is ok; the reference number is 142, the price is 680 kroner, and the travel will start in 4 days)

```
eve rec icm rsent "deliveryso delivery: airport";
```

(the passenger will pick up the ticket in the airport)

```
eve app icm APP_P1 $01;
```

(the database confirms that the ticket will be sent to the airport)

```
eve rec icm rsent "yesnoso choice: 0" ;
```

(the customer does not want to continue the dialogue)

Figure 2.5.1.1. Bottom-up test input to the dialogue description for reservation of a single ticket.

The agreement on formats between database and dialogue description could not be tested automatically during bottom-up test since the two modules could not run together using cio as test surroundings. However, by comparing the format of the output from the database (input for the dialogue handling module) with the format of the output from the dialogue description (input for the database) disagreements could in principle be revealed. Characteristically, however, format problems were not discovered until an early integrated system test was performed.

As soon as possible all system modules were integrated and run together as an entire system, and the bottom-up test was stopped and taken over by a top-down test. The top-down test allowed the functionality of each module to be tested in its real surroundings and the indication of input to the dialogue description was facilitated. The speech recogniser was left out in the top-down test of the dialogue description because it is important that errors can be reconstructed. The speech recogniser is very sensitive to noise and to the way in which an utterance is spoken (voice quality and intonation), which means that one cannot be sure to reproduce input in such a way that it will be recognised as the same input each time. Therefore, messages from the recogniser were simulated through direct textual input to the parser via the Dialogue Communication Manager. Leaving out the speech recogniser means that all misrecognitions which would have been caused by this module are eliminated and that the same input will always create the same output.

The input to and the output from each module were sent as output to the screen by the Dialogue Communication Manager, and could be logged in a script file. The typed input had a format corresponding to what the speech recogniser would produce, i.e. it contained a prefix, the user utterance and a postfix, and it was sent directly to the parser. An example of input is:

```
eve rec icm rsent "ja";
```

To facilitate indication of input a program was constructed, cf. Appendix B, which would allow specification of input as ordinary typed utterances, cf. Figure 2.5.1.2. The program would then expand each piece of input to the format expected by the ICM which would produce input to the dialogue description via the parser.

The example in Figure 2.5.1.2 shows typed test input in the top-down test for the same dialogue (reservation of a single ticket) as in Figure 2.5.1.1. Comments are preceded by # and are included in the file.

```
# test-P1.minimal
# prefix eve rec icm rsent "
# postfix ";
# knows:
ja
# customer:
det er kundenummer et
# persons:
en person
idnummer tre
# route:
rejsen starter i ko8benhavn
til a5lborg
# return:
nej det vil jeg ikke
# date:
mandag den tre og tyvende i ottende
# hour:
klokken elleve ti
# reserve:
# delivery:
hentes
# more:
nej
#
```

Figure 2.5.1.2. Top-down test input for reservation of a single ticket.

2.5.2 The second test phase

When the dialogue description allowed the basic reservations specified in the three test files of the first test phase without system break-downs, a blackbox test was performed. Test data for this test were constructed by the system designer who had been least involved in programming the dialogue description.

Basically, there were three types of reservation to be tested: single tickets, return tickets and discount return tickets. A thorough test of each of these types includes test cases with legal input, borderline cases which may be either legal or illegal, and clearly illegal input. In many cases it was possible to make an exhaustive test of legal key information, i.e. information which should be accepted and not cause error messages. By key information is meant the information asked for by the system, e.g. the name of a destination airport or a customer number. The key information may be embedded in many different formulations of which only a selection was tested along with the dialogue description. Different grammatical formulations were not in focus in the dialogue description test. A thorough test of formulations, i.e. which linguistic formulations lead to complete and relevant semantic objects, belongs to the parser module test, cf. Section 2.4.

The dialogue task structure formed the basis for a specification of what to test. Figure 2.5.2.3 shows the final P2 dialogue task structure. The task structure of P1/P2 has changed somewhat over time but this does not influence the basic idea of how it can be used for constructing test cases:

P1/P2 has system-directed dialogue, so the system will ask a number of questions which the user is expected to answer. The types of question asked by the system may be divided into four categories.

1. The simplest type will invite only a yes/no answer, e.g. “Do you want a return ticket?”
2. A second type is multiple choice questions inviting answers containing elements from an explicit list of alternatives, e.g.: “Shall the ticket be sent or will the traveller pick it up in the airport?”
3. A third type of question invites the user to state a proper name or the like, such as an airport or the user’s own customer number, e.g. “Please state your customer number.”
4. The fourth type is the most open type, i.e. the one which allows the broadest variety of formulations but which still concerns a specific topic, such as date of departure, e.g.: “On which date will the journey start?”

Legal key information in answers to questions belonging to the first three categories can be tested exhaustively. Legal answers to yes/no questions and to multiple choice questions are obviously limited in amount. There is also a limited amount of existing customer numbers, traveller id-numbers, and airport names stored in the database. Only for questions belonging to the fourth group can the key information be expressed in many different ways. These questions concern date and time of departure. For this group we selected a number of different date and time values. Also borderline cases and illegal cases have been tested. Borderline answers are only possible in the last two categories of questions. Examples of cases which have been tested for the four example questions immediately above are:

1. Legal: yes / no
   Illegal: I don’t know
2. Legal: please send it / he will pick it up in the airport
   Illegal: I want the ticket on Monday
3. Legal: all existing customer numbers including the smallest and largest ones
   Illegal: smallest existing customer number - 1 / largest existing customer number + 1 / 1000

4: Legal: August 31 / 31.12 (December 31) / today / on Monday
   Illegal: February 29 1994 / August 32 / 1.13 / yesterday

The three basic reservation types overlap, cf. Figure 2.5.2.1. For instance, customer number and route are needed in all cases whereas a date for the home journey is only applicable to the reservation of return tickets and discount tickets.

User meta-communication was tested, i.e. the keywords *change* and *repeat* were used in every possible position.

The constructed test files revealed a number of bugs in the dialogue description. Such bugs were corrected when they appeared. However, also larger inconveniences were discovered which could not be repaired on the fly. A couple of these were due to disagreements between specification and implementation. But the main part was caused by problems not taken into account by the specification.

Design rationale (DR) frames [Bernsen and Ramsay 1994] were used as a tool for representing the discovered problems and their analysis, cf. Appendix A. The choice of using DR-frames must be seen together with our earlier use of DSD (design space development) frames. A DSD frame represents the commitments made during the design process [Bernsen 1993, Bernsen and Ramsay 1994]. For instance, the design commitments made during the WOZ experiments were retrospectively expressed in a DSD frame, cf. [Report 6a]. A DSD frame may be seen as a snapshot of design commitments made at a certain time in a development process. Usually, a development process encompasses several DSD-frames. DR-frames, on the other hand, are used to represent the reasoning between two succeeding DSD-frames. DR-frames represent problems met with during the development process, violated design commitments some of which may be new and will be added to the next DSD-frame, and reasoning about how to solve the problems and why one solution may be preferred to others.

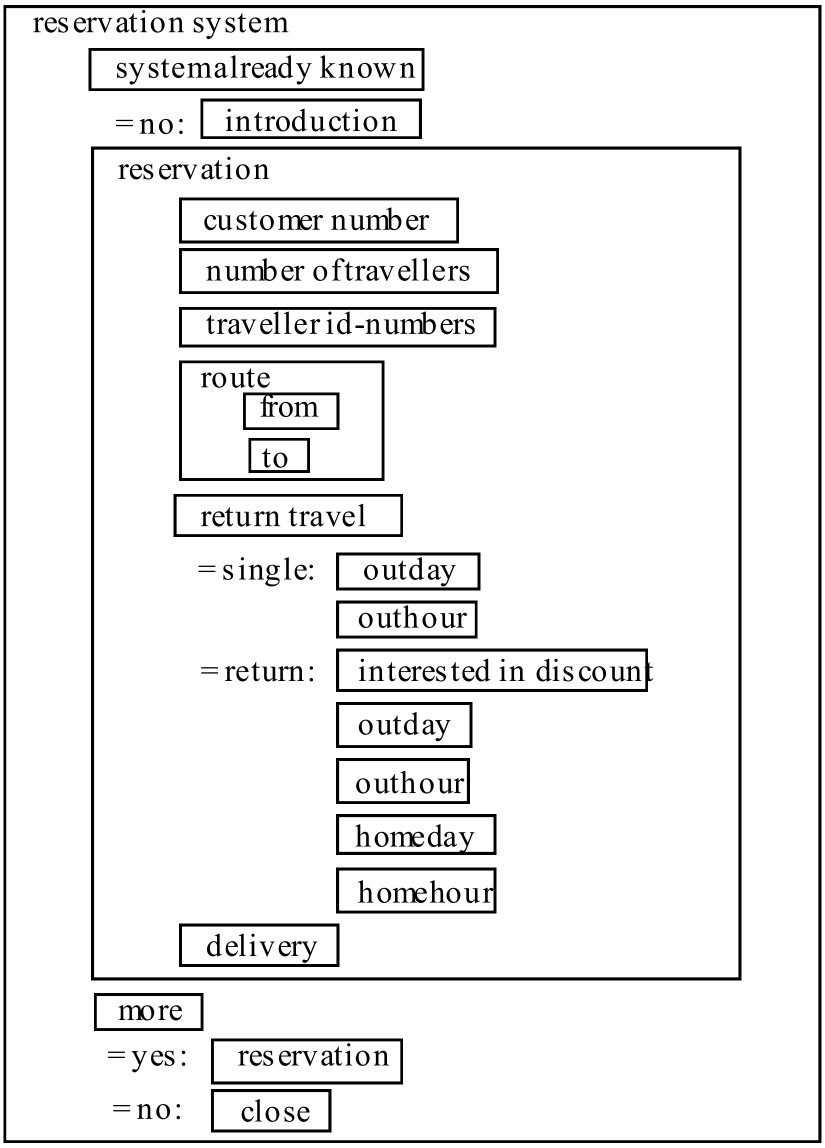

Figure 2.5.2.1. The P2 dialogue task structure.

The problems discovered during the blackbox test in the second test phase are listed below. Each problem is treated in much more detail in its DR-frame in Appendix A.

1. *Cancellation of a reservation:* In P1 it is not possible to cancel a reservation once it has been confirmed by the system.
2. *Correction of an entire reservation:* It is not possible to make corrections if the user encounters errors when the system confirms the entire reservation. Use of the change command after this confirmation will only allow the user to change the time of departure because it is the most recent piece of information provided by the user.
3. *Break off a reservation:* If the desired departure is sold out or if there are no free departures during the entire day it is not possible to break off the reservation task (except by hanging up). P1 will continue to ask for a day and a time of departure until a solution has been found.
4. *Restart:* The only way in which to stop execution of the current reservation task before it comes to its natural end is to hang up.

5. *Wait:* The user cannot suspend the dialogue for a while (e.g. by saying ‘just a moment’ or ‘wait’).
6. *No price information:* Users cannot get the price of the tickets they have reserved.
7. *Repetition of customer number:* The system asks for a customer number every time a new reservation task is started even if the user does not hang up in between.
8. *Booked departures are not mentioned:* If a certain departure is fully booked or the number of free seats is smaller than the number of travellers it is not mentioned by the system when it lists existing departures.
9. *Unavoidable discount*: If the user has indicated to the P1 system that s/he is interested in discount s/he will only be told about discount departures and s/he will not be allowed to reserve anything else than discount tickets.
10. *No discount when not explicitly asked for:* If the user has indicated to have no particular interest in discount, the system will not offer discount tickets even when possible given the departures chosen by the user.
11. *The system does not understand:* All kinds of lack of recognition and understanding are answered by the message “I do not understand”. This is not very helpful and rather annoying in cases where the system often has recognition/understanding problems.
12. *Help:* There is no way for the user to get help on how to continue the dialogue in case of problems which may be caused by the user’s need for more information before s/he is able to answer the latest system question.
13. *Relation between time of day and hour of departure:* P1 pays no attention to whether there should be a correspondence between the time of day the user has asked for and the exact hour of departure s/he indicates or whether, e.g., 9:30 means 9:30 am or pm.
14. *Repetition:* In P1 only the latest system question is repeated when the user asks the system to repeat. In many situations it would be appropriate to repeat the entire most recent system turn including the provided information.
15. *Indication of id-numbers:* Id-numbers must be indicated one at a time and if one of them is not understood the user is asked to start all over again with the id-number of the first traveller.
16. *Lack of flexibility:* P1 is very rigid in the introductory phrases leaving no initiative to the user. The usual situation in a conversation with a human travel agent is much more flexible for the user because s/he has the initiative and decides what to tell (often this is a statement providing number of persons, destination and perhaps the date of departure).
17. *Reference numbers:* Every time the P1 system is restarted it will also restart the numbering of reference numbers from one. This is not very appropriate for demonstrations.
18. *Several tickets for the same person for the same flight:* It is possible to book several tickets for the same person on the same flight during the same reservation task.
19. *Updating the reservation file:* Once a reservation has been written on the reservation file it cannot be changed in P1. Only new and/or revised reservations can be added.

20. *Waiting list:* The possibility of putting users (or rather travellers) on a waiting list if there are no free seats for the moment on the desired flight is not offered.

All elements in this list were considered important but resources were not available for implementing solutions to all the problems. Therefore, only some of them were selected for repair. We analysed how time-consuming it would be to solve each problem, how critical it would be for the user test that it had been solved, and whether the problem could be solved locally at CCS. On the basis of these considerations, solutions to the following problems from the above list were implemented: (2), 6, 7, 8, (9), (14), 15, 17, 19. Numbers in brackets indicate that only a partial solution to the problem was implemented.

While implementing the chosen solutions, the programmer discovered and solved other problems caused by the changes but not immediately foreseen. Also a few new problems were revealed and added to the above list of problems.

2.5.3 The third test phase

When the implementation of solutions to the selected problems was completed, the system was run again with the test files from the second test phase. Some of them had been slightly revised because of changes influencing the task structure.

Two system questions were removed which influenced all return ticket reservations in the test files. One of the removed questions concerned information on discount. If the user had expressed an interest in discount tickets s/he was asked if s/he wanted more information on discount. A positive answer would lead to a lengthy and partly superfluous system turn providing discount information. In the changed system version, this information was reduced and reformulated and given whenever the user had expressed an interest in discount.

The second question which was left out concerned the problem that a reservation may include, e.g., three people travelling out together but only two of them returning together. In the case of a return ticket reservation for more than one person, the system would ask if the passengers travelling home would be the same as those travelling out. However, because of functionality problems which could not immediately be solved, it was decided to remove this question although an implication was that this kind of reservation had to be handled as two separate reservations.

During the third test phase a number of bugs were corrected but no new and unknown larger problems were discovered. However, it became increasingly clear that the use of system-directed dialogue could be a problem in cases where the information expected from the user may depend on information s/he will get from the system later in the dialogue, and vice versa. For example, users may prefer to have information on departure times before they decide on the date of departure and maybe also on whether they want discount tickets. This knowledge is also important when testing the system.

2.5.4 Concluding remarks

It is a general problem of files for testing sequences that the order of the test file contents depends on the program to be tested and must be updated whenever program changes are made which influence the expected input order, even in cases where the input order is insignificant and not part of the specification.

A dialogue should not only be tested through a series of user answers which are independent of one another. The handling of possible dependencies should also be tested.

2.6 The application database

As for the dialogue description, the cio program was used for early testing of the database. When it became possible to run the database together with the other system modules, this was done except when errors had to be corrected. The recogniser was left out for the same reasons as mentioned in Section 2.5.1. Using the entire system apart from the recogniser allowed us to automatically check format agreements between the database and the dialogue description which is the module the database communicates with.

For each test the interaction with the database was logged. Originally, a test file was constructed that contained a broad selection of test cases for initial testing of the database while still using the cio program. However, the file was extended also when running the database together with the other system modules. Whenever problems were registered which seemed to be due to the database the precise query was added to a database test file. Selected cases from the test file were then used as input when errors had to be detected and corrected and the database was run with cio.

Test cases in the database test file were formulated in the SNF format [Wetzel and Torabli 1991] and each case had the expected output attached as a comment. By comparing the actual output and the expected output, the correctness of the database answer was evaluated. Examples of test cases for the database (with the meaning indicated in brackets) are shown in Figure 2.6.1:

```
per icm app app_p0 1;                          # cust +
```

(does customer number 1 exist?)

```
per icm app app_p1 (BINARY1 $01 BINARY1 $00);
                                               # route +
```

(is Copenhagen—Aalborg a valid route?)

```
per icm app app_p2 (1 1);                      # person +
```

(if customer number 1 has attached a potential passenger with id-number 1 then return the initials of this person)

```
per icm app app_p3 ((1993 BINARY1 $08 4) BINARY1 $02);
                                               # date + ((1993 8 4) 2)
```

(is Wednesday August 4 1993 a valid date?)

Figure 2.6.1. Examples of database input. `#` means that the text that follows is a comment indicating the expected output, e.g. `# cust +` means that the customer number exists and is the output from the database.

The database is implemented in C++. Three tools have been very helpful during testing. One tool is a run-time source level debugger (gdb4) from GNU (The Free Software Foundation produces copyright public domain software with free source code). When errors were detected, the debugger was used so that every step in program execution could be watched, which makes it much easier to localise and correct errors. When the debugger was used, the database was run separately together with the cio program.

A second tool is a run-time memory debugger called Purify [Purify 1993]. For a given run it keeps an account of when and where memory is allocated or initialised and when and where memory is used or de-allocated, and it watches that only memory which has been allocated and not yet de-allocated is used and that allocated memory is de-allocated when needed. In other words, it monitors memory leakage and memory usage which is very helpful and efficient.

A third tool is a prompt program for test input. This program was written by one of the system designers, cf. Appendix B.

For a number of basic classes of the database program a glassbox evaluation has been performed. All methods have been tested with different combinations of arguments in order to activate all possible parts at least once.

However, the main effort has been the blackbox test. This test has been complete for all simple queries. For more complex queries, a selection of test cases has been constructed and used. Legal as well as illegal database input in the semantic sense, including borderline cases, has been used. Illegal borderline cases and illegal input was, i.a., used to test the messages from the database which would cause the dialogue description to send out error messages. Figure 2.6.2 shows an example of input from the dialogue description to the database and the database answer that was returned to the dialogue description. The user said, in Danish, “Rejsen starter den 26. januar” (the travel starts on the 26th of January). This was misunderstood by the system which believed that the user said something like “today, the 26th of January”. The dialogue description asks the database if today is the 26th of January. The database answers that today (which was actually Friday the 6th of January) is inconsistent with Thursday the 26th of January. The error message generated by the dialogue description and sent to the user in this case is “today is not the 26th of January”.

```
PER ICM0 APP0 APP_P3
        LIST (
          LIST (
            VOID
            BINARY1 $01
            INT4 26
          )
          VOID
          INT4 0
          STRING10 "UNDEFINED"
        ) ;

EVE APP0 ICM0 APP_P3
        LIST (
          BINARY1 $0b
          LIST (
            LIST (
              INT2 1995
              BINARY1 $01
              INT1 26
            )
            BINARY1 $03
            INT1 20
            VOID
          )
        ) ;
```

Figure 2.6.2. An example of input to and output from the database.

A problem has been the lack of a real specification of what the database should be able to do. To begin with, it was considered a small and easy task to create the database. It would just include information on prices, departure times and customers, and be able to handle reservations. This may seem very straightforward but turned out to be much more complex than expected. Furthermore, the demands on the database's capabilities have changed over time. If an extra piece of information must be returned together with other pieces of information, this requires redesign of the protocols between the database and the dialogue description. This is a disadvantage made worse by the fact that a taylored query language rather than a general one has been used.

2.7 Pre-recorded output

The output module consists of two parts. One part, developed at CCS, is a sub-module in the dialogue description which generates and sends to the player names of output phrases and words (prerecorded at CCS) in the order in which they are supposed to be concatenated. The other part, developed at CPK, is the player which replays the pre-recorded words and phrases corresponding to the received names, cf. Section 2.3.3.

The CCS part of the output module was not tested separately. Rather, it was tested along with the dialogue description. We tested whether all words and phrases actually had been recorded and could be replayed and judged whether intonation in the concatenated output phrases was acceptable. More importantly, the appropriateness of the output phrases in the given environments was judged. When considered inappropriate, new words and phrases were constructed, recorded and added or used to replace old ones.

Over time it appeared to be very difficult to maintain a consistent voice. When one listens to the system's output it is clear that not all phrases were recorded under the same conditions. The output quality could be improved by recording all phrases from scratch on the same day under the same conditions.

3 Conclusion

Glassbox and/or blackbox tests have been performed on the different system components as well as on combinations of components as described in Chapter 2 above. However, the Danish dialogue system has not been blackbox tested in its entirety. All parts of the running system, apart from the speech recogniser, were to a certain extent tested along with the blackbox test of the dialogue component. Errors found were reported to the site at which the component containing the error had been developed. When the bug had been fixed the test was performed again to see how the dialogue behaved.

In the user test presented in [Report 9b], all system components were used apart from the speech recogniser which was substituted by a text recogniser (cf. Section 2.3.1) and a wizard who keyed in user responses. This test revealed a number of design problems. However, only one or two bugs were found.

Appendix A: DR-frames

During the test of the dialogue description, problems discovered were represented in DR-frames (design rational frames) along with an analysis and discussion of possible solutions. This appendix provides the DR-frames referred to in Section 2.5.2.

Design Project: P2		
Prepares DSD No. 8	**DR No.** 1	**Date:** 24.5.94
Design problem: Cancellation of a reservation		
In P1 it is not possible to cancel a reservation once it has been confirmed by the system.		
Commitments involved		
1	The system's task is to make it possible for the user to perform booking of flights between two specific cities and to decide not to book after all if user desiderata cannot be satisfied.	
Justification		
Users may change their mind very quickly, e.g. a traveller may tell the secretary while she is phoning to book that he is not going to travel after all, or the secretary may have booked for another departure than the desired one because this one was sold out, then she checks with the traveller if it is OK before hanging up and the traveller tells her to cancel the reservation. Finally, and more typical, it turns out later that the traveller is not going to travel after all or wants another departure in which cases a new call to the system is required.		
Options		
1	Allow cancellation by introducing the keyword "annuller" (cancel). This keyword can be used immediately after the system's confirmation of the entire reservation. Although not optimal a restriction to a keyword will probably be necessary because of the limited active vocabulary.	
2	Allow cancellation by explicitly asking the user after the confirmation of the entire reservation if s/he wants this reservation. This, however, would in most cases be a redundant question and hence clashed with the design commitment: Avoid superfluous or redundant interactions with users (relative to their contextual needs).	
3	Allow cancellation as a separate functionality of the system, i.e. let it be a separate task which can be performed independently of a reservation task.	
Resolution: Option 3		
Option 1 is better than option 2 since it does not introduce extra turns unless the user wants to cancel. Option 3 is preferred as solution because like option 1 it does not introduce redundant interactions but in contrast to option 1 it allows cancellation to be independent of the most recent reservation task. Furthermore, it does not require the introduction of a new keyword like "annuller" for grammar and recogniser. Instead cancellation can be activated after the introduction to the system or after the question "Do you want more?" by saying e.g. "I want to cancel a reservation.".		

Comments
Option 1 clearly would be an implementationally smaller solution than option 3 and would be closely related to the introduction of the keyword “start forfra” (restart, DR4): The system will forget all the information provided by the user (except the customer number), and then in case of cancellation it will ask if the user wants more. Option 3 requires that the system permits the user to choose between the two tasks of reservation and cancellation after the system introduction. Moreover, the database must be able to retrieve a reservation, send it to the dialogue handler and delete it from the reservation file.
Time estimate for developing and implementing solution
1 week for option 3 (2 days for option 1).
Links to other DRs
2 (correction of an entire reservation), 3 (stop reservation if desiderata cannot be fulfilled), 4 (restart) and 19 (updating the reservation file).
Documentation
Insert into next DSD frame
Option 3. Commitment 1 (partially new).
Status
Maybe do the simplest solution.

Design Project: P2		
Prepares DSD No. 8	**DR No.** 2	**Date:** 24.5.94
Design problem: Correction of an entire reservation		
It is not possible to make corrections if the user encounters errors when the system confirms the entire reservation. Use of the change command after this confirmation will only allow the user to change the time of departure because it is the most recent piece of information provided by the user. In fact the problem here is that the system provides feedback on the time of departure and then immediately after this it provides feedback on the entire reservation of which the time of departure is a subset. Hence it is not clear which of the two feedback utterances a user is referring to when saying “change”. In P1 a change command at this place is always taken to refer to the time of departure.		
Commitments involved		
1	Clear and sufficient system reaction when users start meta-communication.	
Justification		
Without sufficient repair and support mechanisms tasks cannot be satisfactorily performed when something has gone wrong.		
Options		
1	Interpret the change command at the place in question to refer to the entire reservation. Confront the user with each piece of information recorded and ask	

	whether it is correct. In case of incorrect information the user should be allowed to indicate a new value. Since the time of departure is a subset of the entire reservation the user will also have a chance to correct this piece of information.
2	Allow the user to tell which piece(s) of information is (are) wrong. Then it would only be necessary to check these pieces of information plus the ones depending on them with the user.
3	As option 1, but start with the time of departure and whenever the user changes an item s/he is asked whether there is more to be corrected.
Resolution: Option 3	
Option 3 may save turns compared to option 1 and option 2 is not feasible because of the restriction to 100 active words at a time which cannot be relaxed and because we have very limited time resources.	
Comments	
The implementation requires that the reference of the change command used after the feedback on the entire reservation is changed. There must be phrases which will confront the user with the provided pieces of information one by one and a phrase asking whether there is more to be corrected.	
Time estimate for developing and implementing solution	
At least 1 week.	
Links to other DRs	
19 (updating the reservation file).	
Documentation	
Insert into next DSD frame	
Option 3. Commitment 1.	
Status	
Will not be implemented.	

Design Project: P2		
Prepares DSD No. 8	**DR No.** 3	**Date:** 24.5.94
Design problem: Break off a reservation		
If the desired departure is sold out or if there are no free departures the whole day it is not possible to break off the reservation task (except by hanging up). P1 will continue to ask for a day and a time of departure until a usable one has been found. More generally the problem can be formulated as follows: A piece of information from the user is recognised and is semantically meaningful but as regards current resources it cannot be satisfied.		
Commitments involved		
1	The system's task is to make it possible for the user to perform booking of flights between two specific cities and to decide not to book after all if user	

	desiderata cannot be satisfied.
Justification	
It is really not very realistic that users are forced to make reservations even if they cannot get what they want and that they can only avoid it by hanging up. The present solution also means that a user cannot choose to have the out-travel but not the home-travel (e.g. if the desired departure is sold out) because the only way of avoiding a home-travel reservation is by hanging up and so the out-travel is not registered. If the user wants the out-travel s/he must call the system again and ask for a single travel.	
Options	
1	If the user's desiderata cannot immediately be satisfied, e.g. because a departure is sold out or there are no morning flights, then the user should be asked if s/he still wants to continue the reservation (of the home-travel), more precisely the question could e.g. be if the user wants to reserve for another day or departure. The exact phrasing and the continuation if the user says no should depend on where in the dialogue the user is, e.g. if the user cannot find a suitable home-travel this does not automatically imply that s/he does not want the out-travel. In other words the user should be given the possibility of stopping the reservation process in a graceful way and not just by hanging up and the possibility of reserving only a single ticket even if s/he intended to reserve a return ticket when s/he called the system.
Resolution: Option 1	
Comments	
The solution is closely related to cancellation (DR1) and restart (DR4). If the user does not want to reserve for another day or departure and does not want a possible out-travel the information s/he has provided should be cancelled (except the customer number) and s/he should be asked if s/he wants more. If s/he wants the out-travel this is just confirmed in the usual way and the user is asked if s/he wants more.	
Time estimate for developing and implementing solution	
3 days.	
Links to other DRs	
1 (cancellation) and 4 (restart).	
Documentation	
Insert into next DSD frame	
Option 1. Commitment 1.	
Status	
Do the implementation.	

Design Project: P2		
Prepares DSD No. 8	**DR No.** 4	**Date:** 24.5.94

Design problem: Restart	
The only way in which to stop execution of the current reservation task before it comes to its natural end is to hang up.	
Commitments involved	
1	Allow relevant meta-communication facilities.
Justification	
If a dialogue is going really wrong or the user thinks s/he has followed a wrong path, it will be easier and more efficient to just start all over again at once rather than wait until the system finishes the present task and asks if the user wants to do another task. And just hanging up requires a new call to the system.	
Options	
1	Restart could be introduced and triggered by a keyword (e.g. "start forfra") the use of which is allowed everywhere just as *change* and *repeat*..
2	Let the user tell the system to restart in his/her own words whenever s/he wants to.
Resolution: Option 1	
Option 1 is not an optimal solution but it is feasible within the given active vocabulary size in contrast to option 2.	
Comments	
Restart would cause the program to start all over again (i.e. by asking for a departure airport) and cancel all information provided by the user so far except the customer number.	
Time estimate for developing and implementing solution	
3 days. Note that it requires a new word ("start forfra") for grammar and recogniser.	
Links to other DRs	
1 (cancellation), 3 (break off a reservation) and 11 (degradation).	
Documentation	
Insert into next DSD frame	
Option 1. Commitment 1.	
Status	
Will not be implemented (new word model needed).	

Design Project: P2		
Prepares DSD No. 8	**DR No.** 5	**Date:** 24.5.94
Design problem: Wait		
The user cannot suspend the dialogue for a while (e.g. by saying 'just a moment' or 'wait').		
Commitments involved		

1	Allow relevant meta-communication facilities.
Justification	
Some of the secretaries who acted as subjects in WOZ 7 really missed a “wait” function. People often drop in to ask the secretary about something, also when s/he is in the middle of a telephone call. Or s/he may have to check something concerning the reservation with the person who is going to travel.	
Options	
1	Wait could be introduced and triggered by a keyword (“vent”) the use of which is allowed everywhere just as *change* and *repeat*.
2	Let the user tell the system to wait for a moment in his/her own words whenever s/he wants to.
Resolution: Option 1	
Option 1 is not an optimal solution but it is feasible within the given active vocabulary size in contrast to option 2.	
Comments	
Wait could be implemented in the same way as timeout warnings just allowing the user not to respond for a longer time interval. If it is possible to have the recogniser exploit a prioritised focus list (e.g. just containing two levels) it might be a good idea to assign a high priority to the meta-communication commands like “repeat” and “restart” when the user has issued the “wait“ command.	
Time estimate for developing and implementing solution	
3 days.	
Links to other DRs	
3 (break off a reservation) and 4 (restart).	
Documentation	
Insert into next DSD frame	
Option 1. Commitment 1.	
Status	
Maybe do the implementation. The word “vent” is already included in the vocabulary.	

Design Project: P2		
Prepares DSD No. 8	**DR No.** 6	**Date:** 24.5.94
Design problem: No price information		
Users cannot get the price of the tickets they have reserved.		
Commitments involved		
1	It should be possible for users to fully exploit the system’s task domain knowledge when they need it.	

2	Avoid superfluous or redundant interactions with users (relative to their contextual needs).
Justification	
Only some users are interested in getting information on the price. Professional users loose time on an extra dialogue turn if they are asked whether they want it. On the other hand, for users wanting the price information this may be very important.	
Options	
1	Provide full price breakdown information at the end of a reservation task.
2	Ask users if they want to know the price of their reserved tickets.
3	Always inform users about the total price of their reservation (but not its break-down into the prices of individual tickets).
Resolution: Option 3	
There is a clash between the two design commitments because of the existence of different needs in the user population. Option 3 was identified and selected as a compromise between the two relevant design commitments. Option 3 does not require extra turn taking but mentions the price briefly.	
Comments	
Since P1 already computes the price it will be easy also to output this information to the user. It would be a possibility to allow the user to obtain additional price information (a breakdown into the prices of individual tickets) via the help function (see DR 12).	
Time estimate for developing and implementing solution	
Less than 1 day.	
Links to other DRs	
12 (help).	
Documentation	
Insert into next DSD frame	
Option 3.	
Status	
Do the implementation.	

Design Project: P2		
Prepares DSD No. 8	**DR No.** 7	**Date:** 24.5.94
Design problem: Repetition of customer number		
The system asks for a customer number every time a new reservation task is started even if the user does not hang up in between.		
Commitments involved		
1	Avoid superfluous or redundant interactions with users (relative to their contextual needs).	

Justification	
It is annoying for the user to be asked several times about something which the system actually already knows.	
Options	
1	Only ask for a customer number during the first reservation task performed in a dialogue. If more than one reservation task is performed within a dialogue then only check the customer number with the user for every new reservation task by mentioning the number from the first task and asking if it is still this one.
2	Like option 1 but the system should only mention the customer number it will use and proceed directly to its next question without awaiting an answer from the user. If the customer number is not the one to be used the user must say "change" and hence initiate meta-communication to be allowed to indicate another customer number.
Resolution: Option 2	
Option 2 is preferred since this solution saves a system and a user turn in contrast to option 1 and hence is more efficient. Moreover, provided a fragile speech recogniser extra turn taking should be avoided if possible.	
Comments	
Implementationally option 2 is handled by transferring the customer number to the new task object with status (DA, UI) when switching to a new task after finishing the current reservation task.	
Time estimate for developing and implementing solution	
2-3 days.	
Links to other DRs	
Documentation	
Insert into next DSD frame	
Option 2.	
Status	
Do the implementation.	

Design Project: P2		
Prepares DSD No. 8	**DR No.** 8	**Date:** 24.5.94
Design problem: Booked departures are not mentioned		
If a certain departure is fully booked or the number of free seats is smaller than the number of travellers it is not mentioned by the system.		
Commitments involved		
1	Take users' relevant background knowledge into account.	
Justification		
The risk is that the user knows one of the fully booked departure times and asks why		

it has not been offered. The system will probably not be able to understand this question.	
Options	
1	Always insert the phrase “not fully booked” in the system’s formulation of the information.
2	Insert the phrase “not fully booked” in the system’s formulation of the information only if some of the departures in fact are fully booked.
3	List all relevant departures but add “fully booked” or “only X free seats left” after those departures which are either sold out or have too few free seats left for the number of travellers mentioned by the user.
Resolution: Option 3	
Option 2 is better than option 1 because it may be misleading to use the phrase “not fully booked” when there are no departures which are fully booked. However, option 3 is considered better than option 2 because it provides more complete information.	
Comments	
The dialogue handler receives from the database information on fully booked as well as not fully booked departures and on the number of free seats. In front of fully booked departures and departures with too few free seats the relevant one of the two new phrases mentioned under option 3 should be inserted in the output to the user.	
Time estimate for developing and implementing solution	
2 days.	
Links to other DRs	
9 (discount).	
Documentation	
Insert into next DSD frame	
Option 3.	
Status	
Do the implementation.	

Design Project: P2		
Prepares DSD No. 8	**DR No.** 9	**Date:** 25.5.94
Design problem: Unavoidable discount		
If the user has indicated to the P1 system that s/he is interested in discount s/he will only be told about discount departures and s/he will not be allowed to reserve anything else than discount tickets.		
Commitments involved		
1	It should be possible for users to fully exploit the system’s task domain knowledge when they need it.	
2	Provide clear and sufficient information to users on which possibilities they have when it is not otherwise obvious.	

Justification	
The user may know that a certain departure exists but is not aware whether discount is possible. If the user has indicated that s/he is interested in discount and chooses a departure for which discount cannot be obtained s/he will be puzzled by being told that this departure does not exist and may hence pose questions which the system is unable to understand or answer.	
Options	
1	When a user e.g. asks for a departure in the morning then let the system provide all possible departures but clearly indicate which ones can be used if the user wants discount.
2	The system should list non-discount departures only if there is no departure providing the desired discount.
3	Only list discount departures if the user has asked for discount but make clear that they are discount departures.
4	The system should not reject a user reserving for a departure for which discount is not possible. Instead it should tell the user that s/he cannot obtain discount if she insists on that departure and ask if that is okay.
Resolution: Options 1+4	
This problem could be solved by choosing options 1+4 , 2+4 or 3+4. Options 1+4 are preferred because option 1 at once tells the user all the possibilities, including those which are possible if s/he decides not to have discount after all. This may of course be redundant. On the other hand it is a very local solution. The choice of options 2 or 3 would require considerations on how to handle a situation in which a user rejects the offered discount departures, i.e. how should then the non-discount departures be made available to him/her.	
Comments	
The implementation will include a change in the database so that it always informs the dialogue handler on discount as well as non-discount departures and also a couple of output phrases must be changed.	
Time estimate for developing and implementing solution	
3-4 days.	
Links to other DRs	
8 (booked departures are not mentioned).	
Documentation	
Insert into next DSD frame	
Options 1+4. Commitment 2.	
Status	
Do the implementation.	

Design Project: P2

Prepares DSD No. 8	**DR No.** 10	**Date:** 24.5.94
Design problem: No discount when not explicitly asked for		
If the user has indicated not particularly to be interested in discount the system will not offer discount tickets even when possible given the departures wanted by the user.		
Commitments involved		
1	Provide clear and sufficient information to users on which possibilities they have when it is not otherwise obvious.	
2	Avoid superfluous or redundant interactions with users (relative to their contextual needs).	
Justification		
Users should not feel that they are tricked to pay more than necessary because the system withholds information.		
Options		
1	If the user has made a reservation which would allow him/her to get discount the system should ask if s/he wants this.	
Resolution: Option 1		
The decision to insert an extra question is a trade-off between the two mentioned commitments. There is a risk in some cases to have a redundant interaction.		
Comments		
An implementation would require the system to keep track of what kind of discount is possible for the departures chosen in case of return tickets. This requires the database to deliver this information. If discount is possible an extra question should be inserted immediately before the confirmation of the entire reservation (and before the price is computed).		
Time estimate for developing and implementing solution		
4 days.		
Links to other DRs		
9 (discount).		
Documentation		
Insert into next DSD frame		
Option 1. Commitment 1.		
Status		
Do the implementation.		

Design Project: P2		
Prepares DSD No. 8	**DR No.** 11	**Date:** 25.5.94
Design problem: The system does not understand		

All kinds of lack of recognition and understanding are answered by the message "I do not understand". This is not very helpful and rather annoying in cases where the system often has recognition/understanding problems.	
Commitments involved	
1	Clear and comprehensible error messages and repair support from the system.
Justification	
The more precisely users can be told what went wrong and possibly also how to repair it the better the chances are to solve the problem and hence to proceed successfully in the dialogue.	
Options	
1	Introduce graceful degradation. For a more detailed description of this see [Bernsen et al. 1994].
2	Introduce a more varied kind of error messaging. We propose the following variations in system messages when a given user input cannot be recognised/understood a number of times in succession: The first time: just tell that the system did not understand it and ask for repetition. The second time: tell that the system did not understand the input and ask the question again. The third time: tell that the system did not understand the input and provide examples of appropriate answers to the question which the system has asked. The fourth time: tell that the system still does not understand the input and ask if the user wants to continue in spite of understanding problems. If yes then go to step 2, i.e. ask the question again, or reset the degradation, i.e. start from step 1.
Resolution: Option 2	
Option 1 is the best solution but would require too much restructuring to be feasible within the given time limits of P2. Option 2 is less optimal but seems to be a good approximation and would be feasible provided the limited project resources and may certainly increase user satisfaction.	
Comments	
An implementation will require the dialogue handler to keep track of how many times in succession communication fails. In addition to this there is nothing new for step 1 except perhaps a change of the phrase. Step 2 just requires that the latest question can be repeated. Step 3 will take some time since example answers must be elaborated for each possible question in the dialogue. In fact step 3 includes the first step towards a help function. A simple version of "help" could just execute the non-error message part of step 3. If the user does not want to continue in step 4 all information provided by the user so far should be cancelled except the customer number and the user should be asked if s/he wants more, i.e. step 4 is actually a switch to the top task.	
Time estimate for developing and implementing solution	
2 weeks (at least, since there are many phrases).	
Links to other DRs	
12 (help) and 3 (stop reservation if desiderata cannot be fulfilled).	

Documentation
Insert into next DSD frame
Option 2. Commitment 1.
Status
Will probably not be implemented.

Design Project: P2		
Prepares DSD No. 8	**DR No.** 12	**Date:** 26.5.94
Design problem: Help		
There is no way for the user of getting help on how to continue the dialogue in case of problems which may be caused by the user needing more information before s/he is able to answer the latest system question.		

Commitments involved	
1	Ability to communicate that system or user understanding has failed.
2	Separate whenever possible between the needs of novice and expert users (user-adaptive dialogue).
Justification	
Without sufficient repair and support mechanisms tasks cannot be satisfactorily performed when something has gone wrong. There are major differences between novice and expert users, one such difference being that expert users already possess the information needed to understand system functionality.	
Options	
1	A context dependent help function may be a way of adapting a system to novices which is otherwise meant for experts in that it provides no explanations of concepts which perhaps are not well-known to everybody. For example it could be a possibility to provide the explanation on red and green discount only via the help function so that users do not have to listen to it every time they want to reserve discount tickets.
Resolution: Option 1	
Comments	
The introduction of a help function will require a careful analysis of what the systems answers should be in each case where help can be activated. Furthermore, it should be analysed if some of the explanations in P1 should then be left out and only be obtainable via the help function.	
Time estimate for developing and implementing solution	
3-4 weeks (part of the work on new phrases could be shared with DR 11).	
Links to other DRs	
11 (the system does not understand).	

Documentation
Insert into next DSD frame
Option 1.
Status
Will not be implemented.

Design Project: P2		
Prepares DSD No. 8	**DR No.** 13	**Date:** 26.5.94
Design problem: Relation between time of day and hour of departure		
P1 pays no attention to whether there should be a correspondence between the time of day the user has asked for and the exact hour of departure s/he indicates and whether e.g. 9:30 means 9:30 am or pm. In fact the problem has three variations which may be exemplified by: (1) If a user has asked for morning departures and then indicates an hour of departure which turns out only to be possible in the evening then P1 will simply make a reservation for this evening flight. (2) A user may ask for morning departures and then reserve an evening departure. (3) If the user has not indicated a specific time of day but only mentions a time of departure, e.g. 8:15 then first 8:15 am is tried and if there is no departure then 8:15 pm is tried.		

Commitments involved	
1	Provide sufficient feedback on each piece of information provided by the user.
2	Avoid superfluous or redundant interactions with users (relative to their contextual needs).
Justification	
In many cases sufficient feedback is just a repetition of the key-information provided by the user, such as time of departure, but if there is some kind of indirect contradiction feedback may only be sufficient if this conflict is made clear to the user and accepted by him/her.	
Options	
1	Always ask users when there seems to be a contradiction or in cases where it is unclear what the user means.
2	(1) If the user indicates an hour of departure which may be in accordance with the time of day s/he has asked for except that there are no flights e.g. am but only pm at the indicated hour then the system should check with the user if s/he really wants this departure. (2) If the user explicitly asks for an existing hour of departure which is not in accordance with the time of day s/he has indicated to be interested in then accept and only provide the usual feedback (i.e. repeat the hour and then go on to the next question). (3) If the user has not indicated a specific time of day but only mentions a time

	of departure, e.g. 8:15 then the system should see if there is a departure at 8:15 am. If not, then it should try 8:15 pm. If there is a departure at one of these to hours then the system should just provide the usual feedback (no matter which of the possibilities existed).
Resolution: Option 2	
Option 2 is considered to be the best solution since it is assumed that always asking the user explicitly will mean redundant interaction in nearly all cases of variations 2 and 3 whereas for variations 1 it is much more doubtful whether the user really wants the departure the system has found since it is the system that introduces an inconsistency between time of day and hour of departure. In variation 2 the user (or the recogniser) introduces the inconsistency and in variation 3 there is no time of day with which the hour of departure can be in conflict.	
Comments	
The implementation requires that the system keeps track of which time of day (if any) the user has indicated. If the user indicates an hour of departure which may be am as well as pm then the departure found should be compared to the indicated time of day, if any. If there is a contradiction the system should check with the user.	
Time estimate for developing and implementing solution	
3 days.	
Links to other DRs	
Documentation	
[Dybkjær, 25.5.94]	
Insert into next DSD frame	
Option 2. Commitment 1 (partially new).	
Status	
Do the implementation.	

Design Project: P2		
Prepares DSD No. 8	**DR No.** 14	**Date:** 26.5.94
Design problem: Repetition		
In P1 only the latest system question is repeated when the user asks the system to repeat. In many situations it would be appropriate to repeat the entire most recent system turn including the provided information.		
Commitments involved		
1	Clear and sufficient system reaction when users start meta-communication.	
Justification		
When the system has provided information, such as a telephone number or a list of departures, followed by a question and the user says “repeat” it is very likely that it is the information the user asks to have repeated and not only the question.		
Options		

1	Always repeat the most recent system turn entirely.
2	Analyse carefully if there are situations in which it will be most reasonable only to repeat the question and perhaps part of the information, i.e. a context dependent repetition.
Resolution: Option 1	
Option 1 is chosen because it is less time consuming and the profit as regards functionality of choosing option 2 is assumed to be small.	
Comments	
An implementation will be somewhat time consuming because the dialogue history of P1 does not record enough information to support the solution. The recording of the entire latest system utterance must be done in a principled way.	
Time estimate for developing and implementing solution	
2-3 weeks.	
Links to other DRs	
Documentation	
Insert into next DSD frame	
Option 1. Commitment 1.	
Status	
Will not be implemented. Wait until P3.	

Design Project: P2		
Prepares DSD No. 8	**DR No.** 15	**Date:** 26.5.94
Design problem: Indication of id-numbers		
Id-numbers must be indicated one at a time and if one of them is not understood the user is asked to start all over again with the id-number of the first traveller.		
Commitments involved		
1	Avoid superfluous or redundant interactions with users (relative to their contextual needs).	
Justification		
Superfluous interaction is boring and inefficient for the user. The more fragile the speech recogniser is the more reason there is at least not to ask users to start all over again in case of no recognition.		
Options		
1	Allow users to indicate all id-numbers in one utterance.	
2	Ask for id-numbers one at a time but only ask for repetition of those which are not understood.	
Resolution: Option 2		

Option 1 will, if the user utterance is recognised at once, reduce the number of user and system turns if there is more than one traveller. However, this solution will require the grammars to be changed and user utterances will become longer. The speech recogniser has problems with “long” utterances and the error rate can be foreseen to grow which may lead to a situation in which the user often will have to ask for correction or the system will have to ask for repetition. Option 2 will be much simpler to implement, given P1, the utterance length will not be increased but a number of annoying repetitions avoided.
Comments
The most primitive implementation of the solution will be to insert the “not understood” handling as a further hack into the handling of “Person?” (which includes “Person get”. Another fairly primitive implementation will require a local restructuring of P1. A more general implementation requires a principled restructuring of P1 as regards the handling of sub-tasks. The latter possibility will be too time consuming. The second possibility is preferred to the first one because it is not so much of a hack.
Time estimate for developing and implementing solution
1-2 days for the second possibility mentioned under comments.
Links to other DRs
Documentation
Insert into next DSD frame
Option 2.
Status
Do the implementation.

Design Project: P2		
Prepares DSD No. 8	**DR No.** 16	**Date:** 26.5.94
Design problem: Lack of flexibility		
P1 is very rigid in the introductory phrases leaving no initiative to the user. The usual situation in a conversation with a human travel agent is much more flexible for the user because s/he has the initiative and decides what to tell (often it is a statement providing number of persons and destination and perhaps date).		
Commitments involved		
1	Maximise the naturalness of user-interaction with the system.	
2	Unless a naturalness criterion cannot be met for feasibility reasons, it should be incorporated into the artifact being designed.	
Justification		
Probably the system would appear much more natural to users when some flexibility could be allowed in the beginning of a reservation dialogue because then it would correspond much to what is typical in human-human reservation dialogues, namely that the travel agent takes over and asks questions when the user has stated a couple		

of facts on what s/he wants.	
Options	
1	Allow the user to tell what s/he wants in reply to the system's first question ("Do you know this system?").
2	Like option 1, but only allow a limited set of information as a maximum, e.g. number of persons, departure and arrival airports and date.
Resolution: Option 2	
Option 2 is a trade-off between recogniser constraints and desired usability. Leaving more initiative to the user and making it possible to indicate several pieces of information at a time (i.e. enlarging the focus set of the dialogue handler) will inevitably increase users' utterance length which will provide problems for the recogniser. Therefore it will not be possible to allow users to provide any combination of information at this point so option 1 is not feasible. Instead the most common not too long combinations could be allowed such as "I would like to reserve two single tickets to Aalborg".	
Comments	
The implementation requires an analysis of and a decision on which pieces of information to allow from the user after the system's introductory turn. The pieces of information which are allowed must be included in the system's focus set. The system may as default proceed like P1 does now by asking a question for one single piece of information at a time. However, before asking a question it should check whether it already has the information it is going to ask for. If it already has the information it should go on to the next question.	
Time estimate for developing and implementing solution	
3 days (plus new grammars (CST)).	
Links to other DRs	
Documentation	
Insert into next DSD frame	
Option 2.	
Status	
Will not be implemented now. Will perhaps be implemented for P2.	

Design Project: P2		
Prepares DSD No. 8	**DR No.** 17	**Date:** 26.5.94
Design problem: Reference numbers		
Every time the P1 system is restarted it will also restart the numbering of reference numbers from one. This is not very appropriate for demonstrations.		
Commitments involved		
1	Unless a naturalness criterion cannot be met for feasibility reasons, it should be incorporated into the artifact being designed.	

Justification	
The only reason for making this change is that it may appear more realistic to people who attend a demonstration of the system that the first reference number is not one. If the reference number is one it may seem as if it is the very first time the system is used.	
Options	
1	Let the system generate a random number from which to start.
2	Let the system start from a fixed number which is not 1 but e.g. 57.
3	Store the most recent reference number on a file (must be done after each reservation). When the system is restarted the next reference number will be the one on the file plus 1.
Resolution: Option 3	
Option 3 is chosen because of people attending more than one demonstration. Furthermore, consider the situation where the system goes down after a user has made two reservations but still wants to make two more. The user has got two reference numbers (one for each of the two first reservations). When the system is restarted and the user performs the two last reservations s/he will get the same two reference numbers for these two reservations as for the first two ones when choosing option 2 and this is very likely to be the case also when choosing option 1. However this problem is avoided by option 3.	
Comments	
The implementation of the solution will only require a small change in the function which generates reference numbers plus the introduction of a reference number file.	
Time estimate for developing and implementing solution	
Less than 1 day.	
Links to other DRs	
Documentation	
Insert into next DSD frame	
Option 3.	
Status	
Do the implementation.	

Design Project: P2		
Prepares DSD No. 8	**DR No.** 18	**Date:** 26.5.94
Design problem: Several tickets for the same person for the same flight		
It is possible to book several tickets for the same person on the same flight during the same reservation task.		
Commitments involved		
1	Provide sufficient feedback on each piece of information provided by the user.	

Justification	
Serious customers probably would not book more than once for a given traveller. However, recognition errors may cause this to happen and if the user does not pay attention to it during the feedback on who is going to travel it will not be corrected. Furthermore, there may be situations in which two persons try to book for the same traveller, due to misunderstandings or a user tries to book the same ticket twice because s/he is not sure that the first reservation task was perform successfully.	
Options	
1	Let the database check that each id-number is only mentioned once during a reservation.
2	Like option 1 but also let the database check that none of the travellers has booked for the same flight previously.
Resolution: Option 2	
Option 1 will solve the problem with misrecognitions which are perhaps not discovered. Option 2 will solve this problem as well as the one with repeated reservations.	
Comments	
The implementation will require the introduction of an extra check on id-numbers in the database. Moreover, when the departure is known the reservation file of the customer should be checked for already existing reservations for the same flight for the same traveller.	
Time estimate for developing and implementing solution	
1 week.	
Links to other DRs	
Documentation	
Insert into next DSD frame	
Option 2. Commitment 1 (partially new).	
Status	
Will only be implemented if there is time.	

Design Project: P2		
Prepares DSD No. 8	**DR No.** 19	**Date:** 24.5.94
Design problem: Updating the reservation file		
Once a reservation has been written on the reservation file it cannot be changed in P1. Only new and/or revised reservations can be added. This i.a. means that the solution suggested in DR 1 (cancellation) cannot be implemented without a change to the reservation file handling. However, a solution to this problem should not affect the user in any negative way. The moment at which the reservation is written to the reservation file is important in a realistic system where there may be several users booking in parallel. Seats are only booked when the reservation is written to the reservation file. Therefore when the system confirms a reservation it is important that		

this means that the reservation has been written to the reservation file successfully so that the system does not have to tell the user later that there were not enough free seats after all.	
Commitments involved	
1	Ability to handle and execute user corrections in a proper way.
2	Do not tell the user anything for which there is no evidence.
Justification	
If the system promises the user that it can perform corrections not only to separate pieces of information but also to entire reservations it should be able to do it not only at the surface but also behind the interface, i.e. correction should be carried out in reality and not just seemingly since this would never function in a realistic system.	
Options	
1	Only update the reservation file when a task is finished and the system can be certain that there are no more corrections to the information provided.
2	Update the reservation file immediately before the entire reservation is confirmed. When there are changes to the information before the task is finished then make the changes and overwrite the old reservation on the file if the changes were acceptable.
Resolution: Option 2	
Option 1 would be easier to implement because it requires very few changes to the present P1 (only a delay of the update of the reservation file). However, a solution to the problem should not affect the user in any negative way. Option 1 would influence the moment at which the reservation is written to the reservation file. So option 1 is in conflict with commitment 2. Moreover, option 1 does not provide a solution to how to handle previous reservations (e.g. when a user wants to cancel one). Option 2 offers a solution to this problem and does not clash with any of the commitments and is therefore preferred.	
Comments	
The moment for writing on the reservation file can be the same as in P1 but option 2 requires the implementation of some simple file handling that will allow the database to retrieve a reservation from the reservation file and send it to the dialogue handler and to update a previous reservation.	
Time estimate for developing and implementing solution	
4-5 days.	
Links to other DRs	
1 (cancellation) and 2 (correction of an entire reservation).	
Documentation	
Insert into next DSD frame	
Option 2. Commitments 1+2.	
Status	
Will not be implemented.	

Design Project: P2		
Prepares DSD No. 8	**DR No.** 20	**Date:** 16.6.94
Design problem: Waiting list		
The possibility of putting users (or rather potential travellers) on a waiting list if there are no free seats for the moment on the desired flight is not offered.		
Commitments involved		
1	Sufficient task domain coverage.	
2	Make system limitations clear to users from the outset.	
Justification		
The possibility of being put on a waiting list is usually offered by travel agencies and one of our subjects directly mentioned that she missed this functionality.		
Options		
1	Offer users the possibility of being put on a waiting list in case the desired flight is fully booked.	
2	Inform users in the system introduction that a waiting list is not available.	
Resolution: Option 1		
Option 1 is obviously the better solution because it fully solves the problem. Option 2 might just add to cluttering up the system introduction with a lot of talk half of which the user cannot remember after all if the introduction is too long. And there may be other information on what the system cannot do which it would be just as relevant to inform about in the introduction as the missing waiting list.		
Comments		
The implementation of a waiting list would require a new field to be added to each record in the flight file where the number of free seats are registered. When reservations are deleted the database should check if somebody is on the waiting list. If this is the case then the first customer for whom enough free seats are available should be contacted. The system is not prepared for contacting users itself but it could print a message to a travel agent on a screen.		
Time estimate for developing and implementing solution		
5 days.		
Links to other DRs		
Documentation		
Insert into next DSD frame		
Option 1.		
Status		
Will not be implemented. Since the change of reservation task (including cancellation of reservations) is not implemented it does not make sense to put users on a waiting		

```
list .
```

Appendix B: The prompt program

The prompt program was constructed and used during test of the dialogue description and the application database. The program facilitated indication of input to the ICM module by automatically expanding ordinary typed utterances into the format expected by the ICM which would produce input to the dialogue description via the parser. Below is given a very brief description of the program. It is the description which is available on the system if one asks for help.

```
prompt$ prompt -h
        Merges commands from file and standard input to other program.
        Usage:
                prompt {options} file | other-program
        where options are [default]:
                -h          Help [this table].
                -v          Version.
        and file is input of other-program, with the line format:
                [program-command] [# comment]
                [# 'prefix' command-prefix]     // put before subsequent commands
                [# 'postfix' command-prefix]    // put after subsequent commands
                [# 'stop' command]              // default commands for stop
                [# entry ':' ]
        and other-program typically expects line-commands.
        While running the following commands may be used:
                <return>    : send current message to other-program
                ?           : print this description
                '<text>     : send <text> raw to program
                /<name>     :search for group <name>
                +           : next command
                -           : previous command
                *           : use body of current for editing
                :r          : reread input file
                :f          : new input file
                :v          : toggle view comments
                :c          : add comment to current line
                :d          : delete current line
                :s          : save current line set
                :q          : quit
                <text>      : insert <text> as body in list
                [<pre>      : use <pre> as prefix (for <text>)
                ]<post>     : use <post> as postfix (for <text>)
        The current line is marked by ---
        Press return after all commands
```

References

[AT&T Application Note 1989] Linear Prediction Based DTMF Detection for the WE DSP32 Signal Processor Family. AT&T Application Note, AP89-008DMOS, June 1989.

[Baekgaard 1995] Baekgaard, A.: A Platform for Spoken Dialogue Systems. Proceedings of the ESCA workshop on Spoken Dialogue Systems, Vigsø, Denmark, May 30 - June 2, 1995, 105-108.

[Balkan et al. 1994] Balkan, L., Netter, K., Arnold, D. and Meijer, S.: TSNLP: Test Suites for Natural Language Processing. Proceedings of Language Engineering Convention, 6-7 July 1994, ELSNET 1994.

[Bernsen 1993] Bernsen, N.O.: The Structure of the Design Space. In Byerley, P.F., Barnard, P.J. and May, J. (Eds.): Computers, Communication and Usability: Design issues, research and methods for integrated services. Amsterdam, North-Holland, 1993, 221-244.

[Bernsen and Ramsay 1994] Bernsen, N.O. and Ramsay, J.: Design Structure, Process and Reasoning. The Advancement of a Tool for the Development of Design Spaces. Esprit Basic Research project AMODEUS-2 Working Paper RP3-ID-WP28, 1994.

[Brøndsted 1994] Brøndsted, T: Stokastisk og heuristisk sprogmodellering, SPS 9, Aarhus 1994.

[Brøndsted 1995] Brøndsted, T.: The text recogniser TXTREC. Internal Note, Center for PersonKommunikation, Aalborg University, 1995.

[Chomsky 1959] Chomsky, N.: Syntactic Structures, Mouton, The Hague, 1959.

[Chomsky 1971] Chomsky, N: Syntactic Structures, 9th printing, Mouton, The Hague, 1971.

[EAGLES 1994] EAGLES - Evaluation of Natural Language Processing Systems. Draft - Work in Progress. Eagles Document EAG-EWG-PR.2, July 1994.

[Ensigma 1990] Ensigma Ltd.: DSP32C Telephony Board User Guide, version 1.0, UK January 1990.

[Galliers and Sparck Jones 1993] Galliers, J.R. and Sparck Jones, K.: Evaluating Natural Language Processing Systems. Technical Report No. 291, March 1993.

[Jacobsen 1991] Jacobsen, C.N.: SIRTrain Training Software. User Guide, Ver. 2.1", SUNSTAR Grp2-1-1, 1991

[Larsen 1995] : Lars Bo Larsen "Functional Description of the TLI", Internal note, Center for PersonKommunikation, Aalborg University, November 1995.

[Lauesen 1979] Lauesen, S.: Debugging Techniques. Software—Practice and Experience, vol.9, 1979, 51-63.

[Lindberg 1995] Lindberg, B.: "3R Software Reference Guide Ver. 5.0" Center for PersonKommunikation, Aalborg University 1995.

[Purify 1993] Purify User's Guide. Release 2.1, Pure Software Inc.,1309 South Mary Avenue, Sunnyvale, CA 94087, 1993.

[SAM 1992] ESPRIT Project SAM 2589 - Final Report, Year 3, SAM-UCL-G004, June 1992.

[Thompson 1992] Thompson, H. (ed.): The Strategic Role of Evaluation in Natural Language Processing and Speech Technology. Human Communications Research Centre, Edinburgh, 1992.

[Wetzel and Torabli 1991] Peter Wetzel and Klan Torabli: Description of the SPC and SNF Converter. Esprit Project 2094 SUNSTAR, WP II.1.3, March 1991.

[Young 1992]: Young, S.J.: HTK: Hidden Markov Model Toolkit V1.4. Cambridge University 1992.

[Young et al. 1991] Young, S.J., Russell, N.H. and Thornton, J.H.S.: The Use of Syntax and Multiple Alternatives in the VODIS Voice Operated Database Inquiry System. Computer, Speech & Language, Vol. 5, 1991.

Project Reports

The following is a list of project reports from the research programme *Spoken Language Dialogue Systems.*

1. Larsen, L.B., Brøndsted, T., Dybkjær, H., Dybkjær, L., Music, B. and Povlsen, C: State-of-the-art of Spoken Language Systems—A Survey. September 1992.

2. Larsen, L.B., Brøndsted, T., Dybkjær, H., Dybkjær, L. and Music, B.: Overall Specification and Architecture of P1. February 1993.

3. Dybkjær, L. and Dybkjær, H.: Wizard of Oz Experiments in the Development of a Dialogue Model for P1. February 1993.

4. Povlsen, C.: Sublanguage Definition and Specification. April 1994.

5. Brøndsted, T. and Larsen, L.B.: Representation of Acoustic and Linguistic Knowledge in Continuous Speech Recognition. January 1994.

5a. Larsen, L.B. and Steingrimsson, P.: Representation of Acoustic and Linguistic Knowledge in Continuous Speech Recognition. Documentation of Training and Test Databases. To appear 1996.

5b. Brøndsted, T. and Larsen, L.B.: Representation of Acoustic and Linguistic Knowledge in Continuous Speech Recognition. Program Descriptions. May 1994.

6a. Bernsen, N.O., Dybkjær, L. and Dybkjær, H.: Task-Oriented Spoken Human-Computer Dialogue. February 1994.

6b Dybkjær, H. and Dybkjær, L.: Representation and Implementation of Spoken Dialogues. May 1994.

7. Music, B. and Offersgaard, L.: The NLP Module. April 1994.

8. Lindberg, B. and Kristiansen, J.: Real-time Continuous Speech Recognition within Dialogue Systems. December 1995.

9a. Dybkjær, L., Bernsen, N.O., Brøndsted, T., Bækgaard, A., Dybkjær, H., Larsen, L.B., Lindberg, B., Povlsen, C.: Test of the Danish Spoken Language Dialogue System. January 1996.

9b. Dybkjær, L., Bernsen, N.O. and Dybkjær, H.: Evaluation of Spoken Dialogues. User Test with a Simulated Speech Recogniser. January 1996.

9c. Povlsen, C.: Adequacy Evaluation of the Linguistic Module. To appear 1996.

10. Bækgaard, A.: The Generic Dialogue System. January 1996.