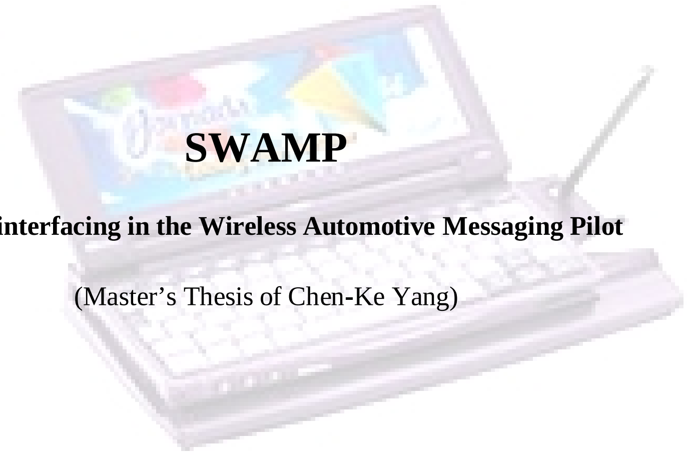


# SWAMP

**Speech interfacing in the Wireless Automotive Messaging Pilot**

(Master's Thesis of Chen-Ke Yang)

Yang, Chen-Ke (chen.yang@cmg.nl or c.k.yang@twi.tudelft.nl)

Master's Thesis, June 2001
"SWAMP:
Speech interfacing in the Wireless Automotive Messaging Pilot"

Delft University of Technology
Faculty of Information Technology and Systems
Knowledge Based Group
Zuidplantsoen 4, 2628 BZ Delft, the Netherlands

CMG Trade Transport & Industry BV
Division Technical Software Engineering (RTSE1)
Kralingseweg 241-249
3062 CE Rotterdam

Keywords: Artificial intelligence, belief desire intention model, context free grammar, dialogue management, dialogue flow, speech interfacing.

## Abstract

*Speech technology is rapidly developing and has improved a lot over the last few years. Nevertheless, speech-enabled applications have not yet become mainstream software. Furthermore, there is a lack of proven design methods and methodologies specifically concerning speech applications. So far the application of speech technology has only achieved limited success.*

*This thesis describes a project done at CMG Trade Transport & Industry. It is called SWAMP and is an example of the application of speech technology in human-computer interaction. The purpose of the project was to build a speech user interface on top of an existing application with a graphical user interface.*

*The reasoning model behind the speech interface is based on the Belief Desire Intention (BDI) model for rational agents. Other important tools that were used to build the speech user interface are the Microsoft Speech API and CLIPS.*

# Table of Contents

# Preface

This master thesis is written as part of my studies at the faculty of ITS at the Delft University of Technology. It covers the work I did on building a prototype speech user interface in the Wireless Automotive Messaging project (WAM) at CMG Trade, Transport & Industry in Rotterdam. My academic supervisor is Drs. Dr. L.J.M. Rothkrantz, Faculty of Information Technologies and Systems, Delft University of Technology. I thank him for his support during the project. My supervisor at CMG is Robert van Egmond and my advisors are Hans de Man and Eric Visser (also from CMG).

First I want to thank CMG for allowing me to work on this magnificent project and letting me free to decide and unleash my creativity upon the project. I am also very grateful to Robert, for his support, feedback and interest in my work. I'd also like to thank Eric for his advice and Hans for taking care of all other details so that I could concentrate on my work. I also owe a lot to my friend Stephen. Our conversations during the rides to work have greatly influenced this thesis. Finally a big wave to all fellow-students and colleagues at CMG for putting up with me during the past twelve months.

Personally I think speech interfacing will become more important in the future and will eventually be a common human-machine interfacing technique. But the time of the triumph of speech interfacing has not arrived yet, but I hope my work and experiences with this project will contribute to draw this day a little closer.

Chen-Ke Yang,
Rotterdam, may 2001

# 1 Introduction

## 1.1 Background

Speech is the most common mode of communication between people. Although speech communication is not a perfect process (misunderstandings, misinterpretations, etc. occur), we are able to understand each other with a very high success rate. Research has shown that the use of speech enhances the quality of communication between humans, as reflected in shorter problem solving times and general user satisfaction [Chapanis 1981]. Furthermore, speaking to humans subjectively seems to be a relatively effortless task [Nusbaum 1995]. The benefits mentioned above are some reasons that have moved researchers to study speech interaction systems between humans and computers. A quality target commonly used by many speech researchers and engineers is HAL, the talking computer in “2001: A Space Odyssey” by Arthur C. Clarke e.g. [Page 1998]. Despite many efforts, speech interfaces have only been successfully applied in a number of specific situations. Generally, it can be concluded that speech interfaces are most successful when applied in situation where:

1. Other interfaces are inappropriate e.g. using a keyboard while driving a car, for RSI patients.
2. The use of speech enables faster task completion than other modes of interaction.

In September 1999 CMG Trade, Transport & Industry (TTI) started the Wireless Automotive Messaging project (WAM-Pilot). Its purpose was to develop new wireless services in the field of traffic and transport and thus explore new business opportunities in this field. To boost the interest of potential clients it was also necessary that these new services could be demonstrated during meetings, conventions etc. The WAM pilot is an application that delivers services for highly mobile clients. Because the clients are mobile, communication is based on wireless communication infrastructure and techniques. The infrastructure used is the GSM network and the technique applied is communication with SMS messages. The current version of the WAM-Pilot is used for demonstration purposes. It has already attracted the interest of potential customers. Despite its success, it appears to have one flaw: the user interface is not suitable.
CMG wants to know what the possibilities of speech technology are and in particular if a speech interface is better suited for the WAM pilot.

## 1.2 Problem description

The WAM-Pilot consists of an application that is based on the Client-Server model. The server is stationary while the client travels with the user. One of the main problems with the WAM-pilot is that the client (an HP Jornada handheld computer) has only two methods for interaction with the user. The user can enter input by:

- Pressing keys on the relatively small keyboard of the HP Jornada.
- Using the GUI of Windows CE on the relatively small display of the HP Jornada.

Obviously, these methods of interacting with the system are not practical for a car driver. Not to mention a danger for the general traffic safety, since the driver must divide his attention between using the interface of the system and driving. Therefore, a more appropriate method for interaction is needed. A speech interface seems to be a likely candidate. The focus of the WAM-Pilot is primarily on car drivers. The goal is to develop as much relevant and interesting new services for this group as possible. Currently there are only four services available (see section 2.3) and this is considered as too few.

Summarised the following problems exist.

1. Problem : The existing user interface is not suited.
   Wish : Integration of a speech interface in the current system.
2. Problem :The amount of services available is too small.
   Wish : Develop new relevant and interesting services.

## 1.3 Deliverables

The services offered by the WAM-pilot must be analysed and the suitability of a speech interface must be studied. If appropriate, a prototype of the speech interface suitable for demonstration must be designed and implemented. Also, new services must be implemented and added to the repertoire of services already available to the WAM-pilot. These new services obviously have to be useful, interesting and compelling to customers. Apart from this thesis, four other documents have been written during the project. First a preliminary literature survey, where the state of the art in speech and other relevant technologies is analysed. Second, a design document containing the design of the application and a special grammar specification document containing the design specifics of the grammar and dialogue flow were made. At last an implementation document was created, describing the final result.

## 1.4 SWAMP

This thesis describes the SWAMP[1] project, which is an extension of the WAM-Pilot project. The SWAMP project contains an application called the SWAMP client. In summary, the SWAMP client is basically nothing different from the WAM client, except that it has a speech interface built on top of it and that it contains a few newly added services. With the addition of a speech interface, the user interface of the SWAMP application becomes multimodal; the user can access the services using both the GUI (as in the traditional way) as well as speech (the new way). Speech interaction between the user and the SWAMP application is based on dialogues. Generally, the user starts a speech interaction by indicating (via speech) what his desires are. The SWAMP application then leads the user through a dialogue in which it tries to retrieve information regarding these desires. If eventually all the necessary information is collected, the application takes the appropriate actions to realise the user's desires. To successfully manage dialogues with a user, the SWAMP application applies Artificial Intelligence (AI) techniques [Boullart 1992] to achieve some kind of reasoning. Although the SWAMP application is a client server application consisting of a SWAMP client and a SWAMP back office, this thesis does not describe the SWAMP back office extensively. The reason is that the SWAMP back office does not differ from the original WAM back office. A more detailed description of the WAM back office is given in previous WAM documentation e.g. [Achterhof 2000], [van Egmond 1999] and [van Breda 1999]. The software that is to be written will primarily be used for demonstrational purposes. Therefore, some obvious requirements have been left out of the scope of the project. The most important of which is background noise. Furthermore issues concerning the security of speech interaction have also been left out.

[1] SWAMP is an acronym for Speech Interfacing in the Wireless Automotive Messaging Pilot

## 1.5 Layout of the thesis

This thesis is built up as follows:
In chapter 2, an overview of the original WAM pilot is given, presenting a brief discussion of the hardware components of both the WAM client and the back office. In addition the communication and information flow is described. Furthermore the services available in the WAM pilot are analysed.
In chapter 3, the services of the WAM pilot are reviewed and their suitability for speech interfacing is determined. Also additional new services are discussed and subjected to the speech interfacing suitability determination test.
Chapter 4 describes the design and implementation of the SWAMP client in general, giving an overview of the architecture of SWAMP, design and implementation decisions and implementation strategy.
Chapter 5 elaborates on the speech interface. It focuses on the essential components of the speech interface (TTS, ASR and dialogue manager) and how they are implemented. Furthermore the communication architecture and mechanisms between the components are presented and discussed.
Chapter 6 discusses the dialogue manager component in greater detail. The focus is on the reasoning model used (BDI) and the AI tools applied (CLIPS) to implement this model.
Chapter 7 describes the usability tests that were conducted near the end of the project. Furthermore a discussion of the results is presented. The thesis ends with conclusions from this work and recommendations for improvement and further study.
The last appendix of this report contains a paper about knowledge-based speech interfacing in the SWAMP project. It is an extensive summary of this report.

# 2 Description of the original system

The WAM pilot is an application that delivers services for mobile users (clients). Because the clients are mobile, communication is based on a wireless communication infrastructure and technique. The infrastructure used is the GSM network and the technique applied is communication with SMS messages. This Chapter describes the original WAM client. It includes an overview of the existing technology, the hardware architecture of the WAM pilot. Furthermore a description of the information flow during communication and the services available in the WAM pilot are described.

## 2.1 Architecture of the WAM pilot

The WAM-Pilot consists of an application that is based on the Client-Server model. The server is stationary while the client travels with the user. Short Messaging Service (SMS) is used for the communication between the server and its clients. Both server and clients have hardware available to send and receive SMS messages. Although communication with SMS messages is not as fast as other means of wireless communication such as (voice) GSM or GPRS, it is very reliable. SMS messages are guaranteed to arrive, but it usually takes a few seconds. In the following sections the architecture and hardware of the client and server are discussed further. A specification of the used hardware can be found in appendix A1.

### 2.1.1 The WAM Client

The client is an HP Jornada handheld computer (Figure 2.1-1) with the Windows CE 2.11 operating system. Each client is connected to a GPS receiver on the first serial port[1] (COM 1) and to the Motor Management System (MMS) of the car on the second serial port (COM 2). The GPS receiver enables the clients to retrieve its location anywhere on the earth, while the motor management system supplies the clients with current car status information such as fuel usage, speed, oil pressure, air bags status etc. In addition, a Nokia datacard is installed in the clients' PCMCIA slot to communicate with the Server. The communication is established by means of SMS messages, which the Nokia datacard is able to send and receive. Figure 2.1-2 gives an overview of the WAM-Pilot client.

*Figure 2.1-1: The HP Jornada handheld computer*

[1] All serial ports on the HP Jornada are compatible with the RS232 standard

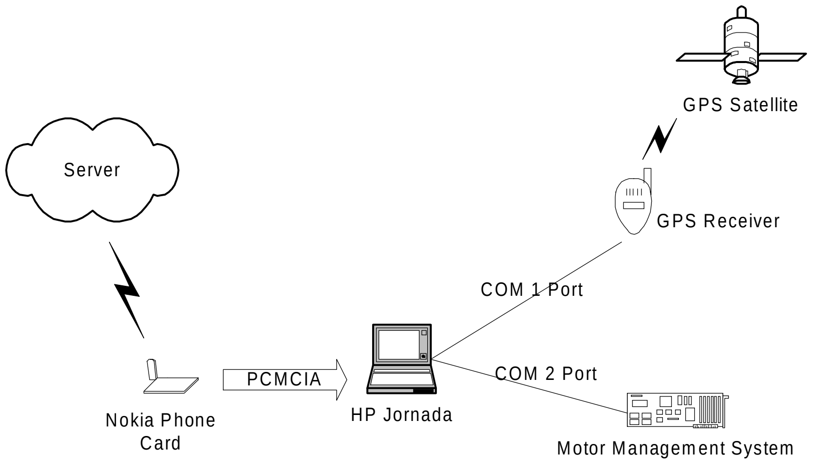


*Figure 2.1-2: Overview of the WAM-Client*

### 2.1.2 The WAM Back Office

Figure 2.1-3 gives a sketch of the current Server system. The server of the WAM-Pilot is a Windows NT workstation called the WAM Back Office (WAMBO). The WAMBO is connected to a Microsoft Access database, where all received messages and other relevant information of the WAM clients are kept. The WAMBO communicates with clients using a Nokia mobile phone (model 6110) which is connected to the serial port of the workstation using a Nokia data cable. The software of the WAMBO intercepts incoming SMS messages from the mobile phone for further processing. Processing an SMS message means parsing the message and interpreting the results to take the appropriate actions. For instance, the GPS co-ordinates of a client can be extracted from an SMS message, and used to show its location on a map. The WAMBO also contains the software (see appendix A6) to send and receive SMS messages to and from the WAM clients.

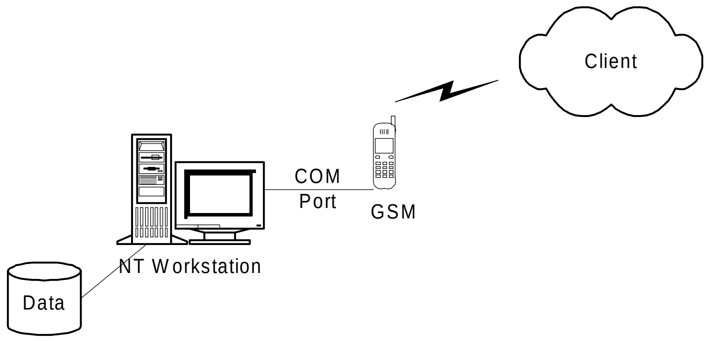


*Figure 2.1-3:Overview of the WAM Back Office*

## 2.2 Information flow

When the user wants to use a certain service offered by the WAM application, he clicks on the appropriate button on the graphical user interface of the WAM client. The WAM client then constructs an SMS message according to a predetermined format containing the users' request and other relevant information and sends it to the server (through the GSM network). Soon afterwards, the back office receives the message and processes the request. Figure 2.2-1 illustrates the typical information flow from the client to the server. The information flow of the response from the server is similar and will not be discussed further.

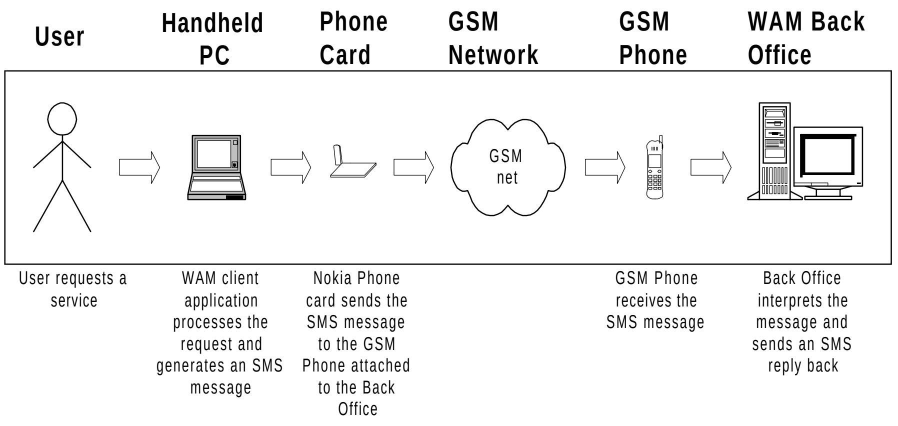

User
Handheld PC
Phone Card
GSM Network
GSM Phone
WAM Back Office
GSM net
User requests a service
WAM client application processes the request and generates an SMS message
Nokia Phone card sends the SMS message to the GSM Phone attached to the Back Office
GSM Phone receives the SMS message
Back Office interprets the message and sends an SMS reply back

*Figure 2.2-1: Typical information flow from client to server*

## 2.3 Services of the WAM Client

In this section a brief description of the services of the WAM pilot is given. For a thorough discussion of the services see discussion in previous reports e.g. [Achterhof 2000].

*Table 1: Services of the WAM pilot*

| Name | Description |
|---|---|
| Login | Before the user can use the services in the WAM-pilot, he must identify himself by supplying his name and the car ID. Furthermore the back office telephone number and trip type are needed (the Project ID must be supplied if it is a business trip). The supplied information is used to identify the client to the back office. |
| SOS call | If something goes wrong during a trip e.g. if the user doesn’t feel well, a traffic accident has occurred etc, the SOS call is activated. This service sends information to the back office, where it is decided what steps will be taken. The SOS service can either be user initiated (user doesn’t feel well) or system initiated (airbags inflated). |
| ANWB Call | This service is used to call the “Algemene Nederlandse Wielrijders Bond” (ANWB) to fix the car if, for instance, the car has broken down on the highway. When the user presses the ANWB button, an SMS message containing his GPS co-ordinates and relevant information is sent to the back office. At reception of an ANWB request message the back office automatically notifies the ANWB. |

| | |
|---|---|
| KM registration | At the beginning of a trip and whenever the user changes from trip type during a trip, a special SMS message containing kilometre count information is sent to the back office. This allows the back office to keep track of the kilometres driven by the user. Moreover, the number of kilometres driven on business trips is attributed to individual projects. During a trip, the user can indicate a change of trip type (from business to private or vice versa). All the kilometres driven from that point on are attributed to the new trip type. |
| Vehicle tracking | The position of a specific car can be requested by the WAMBO. Upon reception of a position request SMS the client sends an SMS reply, which contains its GPS position. This service needs no user interaction thus needing no speech enabling. |

# 3 SWAMP services

This chapter gives an overview of the services in the SWAMP application. The original WAM application already contained some services (described in chapter 2). Nevertheless, the number of services is considered as too few, because more services are needed to attract the attention of more and a broader range of customers. During the SWAMP project a number of new potentially interesting services have been devised. In the following sections the new services are presented. The last section contains an analysis of the suitability of a speech interface for both old and new services.

## 3.1 Elaboration of new services

This section describes the new services that have been added to the WAM-Pilot. Similar to the old services, the new ones use SMS messages for communication with the back office. Consequently, the implemented methods to construct, send and receive SMS messages can be reused by the new services. These methods are described in previous reports (e.g. [Achterhof 2000], [Yang2 2001]) and will not be discussed here. The SMS messages sent to the back office by each service have a predetermined format. This format is presented in the discussion. Because of the limited time available to finish the project, only the client part of the services is implemented. In other words the back office processing of the new services has not been implemented. If SMS messages of the new format are received, they are just ignored.

*Table 2: Overview of the new services in the SWAMP application*

| Service | **Description** |
|---|---|
| Request direction | When the user wants to know the directions to a specific location, the WAM-client retrieves a route (taking into account local traffic information) and presents it to the user. |
| Speed warning | When the user travels at a speed of more than x kilometres per hour, the WAM-client retrieves the local speed limit. If the current speed surpasses the local speed limit a warning is given. |
| Request traffic information | During driving the user might require traffic congestion information to aid him to plan a route to his destination. This service provides him with the needed information. |
| Request important corporate information | While driving the user might require information e.g. appointments, stock information. |

### 3.1.1 Request direction

| Name | **Direction Request Dialog** |
|---|---|
| Files: | DirectionReqDlg.cpp, DirectionReqDlg.h |
| Description: | Dialog containing controls to enable the user to graphically request directions. |
| Exports: | None |
| Uses: | SMS API, GPS API |
| Input: | destination, source (optional) |
| Output: | A route from source to destination |

To support the request direction service, the original WAM client is expanded with a Direction request dialog and a new SMS message type. The direction request dialog contains two selection lists from which the user can choose the source and the destination location. The default value for the source is the current location. After the user clicks the OK button, the GPS co-ordinates of the source and destination are retrieved (using the GPS receiver and/or the location-GPS co-ordinates list). Then an SMS (in the request direction SMS message format) is sent to the back office containing the request. The new SMS message type contains, predetermined, comma separated fields (see Table 3).

*Table 3: Request direction SMS message format*

| Field number | Field contents |
|---|---|
| 0 | Type |
| 1 | Driver ID |
| 2 | Car ID |
| 3 | Source GPS Latitude |
| 4 | Source GPS Longitude |
| 5 | Destination GPS Latitude |
| 6 | Destination GPS Longitude |
| 7 | Date and time |
| 8 | Comments |

### 3.1.2 Speed warning

Whenever the driving speed exceeds a certain threshold value, a message is sent to the back office requesting the speed limit of the current location. If the current speed is higher than the speed limit a warning is given. This service needs no additional GUI components.

*Table 4: The speed warning SMS message format*

| Field number | Field contents |
|---|---|
| 0 | Type |
| 1 | Driver ID |
| 2 | Car ID |
| 3 | GPS Latitude |
| 4 | GPS Longitude |
| 5 | Comments |

### 3.1.3 Request traffic information

| Name | **Request traffic information Dialog** |
|---|---|
| Files: | TrafficInfoDlg.cpp, TrafficInfoDlg.h |
| Description: | Dialog containing controls to enable the user to graphically request traffic information. |
| Exports: | None |
| Uses: | SMS API, GPS API |
| Input: | Street name or location |
| Output: | Traffic information |

For this service a new button is added to the GUI of the client. When this button is clicked a dialog containing controls to enable the user to graphically request traffic information is shown.

After the user has filled in the necessary information, an SMS message conforming to the request traffic information SMS message format (Table 5) is sent to the back office.

*Table 5: The request traffic information SMS message format*

| Field number | Field contents |
|---|---|
| 0 | Type |
| 1 | Information type (local or global) |
| 2 | Driver ID |
| 3 | Car ID |
| 4 | GPS Latitude |
| 5 | GPS Longitude |
| 6 | Comments |

### 3.1.4 Request important corporate information

In order to host the request important corporate information service, the WAM client is expanded with a request corporate information dialog. In this dialog the user can chose between four buttons: appointments, stock, telephone number, and supply.

| Name | **Request Corporate Information Dialog** |
|---|---|
| Files | CorporateInfoDlg.cpp, CorporateInfoDlg.h |
| Description | Dialog containing controls to enable the user to graphically request corporate information. |
| Exports | None |
| Uses | SMS API, GPS API |
| Input | Type of information needed |
| Output | Requested information |

After the user presses the appointment button the request appointments dialog is shown. This dialog contains GUI controls allowing the user to indicate in which time interval the appointments must be in.

| Name | **request appointments dialog** |
|---|---|
| Files | AppointmentReqDlg.cpp, AppointmentReqDlg.h |
| Description | This dialog contains GUI controls allowing the user to indicate in which time interval the appointments must be. |
| Exports | None |
| Uses | SMS API, GPS API |
| Input | Time interval |
| Output | Scheduled appointments in the specified time interval |

If the user presses the request telephone number button the request telephone number dialog is shown. This dialog contains GUI controls allowing the user to indicate whose telephone number he wants to retrieve.

| Name | **request telephone number dialog** |
|---|---|
| Files | TelephoneReqDlg.cpp, TelephoneReqDlg.h |
| Description | This dialog contains GUI controls allowing the user to indicate whose telephone number he wants to request. |
| Exports | None |

| | |
|---|---|
| Uses | SMS API, GPS API |
| Input | Name |
| Output | Telephone number corresponding to the name |

Pushing the stock button invokes a dialog in which the user can select the company in whose stocks he is interested.

| Name | **request stock information dialog** |
|---|---|
| Files | StockReqDlg.cpp, StockReqDlg.h |
| Description | Dialog in which the user can choose the company in whose stocks he is interested |
| Exports | None |
| Uses | SMS API, GPS API |
| Input | Company name |
| Output | Stock information of the company |

Pushing the supply button invokes a dialog in which the user can query the amount of a certain product in the supply-base of the company. This service is interesting for users who order products for their company, but lack the information on the amount of products to be bought because it is only available at the last moment e.g. in Just In Time (JIT) processes.

| Name | **supply-base management dialog** |
|---|---|
| Files | SupplyDlg.cpp, SupplyDlg.h |
| Description | Dialog in which the user can query the availability of a product in the companies supply-base. |
| Exports | None |
| Uses | SMS API, GPS API |
| Input | Product |
| Output | Supply information of the product |

After the required information is supplied, an SMS is sent to the back office containing the request. The SMS message contains, predefined, comma separated fields (see Table 6).

*Table 6: The request corporate information SMS message format*

| Field number | Field contents |
|---|---|
| 0 | Type |
| 1 | Information type (appointment, stock, telephone number) |
| 2 | Driver ID |
| 3 | Car ID |
| 4 | GPS Latitude |
| 5 | GPS Longitude |
| 6 | Target (a person or a company, depending on the value of information type) |
| 7 | Begin time (is only valid if information type is appointment) |
| 8 | End time (is only valid if information type is appointment) |
| 9 | Comments |

## 3.2 Suitability of a speech interface for each service

Before starting to design and implement a speech interface, it should be established that speech recognition is indeed an appropriate interface technology. Therefore an analysis should be done to determine which tasks of the application will benefit most from a speech interface and which will be more effectively handled using other types of interfaces. Table 7 gives the results of this analysis for the services (both old and new) of the SWAMP application. The second and the third column indicate the suitability of a speech interface (SUI) and graphical user interface (GUI) respectively (a "+" sign means more suitable and "-" means less suitable). From the results of the analysis it has been concluded which services are suitable to be supported by a speech interface and which are not. Services for which speech interface support will be implemented are indicated with a "●" sign in the fifth column of Table 7.

*Table 7: Services of the SWAMP project and their suitability for speech interfacing*

| Service | SUI | GUI | Comments | |
|---|---|---|---|---|
| General UI control | ++ | ++ | UI control must be available both with Speech and GUI. Speech is obviously more effective during driving. | ● |
| Login | + | ++ | Logging in is usually done before the driver starts driving. So this is not a driving situation. On the other hand data (such as project codes) can be very annoying to enter with a keyboard. In that case speech may come in very handy. | ● |
| SOS call | ++ | ++ | Speech enabling this service is essential, since a car accident can put a driver in a position in which he cannot reach the Pilot's keyboard (with his hands). | ● |
| ANWB Call | + | ++ | The ANWB service in the WAM-pilot sends a message containing the location and type of problem to an ANWB help service. Voice enabling this service is not essential, but appropriate. | ● |
| KM registration | + | ++ | When initialising KM registration the same situation as with Login occurs. The actual KM registration itself is an automatic process, the user is not aware of it. The user just has to supply some parameters. | ● |
| Vehicle tracking | - | - | This is a BO initialised command. The user is not aware of this. So no speech assistance is needed here. | |
| Speed warning | ++ | - | An audible speed warning has far more effect then a graphical one in drawing the driver's attention that he is speeding. | ● |
| Request directions*[1] | ++ | + | A typical car driving question. Thus it should be supported by the interface. The response is also best presented using speech. | ● |

[1] "*" after the service name indicates a new service

| | | | | |
|---|---|---|---|---|
| Local traffic speed* | ++ | - | System alerts the driver about upcoming speed checks. The use of speech (sounds) is ideal in this case. | ● |
| Request Local traffic Information* | ++ | + | Also a typical car driving question. | ● |
| Request corporate information * | ++ | ++ | Request for the latest updates or changes of the corporate/project status. | ● |

The measurement criteria for the suitability of the speech interface are:

**1. Estimated minimal time/effort required from the user**
Each service requires some parameters to be present before the service can start. For the sake of the measurement we define that a service is successfully accomplished when all parameters are present (what happens afterwards is unimportant for the measurement). The measurement consists of comparing the amount of GUI actions needed to successfully accomplish the service (amount of mouse clicks, popup windows, button presses etc.) against the estimated amount of atomic utterances needed to achieve the same. An atomic utterance is a speech utterance that contains just one parameter (In practice, utterances containing more that one parameter are common). In both cases, the best case scenario to accomplish the task is used (no errors, no misunderstandings, no false recognition etc.). The comparison is somewhat out of place, since SUI and GUI are quite different modalities. Nevertheless, it is used because of the absence of absolute measurement techniques.

**2. Estimated attention requirement**
An important issue to consider is that the attention required from the user to accomplish the service during driving should not be too high, because this can influence the traffic safety. Here also, there are no absolute measurement techniques available. It is obvious that saying a name requires far less attention than selecting the name from a list (especially if it is a long list). Furthermore, the attention requirement is also affected by the ergonomy of the GUI and/or the recognition rate of the speech interface. So, just as in the previous measurement the result is based on a (subjective) comparison between the GUI and the SUI. Table 8 shows the possible attention factors and their attention needs; the small display and small keyboard of the HP Jornada have been taken into account in the GUI actions.

*Table 8: Actions factors and their attention needs*

| Action | Attention need |
|---|---|
| Utterance | Normal |
| Button click | Normal |
| Key press | High |
| Select from list | High |
| Read one line from screen | High (but output is permanent) |
| Speech output | Normal (but output is volatile) |

# 4 The SWAMP application

This chapter gives an overview of the SWAMP design and implementation. The discussion on the design includes the design decisions, requirements, overall architecture and components. The implementation discussion describes important implementation decisions and the implementation strategy.

## 4.1 Design

This section describes the design of SWAMP client. The discussion covers a general overview of the design process. For a detailed report on the design process and results (including UML diagrams) see the SWAMP client design document [Yang2 2001].

### 4.1.1 Objectives

This section discusses the objectives of the speech interface. These objectives have been accumulated from brainstorm sessions and meetings with colleagues. During the compilation of this list, limited consideration has been given to technical possibilities and capabilities available in the current hardware. The reasons for this decision are the rapid advancements in the hardware speed and capabilities and the varying requirements of available speech engines, ranging from 386 to Pentium III computers or from several kilobytes to several hundreds of megabytes disk space. The list of objectives is summarised in Table 9.

*Table 9: The objectives of the SWAMP project*

| NR | Objectives |
|---|---|
| 1 | The speech interface must be integrated in the current WAM-Pilot application. |
| 2 | The speech interface must not affect the current functionality of the WAM-pilot application. |
| 3 | Current services of the WAM-pilot must be accessible through the Speech interface (where applicable). |
| 4 | The speech interface must be stable and demonstrable. |
| 5 | The interface must be English, intuitive and easy to use. |
| 6 | The interface must have real time performance. The time boundary is that the delays should not be annoyingly long. (Unfortunately, no exact boundary could be found for a time that can be considered as “annoyingly long”. So the acceptance test of the application should determine this). |
| 7 | The speech interface should not require too much attention from the user. This implies that an acceptable recognition and intelligibility rate should be accomplished. A typical driver has no problems with dividing his attention between driving and having a conversation with a real passenger. So, the fault must be sought at the speech interface when problems do occur if the real passenger is substituted by a speech interface. |
| 8 | The interface must support undo/cancel/interrupt functions. |
| 9 | The interaction must be dialogue based, so the speech interface can ask for clarification and/or confirmation if a user’s utterance is not or partially understood. |
| 10 | The user must have control over the user interface configuration. |
| 11 | The application must work on a HP Jornada with the Window CE OS. At this moment there is no suitable speech development software available for the CE OS, so the speech interface will be simulated on an NT workstation as demonstration. |

| | |
|---|---|
| 12 | The interface must be speaker-independent (to eliminate the need for extensive training). |
| 13 | A simple mechanism must be implemented to modify and create new vocabulary. |
| 14 | Minimal use of screen area must leave space for other applications. |

### 4.1.2 Approaches

A number of different approaches can be thought of to design the speech interface. This section discusses two rather different approaches: The client-side recognition approach and the server-side recognition approach. As the names suggest, the approaches are distinguished by the location the speech input is processed.

#### 4.1.2.1 Server-side recognition

In the server side approach the speech recognition part of the system resides at the server. Utterances from the user are directly transmitted via the telephone line to the server where they are processed. The reactions are also synthesised at the server and sent back via telephone lines. Figure 4.1-1 shows a graphical representation of this approach.

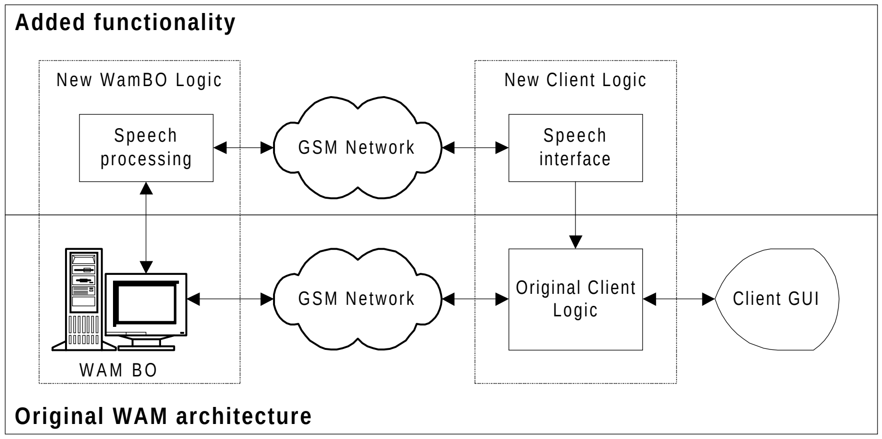


*Figure 4.1-1: Server side recognition approach*

#### 4.1.2.2 Client-side recognition

In this approach speech recognition and speech synthesis are performed at the client. Utterances from the user are processed locally. Reactions are also synthesised locally. As a result, the original communication architecture stays intact. This approach is graphically presented in Figure 4.1-2:

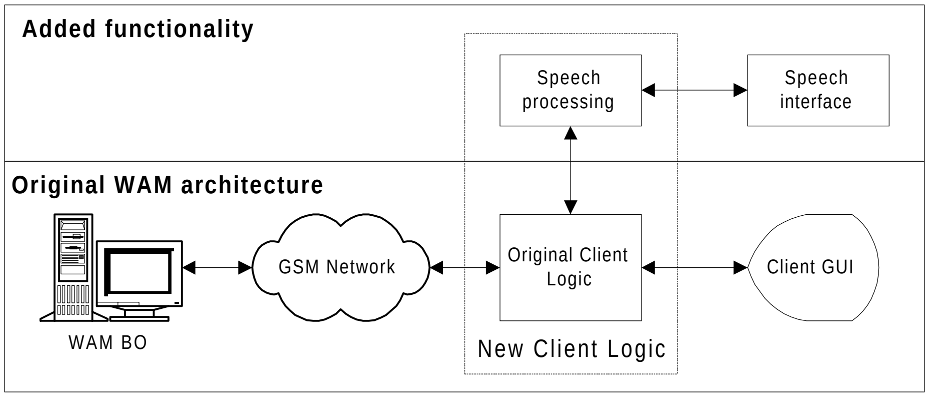


*Figure 4.1-2: Client-side recognition approach*

### 4.1.2.3 Comparison between approaches

Table 10 contains a comparison of the strong and/or weak points between the two approaches (client-side and server-side) presented earlier. The score of each approach is indicated with plusses (meaning strong) or minuses (meaning weak). Based on the results of the comparison it is decided to choose the client-side approach for this project. The comparison clearly indicates that the client-side approach (the approach in which the speech recognition and synthesis components of the system reside at the client) is the better solution at this moment. The potentially better quality of speech recognition and synthesised speech does not weight up against the added complexity and lack of scalability. As the utterances from the user are processed locally and the speech output is also synthesised locally in the client-side approach, the original communication architecture can stay intact, leading to no further added complexity.

*Table 10: Comparison between the Client-side approach and the Server Side approach*

| **Subject** | **Client-side approach** | **Server-side approach** | **Comments** |
|---|---|---|---|
| Complexity | + | - | The server-side approach requires changes in the WAM-BO as well as the WAM-client. Also a new communication channel needs to be developed to transmit voice data. The client-side approach only requires development on the client side, while the existing communication architecture can be used. |
| Licensing and maintenance cost | - | + | In the Client-side approach speech recognition and speech synthesis have to be handled in the client. As a result, the maintenance and licensing costs are very high. In the server-side approach the speech software is only installed in the WAM-BO, greatly reducing maintenance and licensing costs. |

| | | | |
|---|---|---|---|
| Scalability | + | - | The Client-side approach does not require additional resources and processing power from the WAM-BO. Therefore, it is easier to add new users to the system. In fact, the new system is as scalable as in the old situation. On the other hand, if speech processing is done at the server side (with speech recognition and synthesis being processor intensive tasks), the more clients are connected, the more GSM connections and processing power are needed. Degrading scalability of the system. |
| Additional resources | + | - | In the client-side approach, the clients only require additional (software) resources to perform speech recognition and speech synthesis. In the server-side approach, next to the original SMS connection, a separate voice data connection is needed. It cannot be predicted when the user might say something. Therefore this approach requires a constant GSM connection with the Back Office. The Back Office needs the resources to process voice data and the WAM-Client needs resources to send and receive voice data. |
| Integration | + | + | In the client-side approach, voice commands are eventually transformed into the same format as GUI commands. Furthermore a response from the server can be presented graphically (the old way) as well as with speech.<br>In the Server-side approach, the voice commands have a separate communication channel with the WAM-BO, thus speech interface is essentially an independent interface. |

| | | | |
|---|---|---|---|
| Quality | - | + | When recognition is done at the client, the quality of the interface (recognition rate, speech quality) depends greatly on the hardware requirements (CPU power, platform etc.) of the speech software of the client. We can assume that the Back Office is capable of running the most sophisticated speech software, so potentially the server-side approach can supply better speech recognition quality to the system than the Client-side approach. In the server side approach the quality of the voice transmission channel plays the most crucial role. |
| Compatibility | + | + | As the communication architecture stays the same in the client-side approach, the system is still backward compatible. In the server side approach the original architecture is not changed (only a new communication channel has been added), so the system should still be compatible with clients without speech interface. |

### 4.1.3 Elaboration of chosen approach

The speech interface according to the client-side approach consists of three main components. The speech recognition engine, the text to speech engine and the dialogue control. Because each component differs considerably from the others, the components are separately designed and separate design methods will be chosen for each component. Since the chosen approach does not alter the WAM back office nor the communication architecture between the SWAMP client and the back office, these topics are not covered further.

#### 4.1.3.1 Data

The purpose of the speech interface is to support speech interaction between a user and the main application. The term “main application” is used to refer to the logic of the original WAM client, which is described in chapter 2.
First a definition of a typical user of the SWAMP client will be given. This definition is a key consideration in many design decisions. At the start of the project, there was no clear definition of the typical user. Thus, the following (convenient) user has been chosen to represent the typical user:

> The typical user of the system is an adult English speaking male[1]. He is a skilled driver and familiar with current computer and communication technology. Furthermore he travels a lot (privately and/or for the company he works for).

*Figure 4.1-3:Definition of a typical user of the system*

[1] Statistics show that the majority of car drivers is male

Next an explicit state definition of SWAMP client needs to be defined. The state represents what the speech interface knows about the world. Together with external inputs (e.g. user utterances, motor information) this determines the behaviour of the speech interface.
As the world evolves and changes while the SWAMP client is active, this information needs to evolve and be updated too. Since the state is very big and dynamic, artificial intelligence (AI) techniques are used as a tool to represent and maintain this state.
The state of the SWAMP client consists of the following related information structures:

State ***SWAMP*** of

| | |
|---|---|
| ***Driver:*** | ***Driver*** |
| ***AnnoyanceLevel:*** | {NORMAL, ANNOYED} |
| ***FeedbackMode:*** | {normal, silent, verbose} |
| ***AlertLevel:*** | {normal, alert} |
| ***Sessioninfo:*** | ***Driver*** |
| ***DialoguePosition:*** | {***LoginDlg, MainDlg, SOSCallDlg, ANWBCallDlg, KMRegDlg, DirReqDlg, FileInfoDlg, CorpInfoDlg*** } |
| ***Location:*** | ***GPSCoordinates*** |
| ***CurrentAction:*** | Char* |
| ***CurrentGoal:*** | {LOGIN, MAIN, SOS, ANWB, KMREG, DIRREQ, TRAFFICINFO, CORPINF} |
| ***LastSaid:*** | char* |
| ***Retries:*** | int |
| ***Timeout:*** | int |

**End State;**

| | | |
|---|---|---|
| ***Driver***:: | ***Name***: | char* |
| | ***CarID***: | char* |
| | ***ProjectID***: | char* |
| | ***TelefoonNr***: | char* |
| | ***TripType***: | {private, business}; |

***Name:*** char*;
***NameAlias:*** ***Name*** -> UserID;
***Nametable:*** ***NameAlias***-*set*;

***CarID***: char*;
***CarIDTable:*** **CarID**-set;

***ProjectID*** : char*;
***ProjectIDTable:*** **ProjectID**-set;

| | | |
|---|---|---|
| ***LocationName***:: | | char*; |
| ***GPSCoordinates***:: | ***NB***: | char* |
| | ***WL***: | char*; |

***Location*** = ***LocationName*** ***->GPSCoordinates***;
***Locationtable*** = ***GPSCoordinates-set***;

| | | |
|---|---|---|
| **Login::** | **driver:** | **Driver** |
| | **location:** | **GPSCoordinates** |

| | | |
|---|---|---|
| | **loginResult:** | char*; |
| **SOSCall::** | **Location:** | **GPSCoordinates** |
| | **SOSCallResult:** | char*; |
| **ANWBCall::** | **Location:** | **GPSCoordinates** |
| | **ANWBCallResult:** | char*; |
| **KMReg::** | **newTriptype** | {private, business} |
| | **NewProjectID:** | char* |
| | **Kmlevel:** | int |
| | **KMRegResult:** | char*; |
| **DirReq::** | **sourceLocation:** | **GPSCoordinates** |
| | **destinationLocation:** | **GPSCoordinates** |
| | **DirReqResult:** | char*; |
| **TrafficInfo:** | **infoType:** | {local, global} |
| | **InfoParam:** | char* |
| | **Location:** | **GPSCoordinates** |
| | **KMRegResult:** | char*; |
| **CorpInfo:** | **infoType:** | {Telephone, Appointment, stock} |
| | **InfoParam:** | char* |
| | **CorpInfoRes:** | char*; |

#### 4.1.3.2 Components

A system for speech interaction with a user consists of at least three components. The first one is a component to recognise the user's utterance and transform it into a format that can be processed more easily. This component is called the Automatic Speech Recognition or ASR component and it transforms the speech utterance into text. Then a component is needed to process this text to figure out what the user wanted to accomplish with the utterance. This is done in the dialogue manager component. The dialogue manager also takes the appropriate actions as a response to the user's utterance. Actions can be speech responses such as feedback or requests for clarifications, these are sent in text format to the last and final component: the Text To Speech or TTS component were speech is generated from the text to give response to the user. Figure 4.1-4 gives a graphical overview of the components and their interconnection.

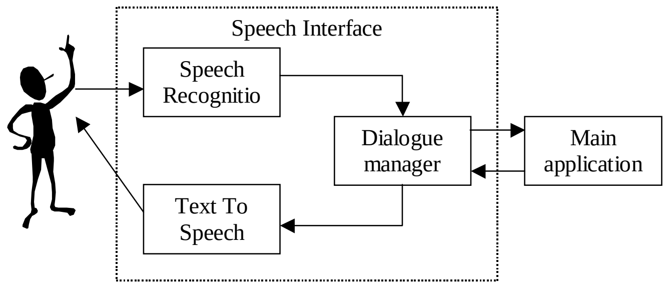


*Figure 4.1-4: Speech interface components*

## 4.2 Implementation

The discussion in this section is limited to a listing of the most important implementation decisions and an elaboration on the implementation strategy applied. Also an overview of the result is presented. For a thorough discussion see the SWAMP implementation document [Yang3 2001].

### 4.2.1 Implementation decisions

In order to contain the complexity of the project some important implementation decisions have been made. They are summarised below.

*Table 11: Implementation decisions*

| Decision | |
|---|---|
| Use Windows NT to build a prototype | Initially it was the intention to build the speech interface to run on Windows CE (see requirements). Due to limitations in software[1] and hardware of the HP Jornada, a prototype will be developed to run under Windows NT first. This step is taken as a safety precaution so that there is something to demonstrate at the end of the project. |
| Use SAPI as middle-ware to implement speech resources | The Microsoft Speech Application Programming Interface version 5.0 (SAPI5) [SAPI] consists of two interfaces: the application-programming interface (API) and the device driver interface (DDI). The SAPI 5.0 API dramatically reduces the code overhead required for an application to access speech recognition and synthesis. Furthermore, the application programming interface delivers access to the speech resources in an independent way, consequently the TTS or ASR component can be replaced by other (better) components without having to change a single line of the SWAMP client's code. The specific reasons for choosing SAPI5 are discussed in section 5.1. |
| Use the CLIPS expert system tool | The CLIPS [CLIPS 2000] expert system tool is designed to facilitate the development of software to model human knowledge or expertise. It has been designed for full integration with other languages such as C. In the SWAMP client, CLIPS will be used to help manage the dialogue with the user. In particular, CLIPS will be applied to do the knowledge processing part of the dialogue manager. |
| Use C++ as programming language | The WAM client was written in C++ and both SAPI and CLIPS support it. So it is only natural that the SWAMP client, as an extension of the WAM client, is only implemented in C++. The development environment is Microsoft Visual C++ enterprise edition. |

[1] Microsoft is very vague on COM and ActiveX support on WindowsCE 2.11 which is required for SAPI5 to work

| | |
|---|---|
| Rapid prototyping | Whenever human factor issues are involved (which is definitely the case in SWAMP) it adds an extra dimension to the problem which is hard to capture in static design models. It can already be anticipated that unforeseen situations will occur as the result of unexpected utterances of the user. To anticipate these unexpected situations a strategy is chosen to develop an early prototype and refine it through extensive iteration. |
| Leave client the same as much as possible | The implementation strategy on the SWAMP client is to leave the old structure and code of the WAM client intact as much as possible, so that the old documentation is still valid. |

### 4.2.2 Overview

Figure 4.2-1 gives a graphical overview of how the speech interface is implemented. All components will be discussed in more detail in chapter 5.

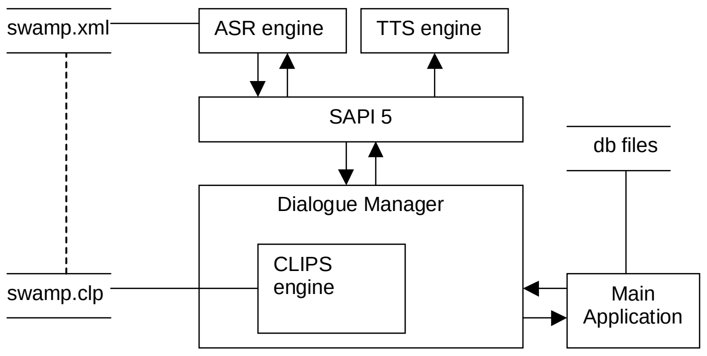


*Figure 4.2-1 Overview of SWAMP implementation*

The ASR engine recognises utterances from the user while the TTS engine synthesises speech to the user. SAPI 5 acts as a communication layer between the dialogue manager and the speech resources (ASR and TTS engine). The CLIPS engine is embedded in the dialogue manager. It can be viewed as the knowledge processing and management unit of the dialogue manager. The main application is the original WAM client modified in such a way that it can communicate with the dialogue manager.

Data can be written to and retrieved from external files:

| | |
|---|---|
| swamp.xml | This file contains a definition of grammar rules of SWAMP. The ASR engine loads this file to know which words or sentences to recognise. |
| swamp.clp | This file contains production rules (constructs). The CLIPS engine uses these constructs to handle dialogue with the user. Swamp.clp contains references to rules in the swamp.xml explaining the dotted line between the files. |
| db files | Represents a set of database files:<br>- name list<br>- project ID list<br>- car ID list<br>- location name to GPS co-ordinates mapping list |

### 4.2.3 Implementation strategy

The implementation strategy applied in the SWAMP application differs from the way traditional systems are implemented because human factor issues are involved. The strategy chosen is to develop an early prototype and refine it through extensive iterations. First a proper speech support framework is built. Then the services are added one by one. In the end, the system as a whole is tested again. Figure 4.2-2 gives an overview of the implementation strategy.

```
iteration point 1
    Implement speech framework
iterate1

iteration point 2
    For each Service do
        iteration point 3
            Dialogue analysis
            Construct scenarios
            Construct a corpus
            Construct a grammar
            Build facilitation in CLIPS
            Implement interaction between GUI and Dialogue manager
            Test scenarios
        iterate3
    end for
    Integral system test
iterate2
```

*Figure 4.2-2: Overview of the implementation strategy*

*Table 12: Implementation steps*

| Implementation step | Description |
|---|---|
| Implement speech framework | This step includes initialisation of external resources (SAPI and CLIPS), implementation of the messaging facility between SUI components and between the components of the dialogue manager. In short, it realises all the necessary conditions to support speech interfacing. |
| Dialogue analysis | Since there are no existing spoken dialogues available, additional information must be extracted from the way the GUI is implemented. In particular the navigation structure in the GUI and the information needed to complete the service is of importance. Flow diagrams depicting the possible paths through a dialogue are used to model the dialogue flow. |
| Construct scenarios | Compile a list of possible scenarios (dialogues) for the service. The scenarios are used to construct a corpus (next step) and to test the SUI in the testing phase. |
| Construct a corpus | Based on the scenarios and dialogue flow, a corpus of about 15 (common) utterances for the service is constructed. |

| | |
|---|---|
| Construct a grammar | The goal in this step is to capture the corpus for the service into some grammar rules. The obtained grammar rules are stored in the "swamp.xml" file. |
| Build facilitation in CLIPS | Construct CLIPS rules to implement the desired behaviour for the service. This includes:<br>Rules that implement the actions to take if a grammar rule is recognised (interfacing rules).<br>Rules that implement actions to lead the user to a successful accomplishment of the service (hidden rules).<br>The CLIPS rules are stored in "swamp.clp". |
| Implement interaction between GUI and Dialogue manager | In this step the implementation of interaction between GUI and Dialogue manager to synchronise speech and GUI actions is implemented. The synchronisation is needed so that:<br>1. The user has some visual indication of what he is doing.<br>2. The content of the GUI is consistent with what is going on. |
| Test scenarios | Test if the dialogue flow goes as planned. Scenarios can be tested using speech emulation as well as real speech. With speech emulation the speech recogniser is disabled and utterances of the user are entered through a keyboard. Emulation filters out unwanted effects, such as background noise, so that only the pure dialogue is tested. |
| Integral system test | Test the system as a whole and let others try it. The purpose is to test whether the speech interface is usable and indeed an improvement on the graphical user interface. |

### 4.2.4 Implementation results

Due to the limited time available for this project, not all implementation goals have been achieved. Nevertheless a working prototype of the described application has been successfully implemented.

The architecture described in section 4.2.2 has been implemented entirely. SAPI5 enables the SWAMP client to 1) recognise spoken speech according to a user defined grammar file and 2) synthesise arbitrary text into speech. Furthermore the voice, speed and pitch of the synthesised speech can be adjusted. The CLIPS engine has also been successfully embedded. It is possible to load CLIPS construct files, send messages to standard input of the CLIPS engine, and receive messages from standard output and standard error. To have a better overview, the client code is split into five functional categories (Table 13). These categories differ in the functions they are to fulfil and the changes necessary in the original WAM client code to achieve this. Since the SWAMP client is an extension of the WAM client, a big part of the WAM clients source code is also used in the SWAMP client. Appendix A5 shows a comparison between the class hierarchy diagrams of the SWAMP client and the WAM client (new classes, modified classes, name changes etc.).

*Table 13: Overview of the functional categories of the SWAMP client's code*

| Category | Description |
|---|---|
| GUI code | This code contains the implementation of the graphical user interface. The only changes made here, are due to the new services and the communication with the dialogue manager. |
| Communication code | Provides the home for communication with the SWAMP back office, GPS antenna and the Motor management system. This code is entirely the same as in the WAM pilot. |
| Recognition code | This code is responsible for the initialisation, control, and destruction of the speech resources. All the code of this category is new. |
| Grammar code | The grammar code defines the grammar of the SWAMP client's speech interface. The grammar code is written in the Microsoft grammar schema format and resides in a file named "grammar.xml". |
| Dialogue management code | The dialogue management code is divided into two parts. The first part is written in CLIPS and resides in a file named "swamp.clp" the second part is written in C++ and controls the communication between the first part and the rest of the application. |

The SWAMP architecture is the skeleton on which speech-enabled services can be implemented. A speech-enabled service consists of a grammar defining the valid utterances, a dialogue flow design specifying the possible dialogue paths, and an implementation of the designed dialogue flows. The grammar of the following services have been designed and implemented: Login (except back office telephone number), SOS call, ANWB call, KM registration (see appendix A3), request direction, and traffic information. The dialogue flow for the login (except back office telephone number), SOS call, ANWB call, KM registration, and request direction services have been designed, implemented and tested. The dialogue flow for the traffic information service has been designed, but not implemented. Appendix A7 shows the dialogue flow diagrams for the designed dialogues. Table 14 summarises the current development status of the speech-enabled services.

*Table 14: Current development status of the services*

| Service | Grammar | Dialogue designed | Dialogue implemented and tested |
|---|---|---|---|
| Login[1] | yes | yes | yes |
| SOS call | yes | yes | yes |
| ANWB call | yes | yes | yes |
| KM registration | yes | yes | yes |
| Request direction | yes | yes | yes |
| Traffic information | yes | yes | no |
| Request corporate information | no | no | no |

[1] Except back office telephone number

# 5 The speech interface

The purpose of the speech interface is to support speech interaction between a user and the main application. The general assumption behind the speech interface is that the user wants to accomplish something with his utterances, in other words he has a certain goal in mind. The set of all services the SWAMP application has to offer however, is just a subset of all the goals the user can have. Goals that don't correspond to a service are beyond the domain of the speech interface and are ignored. In other words, the speech interface is only applicable in a limited domain and will not be able to replace a human conversational partner. Generally, a dialogue is started by the user with an utterance in which he indicates what he wants to achieve: the initial utterance. With each initial utterance, the speech interface tries to find the corresponding service involved. It then tries to accomplish the service by checking whether all the necessary information is available. If this is not the case a dialogue is started to obtain the missing information from the user until the task can be performed.

The speech interface is divided into a number of different pieces of software called components, each of which will be discussed separately:

1 The speech recognition or ASR component:
  Its function is to recognise the user's utterance and transform it into a format that can be processed.
2 The dialogue management component:
  Its function is to process the input from the speech recognition component to figure out what the user wanted to accomplish and take the appropriate actions to realise the user's wishes.
3 The speech synthesis or TTS component:
  Its function is to generate speech output to the user.

The following sections give a discussion of the speech components and how they work together to accomplish the desired behaviour. But first a description of the speech software used is given.

## 5.1 Speech software

Early in the project it was clear that it is impossible to build the TTS component and the ASR component within the time available for the project. To simplify matters, it seemed best to use TTS and ASR engine software from third party vendors. Today, there are various ASR, TTS engines and development tools available to develop speech-enabled applications. Several of these engines and tools have been evaluated. A list a the evaluated software is summed up in appendix A2 The evaluation criteria for the software of choice were:

*Table 15: Evaluation criteria for the speech software*

| Criterion | Comment |
|---|---|
| Ease of use | It should not be overly complicated to use the software. |
| Hard/software requirements | This criterion became less important after the decision was made to build a prototype under NT. |
| Recognition rate | This criterion is only applicable to the ASR engine. The evaluation of the recognition rate depends strongly on the type of recogniser used (user dependent or user independent), the evaluation environment (background noise etc.) and the voice of the user (the performance of recognition engines varies from person to person). |

| | |
|---|---|
| Quality of synthesised speech | This criterion is only applicable to the TTS engine. |
| Price/support | Since the SWAMP application is only a demo prototype, the price of the speech components should be proportional. |
| Ability to recognise specified grammar | For better performance an ASR that accepts a user defined grammar rather then a dictation vocabulary is required. The grammar notation format and its possibilities are also important. |
| Features | Features are properties or capabilities of the software that are interesting but not mandatory for the SWAMP SUI. |

In the end, the Microsoft Speech Application Programming Interface 5.0 (SAPI5) [SAPI 2000] was chosen.

### 5.1.1 Overview of SAPI5

SAPI5 is not an ASR or a TTS engine, but acts as middle-ware between the engines and the application Figure 5.1-1. SAPI 5 consists of two interfaces: the application-programming interface (API) and the device driver interface (DDI). Applications communicate with SAPI5 via the API layer and speech engines communicate with SAPI5 via the DDI layer. The DDI takes care of hardware specific issues such as audio device management, while the API removes the implementation details such as multi-threading. This reduces the amount of code overhead required for an application to use speech recognition and synthesis.

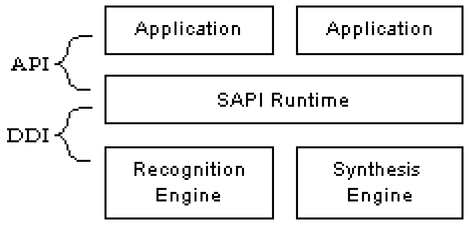


*Figure 5.1-1: SAPI 5 architecture overview*

The advantage of using middle-ware is that the choice of the final ASR and TTS engine can be postponed until a later stadium (e.g. until there is more budget for better engines), provided of course that there are better engines available. In addition, SAPI5 can be downloaded for free and an ASR engine and some TTS voices (Mary, Sam and Mike) are already contained in the package. As for the support of SAPI5, there are two dedicated SAPI newsgroups (microsoft.public.speech_tech and microsoft.public.speech_tech.sdk) available where Microsoft professionals (including the people who built SAPI5) regularly answer questions. Also an extensive manual with several examples and a tutorials is included in the SAPI5 download package. Availability of (better) speech engines from third parties is also good, since many companies have announced support for SAPI5 including Lernout & Hauspie/Dragon Systems Inc., Conversational Computing Corp., Fonix Corp., Fujitsu Ltd., NEC, Toshiba Corp., IBM Corp. [SAPI5 third party]. SAPI5 compatible speech engines from some of these third parties are already available for sale.

The compliance tests from Microsoft verify that engine developers have successfully implemented the required features to be considered compatible with SAPI5, compatible engines can be used by the SWAMP client without trouble. Moreover, as the application programming interface gives access to the speech resources in an engine independent way, the TTS or ASR components can be replaced without having to change a single line of SWAMP application code.

### 5.1.2 SAPI5 usage

Before SAPI5 can be used, it needs to be installed first (see appendix A4 for more information on system requirements and installation notes). The next step is to initialise the speech resources. The initialisation of the engines incorporates the following steps:

| | |
|---|---|
| Step1 | Initialise COM (Common Object Model). |
| Step2 | Create the recogniser object, this provides access to the recognition engine. |
| Step3 | Create recognition context for the engine. A context is a single area of the application needing to process speech (in this case, the entire application). |
| Step4 | Loading grammars and rules. In *Step3*, the grammar was created. This step populates the grammar with rules from an external resource. After this step the initialisation of the recognition engine is complete. |
| Step5 | Create and attach a TTS engine to the recognition context. By attaching the TTS engine to the recognition context, the ability for barge in is provided. |

After the initialisation phase, the application is speech-enabled. The speech processing is done in the background. Whenever there is relevant information from the TTS engine or ASR engine, SAPI collects this information, and returns it back to the application by means of events. Although numerous methods are available to control the execution of the speech resources, not all methods are used in the speech interface. The specific methods used will be discussed when the components using these methods are analysed.

## 5.2 The ASR component

The core task of automatic speech recognition (ASR) is to take a digitised speech signal as input and convert that into recognised words and phrases. To successfully recognise incoming speech the recogniser matches this speech against the grammar. The grammar defines the words and the order of those words that make a valid sentence. The recognition domain is limited to a grammar of valid sentences because this provides better accuracy and performance, and reduces the processing overhead required by the application. The limited grammar also enables speaker-independent processing. The following sections describe the ASR component along with the grammar used by the ASR component, how it is obtained, and how it is used.

### 5.2.1 ASR overview

SAPI 5.0 is based on the common object model (COM), a technique that enables the development of reusable binary software components. Therefore a lot of COM terminology will be used in this overview of the ASR engine. COM is explained in many books e.g. Essential COM [Box 1999]. Figure 5.2-1 shows an overview of the information flow from the utterance of the user until the processing of the recognised words. The figure also shows the role of the ASR engine in this flow.

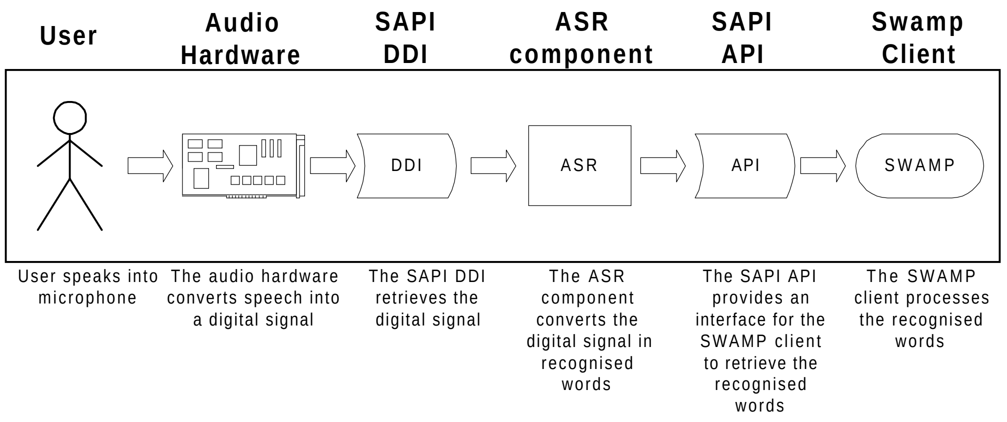

User
Audio Hardware
SAPI DDI
ASR component
SAPI API
Swamp Client
DDI
ASR
API
SWAMP
User speaks into microphone
The audio hardware converts speech into a digital signal
The SAPI DDI retrieves the digital signal
The ASR component converts the digital signal in recognised words
The SAPI API provides an interface for the SWAMP client to retrieve the recognised words
The SWAMP client processes the recognised words

*Figure 5.2-1 : Overview of the information flow from the utterance of the user to the processing of the recognised words*

The SAPI DDI manages the audio hardware and provides a unified interface[1] for the ASR engine to retrieve the digitised speech signals. How different ASR engines implement this interface is left to the ASR engine vendor and is out of the scope of this document. After retrieval of the digitised speech signals, the ASR engine performs recognition algorithms on the signal to extract the spoken words. The algorithms applied vary from vendor to vendor. In ”Artificial intelligence, A modern approach” [Russell 1995] and in the literature survey done prior to this project [Yang1 2001] the most commonly used algorithms are discussed. To share the extracted information with the SAPI5, the ASR engine (which is a COM object itself) must implement the ISpSREngine interface (Figure 5.2-2). The SAPI5 API uses the methods of this interface to obtain the recognised information or to pass details of recognition grammars and tell the engine to start and stop recognition etc. SAPI5 itself implements the interface ISpSREngineSite. A pointer to this is passed to the engine and the engine calls SAPI using this interface to read audio, return recognition results etc. ISpRecoContext is the main interface for speech recognition, it is the speech interface's vehicle for receiving notifications for the requested speech recognition events. Each ISpRecoContext object can take interest in different speech recognition engines and utilise different recognition grammars. Speech applications must have at least one ISpRecoContext instance to receive recognitions. Within an ISpRecoContext an application has the choice of two different types of speech recognition engines (SpRecognizer object). A shared recogniser that could be shared with other speech recognition applications or an in-process (InProc) speech recognition engine for application where speed is key: The SWAMP client uses an in-process recogniser. The SpRecognizer object represents a single SR engine and enables the application to control aspects of the speech recognition (SR) engine.

[1] The term "interface" must be interpreted, in the spirit of COM, as a means to 1) separate definition of the functionality of a COM object from the implementation details of that object or 2) gain access to functionality of the COM object.

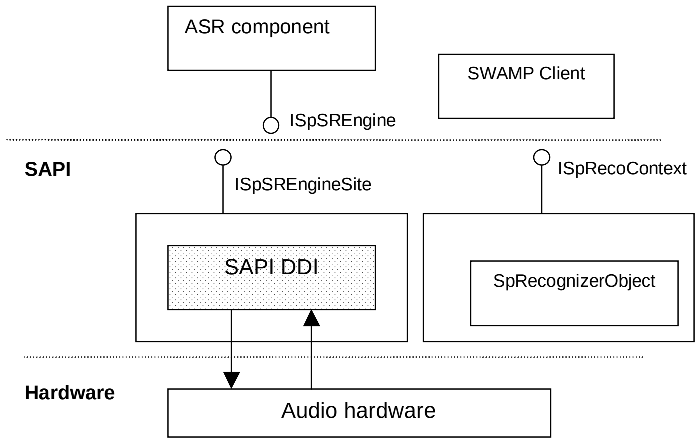


*Figure 5.2-2: Interfaces of SAPI5*

### 5.2.2 The SWAMP Grammar

The SAPI5 design specification requires that all SAPI5-compatible speech engines must be able to understand a context-free grammar (CFG) written in a format specified in the SAPI5 grammar schema. This schema describes the SAPI 5.0 speech recognition grammar format and is based on the XML framework. Since only SAPI5 compliant ASR engines can be used in the SWAMP client, the grammar for the dialogues is also defined according to the schema.
The ASR engine uses the CFG to constrain the words contained in the user's utterance that it will recognise. Furthermore the CFG can be extended with semantic information (property names and property values) declared inside the grammar. This enables the ASR engine to associate certain recognised word string with name/value-meaning representations. The dialogue manager then applies these meaning representation associations to understand and control the dialogue with the user.

#### 5.2.2.1 Syntax

The SWAMP grammar is stored in a grammar file (swamp.xml). Basically the grammar file consists of a set of grammar rules in the grammar schema syntax (see Figure 5.2-3). The complete specification of the schema can be found in the SAPI5 online help file [SAPI5 website].

```
<RULE
    [ DYNAMIC = enumeration : 0 | NO | FALSE | 1 | YES | TRUE ]
    [ ID = string ]
    [ NAME = string ]
    [ TOPLEVEL = enumeration: INACTIVE | ACTIVE ]
>
    [<RULEREF>| <P> | <L> | <O>]*
</RULE>
```

*Figure 5.2-3: Syntax of a grammar rule*

| Attribute | Description |
|---|---|
| DYNAMIC | Indicates that the contents of the rule can change during runtime. |
| ID | This is a unique identifier of the rule. |
| TOPLEVEL | The TOPLEVEL tag within the RULE statement gives a rule a special status. Not only does this identify the rule as being top level, but it also sets the activation state (active or inactive). Only top-level rules can be activated or deactivated. SAPI recognises active rules and conversely does not recognise deactivated ones. The application may change the state of the rules during execution. so if a rule is no longer needed, it may be deactivated |

| Element | Description |
|---|---|
| RULEREF | This element is used inside the contents of a rule definition to reference another defined rule (but not other top-level ones). |
| P* | This element is used to describe the Phrase element. An associated property name and value pair is generated if the contents of this element are recognised. |
| L | Defines a list of alternate phrases from which any one can be used to complete the match. Thus each sub-element within this element represents a possible separate recognition in place of this element. |
| O* | This element is similar to the P element. With the exception that the O element is optional. As the name implies, optional words are not required for a successful rule match. |

Elements labelled with * can contain properties or semantic information within tags (property names and property values). After a successful recognition, the information can be retrieved from SAPI. References to other rules can be *recursive* – i.e., rules can reference themselves, either directly or indirectly.

### 5.2.2.2 Construction of grammar rules

The grammar rules are derived from a corpus of about fifteen utterances per service. Fifteen utterances are too few to build a reliable and robust corpus for a service, but should be enough to allow the demonstration of the general idea of the speech interface in the SWAMP client prototype without making the grammar too complex. Example 1 shows an example of the derivation of the grammar rules for the KM registration service out of a corpus of utterances. The resulting grammar rule is given in pseudo CFG format.

*Example 1: Example of the derivation of grammar rules from a corpus*

Sample utterances (corpus):

| Utterance |
|---|
| It’s private now |
| Set the trip type to/into private |
| Change the trip type to/into private |
| It’s a business trip now |
| Project ID is X |
| It’s a business trip and project ID is X |

| |
|---|
| Bill all kilometres driven from now on project X |
| This part is of the trip is private |
| The drive is now private |
| Did I mention that this was a private trip? |
| It will be a private trip from here on |
| The trip type is Private |
| Trip type is private |

The transformation from utterances to a context free grammar is not difficult. In the worst case the resulting CFG is an enumeration of all the utterances. For performance reasons it is better to try to discover general patterns in the utterances and thus to make the CFG as small as possible resulting in the following pseudo CFG:

| | |
|---|---|
| <Triptype> = | "Private" \| "business" |
| <TriptypeUtterance> = | "Trip" \| "triptype" \| "drive" |
| <Now> = | "now" \| "from now on" \| "from here on" |
| <Project-ID> = | // obtained from database |
| <KM-Triptype> = | ["change" \| "set the" ] <*TriptypeUtterance*> "in" \| "into" \| "to" <*Triptype*> \| ["the" \| "this part of the" ] <*TriptypeUtterance*> "is" <*Triptype*> \| "it's" \| "Did I mention that this is" \| "It will be" \| "I will be making" "a" <*triptype*> <*TriptypeUtterance*> [<*Now*>] |
| <KM-ProjectID> = | "Bill all kilometres driven *now* on project" |
| < KM-Project-ID> = | ["The"] "project I D" [ "is" ] <*Project-ID*> |
| <KM-utterance> = | <KM-TripType> \| <KM-ProjectID> \| <KM-TripType> "and" <KM-ProjectID> |

The pseudo CFG must then be transformed into Microsoft schema language (which is a real CFG) to be of use for the ASR engine. This process is rather straightforward and will not be discussed. Appendix A3 shows the resulting grammar rules in Microsoft schema language.

The grammar rules thus obtained must be stored in a file (in this case: swamp.xml). SAPI5 comes with a grammar compiler that creates binary grammars from XML defined grammars. The SAPI grammar compiler is divided into two parts, a front-end section and a back-end section. The front-end parses the grammar described in XML and optimises the XML formatted text grammar if requested. The front end then calls the back-end compiler to convert the internal representation into the SAPI5 binary format. The binary file (grammar.cfg) is stored as an application resource. During compilation, the grammar compiler gives error messages if the grammar format does not conform to the Microsoft schema for grammar format. This enables the verification of the validity of the grammar. Although the process of deriving grammar rules from a corpus is not very difficult (especially with only fifteen utterances), there are some issues to consider:

**Dynamic content in grammar**
Suppose we want a rule with RULE ID EMAIL that needs to support the phrase "send new e-mail to NAME." The problem is that the phrase "send new e-mail to" is static, and known at design time, but NAME is undetermined and only available during runtime. We could solve this problem by explicitly listing all names in a static grammar rule. Static means this list is defined ahead of execution time and is not subject to change (Although the .xml file may be edited independently of the application, it may not be changed during execution). This is not an elegant solution because the grammar file has to be changed and recompiled every time the user adds a new name to his e-mail list. Luckily SAPI5 also supports dynamic grammars. As the name implies, a dynamic grammar is the opposite of a static one. First, words may be added and deleted during runtime and the list does not need to be predetermined. This allows much greater latitude for applications. To use dynamic grammars, dynamic rule content should be separated from static rule content. Another motivation for the separation is that it makes grammar design more clear and improves initial SAPI grammar compiler performance. In the SWAMP grammar, the email problem could be solved by creating a static rule (with rule ID E-MAIL) that contains the static phrase and a dynamic rule (with RULE ID NAMEDB) that contains dummy values. The list of names, e.g. coming from an address book, can be loaded into the dynamic rule at runtime. The static grammar could then contain a rule reference (RULEREF) to the dynamic rule. When the SWAMP client starts up, it quickly loads the static content. Afterwards it can load the dynamic content when needed (resulting in better performance). Moreover, if the list of names (address book) is updated the run-time grammar of the speech interface is automatically updated along with it.

**Semantics in grammar**
The SWAMP grammar is written in a format specified in the SAPI5 grammar schema, which is a context free grammar (CFG). As CFGs only specify the rules for a valid utterance, another mechanism must be found to convey meaning to the dialogue manager. The mechanism used in the SWAMP client to accomplish this is called semantic tagging. Property name and property value tags are used to tag elements in order to artificially endow some meaning in them.
In the example of the previous paragraph, a semantic property tag e.g. NAME can be attached to the rule reference to the dynamic NAMEDB rule. The dynamic information corresponding to the NAME tag (in other words the name actually spoken out) can be retrieved from the recognised phrase at runtime.

### 5.2.3 Grammar handling

So far grammar rules were discussed as abstract data types. To define a grammar rule properly, the operations that can be performed on them need to be considered. SAPI 5 allows some flexibility in the defined grammar, for example *top-level* rules can be activated or de-activated during run-time. In the following a description is given of what the speech interface must do for each of the six basic grammar rule operations: creation/deletion, load, activation/deactivation and modification.

**Creation and Deletion**
When an application creates a grammar object this is reported to the engine via the *OnCreateGrammar()* method. From this method the engine must also return a pointer, which is used to identify the grammar in later calls from SAPI5. When grammars are deleted the *OnDeleteGrammar()* method is called.

**Loading**
SAPI5 takes full control of loading a grammar when an application asks it to. SAPI5 can load from a file (*LoadCmdFromFile()*), a COM object (*LoadCmdFromObject()*), a resource *LoadCmdFromResource()*), or from memory (*LoadCmdFromMemory()*), and can load either binary or XML forms of the grammar. SAPI5 then notifies the ASR engine about the contents of the grammar through various DDI methods.

**Activation/deactivation of rules**
Rules can be *top-level*, indicating that they can be activated or deactivated during run-time with the *SetRuleState()* function. For instance, only the login grammar rules are active during user log in. If the login procedure is completed successfully, the login rules are deactivated while other rules now become active. This way, only rules that are relevant in the current context will be recognised. Moreover, no interference is received from irrelevant rules firing and the performance and recognition rate of the ASR engine is increased because of a small search space.

**Modification**
Grammar rules with the DYNAMIC attribute set (see Figure 5.2-3), can be modified during runtime with the *ClearRule()* command. The contents of a rule can be cleared first whereupon the *AddWordTransition()* command can be used to add a word in the rule contents. This feature is used to load frequently changing data into the grammar. For example, the list of user names changes frequently, it is not flexible to explicitly enumerate the list in the grammar file. Especially, since the whole application needs to be re-compiled if the list is modified. Instead the list is loaded from a database file into the appropriate grammar rule during runtime.

## 5.3 The Dialogue Manager

The dialogue manager is responsible for a successful dialogue with the user. It decides which utterances the ASR engine must recognise, what the TTS engine must say, and how it must be said. Therefore, it is the part of the program that determines the “face” of the speech interface. Following is a list of tasks the dialogue manager is responsible for in order to accomplish this.

*Table 16: Functions of the dialogue manager*

| | Task | Description |
|---|---|---|
| 1 | Controlling the ASR engine | The dialogue manager controls the ASR engine by indicating which grammar rules should be activated or deactivated at any time. Furthermore the control task includes starting and stopping of speech recognition and loading of dynamic data into the grammar. |
| 2 | Controlling the TTS engine | The dialogue manager generates text messages and sends them to the TTS engine for speech synthesis. Also the dialogue manager is responsible for starting or stopping speech synthesis and controlling speed, volume and pitch of the synthesised speech. |
| 3 | Maintaining a model of the real world | Since dialogues are context sensitive, the dialogue manager must maintain an internal model of the real world to be able to correctly interpret the user’s utterances. This model is temporal and needs to be constructed and maintained at run-time. |

| | | |
|---|---|---|
| 4 | Processing and interpreting recognised speech | As the ASR engine recognises user utterances, it is the dialogue manager's task to interpret and extract the user's goals from these utterances (taking into account its model of the real world). Furthermore the dialogue manager must perform the necessary actions to achieve these goals. |
| 5 | Communicating with main application | This communication is necessary to keep the main application updated on relevant changes due to speech activity of the user or to be updated about relevant changes due to GUI activity of the user. |

The process of extracting the users' intentions from recognised speech (function 4) requires some kind of reasoning. Particularly if the utterances are context sensitive and the dialogue manager's internal model of the real world must be taken into account. The use of artificial intelligence to build and maintain the internal models (function 3) is inevitable. Not surprisingly, artificial intelligence (AI) technology and techniques have indeed been applied to achieve this reasoning. In chapter 6 the dialogue manager and AI techniques used will be discussed in more detail.

## 5.4 The TTS component

The text to Speech (TTS) component (or speech synthesis component) takes a sequence of text words (input text) and produces as output an acoustic waveform (see Figure 5.4-1).

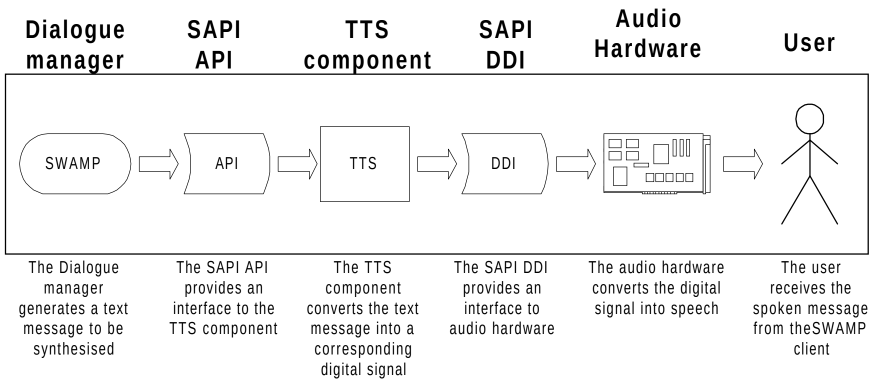

Dialogue manager
SAPI API
TTS component
SAPI DDI
Audio Hardware
User
SWAMP
API
TTS
DDI
The Dialogue manager generates a text message to be synthesised
The SAPI API provides an interface to the TTS component
The TTS component converts the text message into a corresponding digital signal
The SAPI DDI provides an interface to audio hardware
The audio hardware converts the digital signal into speech
The user receives the spoken message from theSWAMP client

*Figure 5.4-1: Overview of the information flow from generation of text messages till generation of spoken message*

When the following piece of C++ code is executed, the sentence "Get back to work" is synthesised with normal pitch, maximal volume and the emphasis on the word "get".

```
Speak(L"<SAPI> <PITCH MIDDLE='5'/> <VOLUME LEVEL='100'/> <EMPH> Get
</EMPH> back to work </SAPI>");
```

Several TTS engines can be installed on the same machine, but only the initialised TTS engine is current. In fact, the "speak" command is a generic SAPI API call to the current TTS engine (which of course is initialised beforehand). The list of available TTS engines is listed in the Windows NT registry in: **HKEY_LOCAL_MACHINE\Software\Microsoft\Speech\Voices**.

The TTS engine used in SWAMP has support for “barge-in”, which allows users to speak at any point in the system, even while the system is still playing prompts, greatly increasing the speed and efficiency of the system. Furthermore the SAPI5 API supports the use of simple speech control symbols (see Table 17) incorporated in the input text.

*Table 17: Speech control elements defined in the SAPI TTS XML schema*

| Element | |
|---|---|
| <CONTEXT> | The context can specify the type of normalisation rules which should be applied to the scoped text. SAPI does not guarantee any predefined contexts. |
| <EMPH> | Places emphasis on the words contained by this element. |
| <LANG> | Changes the LANGID of the scoped text. When the LANGID is changed, SAPI will try to detect if the current voice can handle the new language. If voice does not speak the specified language, then an engine must choose another language it speaks as a best attempt. |
| <PARTOFSP> | The part of speech of contained word(s). The PARTOFSP tag is used to force a particular pronunciation of a word (for example, the word record as a noun versus the word record as a verb). |
| <PITCH> | The scoped/global element PITCH modifies the underlying numerical values of a speech block. Relative attribute values, those preceded by a dash (-) or a plus sign (+), increment the underlying numerical value by the specified amount. SAPI compliant engines have the option of supporting only the guaranteed range of values and behaving as -10 for adjustments below -10 and behaving as +10 for values above +10. |
| <PRON> | Pronounces the contained text (possibly empty) according to the provided Unicode string. |
| <RATE> | Sets the relative speed adjustment at which words are synthesised. |
| <SILENCE> | Produces silence for a specified number of milliseconds to the output audio stream. |
| <SPELL> | Spells out words letter by letter contained by this element. |
| <VOICE> | Sets which voice implementation is used for synthesis of associated input stream text. The best voice implementation given the required and optional attributes will be selected by SAPI. |
| <VOLUME> | The scoped/global elements VOLUME modify the underlying numerical values of a speech block. The underlying value can never be below zero or exceed 100. All negative value entries will result in zero and all values above 100 will result in 100. VOLUME may also receive an absolute value (no '-' or '+' character) of an integer between zero and 100. |

There are two main objects of interest in the TTS Engine: the SpVoice object (SAPI) and the TTS Engine object (see Figure 5.4-2). The SpVoice object implements two interfaces that are of concern – ISpVoice, which is the interface which the application uses to access TTS functionality, and ISpTTSEngineSite, which the engine uses to write audio data and queue events. The TTS Engine must implement two interfaces as well – ISpTTSEngine, which is the interface through which SAPI will call the engine, and ISpObjectWithToken, which is the interface through which SAPI will create and initialise the engine.

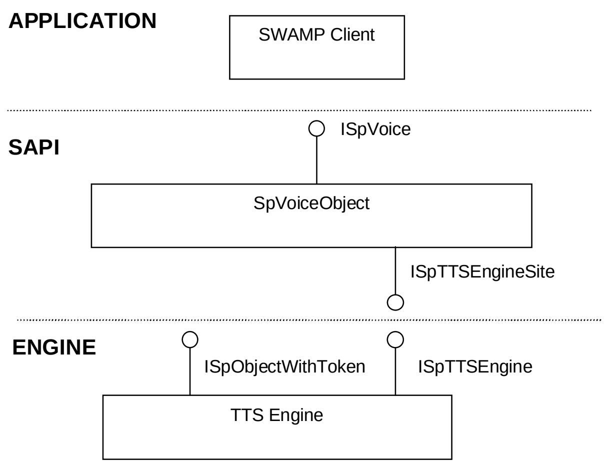


*Figure 5.4-2: SAPI5 TTS relevant objects and interfaces*

# 6 The dialogue manager

In this chapter the dialogue manager will be discussed in more detail. In particular the theoretical background of the techniques behind the dialogue manager, the way the dialogue manager represents knowledge and the way it uses this knowledge to generate the actions to be taken.

## 6.1 Dialogue design

Once the goals of the user are clear, the speech interface must initiate a dialogue to retrieve the required information from the user (if these are not already available). All possible dialogues that the speech interface can initiate must be designed beforehand. This includes speech prompts for each possible situation, and all possible user responses on those prompts. Furthermore design involves the definition of a grammar that captures the syntax of whole conversations into a few simple grammar rules (discussed in 5.2.2.2). Important issues in the design of the dialogues are discussed next.

### 6.1.1 Design approach

The goal of the speech interface is to give a user access to the SWAMP services by means of simple speech interaction. To achieve this, one can choose between two different approaches:
- Demand a longer learning time for the speech interface and require the user to adapt his speaking style.
- Make it easy for the user by allowing an extensive grammar and modelling more and more complex dialogues so that the user can speak to the system as with another human.

Speech User Interface (SUI) designers have learned that humans are extraordinarily flexible in their speech and readily adapt to the speaking style of their conversational partners [Design]. This is not a new finding: think about how easily we adjust our speech depending on whether we are speaking to children or other adults. This flexibility has useful implications for designing the speech interface: after extensive use of the speech interface (as the user gets acquainted with the grammar and has more experience) some dialogues become less and less common. This is because the user will adapt his style of interacting with the speech interface and refrain to only those dialogues that were successful in the past. Because of this finding and the choice of our typical user ("he is familiar with current computer technology") the first approach was chosen: only model the most common utterances and let the user adapt to it. After all, it is not our goal to mimic a real human in human-to-human conversation; this is impossible with the current state of technology anyway.

**What information is needed?**
Each service requires some input parameters before it can be executed. Unlike GUI input, where the parameters are always well defined, speech input can be vague and/or ambiguous. Therefore it is important to impose stringent constraints on speech input to eliminate or detect possible errors e.g. what input is valid, when is the input valid and when not, how and in what sequence must the information be supplied (if important).

**How can the information be obtained from the user?**
The obvious way to obtain information of the user is by ways of speech prompts. Several prompting techniques are commonly applied in speech user interfaces:

*Table 18 : Prompting techniques*

| Technique | Description |
|---|---|
| Open prompt | In this approach open short prompts are presented to the user. Because of these short prompts the dialogues are generally shorter and therefore the goals of the user are accomplished quicker. This type of prompting technique is only suitable for experienced users and a good quality speech recogniser and complete and well-modelled dialogues are essential. Open prompts are the dominating type of prompts that the SWAMP client generates. |
| Closed prompt | These prompts tell the user exactly what they can say (e.g., "What type of call would you like to make? Please say collect, calling card, third number, person to person, or operator"). This type of prompting has resulted in users' having greater success with SUIs. |
| Removable hints | These provide a personalised approach where the application keeps track of how many times a user has successfully answered a prompt. At first, it supplies hints on the prompt by suggesting things the user can say in response to the prompt. Once the user has successfully answered the prompt several times, the hints are removed. Since the user has to identify himself at the start of each session this technique could be applied in the SWAMP client. |
| Layered approach | This is a combination of short open prompts with longer closed prompts that are more direct. For example, when a user doesn't respond to a prompt such as "What do you want to order?" in a predetermined amount of time. The system quickly presents a more directive prompt, such as "You can order X, Y, or Z." This approach is very effective for meeting the needs of both new and experienced users. Inexperienced users get instant help about what they can order, while experienced users can make their orders quickly, guided only by the shorter prompts. |

**Size of the corpus**
While designing a dialogue to retrieve information, not only the questions to ask (prompts) but also the possible responses from the user must be modelled. For the responses on a prompt it must be determined what is expected (what information must be contained in the response?) and what can be expected (what are the valid responses?). To obtain valid responses a proper corpus of dialogues is necessary. A corpus is a compilation of possible dialogues (either real or fictional). If the corpus is too small many possible responses of the user (probably normal and valid ones) are left out and the behaviour of the speech interface in these situations is undefined or unsatisfactory. If the corpus is too large, the grammar and dialogue of the system becomes very complex. In this prototype, the size of the corpus is about fifteen utterances per service.

**Where is the information contained in an utterance?**
Utterances of users are only recognised if it conforms to the grammar of the speech interface. The grammar is written in a format specified in the SAPI5 grammar schema, which is a context free grammar (CFG). CFGs specify how any legal text can be derived from distinguished symbols. However CFGs convey no meaning. Consequently, just because the computer recognises the user’s speech utterance (in other words the utterance is valid according to the CFG) doesn’t mean

it understands the information contained in the speech utterance. Property name and property value tags are used to tag elements to artificially insert some meaning in them. During the design of a grammar it is important to identify the places in an utterance where relevant information is located.

### 6.1.2 Dialogue representation

All the design issues mentioned earlier make modelling dialogues much more complicated and complex then modelling a GUI. Without a proper representation technique, the dialogues can quickly become very complex and unmanageable. In this project dialogues are represented by flow diagrams containing nodes representing start/begin points of a dialogue, boxes representing actions (e.g. an utterance from a user or an action from the system), diamonds representing decisions point and arcs to connect the nodes, boxes and diamonds. A dialogue always begins with a start node and ends with an end node. Within these nodes, the dialogue travels from box to box along the arcs and branching at the decision diamonds. A successful dialogue corresponds to a path in the flow diagram from the start node to the end node.
Speech dialogues are context sensitive. In this representation, the context is defined by the positions within the dialogue flow. Each box represents a certain state or context. The arcs branching from a box indicate the options available within that context and the branches leading to a box define how that context can be achieved. The power of above dialogue representation technique lies in the fact that dialogues are represented in a generic way. E.g. the (user action) boxes define what the user can say at that moment in the dialogue, but not how it must be said (this is defined in the grammar). In this way, a single path in the dialogue flow diagram can represent whole categories of similar dialogues.
A well-modelled dialogue flow diagram is one where each possible dialogue fits in. Table 19 shows an example dialogue for the KM registration service. The flow of this dialogue fits into the flow diagram in Figure 6.1-1[1] (accentuated). This example was chosen because of its simplicity, in practice the dialogues are so complex and the dialogue flow diagrams so large that it is best to split them up into one main dialogue and several smaller sub dialogues. For each sub dialogue a separate dialogue flow diagram is designed and referred to in the main dialogue flow diagram (by means of sub dialogue nodes), thus making the dialogue somewhat manageable. See appendix A7 where the dialogue flow diagrams for all the services are listed. Another use for the dialogue flow diagrams occurs during the testing phase. Since each path from the start node to the end node corresponds to a successful dialogue. The correctness of the implementation of the dialogues can easily be verified by traversing all the paths in the dialogue flow diagrams.

*Table 19: example dialogue*

| |
|---|
| U: Change trip type<br>S: Is it a business or a private trip?<br>U: It’s a business trip?<br>S: OK, what’s the project ID for this business trip?<br>U: Project ID is SWAMP<br>S: Do you want to set the project ID to SWAMP?<br>U: Yes |

[1] The dialogue flow in the figure is not complete, the flow after the “no” (in the lower right corner of the figure) is not modeled yet. Clearly what comes next is context sensitive, as is the action before that (“ask confirmation”). A way to model this is to create 3 new sub dialogue flows (a separate one for each call of “ask confirmation”).

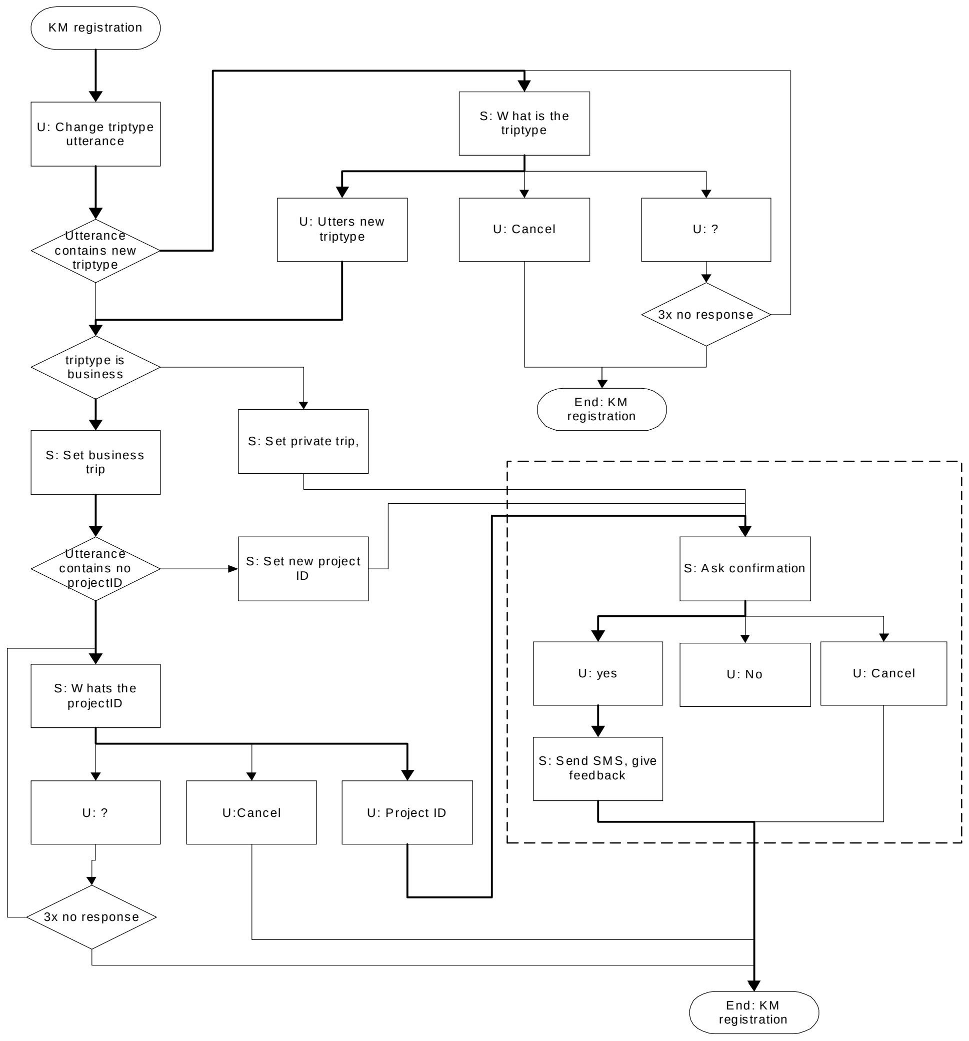


*Figure 6.1-1: a dialogue flow diagram for the KM registration service*

### 6.1.3 Error handling

Fault tolerance is an important issue in dialogue design, since errors and exception are very common and can come in many ways (e.g. the user wasn't listening, misunderstanding by user, misunderstanding by the speech interface, ambiguity in a user's utterance). The way errors are handled in SWAMP is similar to the way humans do during dialogue: by requesting clarification, elaboration, and expansion. Unlike with the GUI, errors can be expected to occur in any dialogue at any position, so this should already be dealt with in dialogue design. Consequently, error handling and prevention mechanisms (e.g. give feedback, ask for confirmation, ask for clarification) must be incorporated in the design. The question however is how and when to apply these error handling mechanisms. This depends on the user, as experienced users may find error handling mechanisms in the dialogue annoying, while new users can't complete a single dialogue without these mechanisms. In section 4.1.3.1 a typical user of the system was defined, the dialogues in SWAMP were designed with this user in mind.

Partial information and Ambiguity
For each service supported by the speech interface, some additional information is needed. For instance, if an appointment with a client must be made, the date and time of the appointment are needed. In human conversation it is quite common to supply partial information and fill in the missing pieces as the dialogue goes along. In the speech interface it is the task of the dialogue manager to detect this and ask the user to supply the missing information and interpret the returned answers to fill in the gaps. Furthermore the dialogue manager must be aware of possible ambiguity in the information of the user and must have mechanisms to detect and deal with it. The information needed and the place where ambiguity can occur depends on the service. It is dealt with (separately for each service) in the dialogue design ([Yang2 2001]).

No-recognition
No recognition occurs when the speech recogniser does not recognise an utterance of the user. This happens when:

| **Problem** | **Solution** |
|---|---|
| The quality of the microphone or sound card is too poor. | Use a better microphone and/or soundcard or do something to suppress the noise. This falls out of the scope of this project. |
| The user speaks too fast or has an accent | The speech recogniser must be trained to adjust to the user (or the other way around). |
| The utterance of the user is not defined in the grammar. | The original question is stated in another form and, if applicable, possible answers are supplied. Technically, the utterance must be added to the corpus and the grammar must be updated. |
| The utterance was not valid | Repeat the question and ask the user to rephrase. |

Mis-interpretation
Except in exceptional situations (during an emergency) confirmation from the user (via the speech recogniser) is needed before any radical actions are taken. The dialogue designer defines the place where these confirmation requests occur (see the dialogue specification document of the SWAMP project).

No response
With each question of the dialogue manager a special timeout value is associated indicating the amount of time the manager must wait for a response from the user. If the time has passed and there is still no response from the user (or the response is not understood) the question is repeated up to a maximum of three times in which the dialogue manager gives up with a suitable (spoken) error message.

## 6.2 Implementation overview

The dialogue manager can be divided into two parts (see Figure 6.2-1):
1. An AI part to do the representation and reasoning with knowledge. An embedded CLIPS expert system shell (CLIPS engine) [CLIPS 2000] is used to implement this part.
2. A C++ part to embed CLIPS and to perform translation and communication between CLIPS and external entities (SAPI and the main application).

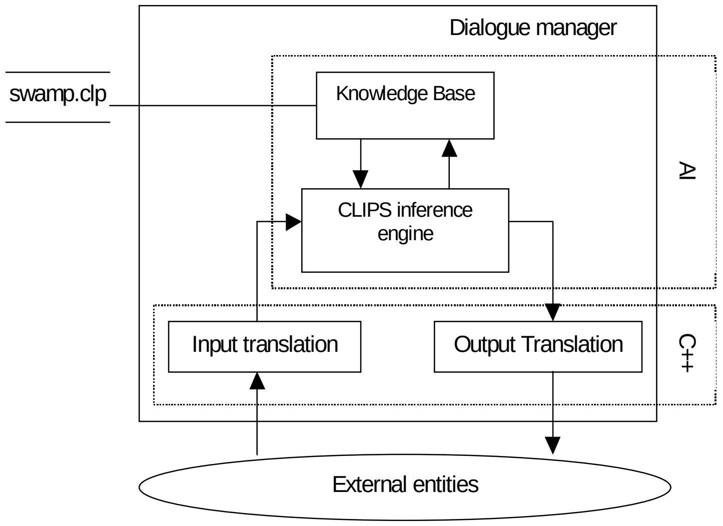


*Figure 6.2-1: Overview of the dialogue manager*

## 6.3 The C++ part

The C++ part is responsible for the communication between the embedded CLIPS engine and the outside world (SAPI and main application). It has a sensing function to sense events from the outside world, a translating function to translate messages from the outside world into a format understandable to the CLIPS engine and an actuating function to translate CLIPS engine messages into actions on the outside world. The outside world for the dialogue manager consists of two entities: the GUI of the main application and SAPI. Communication with the GUI is necessary, because the SWAMP client is a bi-modal application and the different modalities (GUI and SUI) need to be synchronised with each other. SAPI is the location where utterances from the user are recognised and where the text to be synthesised is sent. Since the dialogue processing is done in the dialogue manager some kind of interaction with SAPI is needed. In the following sections, the interaction mechanism between the C++ part of the dialogue manager and the two external entities is described.

### 6.3.1 Interaction with the main application

Whenever the user uses the GUI, he can change the state of the application in such a way that the dialogue manager needs to change its representation of the world. Therefore, the dialogue manager must somehow be notified of the user's action on the GUI. In SWAMP the communication is achieved by means of GUI messages of the application framework. The application framework is devised by Microsoft to build window applications. It generates events in response to GUI actions of the user, such as keystrokes and mouse clicks. The dialogue manager uses the generated events to react on GUI changes. In addition the dialogue manager itself generates events to cause GUI changes. Following is a short description of the synchronisation mechanism.

The application framework in which the WAM-Pilot was built fires an event whenever a GUI action occurs. For example, if the user clicks on a button, the button calls a pre-arranged function in the application. This function is called the event handler of the button and contains the code that implements the appropriate action for the button. The dialogue manager is only interested in actions that actually lead to a change in the dialogue manager's internal model of the world. The communication from the main application to the dialogue manager is realised by adding code in the relevant event handlers, to send a message to the Dialogue Manager whenever the event handler is invoked. All actions performed by the speech interface originate from the AI component. When an action occurs that requires a change in one or more GUI controls the AI components sends a message to the dialogue manager notifying it of this event. The information telling the dialogue manager what exactly needs to be modified in the GUI is contained between the REACT tags of the messages from the AI component (see 6.3.4). After receiving a message from the CLIPS engine, the C++ part scans the information contained within the react tags and performs the actions specified in the message.

### 6.3.2 Sensing recognition events from SAPI

The ASR engine uses events to report information about what is being recognised. There are several events the engine can report. These indicate, for example, that the engine has detected the start or end of speech, or that it has a hypothesis or a completed recognition result. So far, there are more than 30 events. With the *setInterest()* command the type of events that are sensed by the dialogue manager's sensors are restricted to only the relevant events. The dialogue is only interested in the following events:

| Event | Description |
|---|---|
| SPEI_PHRASE_START | SR engine has detected the start of a recognisable phrase. |
| SPEI_RECOGNITION | SR engine's best hypothesis for the audio data. |
| SPEI_FALSE_RECOGNITION | Apparent speech with no valid recognition. |

The most important is the SPEI_RECOGNITION event. The SPEI_RECOGNITION event is fired when a user utterance matches with an active grammar rule. The dialogue manager specifically listens for this event. After receiving this event the receiving application can retrieve the words that have actually been said (and also additional information such as the ID of the rule that fired etc.). SAPI returns this information in a structure: SPPHRASE. The SPPHRASE structure does not only contain the phrase spoken, but also additional information such as the ID of the rule that fired, semantic tag names and values etc.

```
typedef [restricted] struct SPPHRASE
{
   ULONG                           cbSize;
   LANGID                          LangID;
   WORD                            wReserved;
   ULONGLONG                       ullGrammarID;
   ULONGLONG                       ftStartTime;
   ULONGLONG                       ullAudioStreamPosition;
   ULONG                           ulAudioSizeBytes;
   ULONG                           ulRetainedSizeBytes;
   ULONG                           ulAudioSizeTime;
   SPPHRASERULE                    Rule;
   const SPPHRASEPROPERTY          *pProperties;
   const SPPHRASEELEMENT          *pElements;
   ULONG                          cReplacements;
   const SPPHRASEREPLACEMENT      *pReplacements;
   GUID                           SREngineID;
   ULONG                          UlSREnginePrivateDataSize;
   const BYTE                     *pSREnginePrivateData;
} SPPHRASE;
```

| Member | Description |
|---|---|
| `CbSize` | The size of this structure in bytes |
| `LangID` | The language ID of the phrase elements |
| `Wreserved` | Reserved for future use |
| `UllGrammarID` | ID of the grammar that contains the top-level rule used to recognise this phrase |
| `FtStartTime` | Absolute time for start of phrase audio as a 64-bit value |
| `UllAudioStreamPosition` | Start time in the audio stream for this phrase |
| `UlAudioSizeBytes` | Size of audio data in bytes for this phrase |
| `UlRetainedSizeBytes` | Size in bytes of the retained audio data |
| `UlAudioSizeTime` | Length of phrase audio in 100-nanosecond units |
| `Rule` | Information about the top-level rule that was used to recognise this phrase |
| `pProperties` | Pointer to the root of the semantic property tree |
| `pElements` | Pointer to the array of phrase elements |
| `cReplacements` | Number of text replacements. |
| `pReplacements` | Pointer to the array of text replacements |
| `SREngineID` | GUID that identifies the particular SR engine that recognised this phrase |
| `UlSREnginePrivateDataSize` | Size of the engine's private data (in bytes) |
| `PSREnginePrivateData` | Pointer to the engine's private data |

According to the default run-time model of the SAPI the ASR engine continues recognising as long as data is available and it is not explicitly told to stop. Typically, after an engine reports recognition, it will check for grammar changes and then continue reading data and recognising. Recognition stops if an application sets the recognition state (with SetRecoState) to inactive. By inspecting the contents of the SPPHRASE structure, the rule that fired can be obtained. The element’s property name and value can now be retrieved by traversing the *pElements* pointer.

Consider the email problem mentioned in section 5.2.2.2. Suppose we want to retrieve the information contained in the semantic tag NAME from a rule with ID EMAIL. Whenever recognition is received with this rule ID, the property tree (SPPHRASE.pProperties) is searched for the property named NAME. Then the function ISpRecoResult::GetPhrase is called with (SPPHRASEPROPERTY) pNameProp.ulFirstElement and (SPPHRASEPROPERTY) pNameProp.ulFirstElement, and the application can retrieve the exact text that the user spoke into the dynamic rule (e.g. user says "send new e-mail to Harry," and we would retrieve "Harry," user says "send new e-mail to Hermione," and we would retrieve "Hermione,").

### 6.3.3 The translating function

As already mentioned, the dialogue manager is divided into two parts. The first part takes care of the communication with external entities while the second part (the CLIPS engine) does the actual dialogue managing. Because the external entities (SAPI and the main application) differ radically from the dialogue managing part a translation step is needed in the dialogue manager. The translated messages are sent to standard input (stdin) of the CLIPS engine. There are two types of input messages that can be distinguished:

| | |
|---|---|
| Speech related | These messages are a result of utterances from the user. |
| GUI related | These messages are part of the synchronisation process between SUI and GUI. |

Speech related messages sent to CLIPS have the following format:

```
(assert (Type  LangID Rule Confidence Property Value))
```

*Figure 6.3-1: Format of a speech related input message to CLIPS*

| | |
|---|---|
| **Assert** | Assert is a CLIPS command to add a fact to the fact-list. |
| Type | This field indicates the type of the message. The only message type defined at this moment is: RECOGNISED |
| LangID | The language identifier (161 for English) |
| Rule | The ID of rule that fired |
| Confidence | Confidence of the speech recogniser that the rule has indeed fired. SAPI does not demand that all compatible speech recognisers implement this. |
| Property | Property name (optional) |
| Value | Property value (optional) |

The translation component translates SAPI events into CLIPS speech related messages. There exists the following correspondence between SAPI elements and CLIPS input elements:

*Table 20: Correspondence between SAPI elements and CLIPS input elements*

| SAPI element | CLIPS element |
|---|---|
| SPPHRASE.LangID | LangID |
| SPPHRASE.Rule.ulId | Rule |
| SPPHRASE.Rule.Confidence | Confidence |
| SPPHRASE.pProperties.pszName | Property |
| SPPHRASE.pProperties.pszValue | Value |

Now that the correspondence between the elements is known, the translation process is straightforward.

Main application related messages are sent by the event handlers of GUI controls. Each message contains the service the control belongs to, the parameter of the control and the new value of that parameter. Here also, the transformation from main application message to CLIPS message is straightforward. Main application related messages have the following format in CLIPS:

```
(assert (GUIMSG Service Parameter Value))
```

*Figure 6.3-2: format of a GUI related message to CLIPS*

| | |
|---|---|
| **assert:** | assert is a CLIPS command to add a fact to the fact-list. |
| GUIMSG: | This field indicates the type of the message (GUI message in this case). |
| Service: | The service involved. This can be any of the services available. |
| Parameter: | The parameter that has changed. |
| Value: | The new value of the property. |

### 6.3.4 The actuating function

The CLIPS engine sends its output messages to standard output (stdout). These messages are fetched and parsed by the C++ part of the dialogue manager. An output message from CLIPS is a structured string consisting of elements. Each element has a type and a content. The content of an element is always surrounded by tags, which specify where an element begins and where it ends. All elements are optional within a message. Each element of the output message represents an action that must be taken by the actuators (speech resources and GUI).

Example:

```
<say>hello</say>
```

This element has type: say
And contents: hello

Currently, the following tags are defined:

*Table 21: List of defined tags in CLIPS output messages*

| Tag | Description |
|---|---|
| SAY | The content of this type of element contains the text to be synthesied by the TTS engine. The contents of this element is used as main parameter in the Speak() method. |
| ACT | The content of this type of element contains a list of grammar rules that needs to be activated separated by a white space. The C++ part of the dialogue manager calls the activate() method with elements of the list as parameter. |
| DEACT | The content of this type of element contains a list of grammar rules that needs to be de-activated separated by a white space. The C++ part of the dialogue manager calls the deactivate() method with elements of the list as parameter. If the same grammar rule exists in both the ACT and DEACT element of the same message, the rule is first de-activated and then activated again. So the net result is that the rule is activated in the end. |

| | |
|---|---|
| REACT | This element specifies an action to invoke in the main application. It contains an action identifier followed by action parameters. Both the identifier and parameters are fed to the handleaction() method. |
| TIME | A timeout value in milliseconds. The CLIPS engine is notified if no relevant speech event occurred in the period of time specified after reception of the message. |

Actions that can be performed on the speech resources include the activation or deactivation of grammar rules, synthesis of text phrases and other functions to control various properties (volume, speed, pitch etc.) of the synthesised speech. The elements SAY, DEACT and ACT are related to actions on speech resources. The methods used to convert the contents these elements into concrete actions are summarised in Table 22 (only the TTS methods) and Table 23 (only ASR methods). REACT elements are sent to the ActionHandler (Actionhandler.cpp), which contains a generic method to correctly modify SWAMP client's GUI controls specified in the contents of the REACT element (Table 24).

*Table 22: Methods used to control the TTS engine.*

| Method | Description |
|---|---|
| Speak(WCHAR ***phrase**) | Synthesise the contents of the string **phrase.** In addition information such as emphasis, pitch etc. can be embedded in the string using xml tags. |
| SetVolume(int **level**) | Set the volume of the synthesised output to **level** (a value between 0 and -100). |
| ChangeVoice(Cstring **voice**) | Change the current TTS engine, to the TTS engine named **voice**. Of course **voice** must be installed and SAPI compliant. |
| SetSpeed(int **level**) | Sets the talking speed of the TTS engine (a value between –10 and 10). |

*Table 23: Methods used to control the ASR engine*

| Function | Description |
|---|---|
| ActivateSAPIRule(WCHAR ***rule**) | Activate the SAPI rule with the name **rule.** |
| DeactivateSAPIRule(WCHAR ***rule**) | De-activate the SAPI rule with the name **rule.** |
| StopASR() | (Temporary) stops the recognition engine. (Only has effect if the engine is active) |
| LoadGrammar(Cstring **file**) | Load the grammar from the file named **file** |
| LoadUserNames() | Dynamically load the list of usernames into the grammar. |
| LoadLocations() | Dynamically load the list of Location names into the grammar. |
| LoadProjectIDs() | Dynamically load the list of Project ID's into the grammar. |
| LoadCarIDs() | Dynamically load the list of Car ID's into the grammar. |

*Table 24: Methods used to manipulate GUI controls*

| Function | Description |
|---|---|
| HandleAction(Cstring **id** Cstring **param**) | The **id** parameter defines what action to take and the **param** parameter specifies the parameters for that action. |

## 6.4 AI part

In Figure 6.2-1 a graphical overview of the architecture of the dialogue manager was given. The figure showed the division of the dialogue manager in a C++ part and an AI part. The function of the AI part is to represent and reason with the knowledge available so that natural dialogues with the user can be achieved. The process of extracting the users' goals from recognised speech and taking the appropriate actions to achieve those goals requires some kind of reasoning. In particular given the fact that the users' utterances are context sensitive and sometimes ambiguous. Therefore an internal model of the real world must be maintained within the dialogue manager, because only then can the utterances be put in the correct and proper context (see Example 2). In the AI part of the dialogue manager used artificial intelligence tools were used as an aid to build and maintain this model.

*Example 2: An example of a context sensitive user utterance*

If the user utters the phrase: "How long?" he can mean (depending on the context):
- How long is the traffic jam?
- How long do we have to wait?

Further applications of reasoning in the dialogue manager include questions on when and how to prompt for information (section 6.1), when to ask for confirmation and clarification and how to handle errors and ambiguity in speech (section 6.1.3) etc. The tool used to implement the AI part is CLIPS. In this section the chosen reasoning model and the implementation of this model in CLIPS will be discussed.

### 6.4.1 The reasoning model

In the design of systems that are required to perform high-level management and control of tasks in complex dynamic environments a number of different approaches have emerged as candidates for reasoning models. In the choice for a suitable reasoning model for the dialogue manager (one that is capable of adequately describing the reasoning behaviour of the dialogue manager), it is important to distinguish practical reasoning from theoretical reasoning. Theoretical or deductive reasoning is concerned with valid inference: with what follows from the literal meaning of the premises [Bell J 1992]. Practical or pragmatic reasoning is concerned with what follows from the premises given a context and is more directed towards actions. Clearly the latter type of reasoning model is more suitable for the dialogue manager. Ultimately, the Belief-Desire-Intention (BDI) model [Wooldrige 2000] is chosen as the reasoning model for the dialogue manager. The BDI model has its roots in the philosophical tradition of understanding practical reasoning in humans. It gets its name from the fact that it uses beliefs, desires and intentions in rational action. What makes the BDI model particularly interesting is that it combines three important elements:

1. It is founded upon a well-known and highly respected theory.
2. It has been implemented and successfully used in a number of complex fielded applications
3. The theory has been rigorously formalised in a family of BDI logics.

Figure 6.4-1 describes an algorithm of a basic control loop for a BDI agent; this control loop fits the desired behaviour of the dialogue manager quite well. In an implementation of a dialogue manager according to this model, the dialogue manager continuously executes a cycle of observing the world and updating its beliefs, deciding what intention to achieve next, determining a plan of some kind to achieve this intention, and then executing the plan. The purpose of the above and following (section 6.4.3) formalisation is to build a formally verifiable and practical system. This formalisation can be used to specify design, and verify that the system, when placed in the right environment, will exhibit all and only the desired behaviours.

```
Algorithm BDI agent control loop
    1. While true
    2.     Observe the world;
    3.     Update internal world model
    4.     Deliberate about what intention to achieve next;
    5.     Use means-ends reasoning to get a plan for the intention;
    6.     Execute the plan
    7. End while
```

*Figure 6.4-1: A basic BDI agent control loop*

### 6.4.2 The CLIPS engine

In this project, CLIPS is used to implement the BDI reasoning model discussed earlier. CLIPS is a rule–base expert system tool developed by the Software Technology Branch (STB), NASA/Lyndon B. Johnson Space Centre. The reason why it is chosen instead of a normal procedural language is very simple: The common approach to building applications is to understand the desired behaviour first, and then try to code this behaviour. Using a procedural language (e.g. C++), the specification of what should be done as well as how it should be done must be coded. For a complex task the control logic quickly becomes difficult to write, debug and maintain. A rule-based expert system such as CLIPS allows us to avoid, to some degree, the last step. Furthermore CLIPS is freeware and can be easily embedded into a C++ application. The CLIPS engine consists of three basic elements:

*Table 25: Basic elements of the CLIPS engine*

| Element | Description |
|---|---|
| Facts | Initial facts are defined in the swamp.clp file, while new facts are asserted and retracted at runtime. All input from SAPI or the main application are also represented as facts in CLIPS. This explains the “assert” keyword in the messages to the CLIPS engine (section 6.3.3), as assert is the CLIPS command to insert a fact into the fact-list. |
| Knowledge-base | The knowledge base is the knowledge representation component in the CLIPS engine. Knowledge is represented in the form of rules, which specify a set of actions to be performed for a given situation. The rules are defined in swamp.clp. |

| | |
|---|---|
| Inference engine | The inference engine is provided by CLIPS, it takes care of the matching of the facts against the rules. Thus it controls the overall execution of the CLIPS engine. |

### 6.4.2.1 CLIPS data structures

In principle there are just two types of information structures in CLIPS: facts and rules. Facts represent a single chunk of truth within the CLIPS runtime. Each fact can be added using an *assert* command, deleted using *retract* command or modified using *retract* followed by an *assert* command. Rules specify a set of actions to be performed in a given situation. Rules are static and cannot be changed. A rule is similar to an IF THEN statement in procedural programming, therefore it is nothing more then a description of a set of conditions and a set of actions to take if the conditions are true. A rule in CLIPS has the following form:

```
(defrule <rule-name>  [<comments>]
     [<declaration>]                    //Rule Properties
     condition 1
     :                                  //Invocation condition
     condition n
=>
     action 1
     :                                  //Body
     action n
)
```

*Figure 6.4-2: Syntax of a CLIPS rule*

A rule in CLIPS can be characterised by a body and an invocation condition. The body (corresponding to the Right Hand Side (RHS) or the THEN part in an IF THEN statement) is a list of actions that can be performed to achieve a particular state (corresponding to a desire in the BDI model). The invocation condition defines the circumstances under which the dialogue manager should consider the rule (corresponding to the Left Hand Side (LHS) or the IF part in an IF THEN statement). There are four kinds of rules:

| | |
|---|---|
| Speech reaction rules | These rules contain actions that react to an utterance of the user. It is characterised by the RECOGNISE keyword in the invocation condition of the rule. The body of the rule contains actions to take as a result of the utterance. |
| GUI reaction rules | These rules contain actions to perform as a result of GUI messages. They are characterised by the GUIMSG keyword in the invocation condition of the rule. The actions in the body of these rules consist of updates of facts (the internal presentation of the real world) brought about by actions on the GUI. |
| Management rules | The purpose of management rules is to manage the state of the system. |
| Belief rules | These rule describe the general relationship (implication) between certain facts. |

### 6.4.2.2 CLIPS execution

Execution of CLIPS can be represented as a sequence of Recognise-Select-Act cycles:

```
While (true)
    Recognise    //The current facts are matched against the invocation
                 //condition of the rules.  Rules that match are //"instantiated"
    Select       //Selection of the instantiation to be fired
    Act          //Executing the body of the selected instantiation
End while;
```

*Figure 6.4-3: overview of the CLIPS execution cycle*

In the recognise step, the facts are matched against the left hand side of the rules. The inference engine takes care of this matching. It is easy to see that the facts are the data that stimulate execution of the CLIPS engine. In other words the CLIPS engine is data-driven. Execution cycles continue as new facts are asserted and old facts are retracted during the course of the execution. If the left hand side of a rule matches the facts, the rule becomes instantiated.
During one execution cycle more then one rule can be instantiated. In this case a selection needs to be made among the instantiated rules. The selection process is called conflict resolution. The precedence of an individual rule within conflict resolution can be set with the "declare salience" declaration: a higher salience gives a rule a higher priority. Otherwise the depth first strategy (activated rules are placed above all rules of the same salience) is the strategy for conflict resolution. Once an instantiation has been selected, the body of the instantiation is executed in the last step of the cycle.

### 6.4.2.3 Embedding CLIPS

The CLIPS systems may be executed in three ways: interactively using a simple, text-oriented, command prompt interface; interactively using a window/menu/mouse interface; or as embedded application in which the user provides a main program and controls execution of the system [NASA1 1993]. In this project, CLIPS is used in the latter fashion: as an embedded system in the SWAMP client application. Needless to say, the SWAMP client acts as the main program that controls the execution of CLIPS. The SWAMP client first initialises CLIPS by calling the function *InitializeCLIPS*. Then the appropriate constructs (in the file swamp.clp) are loaded. After that a reset and a run command is issued to reset the runtime environment and build the initial model of the world. This concludes the initialisation; the embedded CLIPS system is now ready to accept events. Whenever an event occurs the translation function of the dialogue manager translates it into a corresponding fact and sends it to CLIPS. Once again the run command is issued to allow the appropriate rules in CLIPS to execute as the result of the appearance of the new fact. Afterwards the output (and/or errors) of the CLIPS engine is collected and processed. Above discussion is summarised in Figure 6.4-4.

```
Initialise CLIPS
Load "swamp.clp"            Loads a set of constructs into the CLIPS
Reset                       Resets the CLIPS environment
Run                         Run initial rules
While (event)
    Sendinput(event)        Send input to CLIPS
    Run                     Allow rules to execute, as a result of the new input
    Receiveoutput(stdout)   Receive output from CLIPS
    Receiveoutput(stderr)   Receive errors from CLIPS
    Process output
End while
Clean up CLIPS
```

*Figure 6.4-4: Embedded CLIPS control loop*

When running CLIPS as an embedded application, many of the capabilities available in the interactive interface (the un-embedded version of CLIPS) are also available through function calls. Calls to CLIPS are made like any other subroutine. The functions are documented in the advanced CLIPS programming manual [NASA2 1993]. Prototypes for these functions can be included by using the clips.h header file. Some of these functions are used in the SWAMP client to collect the status of the CLIPS engine (fact list, agenda etc.) for debugging needs.

### 6.4.3 Knowledge representation

There is a straightforward mapping between entities in the BDI model and the abstract information structures of the dialogue manager. Beliefs correspond to information the dialogue manager has about the real world: the state of SWAMP (section 4.1.3.1). These beliefs may be incomplete or incorrect. Desires represent the goals of the dialogue manager: successful dialogues with the user. Finally, the intentions represent the desires the dialogue manager has committed to achieving. As the reasoning part of the dialogue manager is implemented in CLIPS, there must also exist a correspondence between concrete CLIPS data structures and the entities in the BDI model. The relationship between each entity in the BDI model and its CLIPS counterpart is discussed next.

Beliefs
Beliefs in the BDI model are implemented as facts and rules in CLIPS. Facts are used to construct the dialogue manager's internal representation of the world. Facts can be seen as propositions and thus can only consist of a set of literals without disjunction or implication. Therefore special rules (belief rules) are used to complete the representation of beliefs. Belief rules represent the general relationship between facts (e.g. IF utterance=help THEN AlertLevel=high).

Desires
One way of modelling the behaviour of BDI reasoning [Rao 1995] is with a branching tree structure, where each branch in the tree represents an alternative execution path. Each node in the structure represents a certain state of the world, and each transition a primitive action made by the system, a primitive event occurring in the environment or both. In this formal model, one can identify the desires of the system with particular paths through the tree structure.
The above description of the branching tree structure is logically similar to the structure of the dialogue flow diagrams described in section 6.1.2. In fact, both structures represent exactly the

same: a path through the dialogue flow diagram is a successful dialogue, which is also a desire and therefore a path through the branching tree of the BDI reasoning model. As a result, the dialogue flows diagrams can be treated as the structures that describe the behaviour of the dialogue manager. They are directly implemented in CLIPS rules, each rule corresponds to a branch in the dialogue flow. Rules are both the means for achieving certain desires and the options available for the dialogue manager. Each rule has a body describing the primitive sub goals that have to be achieved for rule execution to be successful. The conditions under which a rule can be chosen as an option are specified by an invocation condition. The set of rules that make up a path through the dialogue flow, correspond to a desire.

Intentions
The set of rules with satisfied invocation conditions at a time *T* (the set of instantiated rules) correspond to the intentions of the dialogue manager at time *T*. Obviously the intentions of the system are time dependent. The dialogue manager adopts a single-minded commitment strategy, which allows continuous changes to beliefs and drops its intentions accordingly. In other words the intentions of the system can be affected by the utterances of the user in contrast to blind commitment in which an intention is always executed no matter changes in beliefs.

### 6.4.4 Heuristics for the translation from dialogue flow diagram to CLIPS rules

In the previous section it was shown that the desires of the dialogue manager component in the SWAMP client can be represented by dialogue flow diagrams. The flow diagrams are systematically translated into an executable system formulated in CLIPS rules. This section discusses the implementation of the desires. In particular the heuristics used for the translation from dialogue flow diagrams to CLIPS rules.

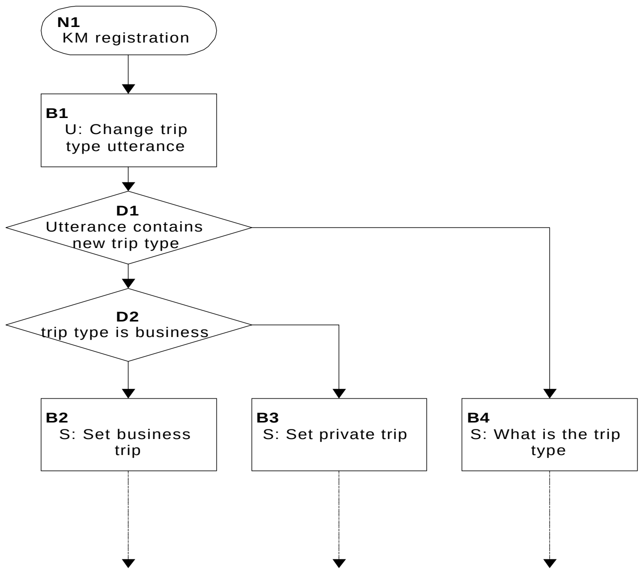


*Figure 6.4-5: Dialogue flow diagram for the heuristics example*

Suppose we must transform a dialogue flow diagram as in Figure 6.4-5. (The start of the KM registration service). This dialogue is initialised when the user utters a phrase that matches the grammar for a change trip type utterance (box **B1**). Notice that box **B1** has 3 branches (to the

boxes **B2**, **B3** and **B4**), furthermore we see that the action in **B1** is a speech action from the user. From this we conclude that the dialogue flow should be implemented using 3 speech rules. The invocation conditions for each rule are the evaluated values of the expressions in the decision diamonds **D1** and **D2**. The body of each rule contains the actions specified in the corresponding destination boxes. Furthermore, the body of the rules also contain actions to anticipate what follows after the action e.g. after box **B4** the user must supply the new trip type so the grammar rules for trip type utterances should be activated.
The resulting 3 CLIPS rules are presented next.

The first rule corresponds to the branch from box **B1** to **B2** in Figure 6.4-5. The property name TripType with value business satisfies condition **D1** and **D2**. The actions taken satisfy **B2** (between the <SAY> tags) and anticipate future utterances of the user by activating the yes-no grammar rule. The other actions in the rule body are used to update the internal representation of the world.

```
(defrule KM_Registration_Business
   ?in<-(RECOGNISED 161 VID_KMREG_TRIPTYPE 50 TripType business)
   ?pos<-(POSITION MAIN RUNNING)
   =>
   (printout t "<SAY>Do you want set the triptype to business?</SAY>
                <ACT>VID_YESNO</ACT>
                <DEACT>"?*Mainrules*"</DEACT>
                <REACT></REACT>" crlf)
   (retract ?in)
   (retract ?pos)
   (assert (POSITION MAIN KMREG))
   (assert (WANT CONFIRM))
   (assert (QUESTION KMREG business))
)
```

The second rule corresponds to the branch from box **B1** to **B3** in Figure 6.4-5. The property name TripType with value private satisfies condition **D1** but not **D2**. The actions taken satisfy **B3** (between the <SAY> tags) and anticipate future utterances of the user by activating the yes-no grammar rule. The other actions in the rule body are used to update the internal representation of the world.

```
(defrule KM_Registration_Private
   ?in<-(RECOGNISED 161 VID_KMREG_TRIPTYPE 50 TripType private)
   ?pos<-(POSITION MAIN RUNNING)
   =>
   (printout t "<SAY>Do you want set the triptype to private?</SAY>
                <ACT>VID_YESNO</ACT>
                <DEACT>"?*Mainrules*"</DEACT>
                <REACT></REACT>" crlf)
   (retract ?in)
   (retract ?pos)
   (assert (POSITION MAIN KMREG))
   (assert (WANT CONFIRM))
   (assert (QUESTION KMREG private))
)
```

The third rule corresponds to the branch from box **B1** to **B4** in Figure 6.4-5. Because there is no property name both conditions **D1** and **D2** fail. The actions taken satisfy **B4** (between the <SAY>

tags). Future utterances are anticipated by the activation of the VID_TRIPTYPE_ONLY grammar rule. The other action commands serve as updates of the internal model.

```
(defrule KM_Registration_No_Triptype
 "User wants to register KM level but no trip type is given"
   ?in<-(RECOGNISED 161 VID_KMREG_TRIPTYPE 50)
   ?pos<-(POSITION MAIN RUNNING)
   =>
   (printout t "<SAY>Is it a business or a private trip?</SAY>
                  <ACT>VID_TRIPTYPE_ONLY</ACT>
                  <DEACT>"?*Mainrules*"</DEACT>
                  <REACT></REACT>" crlf)
   (assert (POSITION MAIN KMREG))
   (retract ?pos)
   (retract ?in)
)
```

# 7 Test

Testing and evaluation play an important part in the software development cycle. The main reason for testing software is to determine whether the specifications have been met [Pressman 1994].Testing of a speech interface in this case has an additional goal: to determine the usability of the speech interface and whether the interface is indeed an improvement on alternative interfacing methods. For applications that are designed to run while the user is simultaneously performing other tasks, yet another test can be conducted. This test is needed to determine that the application does not pose too much distraction to the user, that he is unable to concentrate on the most important task at hand. A common method used to test this is with workload assessment techniques [Yang1 2001]. Because the chosen implementation strategy of the SWAMP client is rapid prototyping with extensive iterations, testing was already done during implementation. But these tests consisted mainly of functional and dialogue testing. The final test of the SWAMP client prototype however, is a usability test. This chapter discusses the usability tests that were performed (after an acceptable prototype was completed).

## 7.1 Test parameters

The main reason for testing software is to determine whether the specifications have been met. In order to test an application according to this definition, some questions have to be answered (e.g. what is the application to be tested?, what are the specifications of the application?).
Having these answers on paper can save a lot of time and trouble during testing, furthermore it serves as a basis through which the test results must be viewed and test results from other similar applications can be compared. This section gives the answer to the most important of these questions.

| | |
|---|---|
| Test environment | The test environment is a silent room with only the tester, the test subject, and a computer running the SWAMP client present. |
| Test procedure | • At the start of the usability test the test subject gets a description of the SWAMP application and its services. Furthermore he is informed about the test procedure.<br>• The application to be tested (the SWAMP client) is running on a laptop (PIII 500). During the test GUI input can be given through the keyboard of the laptop and a mouse. SUI input is given through a noise-filtering microphone and speech output comes from the internal speakers of the laptop.<br>• During a test session the tester sits next to the test subject. The test subject gets a list of tasks to perform using the GUI as well as the SUI (but not in combination).<br>• Each test run is repeated three times but under different circumstances (see section 7.3). During each run, the user has to perform the tasks two times, once using only the GUI and once using only the SUI.<br>• The tester writes down the time it takes to successfully perform each task, furthermore the number of utterances, false recognitions, and successful recognitions are noted.<br>• If the user gets stuck somewhere in the process the tester is allowed to complete the task for him, so that the test subject can continue with the next task. |

| | |
|---|---|
| Scope | The attributes tested are: grammar, quality of the speech engines, quality of dialogue manager, completeness of the designed dialogues. |
| Quality attributes | Usability of the speech interface |
| Test Basis | The speech interface of the SWAMP client |
| Acceptation criteria | No acceptation criteria are defined, we want to get an indication of the usability of the speech interface in comparison to the graphical user interface. |
| Number of test subjects | 12 |

## 7.2 Number of test-subjects

Research [Nielsen 1993] has shown that the number of usability problems found in a usability test with *n* users is:

$$N*1-(1-L)^n$$

Where *N* is the number of usability problems in the design and *L* is the proportion of usability problems discovered while testing a single user. The typical value of *L* is 31%, averaged across a large number of projects studied. Figure 7.2-1 shows the curve of the equation for *L*=31%. The figure shows that as more users are added, less and less new problems will be found, so there is no real need to test further. After the fifth user, it is a waste of time by observing the same findings repeatedly. Time can be better spent to solve the usability problems encountered. To get statistically significant results, the choice of the amount of test-subjects has been increased to twelve. The test subjects were students from the University and employees of CMG. There were 10 males and 2 females ranging in age from 19 to 30 with little or no experience with the speech interfacing technology in the SWAMP client. Since the subjects were novices, any qualitative differences between speech and other media were more likely to be noticed. Subjects with prior experience may have already adapted to these differences and could overlook them during the experience.

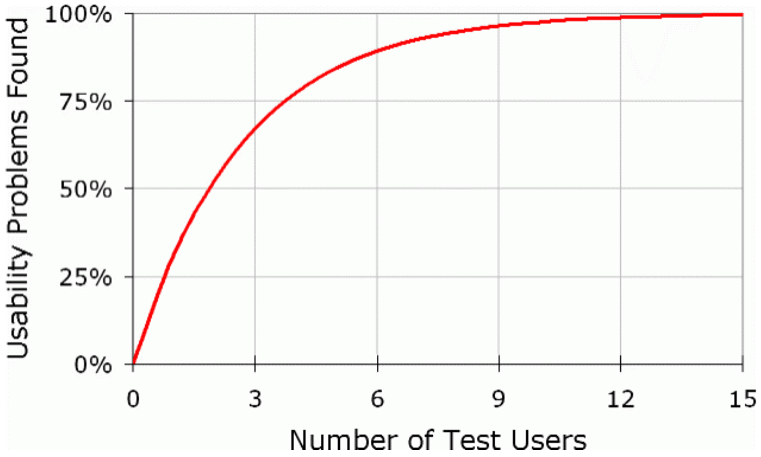


*Figure 7.2-1: Curve of the number of usability problems found in usability test with* L=0.31

## 7.3 Test scenarios

During the usability test, the test subject is asked to use the SWAMP client to perform a set of tasks according to a predefined scenario. The list of tasks is presented on a sheet of paper (Table 26). The tasks are written in such a way that it does not reveal the underlying grammar of the speech interface. The test is divided into three test runs. In each test run the test subject has to perform the same tasks first using only the GUI and then using only the SUI. Each test run is conducted under different circumstances:

| **Run number** | 1 |
|---|---|
| **Situation** | Without prior knowledge of the SWAMP clients' grammar. |
| **Description** | In this situation, the test subject must try to perform the tasks led only by his intuition. This gives the grammar designer a chance to test whether the corpus is complete enough or needs to be extended. As mistakes and exceptions are bound to occur, the dialogue designer can also deduce if the designed dialogues are robust enough. Furthermore, the user gets the chance to familiarise with the system. |

| **Run number** | 2 |
|---|---|
| **Situation** | With knowledge of the grammar of the SWAMP client. |
| **Description** | After the previous test run, the test subject gets an evaluation of his performance during the run. The tester explains to him what went well but in particular what went wrong and how he can correct that. During the second test run the user has to perform the same tasks of the previous run again. Usually there is an improvement in the success rate of the dialogues compared to the first test. The amount of improvement in the second test run gives an indication of the willingness of the users to adapt to the speech interface. Also, the comments the test subject has after this test are a better indication of the probability of acceptance of the speech interface in real circumstances. |

| **Run number** | 3 |
|---|---|
| **Situation** | Same as run number 2, but with another attention demanding task to do. |
| **Description** | This test set-up is based on the workload assessment technique called: Secondary task measures [Yang1 2001]. In this situation the user concurrently performs two tasks. The first task is to perform the tasks specified in Table 26 with the SWAMP client (once with SUI and once with GUI). The second task is to play a game of Tetris (level 1) on a mobile phone. This task is chosen to resemble car driving (at least one hand of the test subject is always occupied, the test subjects must make decisions at times, and the tasks does not require to have his constant attention). The results obtained from the previous test run serve as baseline scores, which can be used later as a comparison. The degree of the decrement in performance, when compared to the baseline scores provides a measure for the workload of the task. |

*Table 26: List of tasks to perform for the usability test*

1. Start the swamp client and log in with the following values:
- Name = <Your name>
- Project ID = SWAMP
- Car ID = Ferrari
- Back office telephone number = 0651061625

(The original values where Tu, Timenet, Neon, 0651061625 respectively)

2. Once logged in change your trip type into private

3. Ask for directions to CMG Rotterdam

4. Change trip type into business with project ID timenet

5. Something is wrong with the car: call ANWB

6. Arghh, call help

## 7.4 Results and discussion

The previous sections presented the test method and other test parameters of the usability test conducted on the speech interface of the SWAMP client. In this section the results of the usability tests will be presented and discussed.

### 7.4.1 Recognition rate

*Table 27: Number of false and successful recognitions of utterances for each of the test subjects per test run.*

| Test Run number | False Reco | Good Reco | Test Run number | False Reco | Good Reco |
|---|---|---|---|---|---|
| 1 | 22 | 22 | 2 | 16 | 21 |
| 1 | 30 | 16 | 2 | 13 | 17 |
| 1 | 15 | 20 | 2 | 16 | 20 |
| 1 | 23 | 13 | 2 | 5 | 16 |
| 1 | 18 | 11 | 2 | 27 | 18 |
| 1 | 18 | 9 | 2 | 13 | 18 |
| 1 | 21 | 23 | 3 | 7 | 19 |
| 1 | 32 | 16 | 3 | 4 | 22 |
| 1 | 26 | 22 | 3 | 4 | 21 |
| 1 | 26 | 10 | 3 | 16 | 20 |
| 1 | 7 | 16 | 3 | 9 | 18 |
| 1 | 33 | 1 | 3 | 18 | 18 |
| 1 | 11 | 15 | 3 | 2 | 20 |
| 2 | 12 | 24 | 3 | 3 | 19 |
| 2 | 7 | 20 | 3 | 8 | 21 |
| 2 | 9 | 19 | 3 | 12 | 17 |
| 2 | 13 | 16 | 3 | 3 | 17 |
| 2 | 8 | 16 | 3 | 24 | 23 |
| 2 | 11 | 21 | | | |

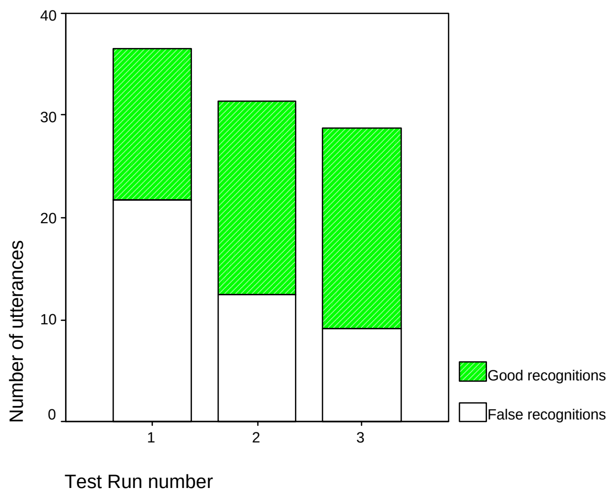


*Figure 7.4-1: Average number of (good/false) utterances needed to complete the six tasks per test run*

Figure 7.4-1 shows a graph containing the average number of utterances uttered by the test subjects in each test run. Furthermore the ratio of good and false recognitions (recognition rate) is indicated in the figure. The result is not a good measurement for the recognition quality for the

speech recognition engine, because some false recognitions occurred as a result of utterances made up of grammar that was not modelled in the grammar of the dialogue manager. This occurred more frequently in the first test run then in the others. Furthermore some false recognitions occurred as result of plain misuse of english grammar by test subjects!
The results do show that the number of utterances needed to successfully complete the tasks decreases with each test run (as the user becomes more familiar with the grammar of the speech interface). It can also be seen that the percentage of successfully recognised utterances also increases with each test run. Furthermore the added attention needs in test run 3 has no significant impact on the recognition rate.

### 7.4.2 Time to completion

*Table 28: Time needed to complete the tasks with the GUI and with the SUI by each of the test subjects per test run.*

| Test Run number | GUI time (s) | SUI Time (s) |
|---|---|---|
| 1 | 118 | 301 |
| 1 | 90 | 360 |
| 1 | 98 | 327 |
| 1 | 79 | 228 |
| 1 | 88 | 357 |
| 1 | 115 | 330 |
| 1 | 58 | 271 |
| 1 | 175 | 329 |
| 1 | 84 | 364 |
| 1 | 58 | 147 |
| 1 | 39 | 224 |
| 1 | 98 | 327 |
| 1 | 55 | 230 |
| 2 | 37 | 154 |
| 2 | 36 | 116 |
| 2 | 48 | 210 |
| 2 | 0 | 164 |
| 2 | 37 | 211 |
| 2 | 36 | 176 |
| 2 | 53 | 134 |
| 2 | 55 | 143 |
| 2 | 38 | 180 |
| 2 | 44 | 169 |
| 2 | 76 | 225 |
| 2 | 37 | 245 |
| 3 | 82 | 210 |
| 3 | 53 | 124 |
| 3 | 77 | 144 |
| 3 | 45 | 109 |
| 3 | 84 | 242 |
| 3 | 70 | 125 |
| 3 | 54 | 128 |
| 3 | 96 | 189 |
| 3 | 68 | 112 |
| 3 | 67 | 151 |
| 3 | 87 | 108 |
| 3 | 78 | 228 |
|  |  |  |

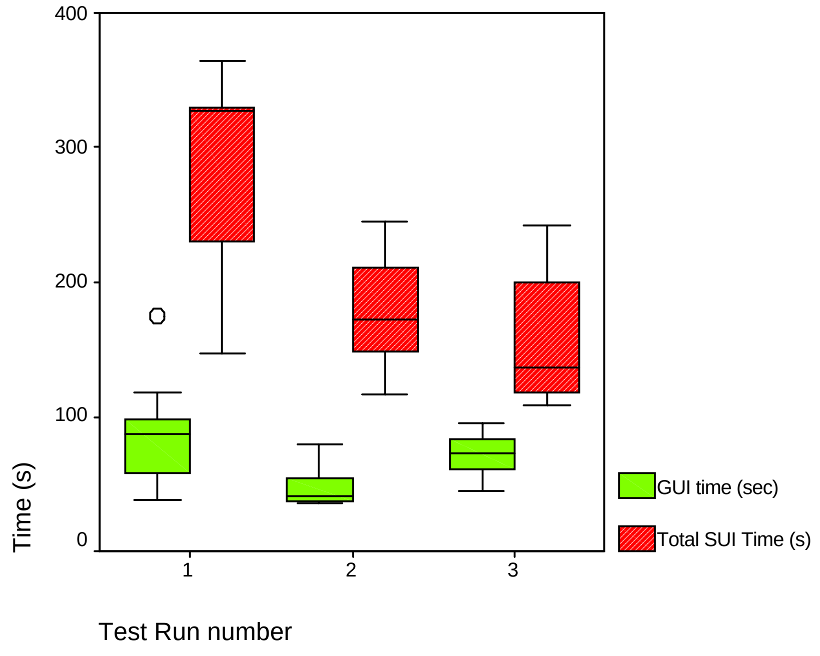


*Figure 7.4-2: Box plot of the time needed to complete the tasks with the GUI and with the per test run.*

Figure 7.4-2 shows a box plot of the time needed to complete the tasks with the GUI as well as with the SUI. Both SUI and GUI show a decrease of the time needed to complete the tasks, as the user gets more familiar with the interfaces. The decrease in the needed time is greater with the SUI then with the GUI. As can be seen, test subjects accomplish the tasks faster with the GUI by default. On the other hand, the attention needs for working with the GUI is greater than with the SUI, this is reflected by an increase in completion time in the third test run (while that of the SUI still shows a decrease).

### 7.4.3 Success rate

*Table 29: Number of test subjects to successfully complete each task per test run.*

| Test Run number | 1 | 2 | 3 |
|---|---|---|---|
| Task 1 | 11 | 11 | 12 |
| Task 2 | 10 | 12 | 12 |
| Task 3 | 5 | 10 | 11 |
| Task 4 | 10 | 12 | 12 |
| Task 5 | 1 | 12 | 12 |
| Task 6 | 10 | 12 | 12 |

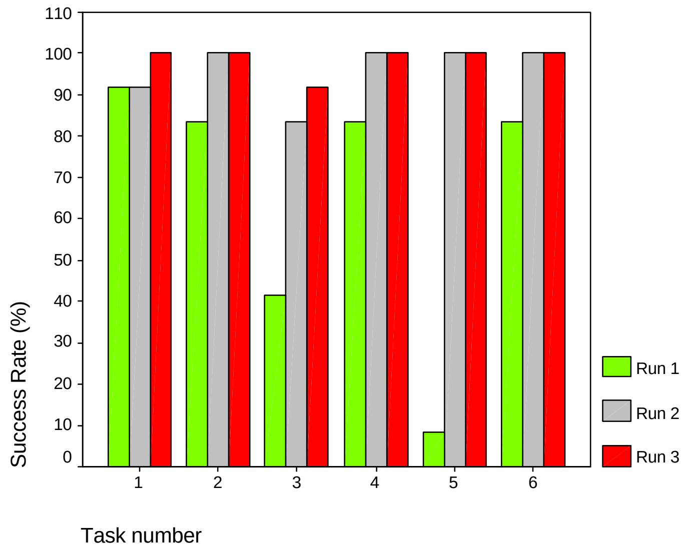


*Figure 7.4-3: Success rate per task for each test run.*

Table 29 shows the number of test subject (out of 12) to successfully complete each task per test run. The percentage version of this data (success rate) is shown graphically in Figure 7.4-3. The figure shows that the success rate increases dramatically between the first and second run. Between the second and third run the increase is much less apparent. This indicates that the speech interface is not very intuitive, but once the user gets it right once (if he successfully completes a tasks in a test run), he will have much more success with the task in the next run.

### 7.4.4 Other results

Apart from the numerical data, test subjects' comments on the user friendliness and usability of the speech interface are very important. The overall impression of the test subjects is that interface is very usable and works fine if everything goes well (if utterances are recognised, dialogues go smoothly etc.). But if something goes wrong (if an exception occurs e.g. false recognition, mis-understanding) it is very difficult and/or frustrating to correct the error.
The most heard comment on the user friendliness was the lack of good and clear feedback in exception situations e.g. the test subjects would like to know (at each moment) what the speech interface expects from them for example a list of speech options on the computer screen or a warning if the user keeps saying the wrong things.
The lack of speech options (because of a limited corpus) however, presented no problems for the test subjects. The test subjects automatically adapted to the grammar of the speech interface with each test run. This is reflected by a change in intonation and speaking style, leading to a higher recognition and success rate.
After the test, most test subjects would prefer to use the speech interface rather than the GUI in car driving interaction. The reason for this is that the SUI is much more natural and less attention intensive then the GUI. The opinions on the representativeness of first level Tetris with car driving however were divided. Most test subject stated that car driving is more attention intensive then playing Tetris.
In the eyes of the test subjects the unreliability of the recognition and the difficulty in correcting errors pose the biggest problems for the acceptance of the speech interface for the big public.

# 8 Conclusions and recommendations

## 8.1 Conclusions drawn from this work

First it can be concluded that the speech interface is capable of handling simple dialogues well and graciously. Furthermore, the use of CLIPS as knowledge processing component makes the dialogue manager very flexible. Nevertheless, no decisive conclusion can be drawn about the question if the SUI is better suited then the GUI, since the speech interface is not thoroughly tested and certain key factors (e.g. background noise) have been left out of the scope of the project. The usability test, however shows that once these technical problems are solved, the SUI is definitely more suitable then the GUI in car driving interactions.
Although the synthesised speech did not sound very well it was still understandable, the performance of the ASR engine was quite good. An explanation for this is the use of a specialised grammar that limits the words that are recognised. Combined with the flexibility of SAPI5 and its ease of use, the choice of third party speech software was a good one.

The complexity of the dialogues grows dramatically as more services are added. This is caused by the fact that the environment gets more complex as more and more environment variables have to be taken into account into the reasoning process. The user is presented with more choices, and all choices need to be handled. E.g. the user can jump from one service to another (with more services, more jumps are possible) and the interface has to take care of a graceful transition from one service to another. The use of generic dialogue flow diagrams to visualise and model dialogues has contributed greatly to the containment of the complexity. Furthermore, these dialogue flow diagrams can easily and systematically be translated into executable CLIPS rules.

The resulting grammar file and CLIPS file however, are still very big, complex and hard to read. As the files get bigger, it also becomes easier to make mistakes (for this reason the size of the corpus per service has been kept low). The problem is also increased by the fact that CLIPS has no rigid syntax checking facility. There is also no strong typing and variable names do not have to be declared beforehand. This makes it very hard to spot and correct a typing mistake. The conclusion is that it is hard to build a complex robust reasoning system, without help of a special tool. Therefore, it is my opinion that dialogue management building tools needs to be used to aid the construction of dialogue grammars and (or especially) the construction of the reasoning system. A survey of six tools providing functionality for dialogue management tasks can be found in [disc 2000].

Initially UML was used as design methodology to design the speech interface, but it did not work out as well as expected. In the end, the design was hardly used and had to be updated and modified constantly. My conclusion is that a good methodology to design and model speech dialogue systems is needed. Traditional methodologies are not sufficient: it is not suitable to represent dialogue driven systems, which are essentially dynamic and non-deterministic, with static design models.

## 8.2 Remaining work and possible improvements

This section discusses the remaining work and possible improvements on the SWAMP application. The discussion is divided into speech interface related improvements and other kinds of improvements.

Speech interface related

| **Improvement** | **Description** |
|---|---|
| Bigger corpus | A corpus of about fifteen utterances was used for each service. This is too small to construct a reliable and robust grammar. A larger corpus of utterances per service is needed. On the other hand, a larger corpus also increases the complexity of the grammar. |
| Measure attention demands | Attention demands for interactions with the speech interface are not measured. It is very important to verify that working with the SWAMP client does not cause information overload. Since one of the goals of allowing the use of speech to access to the services of SWAMP was to increase the safety of the driver it has to be tested if this is indeed the case. |
| Multiple rule recognition | One of the most annoying deficiencies of the speech interface is that it cannot recognise an utterance that conforms to a combination of grammar rules (unless explicitly specified in the grammar file). This is not caused by a flaw in the design or implementation of SWAMP, it is just not supported by SAPI5. Consequently, utterances that conform to a combination of grammar rules are also not handled by the dialogue manager. |
| Layered prompting approach | One of the most heard comments on the speech interface was the lack of good clear feedback in exception situations. The layered approach to designing dialogues is a combination of short conversational prompts with longer prompts that are more direct. E.g. when the user does not respond in a predetermined time, the system quickly presents a more directive prompt. |
| Dialogue management tool | One of the conclusions drawn from the work was that dialogue management tools are needed to:<br>- Check the consistency and completeness of dialogues<br>- Give better overview of the implementation process<br>- Automatic generation of grammars |

Other improvements

| **Improvement** | **Description** |
|---|---|
| Port to CE | The SWAMP client is developed on an NT platform and is meant to be a prototype to demonstrate speech interfacing. Special emphasis is put on the word prototype because it is far from a complete commercial product. First it has to be ported to CE. |

| | |
|---|---|
| Build a better communication method between SUI and GUI | No consideration was given on the possibility of speech interfacing during the design and implementation of the GUI. As the Speech interface had to be build to work next to and together with the GUI this resulted in difficulties in communication and synchronisation between both interfaces during implementation. Since the GUI was already there and the speech interface had yet to be build. Eventually, the synchronisation was established using the hack and crack method: identify where mis-synchronisation occur during testing and then solve problem. This is not the most elegant solution, as the source code becomes unreadable. |
| Implement new services in the Back Office | New services are implemented in the SWAMP client, but not in the SWAMP server. This has yet to be done, but shouldn't be a big problem. |
| Background noise | The problem of background noise has been left out of the scope of the project. Nevertheless it is a serious requirement of the SWAMP client to be able to function around heavy background noise, since this is quite common in its destined working environment. Without the ability to function under noisy circumstances the speech interface in SWAMP has no commercial value. |

# 9 References

## 9.1 Books

[Boullart 1992]
Boullart, L., “A gentle Introduction to Artificial Intelligence”, In Boullart. L, et al. (eds), in Boullart, L. et. al. (eds.), “Application of artificial intelligence in process control”, 1992, Pergamon Press ltd., Oxford, England p5-40.

[Chapanis 1981]
Chapanis, A. *“Interactive Human Communication: Some lessons learned from laboratory experiments”*, 1981, In: Shackel, B. (eds). “Man-Computer Interaction: Human Factor Aspects of Computers and people”, Rockville, MD: Sijthoff and Noordhoff, pp. 65-114.

[Nusbaum 1995]
Nusbaum, H. C. et al., “*Using Speech recognition systems: Issues in cognitive Engineering*”, In: Syrdal A. et al. (eds), “*Applied Speech Technology*”, 1995, Boca Raton, CRC press, pp. 127-194.

[Wooldridge 2000]
Wooldridge, M, “*Reasoning about Rational Agents*”, The MIT Press, Cambridge, Massachusetts, 2000.

[Rudnicky 1995]
Rudnicky, A. I., “*The design of spoken language interfaces*“, In: Syrdal A. et al. (eds), “*Applied speech technology*”, Boca Raton, CRC press, 1995, pp. 403-427.

[Box 1999]
Box, D., “*Essential COM*”, Addison Wesley Longman Inc., Reading Massachusetts, 1999.

[Russell 1995]
Russel, S et. al., *“Artificial Intelligence: a modern approach”*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1995.

[TCG 1998]
Test Consultancy Group, “*Testen voor I&I’ers, De theorie in praktijk*”, Testconsultancy Groep, Groningen, may 1998.

[NASA1 1993]
Software Technology Branch Lyndon B. Johnson Space Center, *“CLIPS Reference Manual Volume 1, Basic Programming Guide Version 6.0”*, Software Technology Branch Lyndon B. Johnson Space Center, June 1993.

[NASA2 1993]
Software Technology Branch Lyndon B. Johnson Space Center, *“CLIPS Reference Manual Volume2, Advanced Programming Guide Version 6.0”*, Software Technology Branch Lyndon B. Johnson Space Center, June 1993.

[Pressman 1994]
Pressman, R., *“Software Engineering: A Practitioner’s Approach”*, McGraw-Hill Companies, 1994.

## 9.2 Papers / articles/ reports

[Page 1998]
Page J.H., "*The Laureate Text-to-speech System Architecture And Applications*", In: Westall, F.A. et al (eds), *"Speech Technology for Telecommunications"*, 1998, T.J. International ltd, Padstow Cornwall, pp. 127-148.

[Nielsen 1993]
Nielsen, J., et. al., "*A mathematical model of the finding of usability problems*," *Proceedings of ACM INTERCHI'93 Conference* (Amsterdam, The Netherlands, 24-29 April 1993), pp. 206-213.

[Achterhof 2000]
Achterhof, I., "*Ids scriptie, The WAM-Pilot project*", CMG TTI, Rotterdam, 2000.

[van Egmond 1999]
van Egmond, R., "*Scriptie Wireless Automobile Messaging Pilot*", CMG TTI, Rotterdam, 1999.

[van Breda 1999]
van Breda, E., "*Scriptie, WAM-Pilot T3*", CMG TTI, Rotterdam, 1999.

[Yang1 2001]
Yang, C.K., "*Speech interfacing in the Wireless Automotive Messaging Pilot, Literature survey"*, CMG TTI, Rotterdam, 2001.

[Yang2 2001]
Yang, C.K., *"Speech interfacing in the Wireless Automotive Messaging Pilot, Design Document"*, CMG TTI, Rotterdam, 2001.

[Yang3 2001]
Yang, C.K., "*Speech interfacing in the Wireless Automotive Messaging Pilot, grammar specification*", CMG TTI, Rotterdam, 2001.

[Bell J 1992]
Bell, J., "*Pragmatic reasoning, a model-based theory*", Applied Logic Group, Computer Science Department, Queen Mary and Westfield College, University of London, London, 1992.

[Rao 1995]
Rao, A. et al., *"BDI Agents: From Theory to Practice"*, Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95), San Francisco, USA, June 1995.

## 9.3 Internet

(The links were last checked at: may 31st 2001)

[SAPI5 website]
Microsoft speech.net technologies home; http://www.microsoft.com/speech/.

[SAPI5 third party]
This page contains a list of speech development or speech related companies who have announced support for SAPI5; http://www.microsoft.com/speech/thirdparty/.

[CLIPS website]
CLIPS A Tool for Building Expert Systems; August, 1999; http://www.ghg.net/clips/CLIPS.html.

[Disc-2 website]
Spoken Language Dialogue Systems and Components: Best practice in development and evaluation; Esprit Long-Term Research; March 2000; http://www.disc2.dk.

[Design]
Design Guidelines for Voice User Interfaces;
http://www.microsoft.com/speech/technical/design.asp.

**List of tables and figure**

**Table of abbreviations**

Glossary of terms and abbreviations

| | |
|---|---|
| AI | Artificial Intelligence |
| ANWB | Algemene Nederlandse Wielrijders Bond |
| API | Application programming interface, the "top" side of the SAPI 5.0 middleware |
| ASR | Automatic Speech Recognition |
| BDI | Believe Desire Intention Model |
| CFG | Context-free grammar, specifies the rules that make valid sentences in a language |
| CLIPS | C Language Integrated Production System. An expert system tool |
| COM | Component Object Model, a technique that enables the development of reusable binary software components |
| DDI | Driver development interface |
| DTD | Document Type Definition |
| GPS | Global Positioning System. A series of 24 geosynchronous satellites that continually transmit their position. GPS is used in personal tracking, navigation, and automatic vehicle location technologies |
| GSM | Global System for Mobile communication. Digital cellular standard used throughout the world, and the primary standard in Europe and Southeast Asia |
| GUI | Graphical User Interface |
| HCI | Human Computer Interaction |
| HTTP | HyperText Transfer Protocol |
| MMS | Motor Management System |
| PCMCIA | Personal Computer Memory Card International Association |
| SAPI | Microsoft Speech Application Programming Interface. SAPI 5.0 is middleware that provides an API for applications and a DDI for speech providers |
| SMS | Short Message Service. Electronic messages on a wireless network |
| SWAMP | Speech interfacing in the Wireless Automotive Messaging Pilot |
| top-level rules | top-level grammar rules have the TOPLEVEL keyword set, indicating that they can be activated or deactivated during run-time |
| TTI | CMG Trade Transport and Industries |
| TTS | Text-to-Speech, also called speech synthesis |
| UML | Unified modelling language |
| WAMBO | WAM Back Office |
| WAM-Pilot | Wireless Automobile Messaging Pilot |
| XML | Extensible Mark-up Language. A streamlined version of Standard Generalised Mark-up Language (SGML), XML is regulated by the World Wide Web Consortium. XML can make far more advanced use of data, and create more advanced links, than HTML |

## APPENDIXES

### A1: Hardware specifications

HP Jornada

| | |
|---|---|
| **Product Number** | HP Jornada 680 (F1262A ABB) |
| Processor | 133MHz 32-bit Hitachi SH3 processor |
| Memory | 16MB SDRAM |
| Display | 6.5-in (16.7-cm) CSTN screen<br>640 x 240 x 65,536 colors on screen, 0.23mm dot pitch |
| Input | Large (76% full-size) keyboard<br>Embedded numeric keypad<br>Touch screen |
| Communications | High-performance internal modem 56Kbps, v.901<br>Internet e-mail support: POP3, IMAP4, SMTP, and LDAP<br>E-mail attachment support |
| Power | One rechargeable Lithium-Ion battery<br>One 3V CR2032 coin-cell backup battery<br>Up to 8 hours2 of battery life |
| Ports/Slots | One IrDA infrared port<br>One RS232C serial port<br>One RJ11 modem port<br>One PC Card Type II card slot (PCMCIA)<br>One CompactFlash Type I card slot |
| Sound | Audio speaker and microphone<br>Built-in voice recording |
| OS | Microsoft Windows CE Services 2.2 or ActiveSync® 3.0 |
| Size and Weight | 7.4 x 3.7 x 1.3 in (18.9 x 9.5 x 3.4 cm)<br>1.1 lbs (510 g) with battery |

Garmin GPS receiver

| | |
|---|---|
| Model | Garmin GPS 35/36 |
| Satellites | MAX 12 |
| Update rate | 1 sec |
| Acquisition time | 12 sec warm<br>45 sec cold |
| Position accuracy | 3m RMS with differential GPS<br>15m RMS Non-differential GPS (100m with selective availability on) |
| Velocity accuracy | 0.2 m/s RMS steady state (subject to selective availability) |
| Dynamics | 999 knots velocity, 69 dynamics |
| Communication | 2 x RS-232 compatible full duplex communication channels |

## A2: Evaluation results of Speech Software

| Name | Voice Xpress SDK version 4.5 |
|---|---|
| Producer | Lernhout&Hauspie |
| Type | Speaker independent, continuous |
| Supported languages | English, German, Dutch, French, Flemish |
| Supported Programming languages | Visual Basic 5.0+, Visual C++ 5.0+, Delphi 4, C++ Builder 4, Powerbuildre 6, (Java) |
| Supported platforms | Windows 95/98, Windows NT |
| Price | - |
| Compatible with MS SAPI | MS SAPI 3/4 |
| Min. system requirements | - |
| End-user's min. system requirements | Pentium 166 Mhz with MMX<br>Windows 98/95, Windows NT 4.0 with Service Pack 3 or later<br>48 MB RAM<br>Installed end-user license for Voice Xpress<br>Creative labs soundblaster 16 or compatible sound card<br>130 MB disk space<br>microphone<br>CD-ROM drive |
| Contact inf. | Email: sdk@lhsl.com<br>Website: http://www.lhsl.com/voicexpress/sdk/ |
| Comments | - |
| Grammar support | unknown |

| Name | Speech SDK 1.0 |
|---|---|
| Producer | Philips |
| Type | Continuos, speaker dependent (needs initial training) |
| Supported languages | Language and model adaptive |
| Supported Programming languages | C/C++, Visual Basic, Delphi and other environments supporting ActiveX controls. |
| Supported platforms | Windows 95/98, Windows NT |
| Price | - |
| Compatibility | Supports SAPI |
| End-user's min. system requirements | - |
| Min. System requirements | Windows95/98: Pentium II 266 MMX, 64 MB memory<br>Windows NT 4.0: Pentium II 266 MMX, 96 memory<br>Soundblaster 16 compatible soundcard supporting 16 bit recording. |
| Contact inf. | E-mail: SpeechSDK@philips.com<br>Website: http://www.speech.philips.com |
| Comments | A SDK or a trial version of it could not be obtained. So this could not be tested. |
| Grammar support | Routines to integrate new words into the user's context are provided |

| **Name** | **IBM Embedded ViaVoice, multiplatform edition** |
|---|---|
| Producer | IBM |
| Type | Speaker independent, continuous |
| Supported languages | U.S. English |
| Supported Programming languages | C |
| Supported platforms | WinCE |
| Price | - |
| Compatibility | Java-JSAPI, Not SAPI compatible |
| End-user's min. system requirements | 242-330KB DRAM<br>1.4-2.0MB ROM of flash |
| Min. System requirements | 90 MIPS required |
| Contact inf. | e-mail: VoiceClientSystems@us.ibm.com<br>Website: www.ibm.com/software/voice |
| Comments | Embedded ViaVoice is specifically developed for mobile devices. Couldn't obtain trial version |
| Grammar support | Yes (with VoiceXML) |

| **Name** | **smARTspeak CS** |
|---|---|
| Producer | ART (Advanced Recognition Technologies) |
| Type | Speaker dependent, continuous |
| Supported languages | language independent |
| Supported Programming languages | ANSI C |
| Supported platforms | Windows(98/CE/NT), EPOC 32, MagicCap |
| Price | - |
| Compatibility | - |
| End-user's min. system requirements | 2-7 MIPS |
| Min. System requirements | - |
| Contact inf. | e-mail: europesales@artcomp.com<br>Website: www.artrecognition.com |
| Comments | Features noise immunity<br>The smARTCar version is special for automotive systems |
| Grammar support | Yes |

| **Name** | **Voice Tools 6.0** |
|---|---|
| Producer | Speech Solutions, Inc |
| Type | Speaker independent |
| Supported languages | English |
| Supported Programming languages | Visual C++, Visual Basic |
| Supported platforms | Windows 9x/NT |
| Price | Evaluation |
| Compatibility | SAPI 4 |
| End-user's min. system requirements | - |
| Min. System requirements | - |
| Contact inf. | Email: getinfo@speechsolutions.com<br>Website: http://www.speechsolutions.com/ |

| | |
|---|---|
| Comments | Speech solutions is a set of 10 ActiveX components. The activeX components work quite good, and there exists a grammar and vocabulary tool to define a tailor made grammar and the words to recognise. |
| Grammar support | Yes |

| **Name** | **Sphinx 2** |
|---|---|
| Producer | Carnegie Mellon University |
| Type | Speaker-independent continuous speech recognition |
| Supported languages | English |
| Supported Programming languages | C, C++ |
| Supported platforms | Linux, FreeBSD (in Linux emulation), SunOS HP/UX, Digital Unix, Windows NT |
| Price | Freeware |
| Compatibility | - |
| End-user's min. system requirements | - |
| Min. System requirements | - |
| Contact inf. | Sphinx is now under Open source development at http://www.sourceforge.com |
| Comments | The CMU Sphinx Recognition System is a library and a set of examples and utilities for speech recognition.<br>This is an early release of a research system. The APIs and function names are likely to change, and several tools need to be made available to make this all complete.<br>Although the sphinx was originally written to run under a linux platform, it is possible to compile the program under windows. But the application could not be made to compile under Windows NT. |
| Grammar support | Yes |

| **Name** | **Dragon Naturally Speaking SDK** |
|---|---|
| Producer | Dragon Systems, Inc |
| Type | User independent |
| Supported languages | English |
| Supported Programming languages | Visual Basic, Delphi, C++ |
| Supported platforms | Windows 95, Windows 98, Windows NT (with SP3 or greater) |
| Price | 49 US$ |
| Compatibility | ActiveX Controls or COM interfaces with support for a subset of Microsoft's Speech API (4.0a) |
| End-user's min. system requirements | 64 MB Memory<br>200 MB disk space<br>CD Drive<br>Creative Labs Sound Blaster 16 or equivalent sound card supporting 16-bit recording. |
| Min. System requirements | IBM-compatible with 200 MHz Intel Pentium |

| | |
|---|---|
| | processor with MMX or equivalent.<br>64 MB Memory (96 recommended)<br>256 MB disk space<br>CD Drive<br>Creative Labs Sound Blaster 16 or equivalent sound card supporting 16-bit recording. |
| Contact inf. | Website: http://www.dragonsystems.com<br>Http://developer/dragonsys.com |
| Comments | Dragon NaturallySpeaking 3.52 or later must already be installed on the system before the SDK can be used. |
| Grammar support | Yes |

| **Name** | **VoiceAction 2.0.000** |
|---|---|
| Producer | United Research Labs |
| Type | User independent |
| Supported languages | English |
| Supported Programming languages | ActiveX supporting environment |
| Supported platforms | Windows 95/98/NT |
| Price | 199 US$ |
| Compatibility | 32 Bit Application ActiveX Control |
| End-user's min. system requirements | - |
| Min. System requirements | - |
| Contact inf. | Customer Support Site : http://www.research-lab.com<br>Website: http://www.research-lab.com/<br>Contact e-mail : urlabs@pn2.vsnl.net.in<br>Contact phone : 0091205888749<br>Contact fax : 0091205655044 |
| Comments | The ActiveX components are easily installed and registered, but using them requires registration. Furthermore there are some sample applications along with the ActiveX components, but these samples are in Visual basic. |
| Grammar support | allows users to build their own basic vocabulary database and design their own artificial intelligence language database of words |

| **Name** | **Chant® SpeechKit™ Version 2.1.2** |
|---|---|
| Producer | Chant, Inc. |
| Type | Continuous |
| Supported languages | - |
| Supported Programming languages | C/C++, Delphi, Java, Smalltalk, Visual Basic |
| Supported platforms | Win 9x/NT |
| Price | - |
| Compatibility | Supports Microsoft's Speech API (SAPI) and IBM's Speech Manager API (SMAPI). |
| End-user's min. system requirements | |

| | |
|---|---|
| Min. System requirements | Pentium 90 MHz or faster<br>Windows 95/98/NT<br>64 MB RAM<br>30 MB hard drive<br>CD-ROM<br>VGA or higher resolution monitor<br>SAPI or SMAPI compliant speech engines<br>Microphone<br>C/C++, Delphi, Java, Smalltalk, or Visual Basic development environment. |
| Contact inf. | Website: http://www.speechkit.com<br>E-mail: online@chant.net |
| Comments | Chant does not come with an own recognition engine but uses the engines already installed. Chant is not a speech recognition engine but rather a layer of software on top of the SAP/MSAPI. |
| Grammar support | Yes |

| **Name** | **IPI speech recognition developers kit** |
|---|---|
| Producer | IPI speech technologies |
| Type | Speaker dependent, command and control. |
| Supported languages | Language independent |
| Supported Programming languages | Visual C++ 4.2, C |
| Supported platforms | Windows 95/98/NT |
| Price | - |
| Compatibility | Easily portable to numerous IC's |
| End-user's min. system requirements | - |
| Min. System requirements | - |
| Contact inf. | Website: http://www.ipispeech.com<br>President: hboyette@ipispeech.com<br>Director overseas operations: Sergey Gladkov sgladkov@ipispeech.com<br>Sales enquiries: sales@ipispeech.com |
| Comments | Emails have been sent to both director of overseas operations and sales enquiries, but no replies have been received yet. |
| Grammar support | |

| **Name** | **ISIP ASR prototype system** |
|---|---|
| Producer | Institute for Signal and Information Processing Mississippi State University |
| Type | - |
| Supported languages | - |
| Supported Programming languages | C++ |
| Supported platforms | SunOS 5.7 (Solaris 2.7), cygwin unix-like interface for Windows, Linux |
| Price | Public domain |
| Compatibility | - |

| | |
|---|---|
| End-user's min. system requirements | - |
| Min. System requirements | 128 megabytes of RAM, and processors running at 50 megaHertz |
| Contact inf. | Email: help@isip.msstate.edu<br>Website: http://www.isip.msstate.edu |
| Comments | Fully functional speech recognition system that includes an acoustic front-end based on mel-frequency cepstral coefficients and their derivatives, an acoustic training module capable of Viterbi and Baum-Welch HMM training, and decision tree-based phonetic state tying. The decoder is an efficient, one-pass, lexical tree-based, hierarchical Viterbi-style decoder capable of handling cross-word triphones and N-gram language models |
| Grammar support | - |

## A3: Grammar rules for KM registration

```
<!-- *********************KM registration ************************** -->

<RULE NAME="VID_NOW">
        <L>
                <P>now</P>
                <P>from now on</P>
                <P>from here on</P>
        </L>
</RULE>

<RULE NAME="TRIPTYPE_UTTERS">
        <L>
                <P>trip</P>
                <P>drive</P>
                <P>trip type</P>
        </L>
</RULE>

<RULE ID="VID_KMREG_TRIPTYPE" NAME="VID_KMREG_TRIPTYPE" TOPLEVEL="INACTIVE">
        <L>
                <P>
                        <O>
                                <RULEREF REFID="VID_StartPolite"/>
                        </O>
                        <L>
                                <P>It is</P>
                                <P>Its</P>
                                <P>Did I mention that this is</P>
                                <P>I will be making</P>
                                <P>It will be</P>
                        </L>
                        <P>a</P>
                        <P>
                                <RULEREF NAME="VID_LOGIN_TRIPTYPE"
PROPNAME="TripType"/>
                        </P>
                        <P>
                                <RULEREF NAME="TRIPTYPE_UTTERS" PROPNAME="dontcare"/>
                        </P>
                        <O>
                                <RULEREF NAME="VID_NOW"/>
                        </O>
                </P>
                <P>
                        <O>
                                <RULEREF REFID="VID_StartPolite" PROPNAME="dontcare"/>
                        </O>
                        <L>
                                <P>the</P>
                                <P>this part of the</P>
                        </L>
                        <O>
                                <RULEREF NAME="TRIPTYPE_UTTERS" PROPNAME="dontcare"/>
                        </O>
                        <P>is</P>
                        <O>
                                <RULEREF NAME="VID_LOGIN_TRIPTYPE"
PROPNAME="TripType"/>
                        </O>
                </P>
                <P>
                        <O>
                                <RULEREF REFID="VID_StartPolite" PROPNAME="dontcare"/>
                        </O>
                        <L>
                                <P>change</P>
                                <P>set</P>
                        </L>
                        <O>the</O>
                        <O>
                                <RULEREF NAME="TRIPTYPE_UTTERS" PROPNAME="dontcare"/>
                        </O>
                        <L>
                                <P>in</P>
                                <P>into</P>
                                <P>to</P>
                        </L>
                        <P>
                                <RULEREF NAME="VID_LOGIN_TRIPTYPE"
PROPNAME="TripType"/>
                        </P>
                        <O>
                                <RULEREF REFID="VID_EndPolite" PROPNAME="dontcare"/>
                        </O>
                </P>
        </L>
</RULE>
```

## A4: SAPI5 system requirements and installation notes

Supported operating systems are:

- Microsoft Windows 2000 Professional Workstation, English edition or English edition with Japanese or Simplified Chinese Language support.
- Microsoft Windows Millennium edition.
- Microsoft Windows ® NT Workstation 4.0, service pack 6a, English, Japanese, or Simplified Chinese edition.

- Microsoft Windows 98 (Windows 95 is not supported).

Software Requirements

- Microsoft Visual C++ 6.0, service pack 3 or later version.
- Platform SDK (PSDK) April 2000 or later edition. Compiling SDK projects requires components of the PSDK. Within Microsoft Visual C++ 6.0, the PSDK include directories must be listed before the Visual C++. You can change the order in Tools->Options menu under the Directories tab. Move PSDK directories above all Visual C++ directories, if needed. To save disk space, you can load a minimal configuration. This includes enabling only the following two options:
    - Configuration Options
    - Build Environment

    These options require 13 MB on the system drive and another 80 MB on any other drive. No other options are needed. You can download the PSDK from http://msdn.microsoft.com/downloads/sdks/platform/platform.asp.
- Microsoft Internet Explorer 5.0 or later version. Users of NT4 with any version of the service packs require Microsoft Internet Explorer 5.5 or later. You need this to read the online documentation and for executing Microsoft XML. You can download the latest version of Microsoft Internet Explorer from the web at http://www.microsoft.com/ie.

Hardware Requirements

- A PentiumII\PentiumII-equivalent or later processor at 233 MHz with 128 MB is recommended.
- SAPI 5.0 can now take advantage of a machine and operating system that supports multiple processors, including all those mentioned above. Additionally, you can use SAPI 5.0 in a distributed application environment.
- Not all sound cards or sound devices are supported by SAPI 5.0, even if the operating system supports them otherwise.

The follow table outlines the RAM usage:

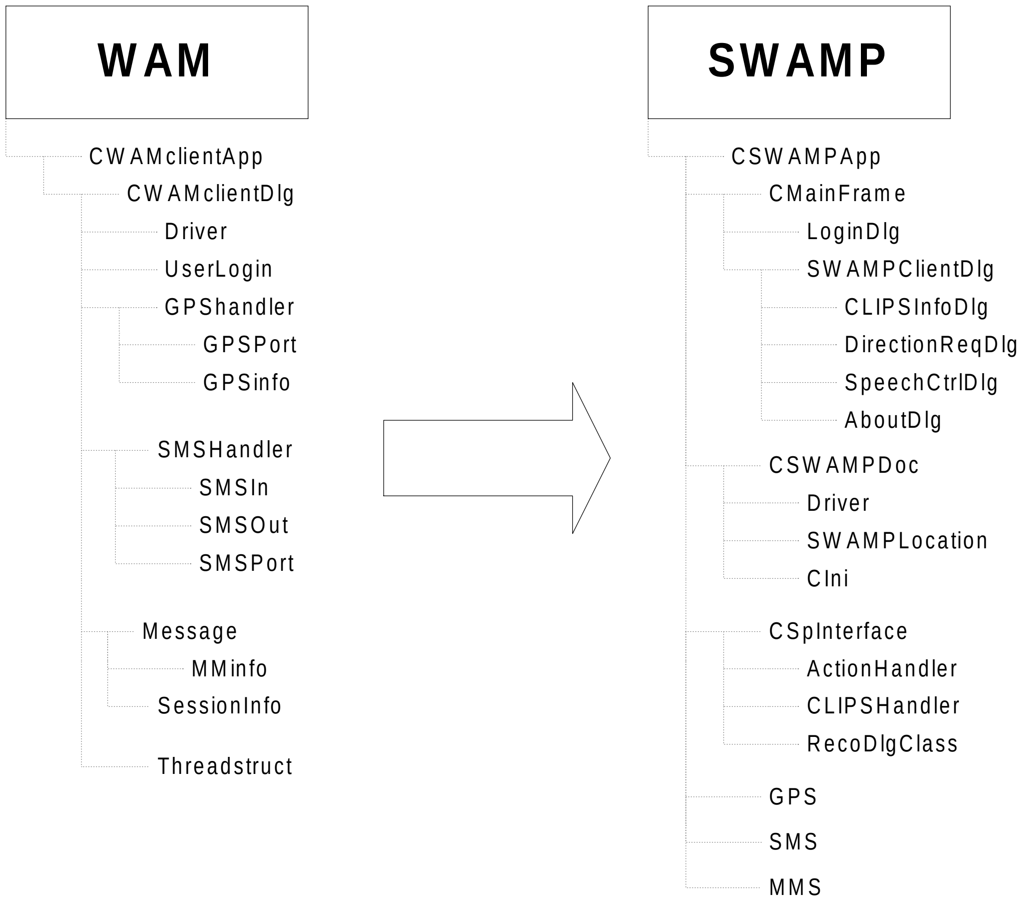
| **Component** | **Minimum RAM** | **Recommended RAM** |
|---|---|---|
| TTS Engine | 14.5 Mb | 32.0 Mb |
| SR C&C | 16 Mb | 32 Mb |
| SR Dict | 25.5 Mb | 128 Mb |
| SR Both | 26.5 Mb | 128 Mb |

The follow table outlines the disk usage:

| **File Name** | **Approximate File Size** | **Setup Merge Names** |
|---|---|---|
| Sapi.dll & Sapisvr.exe | .5Mb | Sp5.msm |
| Sapi.cpl | 36k | Sp5Intl.msm |
| SR Engine | 1.7Mb | Sp5Sr.msm |
| C&C Datafiles | 13.4Mb | Sp5CCInt.msm |
| Dictation Datafiles | 33Mb | Sp5DCInt.msm |
| TTS Engine & voices | 7.8Mb | Sp5TTInt.msm |

## A5: Class hierarchy

The following figure shows the transformation from WAM classes to SWAMP classes



## A6: Software

### Development Software

| Name | Location |
|---|---|
| Microsoft Visual Studio 6.0 Enterprise edition | MSDN, Office Test Platform & Development Tools, disc 1, March 1999 |
| Microsoft speech application development Kit 5.0 (MSAPI 5.0) | The development kit can be downloaded from the microsoft website |
| Microsoft driver development kit (ddk) for Windows 98/ME/NT/2000 | Downloaded from the microsoft website |
| Platform SDK | Downloaded from the microsoft website |
| CLIPS MFC wrapper libraries | Downloaded from the CLIPS website |
| Motor management libraries | These were included in the WAM-pilot |

**Software for Windows CE**

| Name | Location | Comments |
|---|---|---|
| Windows CE toolkit for Visual C++ 6.0 | MSDN, Office Test Platform & Development Tools, disc 14, April 1999 | During installation, at type of installation choose the "specific processor" option, In the following screen choose SH3 and SH4 processor. (Installation of x86 emulation is recommended but not mandatory.) |
| Visual C++, H/PC 2.11 | | Location: MSDN Development platform, disc 1, April 1999 Comments: install H/PC pro 2.11 in \WinCE\HPCp_sdk\ directory |

**Miscellaneous software**

| Name | |
|---|---|
| AND data | Road Data |
| Clavis Map | Maps |
| MS Acess (MDAC 2.11) | Database drivers |
| Nokia datasuite 2.0 & 3.0 | Software for the communication with the phone through the data cable. |
| Microsoft ActiveSync 3.0 | Downloading the application to the HP Jornada |
| Win CLIPS 6.0 | Testing of CLP file |
| XML Spy 3.5 | XML files editing tool, used for syntax checking of XML file |

## A7: Dialogue flow diagrams

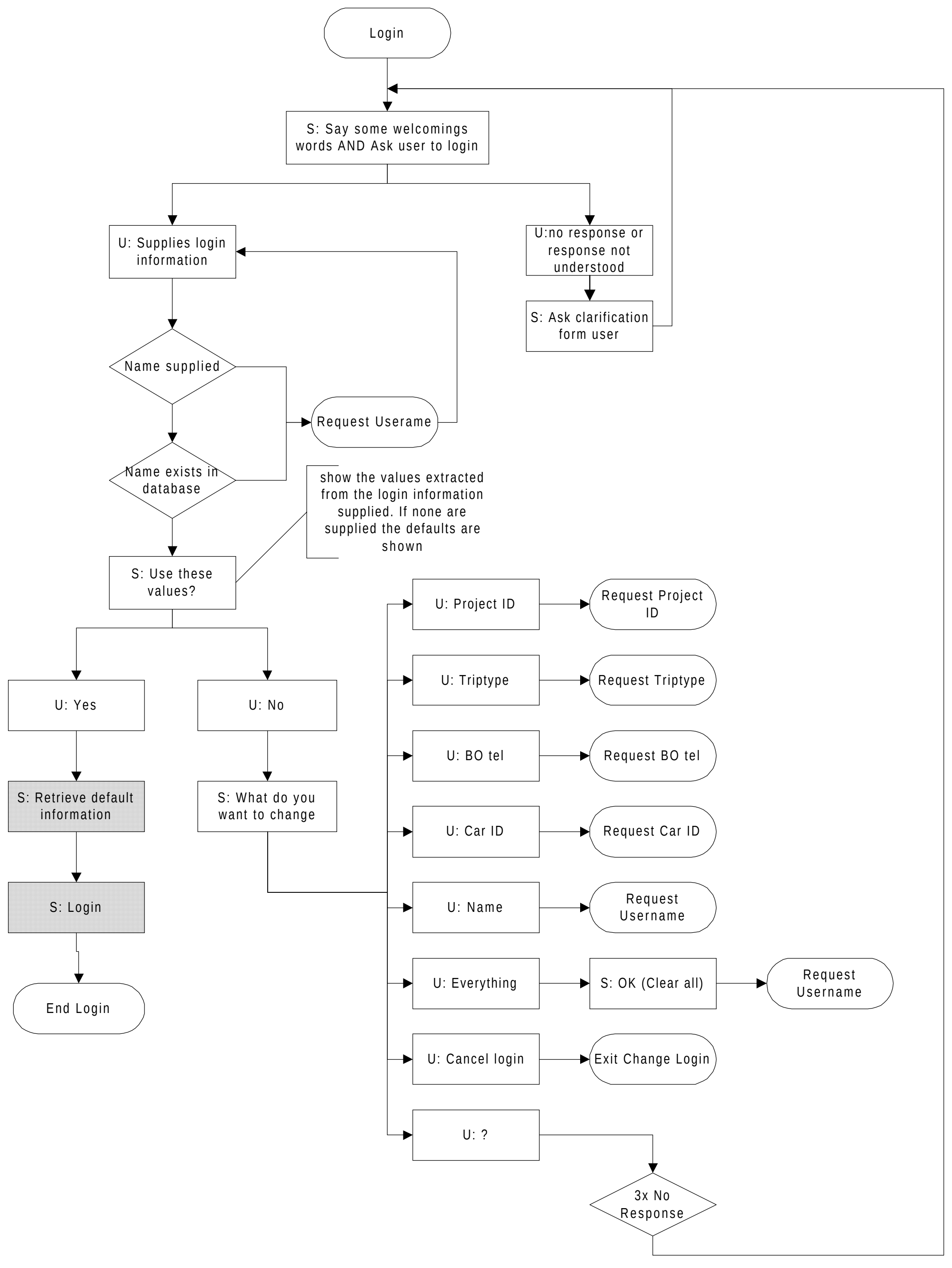


*Figure 0-1: dialogue flow diagram for main login dialogue*

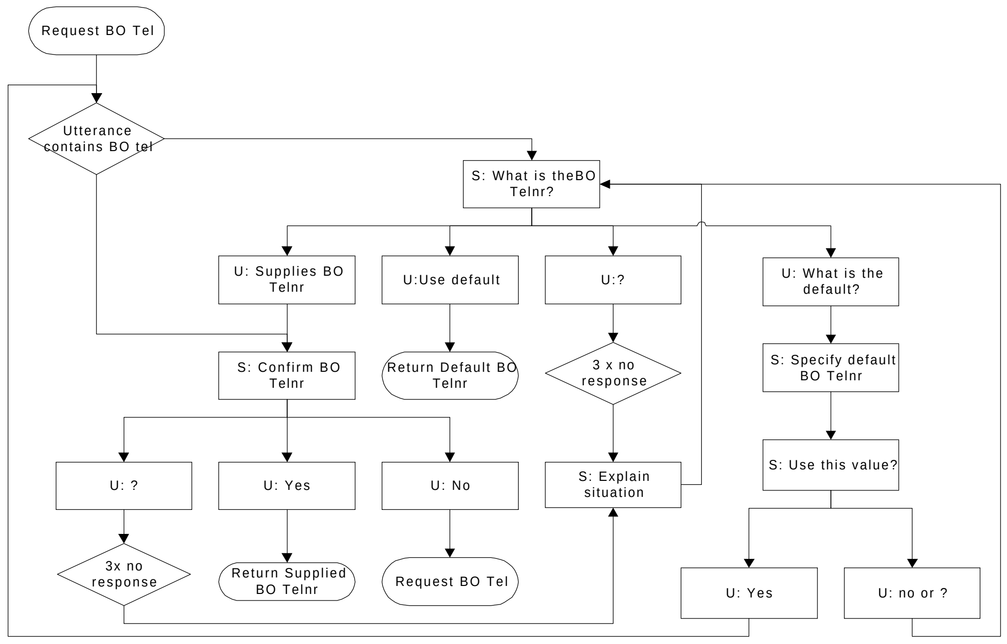


*Figure 0-2: Dialogue flow diagram for the request Back Office telephone number sub dialogue of the login dialogue*

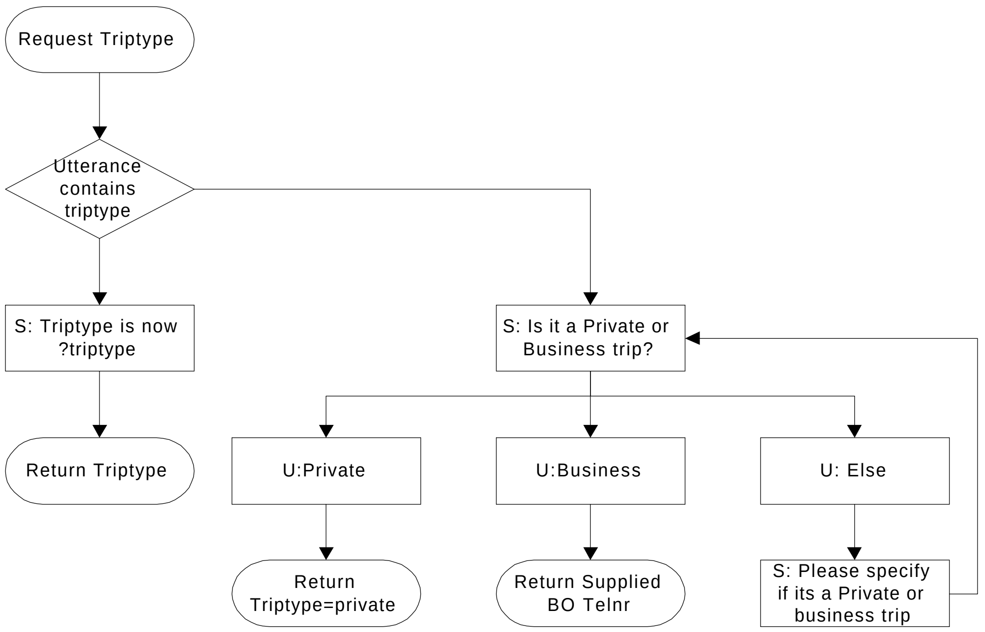

Request Triptype
Utterance contains triptype
S: Triptype is now ?triptype
S: Is it a Private or Business trip?
Return Triptype
U:Private
U:Business
U: Else
Return Triptype=private
Return Supplied BO Telnr
S: Please specify if its a Private or business trip

*Figure 0-3: Dialogue flow diagram for the request trip type sub dialogue of the login dialogue*

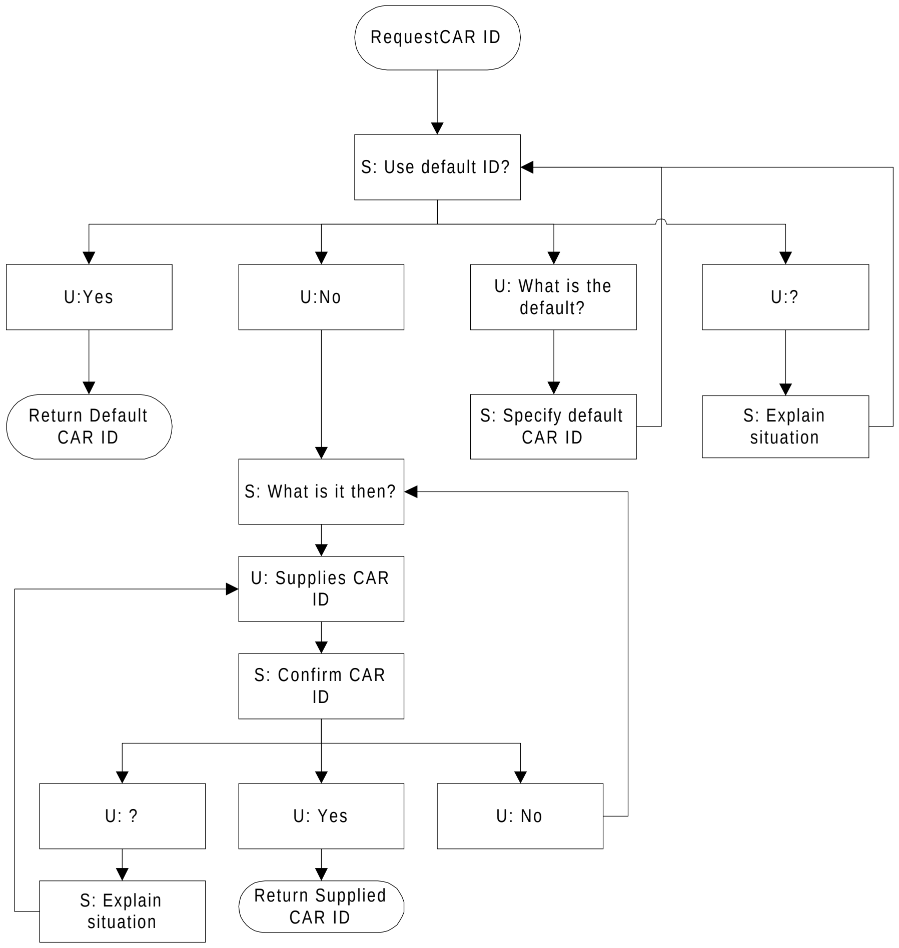


*Figure 0-4: Dialogue flow diagram for the request car ID sub dialogue of the login dialogue*

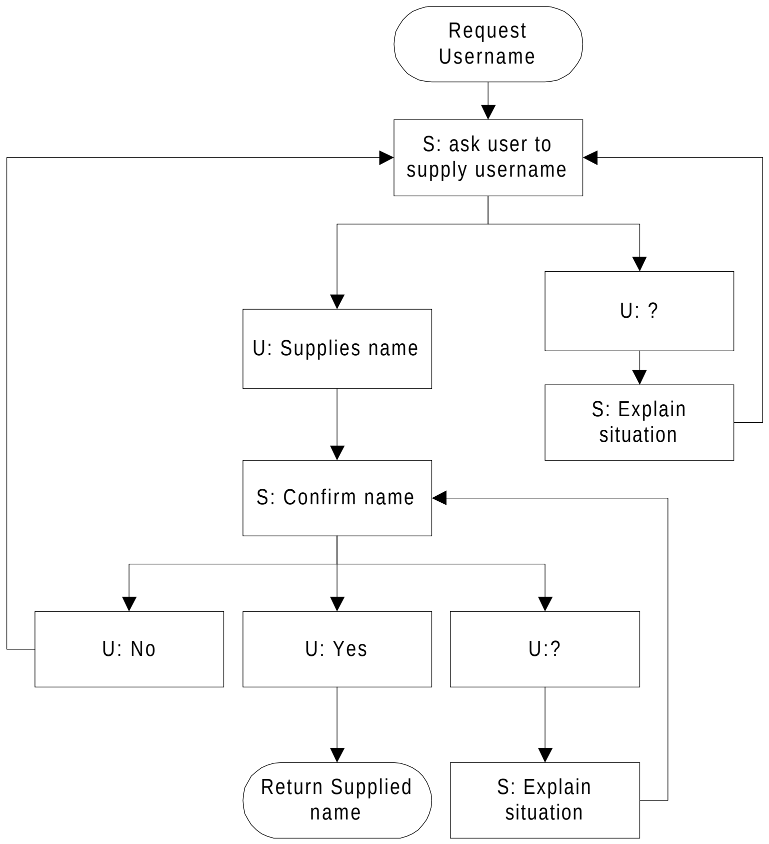


*Figure 0-5: Dialogue flow diagram for the request Username sub dialogue of the login dialogue*

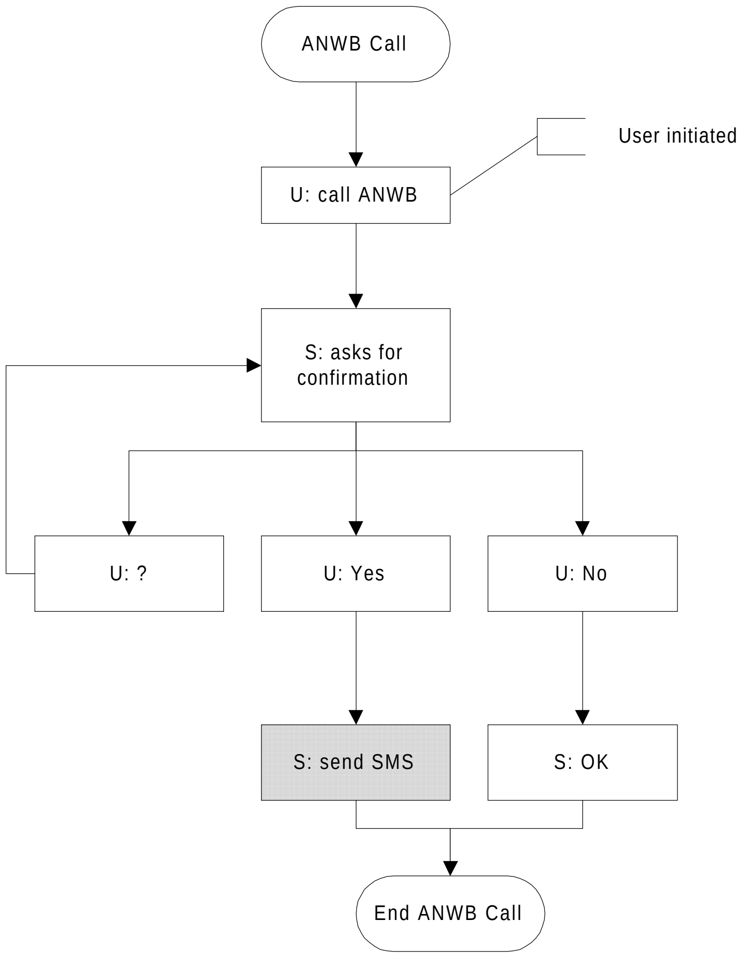


*Figure 0-6: Dialogue flow diagram for the ANWB call dialogue*

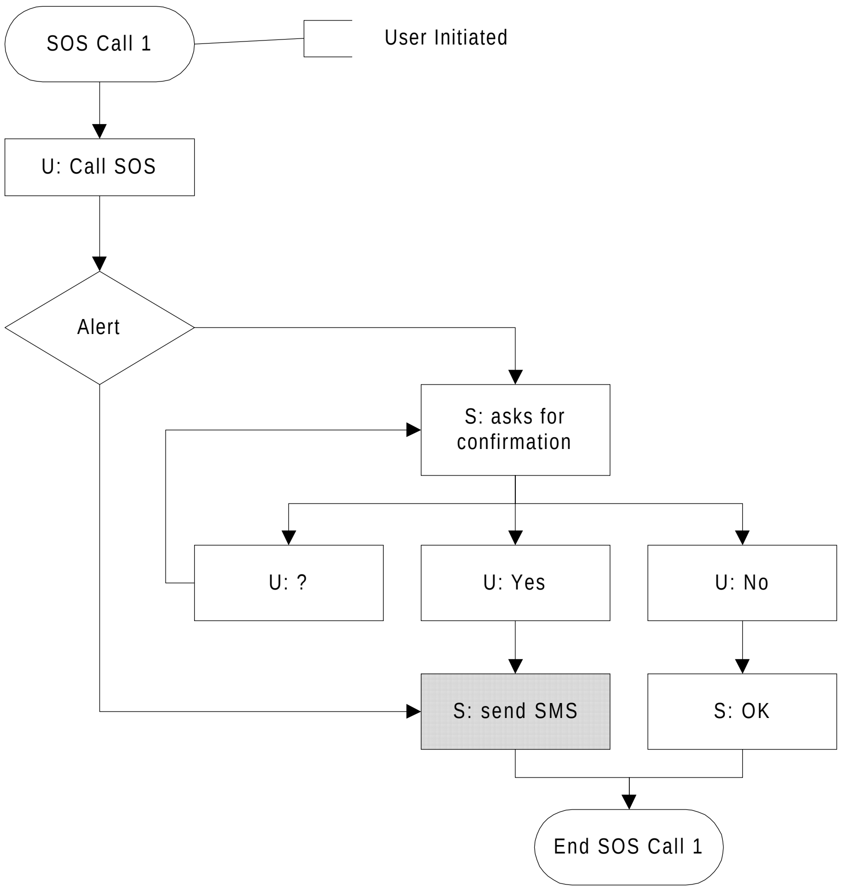


*Figure 0-7: Dialogue flow for the user initiated version of the SOS call dialogue*

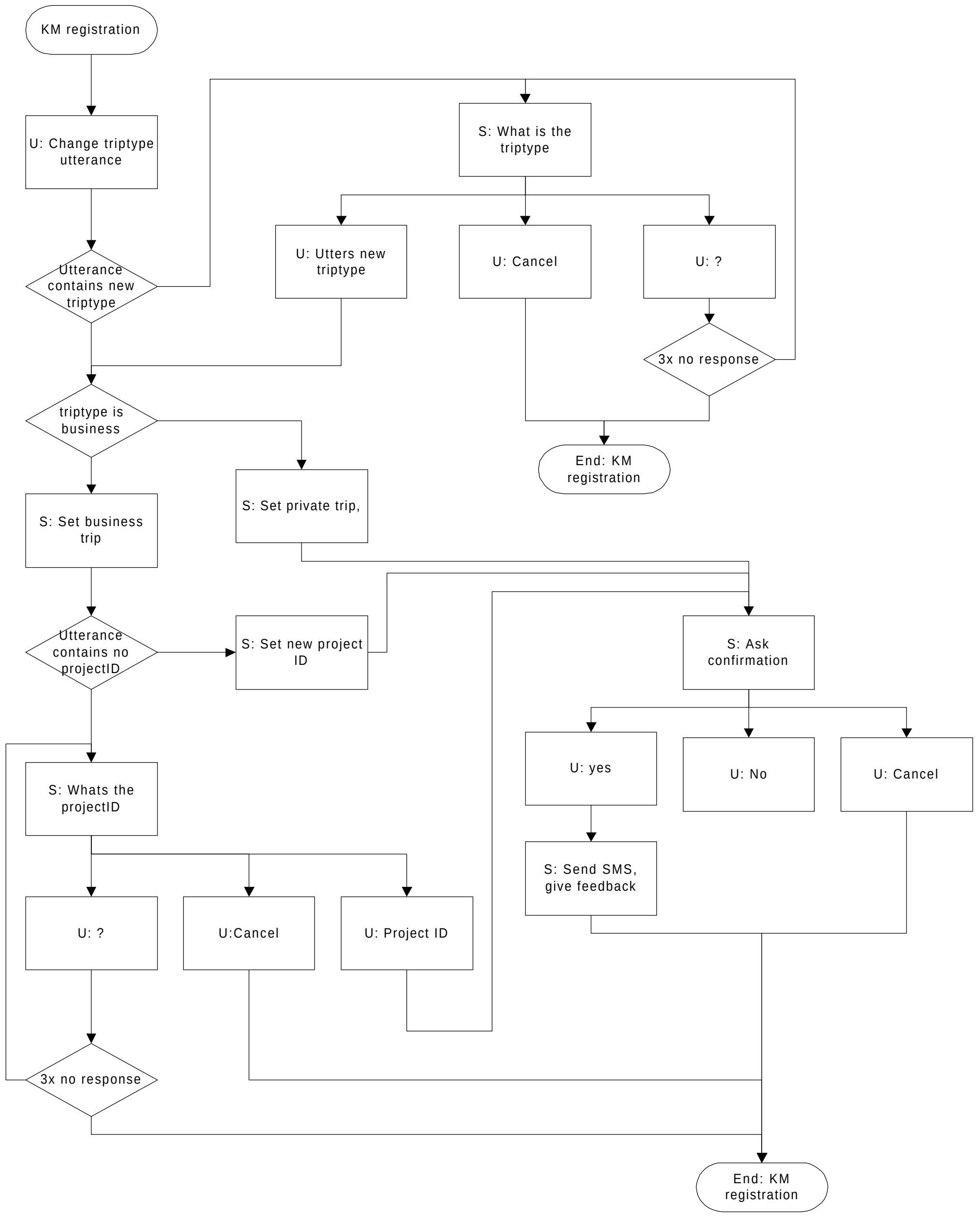


*Figure 0-8: Dialogue flow diagram for the KM registration dialogue*

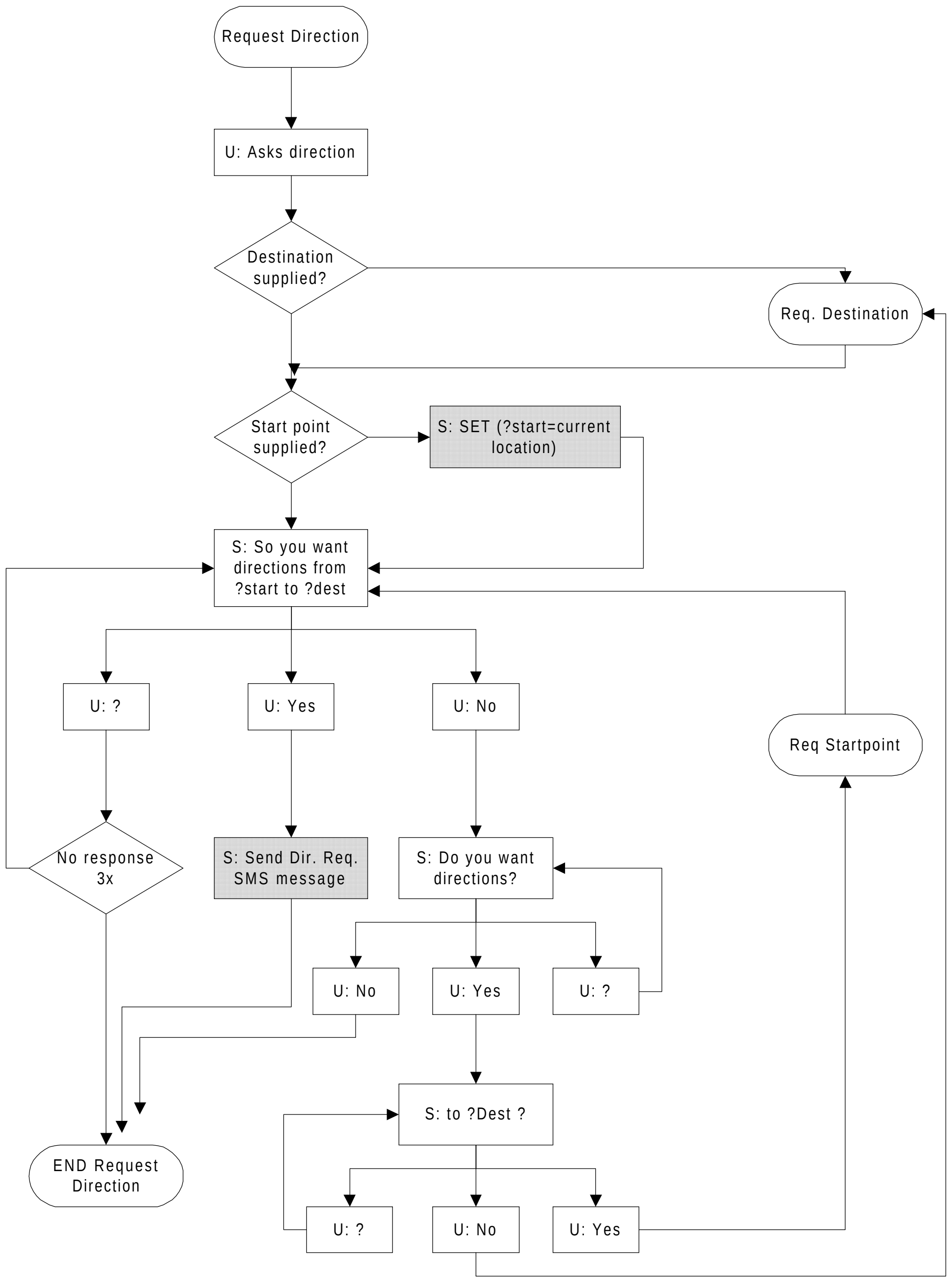


*Figure 0-9: Dialogue flow diagram for the direction request main dialogue*

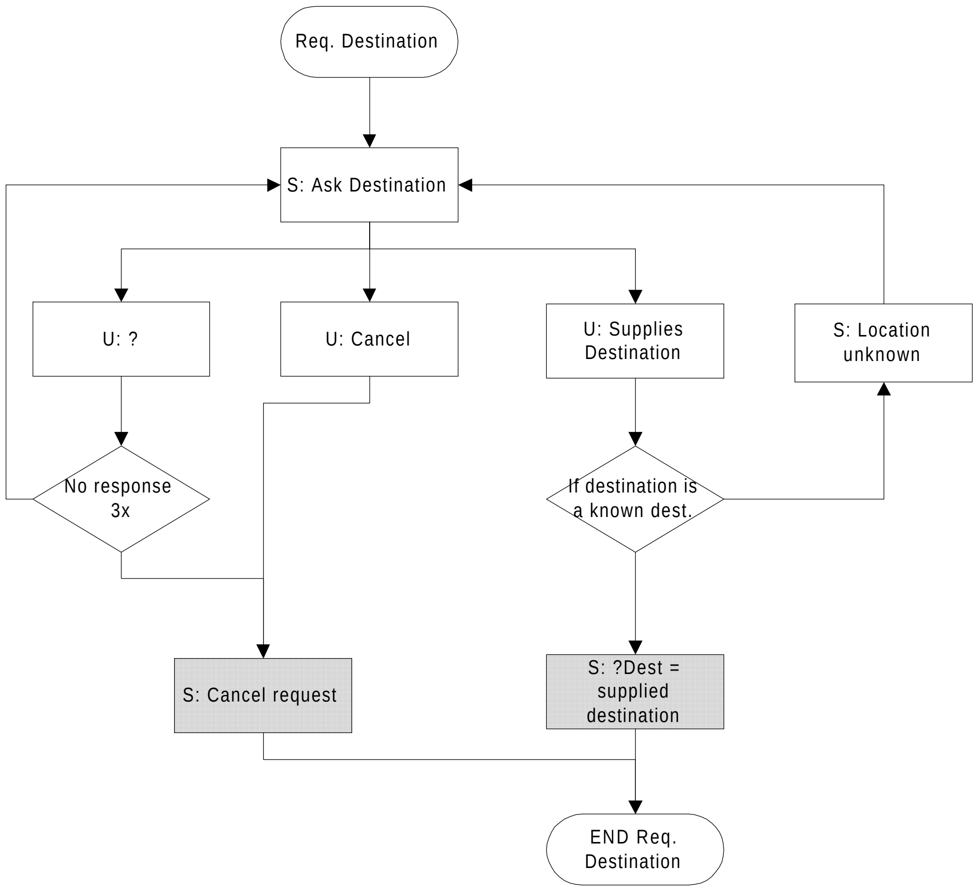


*Figure 0-10: Dialogue flow diagram for the request destination sub dialogue of the request directions dialogue*

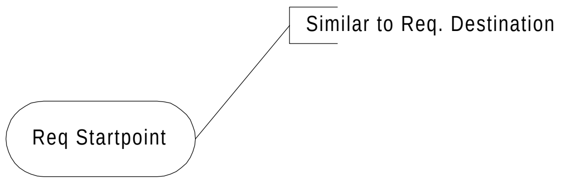


*Figure 0-11: The dialogue flow diagram for the request start point sub dialogue is similar to that of the request destination sub dialogue*

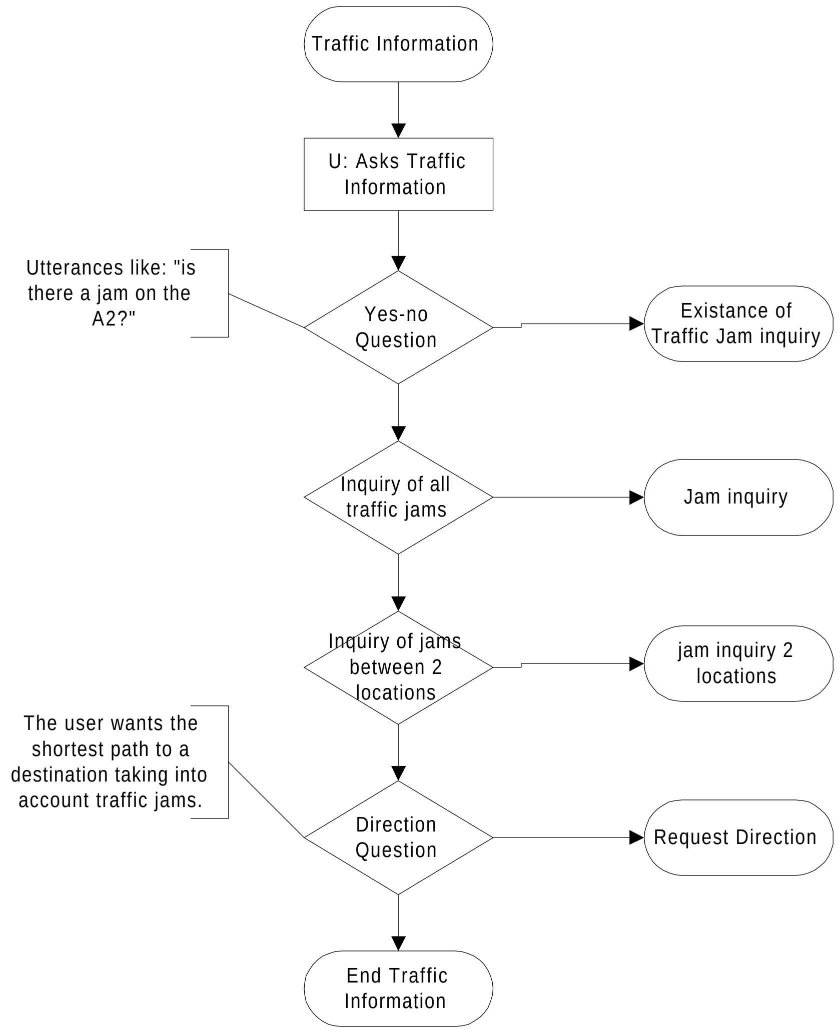

Traffic Information
U: Asks Traffic Information
Utterances like: "is there a jam on the A2?"
Yes-no Question
Existance of Traffic Jam inquiry
Inquiry of all traffic jams
Jam inquiry
Inquiry of jams between 2 locations
jam inquiry 2 locations
The user wants the shortest path to a destination taking into account traffic jams.
Direction Question
Request Direction
End Traffic Information

*Figure 0-12: Dialogue flow diagram for the request traffic information dialogue*

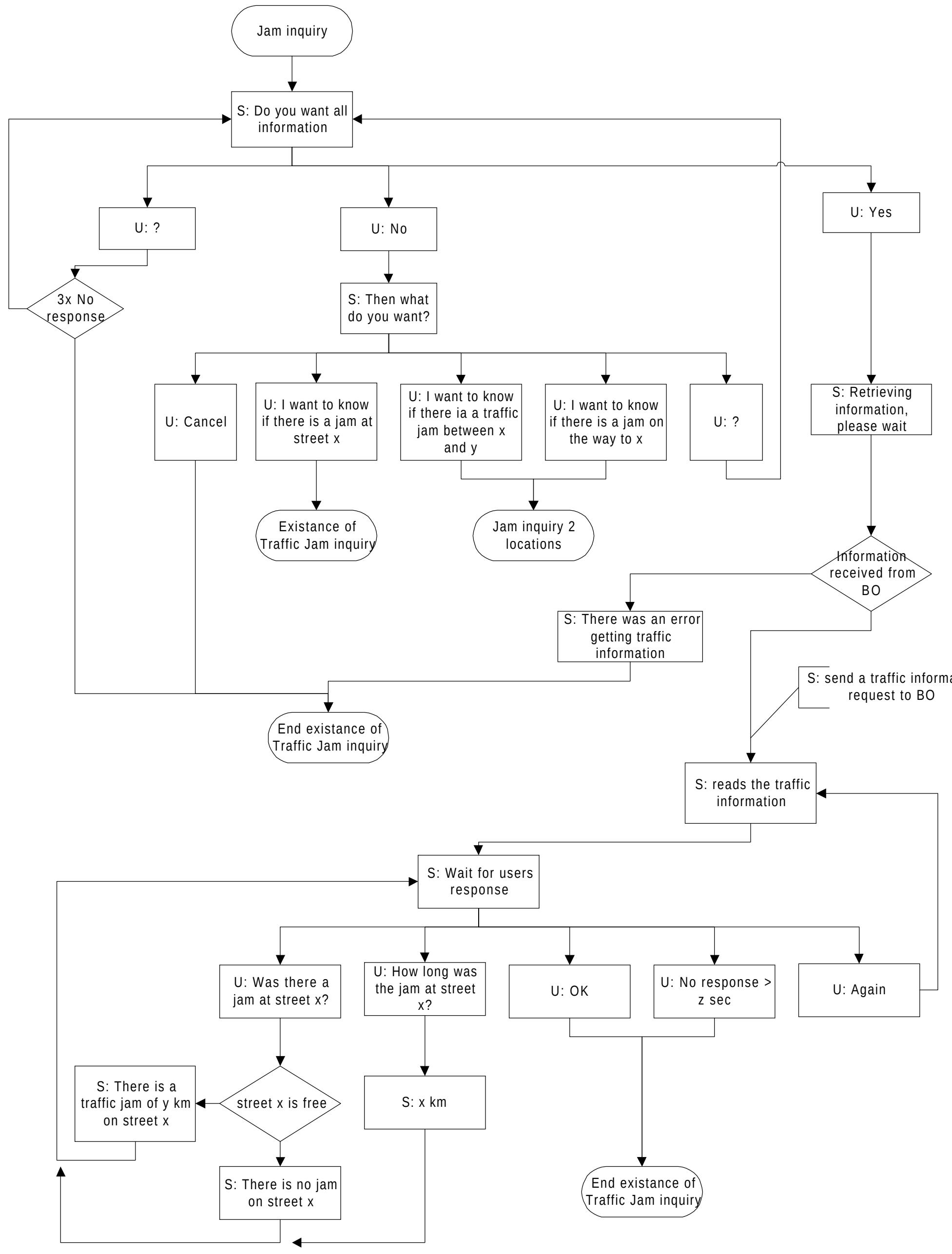

Jam inquiry
S: Do you want all information
U: ?
U: No
U: Yes
3x No response
S: Then what do you want?
U: Cancel
U: I want to know if there is a jam at street x
U: I want to know if there ia a traffic jam between x and y
U: I want to know if there is a jam on the way to x
U: ?
S: Retrieving information, please wait
Existance of Traffic Jam inquiry
Jam inquiry 2 locations
Information received from BO
S: There was an error getting traffic information
S: send a traffic information request to BO
End existance of Traffic Jam inquiry
S: reads the traffic information
S: Wait for users response
U: Was there a jam at street x?
U: How long was the jam at street x?
U: OK
U: No response > z sec
U: Again
S: There is a traffic jam of y km on street x
street x is free
S: x km
S: There is no jam on street x
End existance of Traffic Jam inquiry

*Figure 0-13: Dialogue flow diagram for the traffic jam sub dialogue*

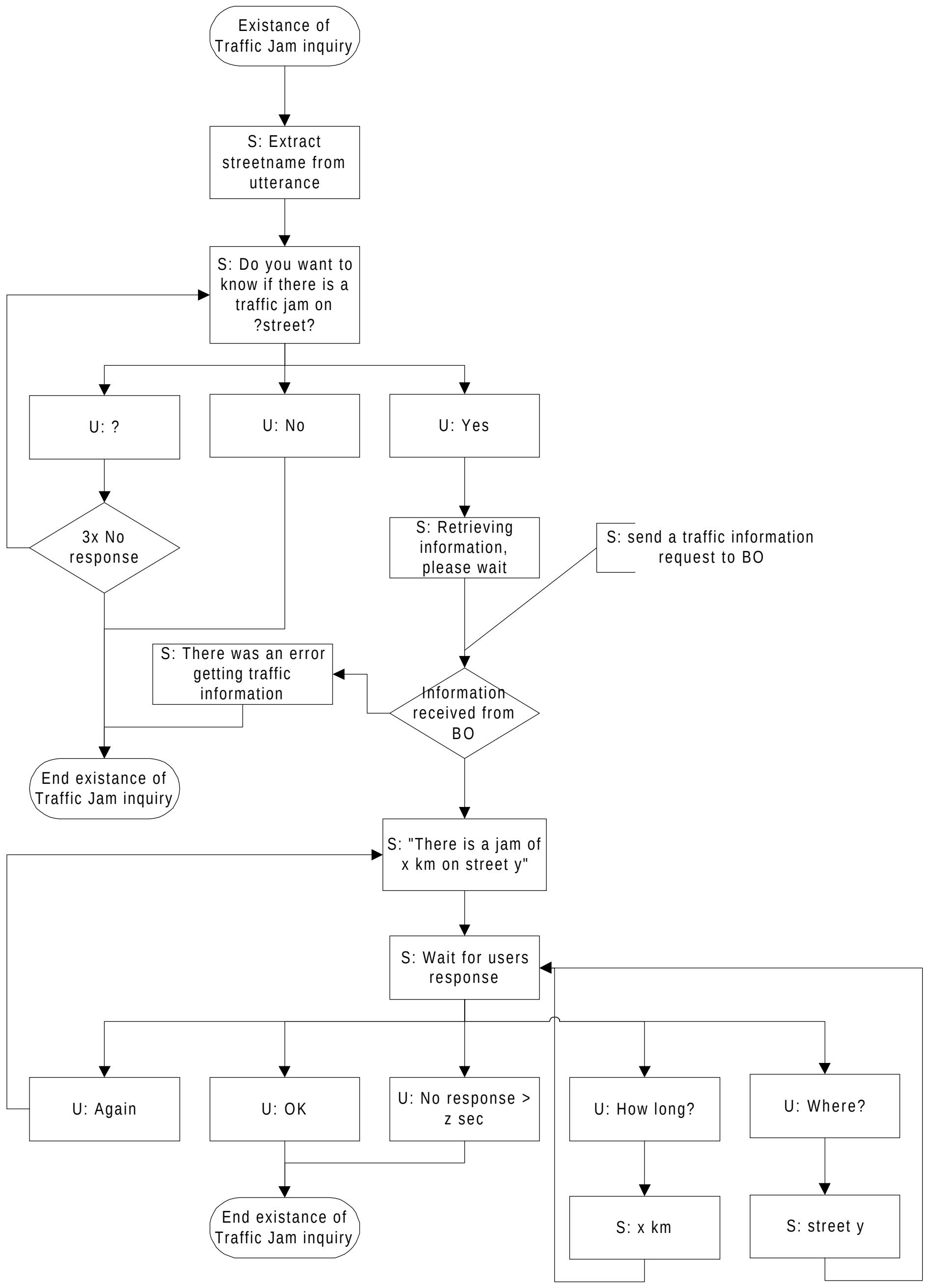

Existance of Traffic Jam inquiry
S: Extract streetname from utterance
S: Do you want to know if there is a traffic jam on ?street?
U: ?
U: No
U: Yes
3x No response
S: Retrieving information, please wait
S: send a traffic information request to BO
S: There was an error getting traffic information
Information received from BO
End existance of Traffic Jam inquiry
S: "There is a jam of x km on street y"
S: Wait for users response
U: Again
U: OK
U: No response > z sec
U: How long?
U: Where?
End existance of Traffic Jam inquiry
S: x km
S: street y

*Figure 0-14: Dialogue flow diagram for existence of traffic jam sub dialogue*

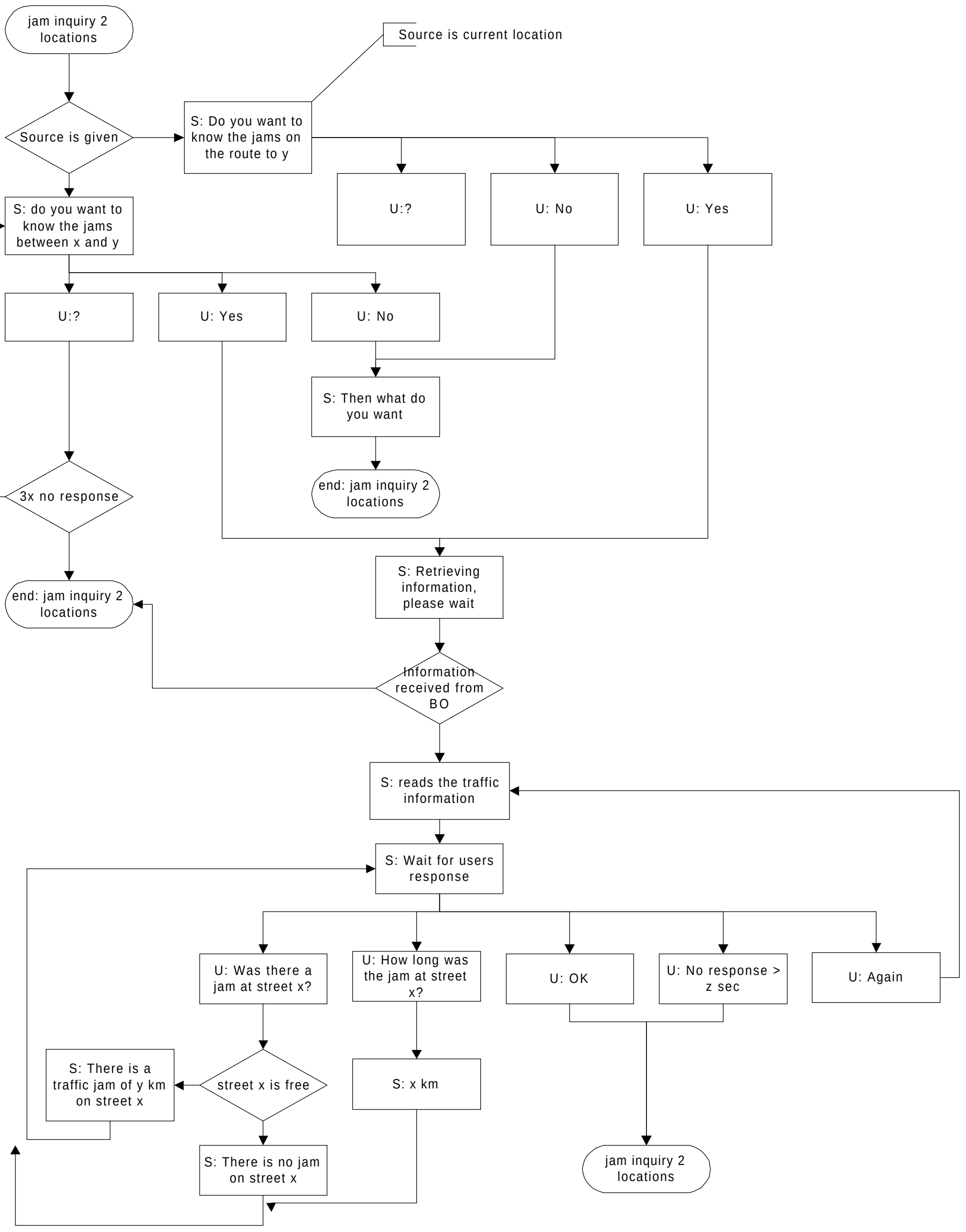

jam inquiry 2 locations
Source is current location
Source is given
S: Do you want to know the jams on the route to y
U:?
U: No
U: Yes
S: do you want to know the jams between x and y
U:?
U: Yes
U: No
S: Then what do you want
end: jam inquiry 2 locations
3x no response
end: jam inquiry 2 locations
S: Retrieving information, please wait
Information received from BO
S: reads the traffic information
S: Wait for users response
U: Was there a jam at street x?
U: How long was the jam at street x?
U: OK
U: No response > z sec
U: Again
S: There is a traffic jam of y km on street x
street x is free
S: x km
S: There is no jam on street x
jam inquiry 2 locations

*Figure 0-15: Dialogue flow diagram for traffic information on a route sub dialogue*

## A8: Paper