A LANGUAGE FOR INTERACTIVE SPEECH DIALOG SPECIFICATION

by

Ira Scharf

S.B., Massachusetts Institute of Technology
(1989)

SUBMITTED TO THE DEPARTMENT OF
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

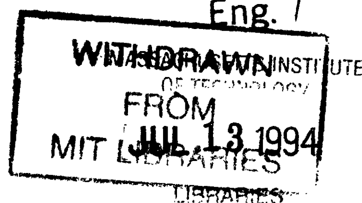
© Ira Scharf, 1994
All Rights Reserved.
The author hereby grants to M.I.T. permission to reproduce and
to distribute copies of this thesis document in whole or in part.

Signature of Author ______________________________
Department of Electrical Engineering and Computer Science
May 12, 1994

Certified by ______________________________
Christopher M. Schmandt
Thesis Supervisor

Certified by ______________________________
J. Robin Rohlicek
Company Thesis Supervisor (BBN)

Accepted by ______________________________
F. R. Morgenthaler
Chairman, Department Committee on Graduate Students



A Language for Interactive Speech Dialog Specification

by

Ira Scharf

Submitted to the Department of Electrical Engineering and Computer Science on May 12, 1994 in partial fulfillment of the requirements for the degree of Master of Science

Abstract

Developing a complex spoken dialog application for speech recognition systems using currently available tools requires careful coordination of three separate components: the recognition grammar, the interpreter for the recognition output, and the dialog control logic. Each of these components is an integral part of an interactive speech application; however specifying each one separately makes building speech applications cumbersome, time consuming, and error prone. This thesis describes a language which allows the specification of dialog structure and interpretation in one concise manner. As part of this thesis a compiler for the language has been developed which generates a recognition grammar and application code to control the dialog flow and recognition parsing from this common specification.

Thesis Supervisor: Christopher M. Schmandt
Title: Principal Research Scientist, MIT Media Laboratory

Thesis Supervisor: J. Robin Rohlicek
Title: Division Scientist, Bolt Beranek and Newman Inc.

Acknowledgments

Many people have contributed their efforts and ideas to this work, and I would not consider it complete without gratefully acknowledging their contributions.

I would first like to thank Robin Rohlicek, my thesis advisor at BBN, for his guidance and encouragement. Over the course of this project I have benefited tremendously from his knowledge, clear thinking, and pleasant nature.

I would also like to thank Chris Schmandt, my thesis advisor at MIT, for all his help and support.

Many of my colleagues in the BBN Speech Group provided a great deal of help along the way. In particular, I would like to thank Will Sadkin for his advice on numerous implementation issues; Larry Denenberg for some very helpful design suggestions; and Bruce Papazian for his many ideas about what a dialog specification language should look like. Others in the group who provided many helpful suggestions include Mike Vande Weghe, Dan Ellard, Sue Hamilton, and Kristin Kupres.

I would like to thank Bolt Beranek and Newman Inc for funding this research, and particularly John Makhoul and Mike Krasner for sponsoring this program from the onset. This work was conducted at the Speech Products Group of BBN Laboratories, in association with the MIT Media Laboratory.

Finally, I would like to thank my family, for helping me reach this point.

To my parents

Contents

Abstract **2**

Acknowledgements **4**

1 Introduction **9**

1.1 Definition of Problem 9

1.2 Today's Solution 9

1.2.1 Deficiencies of Current Practice 10

1.3 Overview of New Approach 12

1.4 Outline of the Thesis 14

2 Background **16**

2.1 IVR Systems 16

2.1.1 Application Generators for IVR Systems 17

2.2 Speech Recognition Systems 18

2.2.1 Capabilities of the HARK Speech Recognition System 19

2.3 Integrating Speech and IVR Systems 22

2.3.1 Limitations of Extending Current IVR Application Generators . . 24

3 Functional Description **26**

3.1 State Machine Model 26

3.2 Dialog State Inputs 27

3.2.1 Speech Inputs 27

3.2.2 Dynamic Control of Valid Speech Inputs 29

3.2.3 Application Event Inputs 32

3.3 Dialog State Transitions 34
3.3.1 Application Control of Dialog State Transitions 34
3.4 Handling of User Feedback Within the Dialog 35
3.5 Assigning Actions and Prompts to State Transitions 36
3.5.1 Conditional Actions and Prompts 37
3.6 Dynamic Expansion of Non-Terminals 38
3.7 Global State 40

4 Implementation **42**
4.1 Dialog Compiler 42
4.1.1 Language Parser 42
4.1.2 Code Generation 43
4.2 Dialog Manager 49
4.3 Speech Application 50
4.3.1 Activating the Current Dialog State 50
4.3.2 Registering Speech Input 50
4.3.3 Registering Application Event Inputs 51
4.4 User Supplied Library 52

5 Using the Language **53**
5.1 Dialog Definition 53
5.2 State Definition 54
5.3 Prompts 56
5.4 Speech Input 58
5.4.1 Dynamic Activation of Speech Inputs 60
5.5 Application Events 62
5.5.1 Special Reserved Event Labels 62
5.6 Action Functions 65
5.6.1 Conditional Actions 66
5.7 Next State 68
5.7.1 Conditional Next 69
5.8 Summary of Input Components 70

6 Sample Application: A Voice Dialer **71**
6.1 Description of the Application 71
6.2 Designing the Dialog 72
6.3 Implementing the Dialog Specification 75

7 Future Directions **81**
7.1 Graphical User Interface 81
7.2 Interactive Development Environment 82
7.2.1 Dialog Simulation 82
7.2.2 Prompt Manager 82
7.3 Parameterized States 83
7.4 Talk Ahead 84
7.5 Integrating Speech into a Graphical User Interface 87

A Voice Dialer Application Source **90**

B BNF Grammar for the Dialog Specification Language **109**

List of Figures

1.1 Existing Development Methods for Spoken Dialog Systems 11
1.2 Overview of the Development Environment Using the Dialog Specification Language 13

2.1 Example IVR State Diagram 17
2.2 Example Grammar Structure 20
2.3 Grammar Structure With Labeled Arcs 21
2.4 Grammar Structure With Defined Regions 21
2.5 Grammar With Rejection 23

3.1 Single Dialog State 27
3.2 Application Control Flow 28

4.1 Top Level Structure of the Generated Grammar 44
4.2 Single State Subgrammar 45
4.3 Dialog State Table 46
4.4 Compiler Generated Output Files 48

5.1 Dialog Definition Command 54
5.2 State Definition Command 55
5.3 Dialog State Showing Transition Prompts 57

6.1 Simple Two State Dialog 73
6.2 Four State Dialog 74

7.1 Banking IVR Application 85
7.2 Grammar with Talk Ahead Enabled 86

Chapter 1

Introduction

1.1 Definition of Problem

Developing a complex spoken dialog application using today's tools requires coordination of three separate components: the recognition grammar, the interpreter for the recognition output, and the dialog logic control. Each of these components is an integral part of an interactive speech application; however specifying each one separately makes building speech applications cumbersome, time consuming, and error prone. An aid to development of a spoken dialog application, an application generator, is the focus of this thesis. Central to this application generator is a specification language which allows the application developer to specify all three components of the application in a single place using a common syntax. This dialog specification language provides a dramatically more efficient method for rapidly developing complex spoken dialog applications, and alleviates many of the inconsistencies prevalent using today's tools.

1.2 Today's Solution

Many speech recognition systems can be configured to accept a restricted syntax, specified in some form of grammar. The grammar improves recognition accuracy by not allowing the recognizer to consider syntactically invalid hypotheses. Once the grammar is specified it is also the responsibility of the application developer to build an application which

can effectively parse the outputs from the recognizer and execute appropriate actions, including changes in the state of the dialog, based on what the user spoke.

The paradigm of specifying the grammar and the actions separately is adequate only for the simplest of speech applications. More complex applications, however, require intricate logic to control an ongoing dialog with the user. For example, an interactive user dialog might require different parts of the grammar to be enabled at different times in the dialog, depending on what information the application is expecting from the user. The application must keep track of the state of the dialog in order to interpret the recognition output correctly.

Given the complexity of such spoken dialog applications, the task of implementation is often divided among several different developers or teams. A human factors expert, for instance, may be more qualified to define the recognition grammar, while other segments of the application relating to controlling the dialog, communicating with the recognizer, and interpreting the recognition output, are usually built by a team skilled in software engineering. The overhead involved in integrating these segments, possibly developed by different groups, can add to the complexity of the system.

1.2.1 Deficiencies of Current Practice

Using current techniques the dialog logic is spread throughout the application. The grammar file defines only the structure of the sentences to be recognized by the application. Within the application the developer writes a parser to interpret the recognized word string, a dispatcher to execute the appropriate actions for each interpreted recognition, and a state machine to control the flow of the dialog. The implementation of each segment is naturally dependent upon the implementation of each of the other segments. Figure 1.1 shows a typical software development environment for spoken dialog systems using the current development model.

The main problem with the existing model is that the logic of the dialog is now spread between these three separate segments, allowing room for inconsistencies to develop as the syntax or dialog flow are modified. When changes are made it is critical that all areas of the application maintain consistency with each other. If, for example, the developer adds a new word sequence to the grammar, it may be necessary to update the application

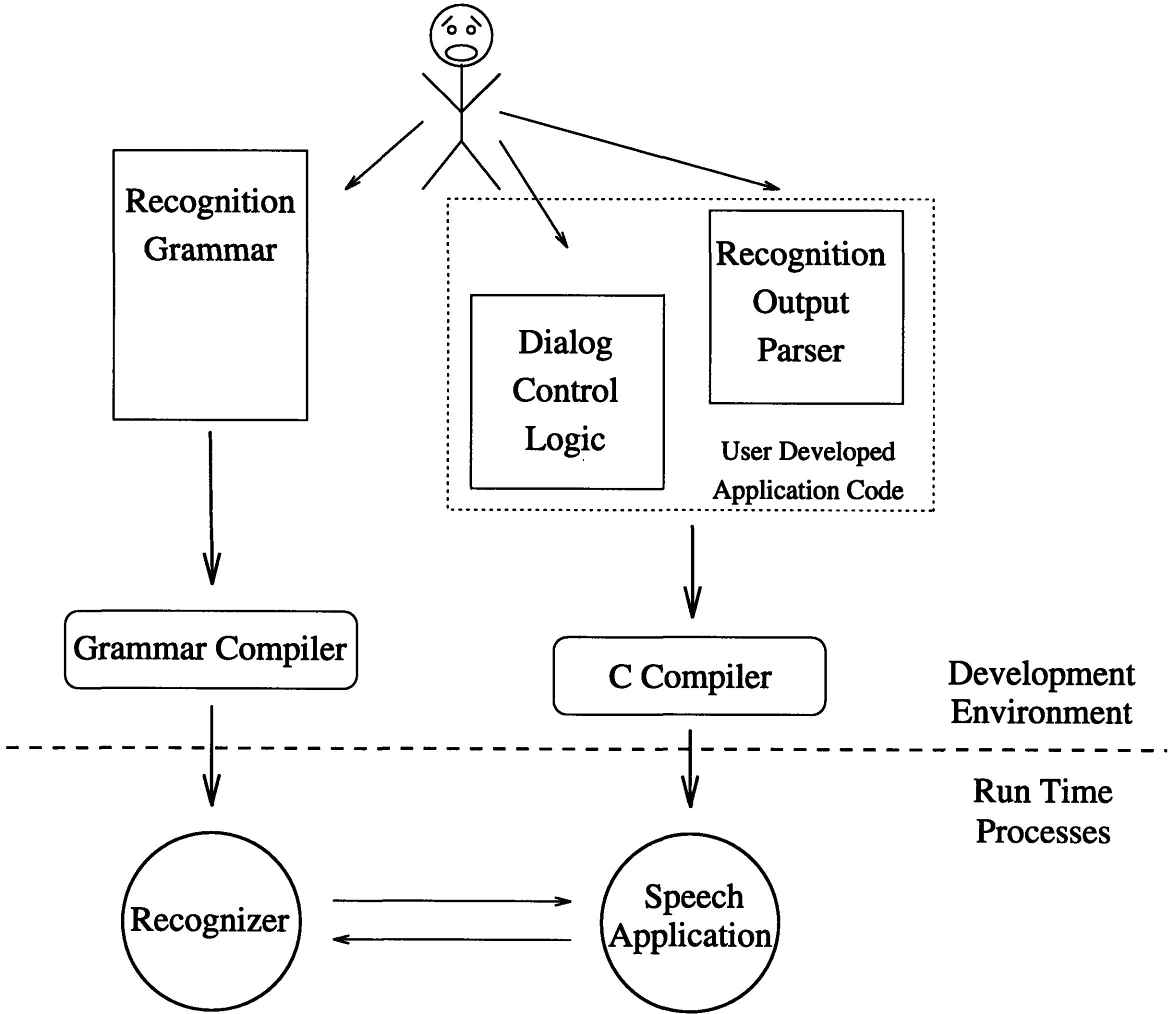


Figure 1.1: Existing Development Methods for Spoken Dialog Systems

code as well to reflect the change. Moreover, there is no single place to look for a concise representation of the dialog flow, or to determine which actions are executed for each valid phrase in the grammar.

Using such a development environment spoken dialog applications are difficult to design, implement, and modify. Each change to the system must be carefully considered for its potential impact on other segments of the application. Iterations of dialog designs to improve the human-computer interface are best accomplished using a dialog simulator, rather than the actual system, because of the cumbersome methods for modifying the flow of the dialog in the application; usually only the final dialog is implemented in the application. In short, the current development process is tedious, time consuming, and prone to errors resulting from inconsistencies within the separate modules of the application.

1.3 Overview of New Approach

This thesis describes a high-level dialog specification language which allows a speech application builder to specify in one place the structure and logic of the dialog, the allowable or interpretable spoken utterances, and the specific actions to take for each utterance. From this single specification a speech grammar file, a recognition output parser, and a dialog tracker are generated automatically using a compiler that has been built as part of this thesis project.

The design goal for the specification language is to define dialog information and interpretation in one concise, efficient manner. The entire dialog can generally be expressed in one file. Using the compiler for this language the generated grammar, parser, and dispatcher are guaranteed to be consistent since they are all derived from the same source. This common specification allows the dialog to be treated as one consistent unit.

Figure 1.2 shows the development environment using the application generation system described in this thesis. Using the dialog specification language this integrated environment alleviates the need for the developer to maintain pieces of the dialog across various segments of the application. The developer specifies the dialog flow, prompts,

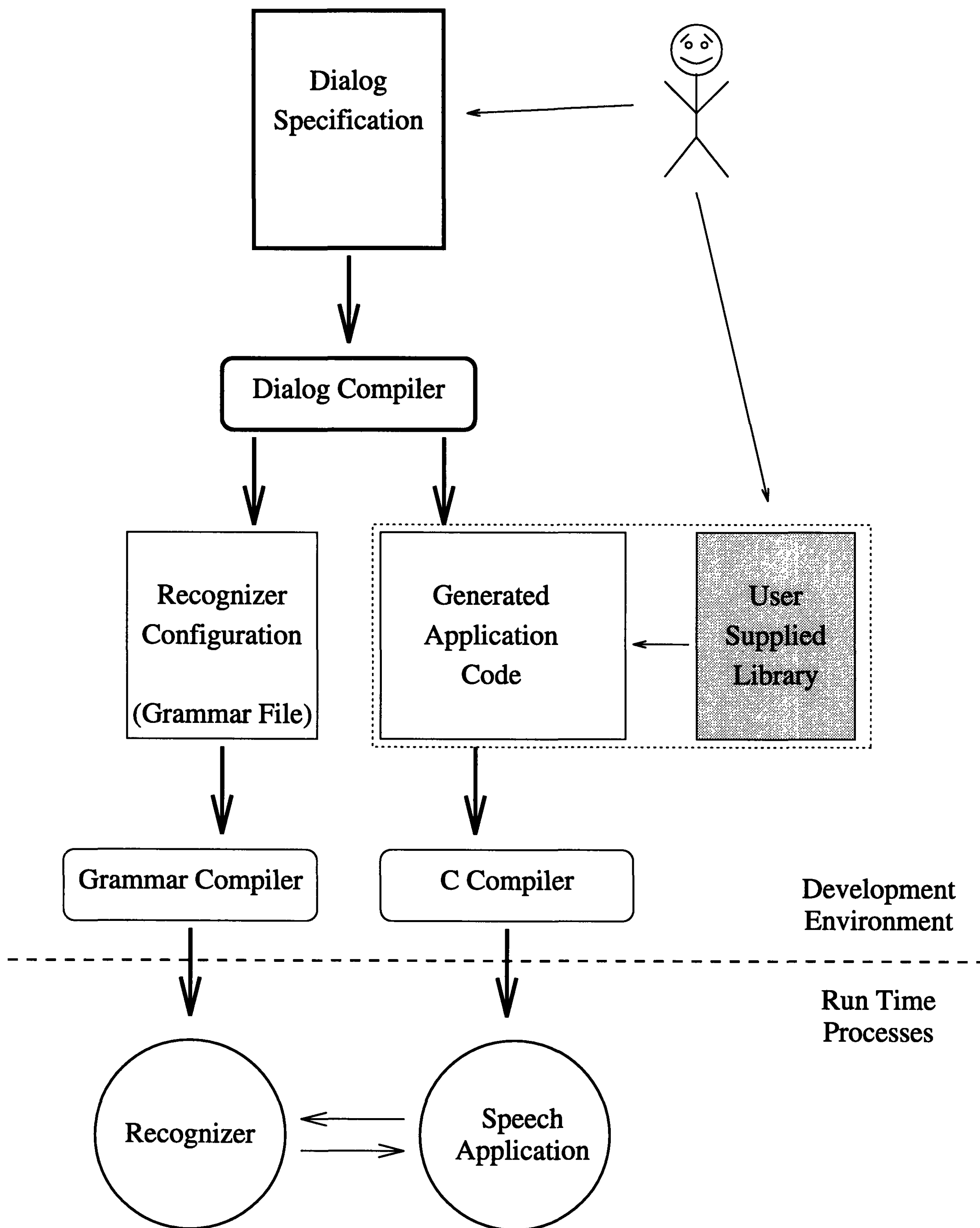


Figure 1.2: Overview of the Development Environment Using the Dialog Specification Language

actions, and dialog transitions all within the dialog specification. The dialog compiler reads the specification file and automatically generates the necessary segments of the application which are involved in maintaining an interactive user dialog. The only modules of code the developer is required to provide are the custom functions which implement specific actions that are executed during the course of the dialog. These modules are referenced in the dialog specification and linked in automatically by the dialog compiler as the application is built.

Using a single dialog specification, applications now become much easier to specify and implement. In addition to having one common specification, as discussed above, the dialog language makes it easier to define higher level aspects of dialog in a consistent manner. For example, particular features of the dialog such as help messages, rejection of syntactically invalid utterances, and backup or "undo" can be handled in a consistent manner throughout the dialog. Additional features, such as "talk ahead", the ability for an experienced user to respond ahead of the prompts, now also become much easier to incorporate into the dialog.

1.4 Outline of the Thesis

The next chapter provides background information on the fields of speech recognition systems and application generators for voice dialog systems. This material should be particularly helpful to those not familiar with the state of the art in these fields for understanding the work done in this thesis.

Chapter 3 contains a functional description of the dialog specification language. It includes a discussion of the various features contained in the language and of the factors which contributed to its design.

Chapter 4 contains a detailed discussion of the implementation of the language and the associated compiler.

Chapter 5 provides a detailed description of the language syntax, intended to be used as a *User's Guide* for the language.

Chapter 6 goes through a sample application implemented using the dialog specification language.

Chapter 7 outlines some directions for future related work and describes possible improvements to the language which were not within the scope of this thesis.

Chapter 2

Background

This chapter provides some important background information relating to the fields of speech recognition, interactive telephone systems, and application generators. This chapter is intended to present enough material so that those who are not familiar with these fields can nonetheless follow and understand the work presented in this thesis.

2.1 IVR Systems

This thesis focuses on interactive telephone applications to demonstrate the suitability of a dialog specification language in developing complex spoken dialog systems, although spoken dialogs are applicable to a wider range of systems and situations. This type of application is often referred to as an Interactive Voice Response (IVR) system. Current IVR applications use voice prompts to conduct a spoken dialog with the user and allow the user to communicate back to the computer by using the twelve DTMF touchtone keys on a telephone keypad. Using touchtone keys the user can traverse through voice prompts and menus to access a variety of functions, for example retrieving information from a particular database.

IVR applications are often modeled as state machines. Each state represents a particular point in the dialog and the touchtone keys are used to control the transitions to other states. Figure 2.1 shows how a menu based IVR application can be modeled as a series of states and transitions. Each time the user presses a touchtone key the dialog

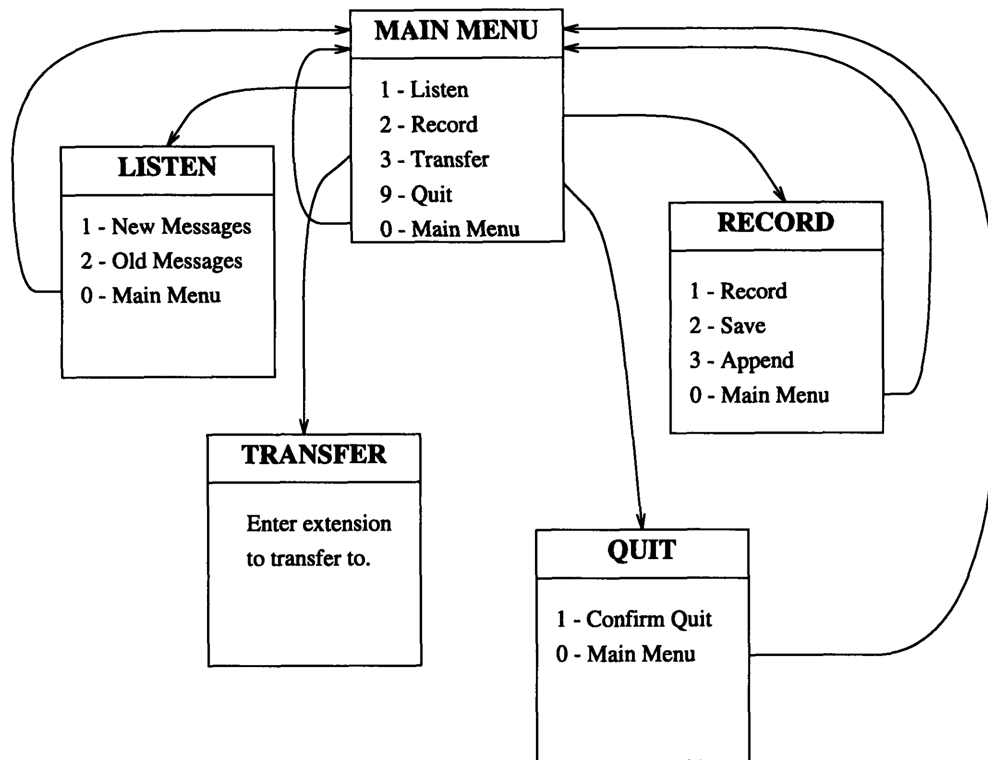


Figure 2.1: Example IVR State Diagram

transitions to a new menu until the user has reached the information or function they were looking for.

2.1.1 Application Generators for IVR Systems

A large number of application generators exist today for building telephone IVR systems. Many of these systems adhere to the state machine model described above to represent the application call flow and the dialog transitions. Application generation systems are particularly helpful in building IVR systems since they provide the system developer a simple means to specify an arbitrarily complex dialog.

Some IVR application generation systems which are currently available include Mediasoft's Interactive Voice System (IVS) generation facility and Infologue's V.A.S.E. system. Both of these systems provide a graphical user interface (GUI) for defining the state machine and the call flow, however they also allow an escape to some high

level language (such as C) to express more complicated logic. The events which drive the IVR state machine are the twelve DTMF touchtone inputs, plus a handful of other system events such as timeout and hangup. These systems as yet do not handle the integration of recognized speech inputs.

A more automated class of application generators are the ROLM *Call Process Developer* [Rol89], and ATS's *VoiceMagic* [Luh91]. These systems are designed to build standard applications by asking the developer to define a few options from a predetermined set of choices. Application generators of this type are useful for building only the simplest classes of applications. Applications which require specialized or custom features may force the developer to modify the generated code by hand, which often can more tedious and time consuming then writing the entire application from scratch.

Other types of IVR system building tools include simple script languages which provide basic functions for application developers using English like syntax, which is then interpreted at run time. Script languages are designed generally for a specific type of application and do not generalize well to a broader class of problems. While providing a relatively simple interface, such systems also lack the ability to express more complicated application logic.

Some specialized high level languages exist, for example *Fante* [Ren92], however such languages have their limitations, and as [Luh91] remarks, "sometimes even intermediate vendor programming languages are insufficient to get down to the nitty-gritty of a particular IVR application." The limitations of special purpose high level languages arises from the tradeoff between the ease of use of the language and its flexibility. BT Laboratories has developed a dialog constructor for speech-based applications [TWW93], however it's purpose is primarily for dialog design and evaluation, and it cannot be used to generate the application which implements the dialog.

2.2 Speech Recognition Systems

There are a wide range of speech recognition systems available today offering a variety of features and capabilities. Simple speech recognition systems provide only limited vocabulary, isolated word, speaker dependent recognition. Such limitations of the recognition

system place constraints on the capability of applications built around them. Examples of these simple recognition systems available today include Verbex Listen, VCS, and VPro.

A more sophisticated class of speech recognizers can provide continuous speech, medium to large vocabulary, real-time, speaker independent recognition. Included in this category of recognizers is the BBN HARK recognition system, as well as IBM ICSS, and the TI Dagger recognition system. This thesis has been developed using the BBN HARK recognizer, and the next section outlines some of the key capabilities provided by this system.

2.2.1 Capabilities of the HARK Speech Recognition System

One of the important features of the HARK recognizer is that the configuration of the recognizer can be dynamically modified at run time. Part of the recognizer's configuration is the recognition grammar. The HARK recognizer relies on a phrase structured grammar definition to evaluate possible recognition theories.

A tool for building the grammar specification is provided with HARK and is called the HARK Prototyper. The Prototyper uses a modified *Backus Naur Form* (BNF) syntax to express the recognition grammar. A grammar is written as a series of rules; each rule consists of an an expression, or formulation, bound to a unique non-terminal symbol. The following is an example of a Prototyper definition.

```
$DIGIT : one | two | three | four | five
       | six | seven | eight | nine | zero
       ;
$PHONE_NUMBER : $DIGIT*
              ;
$TOP : phone $PHONE_NUMBER
     | phone home
     | phone my office
     ;
```

The symbols preceded by **$** are non-terminals. To the right of each non-terminal the expression can be a sequence of other non-terminals and terminals. Terminals in the

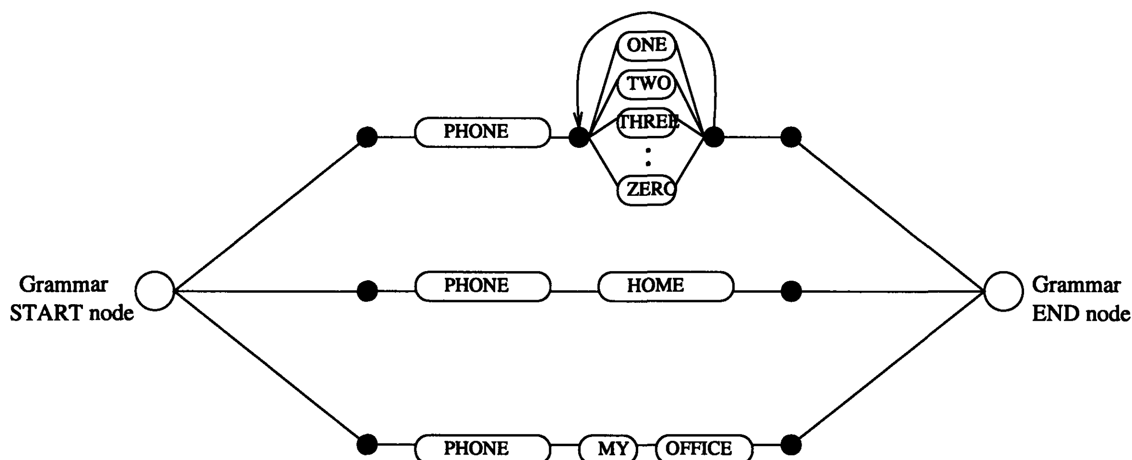


Figure 2.2: Example Grammar Structure

Prototyper grammar are valid words for recognition. The structure pictured in Figure 2.2 shows a representation of the grammar built from the specification in the example.

The Prototyper provides several features for building a grammar which can be dynamically modified at run time. The arcs in the grammar, for example, can be labeled in the Prototyper specification, so that at run time the weights (costs) of the arcs can be modified. Setting the weight of an arc in the grammar to zero inhibits the recognizer from considering any theories in that section of the grammar.

In figure 2.3, `A`, `B`, and `C` are labeled arcs in the grammar. The weights of those arcs can be manipulated at run time to enable or disable those particular sections of the grammar.

The Prototyper has an additional capability which allows any group of words in the grammar to be defined as a contiguous region, and assigned a region number. Each word in the grammar has an associated region number, and these region numbers are returned by the recognizer with the recognition result. In figure 2.4, region 1 contains all the phone number digits, and region 2 contains the words `phone my office`. Since the region numbers associated with each word are returned with the recognition they can often be important in helping the application interpret the recognition result by identifying which areas of the grammar were traversed during recognition.

Words in the grammar can be annotated with special tags which are also returned with the recognition output. Tags can be used to code the semantic meaning or interpretation

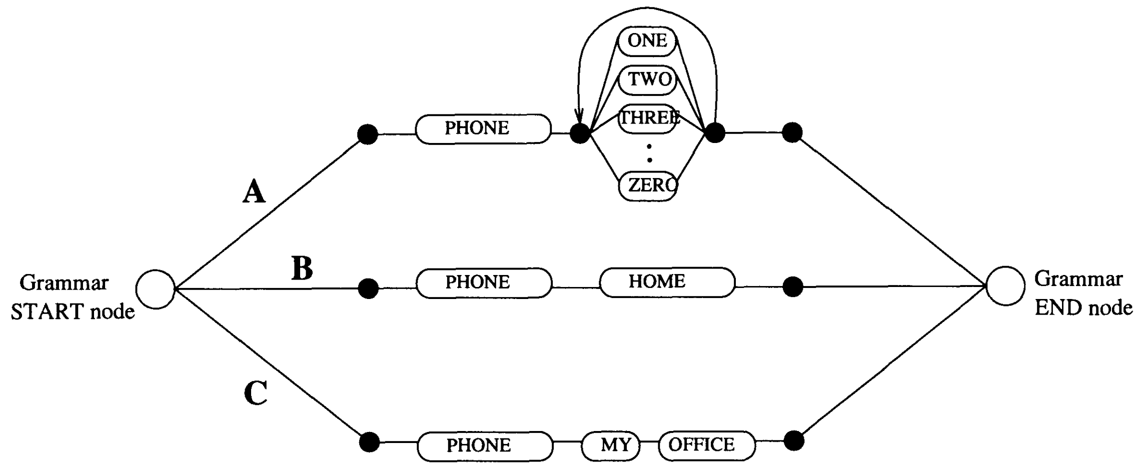


Figure 2.3: Grammar Structure With Labeled Arcs

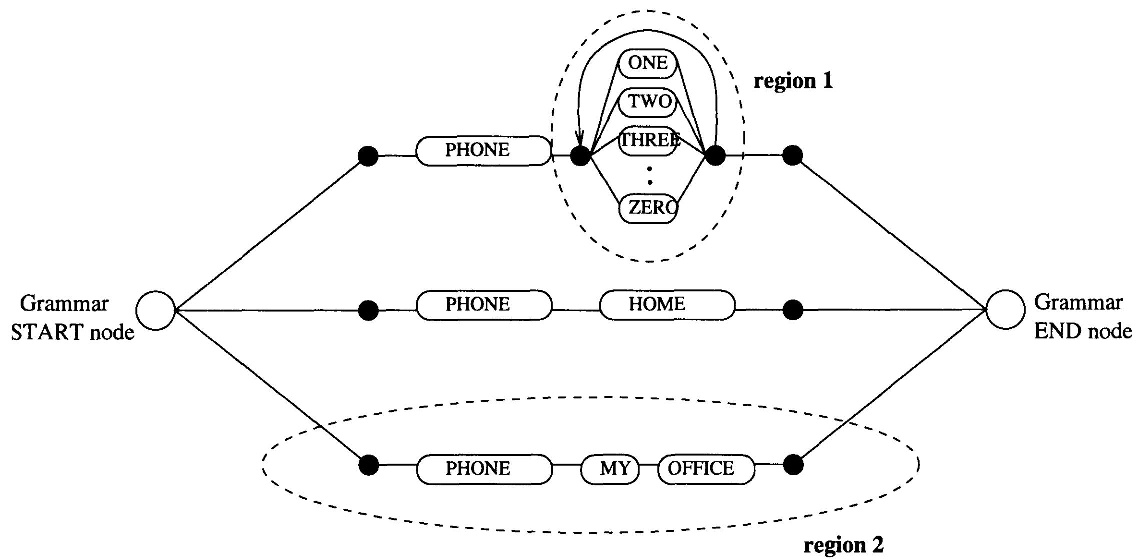


Figure 2.4: Grammar Structure With Defined Regions

of a word directly in the grammar. Consider the following example.

```
$VOLUME : raise the volume/VOLUME_UP
        | increase the volume/VOLUME_UP
        | volume up/VOLUME_UP
        ;
```

In this example of a grammar fragment, the tag **VOLUME_UP** is returned with the recognition result when any of the three utterances listed are recognized, implying that their meaning to the application is the same. A well written application would not need to look at the exact words returned in the recognition result, simply at the tag, in order to determine how to process the command.

A typical output from the HARK recognizer might look as follows.

```
utterance: [SILENCE] raise the volume|VOLUME_UP [SILENCE]
regions: -1 1 1 1 -1
```

Tags associated with words are separated by a vertical bar | from the words. The region numbers for each word are returned in a separate list following the list of words.

The HARK Prototyper can automatically build a section into the grammar used for rejecting out of set utterances. This "alternate grammar" is a special subgrammar which is placed in parallel with the rest of the grammar to aid the recognizer in filtering out utterances which are out of set. Figure 2.5 shows the typical placement of this alternate grammar in the grammar. If the recognition theory goes through the alternate grammar then the recognizer returns an indication that the utterance was out of set, or rejected.

For more information on particular features and capabilities of the HARK recognizer, refer to the *HARK Recognizer System Integrator's Guide* [BBN94] and the *HARK Prototyper User's Guide* [BBN93].

2.3 Integrating Speech and IVR Systems

The integration of speech recognition into IVR type applications will undoubtedly have a tremendous impact on the usability of such systems. Input to an IVR system is currently limited to the twelve keys on the telephone keypad. Integrating speech recognition can

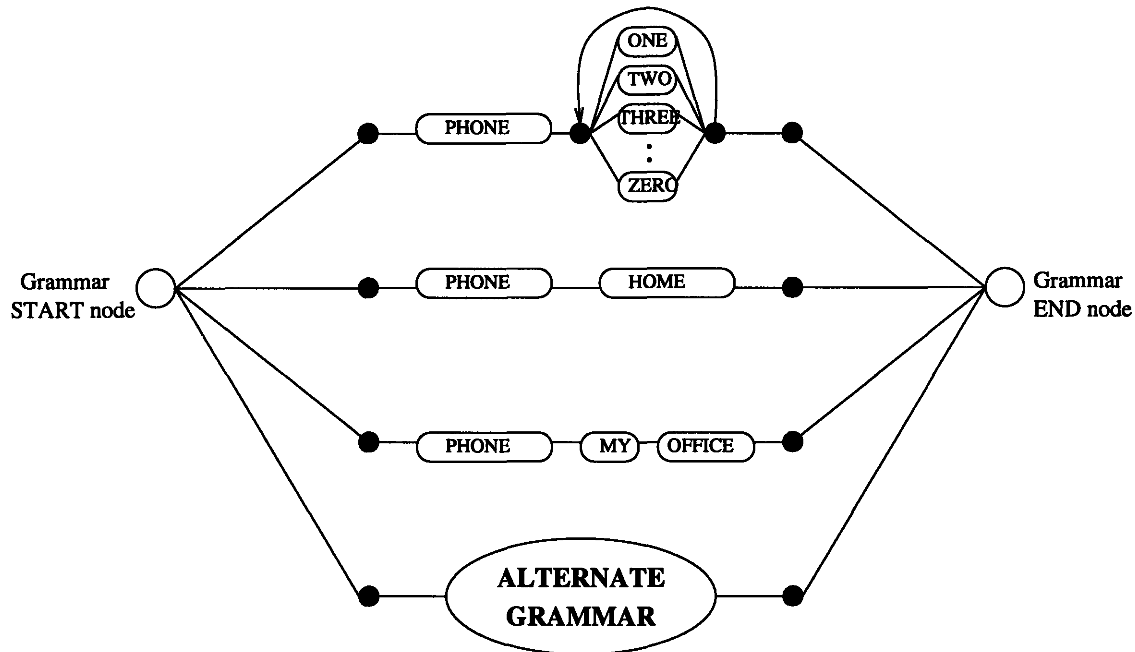


Figure 2.5: Grammar With Rejection

provide the user with a more natural, open-ended interface to the system. The dialog need not be as structured as with existing applications, and users will be able to access information much quicker with speech than with traditional DTMF touchtone inputs. Speaker verification can also be introduced into applications which may require secure access [Luh91].

Integrating speech into IVR applications will place a tremendous demand on the tools used to generate applications. Specifying the valid inputs in a speech application is not as simple as specifying one of twelve keys on a touchpad. In addition, speech systems should be able to reject invalid command syntax. With touchtone IVR systems, the user is limited to entering only one of 12 valid keys, or pressing nothing. This results in 13 possible inputs which should be considered at each level in the dialog. Speech input, however, can include an arbitrary set of valid utterances, and the set of valid inputs can change at each step in the dialog. An application building tool for a speech enabled IVR system must be capable of handling these additional intricacies.

2.3.1 Limitations of Extending Current IVR Application Generators

A class of IVR application generation tools claim to integrate speech into their systems as well as the DTMF touchtone inputs. The speech input, however, is very basic; most systems only accept the speech equivalents of the twelve touchtone keys — i.e., "say one, two ..." plus possibly a few extra keywords like "help", or "balances".

These systems employ very simple speech recognition systems, in which the entire vocabulary for the application is active at all times. The recognizer is not controlled by the application and the active vocabulary is not dependent on the state of the application. This limits the recognition to very small vocabulary systems, and hinders the recognition accuracy.

More robust speech recognition systems exist today, including those in use in the research environment. Because the vocabulary for such systems can contain any arbitrary spoken utterance, there are no generic application builders for large vocabulary speech recognition systems. The application state machine logic must be hand coded using some general purpose high level language, like C. In addition, any changes to the recognizer configuration during run time, affected by the application state, must be done explicitly by the application. The parsing of the recognition output must also be hard coded into the application, and carefully implemented to be consistent with the recognition grammar.

In addition, many of the existing IVR application builder tools only allow the developer to specify simple tree structured logic to model the dialog. They are not designed to allow the developer to specify intricate branching and complex dialogs for their applications. Even now, without the added intricacies of a speech dialog, certain complicated IVR applications cannot be built with existing tools because of these inherent limitations. While speech has the potential to provide tremendous benefits to existing IVR applications, it is clear that current tools for building applications will not be sufficient. A new generation of tools is required giving the developer the capability to specify intricate dialogs between human and machine.

Prior to this work, no spoken dialog application generator has been developed that addresses all of these issues. The tools available today will be simply inadequate for building the complicated interactive speech systems of tomorrow. Even today, those developers on the leading edge of speech application development are forced to tediously

code their applications in C, or some other general purpose programming language, for lack of an alternative, and at great cost. Once speech applications become more common, the need for rapid development of such systems will grow quickly. A dialog specification language will be the critical component for prototyping and developing complicated systems of this type.

Chapter 3

Functional Description

This chapter presents a functional description of the dialog specification language. It provides an overview of the features of the language and the factors which influenced its design. A more detailed discussion of the language implementation is covered in the next chapter on implementation.

3.1 State Machine Model

Using the dialog specification language, a spoken language dialog is modeled as a sequence of transitions between a defined set of states. Each state represents one point in the dialog where the user can supply input, and the transitions between states comprise the flow of the dialog. The dialog state model is similar to an IVR system state model, shown in figure 2.1, except the transitions can be triggered by speech input, not only touchtone input.

States transitions are input driven — the dialog remains in its current state until an input, usually a speech input, triggers the transition into another state. Dialog states can be considered wait states. The dialog state machine waits in a particular state until one of the expected inputs is received, and the input determines which state to move to next.

A more detailed model of a dialog state is shown in figure 3.1. Each state consists of an inbound arc, a list of inputs, and one outbound arc for each of the listed inputs. In the dialog specification language it is possible to specify actions and prompts which are

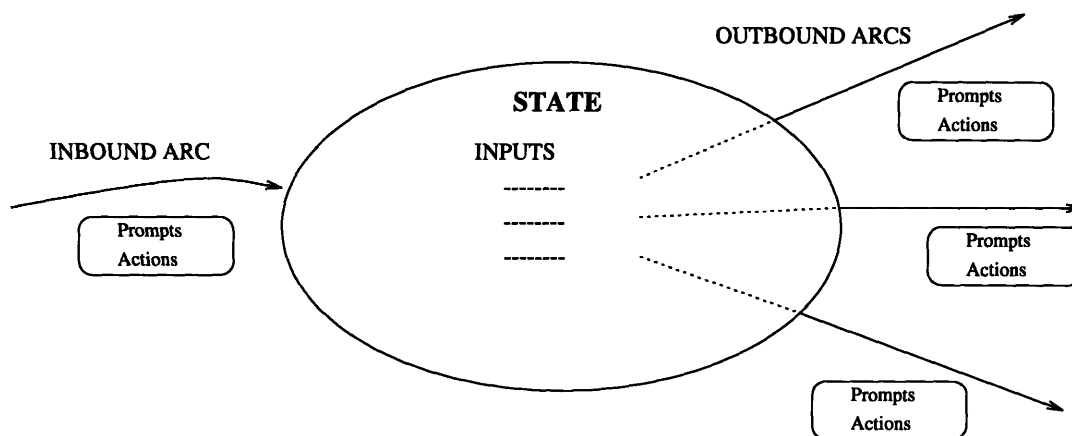


Figure 3.1: Single Dialog State

executed on the transition arcs either into or out of a state. The choice of outbound arc, and its associated actions and prompts, is determined by the particular input received.

The control flow of the application is dictated by the flow of the dialog state machine. The actions associated with state transition arcs are part of the code supplied by the developer. The state machine calls those user functions at the appropriate time in the dialog. A flowchart showing the application control flow coming into a state and leaving a state is shown in figure 3.2.

3.2 Dialog State Inputs

States can contain two different types of inputs, speech and application events. Speech inputs are spoken by the user and returned to the application by the recognition system. Application event inputs can be any other source of input received by the application.

3.2.1 Speech Inputs

Each state has a list of possible grammar specifications, or spoken phrases, which are accepted in that state. In the current implementation, the spoken phrases are defined in the state in the same syntax as HARK Prototyper rules [BBN93]. In the simplest case, each spoken phrase is associated with a single arc leaving the state, and this outbound

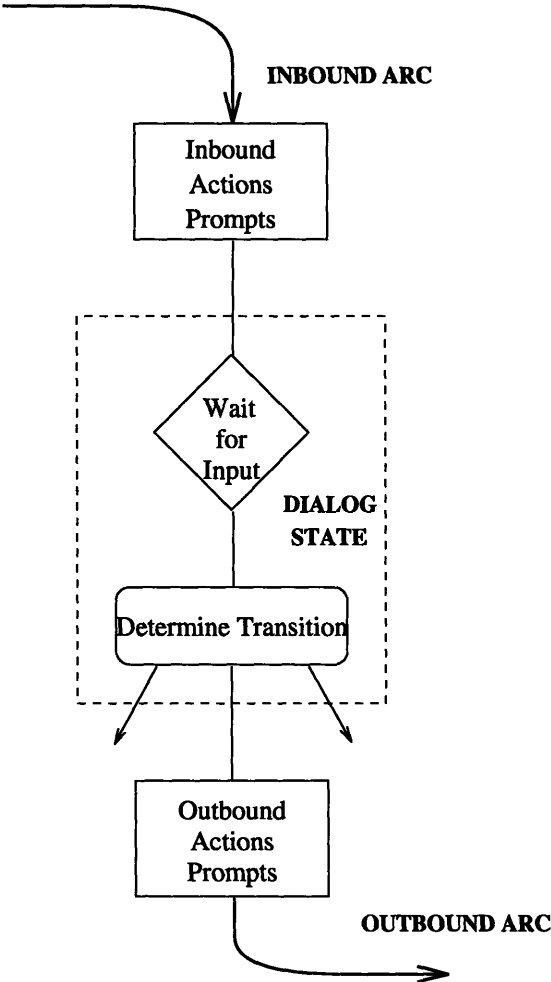


Figure 3.2: Application Control Flow

arc can be assigned actions and prompts which will be executed if that input is received. More complicated control of the dialog flow is discussed in section 3.3.

3.2.2 Dynamic Control of Valid Speech Inputs

At any point in the dialog, the recognizer considers a certain set of recognition inputs as valid at that time. By limiting the number of valid inputs the recognizer has a easier time choosing a possible match for the user's input. For example, if the current dialog state is asking for a confirmation, then there may only be three valid inputs at that state: yes, no, and cancel. Limiting the choice of inputs helps the accuracy of the recognition system.

When a state specification is written, it is assumed that any time that state is active in the dialog all inputs in that state are also active. The state specification describes a static structure which does not change at run time.

Sometimes, however, it is desirable to alter the set of active speech inputs in a state dynamically during run time. The dialog may progress to a point where the application knows that certain inputs may not be valid at that time, at the user would be unlikely to say them. In such a situation, one approach would be to simply leave the static state definition alone and ignore the few extra inputs which are active but are actually not valid. This would be reasonable if our speech recognition system were capable of delivering 100% accurate recognition results. In such a case it wouldn't be of concern that the recognizer was listening for a few additional inputs that were not likely to be spoken, because the recognizer would only return what the user actually spoke.

Speech technology, however, is not yet at the stage where we can count on 100% accurate recognition. Even the most constrained recognition problems still must assume a marginal error rate in recognition accuracy.

The most potentially damaging type of recognition error for the application is a substitution error, where the user speaks an utterance that is in set, and the recognizer returns a different in set response. One way for the application to minimize substitution errors is to constrain the set of valid inputs as much as possible. If there are extraneous inputs active in the grammar, the recognizer is more likely to return one of those inputs erroneously. If those extra inputs are removed from consideration we have effectively

increased the recognition accuracy by reducing potential errors.

The dialog specification language supports dynamic enabling and disabling of speech inputs. This feature allows the application to control when certain inputs are enabled and to dynamically modify the configuration of the recognition grammar.

Certain dialog states may have several possible choices for speech inputs, however based on information available at run time the application can determine that some of those choices are not valid some of the time. Consider the following example of an automated dialer application, where the user has asked the system to call his friend Ben Bitdiddle. The computer looks up Ben in the Rolodex database and finds that Ben has two numbers listed, home and office. The computer then queries the user as to which one is desired. The dialog might look like this:

```
User:       Please call Ben Bitdiddle.
Computer:   Which number please?
User:       Office.
Computer:   Dialing Ben Bitdiddle at the office.
```

The dialog in this example has traversed two states — the first where the user gives his initial request, and the second where the computer gets additional information about which number to call. The specification for the second state might be written as follows (leaving out the actions to take for each input):

```
(define-state WHICH_NUMBER
  (prompt "Which number please?")
  (input "home"
         "office"
         "fax"
         "pager"
         "car phone"))
```

The specification for this example dialog state has to be generic enough to accept other types of phone numbers when appropriate. Some entries in the database may have a car phone number listed, or possibly a fax number. Here the specification allows for

five possible valid types of numbers, even though Ben Bitdiddle only has two, home and office.

What if the user asks the system to call Ben on his car phone? The application has already determined that Ben does not have a car phone. Either the user asked for a number which did not exist, or the recognition was inaccurate and heard car when the user may have said office. A simple solution to this problem would be to pass the recognition result to the application and allow the application to determine if the result was valid at that time. If the user asked for an invalid choice, the application could request that the user make a different selection. If the user was incorrectly recognized, then the frustrated user would have to repeat his original choice.

The dialog specification language provides a mechanism which allows the application to dynamically control which recognition inputs are valid in a given dialog state. The mechanism involves augmenting the dialog specification with function predicates which are called at run time to determine if a particular recognition input should be active at that time. This gives the application complete flexibility in enabling or disabling particular utterances depending on particular application state. Here is an example of what the augmented dialog specification might look like:

```
(define-state WHICH_NUMBER
  (prompt "Which number please?")
  (input "home"
             (enable home_phone_p())
           "office"
             (enable office_phone_p())
           "fax"
             (enable fax_p())
           "pager"
             (enable pager_p())
           "car phone"
             (enable car_phone_p())))
```

The predicate functions are written by the application developer. They can access whatever internal application state necessary in order to return appropriate values to

control the grammar. In this case, any predicate which returned a value of true would enable that input as valid, returning a value of false would disable the input. Going back to the example of calling Ben Bitdiddle, the application would look up Ben in the Rolodex and determine that only a home number and an office number were on file. In this case the predicates **`home_phone_p()`** and **`office_phone_p()`** would return true, while the rest would return false.

For any inputs which do not need to be accessed and dynamically set by the application the predicate can be omitted, which leaves the input enabled whenever that state is active. If, for example, every name in the Rolodex file always has a home phone number, then the **`home_phone_p()`** predicate is not necessary, and that enable option can be omitted. This approach can also be easily extended to allow the functions to return any floating point number, which would be set as the bias, or probability that the input is expected. For simplicity, however, the current implementation will assume the enable functions are predicates.

3.2.3 Application Event Inputs

In many applications, speech input is not the only form of input available to the user. Applications may also allow the user to type requests to the keyboard or mouse click graphical buttons in addition to providing speech input. Telephone applications may allow touchtone input along with speech.

The dialog specification language provides an easy mechanism to specify any other type of input available in the application right along with speech inputs, and to specify the same actions and dialog transitions which can be specified for speech input. The dialog language supports a concept of *application events*. Application events can be any events which occur within the application, possibly from other input sources, and are reported to the dialog manager in a similar fashion as recognition events are reported. This allows the application developer to specify both speech and non-speech events directly in the dialog specification. This unified specification makes it very easy to design an entire mutli-modal application using a single specification.

Application events can be a very powerful and flexible tool for the application developer. They provide the ability to define applications which accept many different forms

of input and have the entire dialog flow still specified in one place. These application events are fully integrated into the dialog state machine and can affect the dialog flow in exactly the same way as speech input.

Consider the following example which shows a sample telephone application where touchtone input is integrated directly with speech input.

```
(prompt "Say listen or touch 3 to hear your messages")
(input "listen"
         (next LISTEN_MESSAGES))
(event "TOUCHTONE_3"
         (next LISTEN_MESSAGES))
```

When the application dialog is in this state, the user has the option of either speaking "listen" or touching the number 3 to listen to their messages. It is clear by the specification that they will both have the exact same effect.

Application events can also be used to control the dialog in different ways than the speech input. Consider the following example.

```
(prompt "Speak the name of the person you wish"
        "to call or touch any button to cancel")
(input "[call] $NAME"
           (action lookup_name($1))
           (next DIAL_NAME))
(event "ANY_TOUCHTONE"
           (prompt "Canceled")
           (next MAIN_MENU))
```

This example shows how a touchtone can be used by the user to cancel the request to dial, while speaking the name of a person will dial that person. In both cases, the event labels, **TOUCHTONE_3** and **ANY_TOUCHTONE**, are arbitrary, and defined by the application developer. It is expected that the application will monitor the event sources and inform the dialog manager when such an event has occurred.

3.3 Dialog State Transitions

The transitions from one state to the next are defined for each input listed in the state. Every input, both speech inputs and application events, must have a state transition defined, although inputs may have their transition defined back to the same state, which is the default if no other state transition is given.

3.3.1 Application Control of Dialog State Transitions

The dialog specification language supports a powerful feature which allows the application to dynamically control dialog state transitions. Instead of defining state transitions which will always occur after the given input or event is received, it is possible to also define state transitions which are conditioned upon the result of some application dependent expression, that could in turn depend on the spoken utterance.

This conditional transition capability gives the application developer the flexibility to branch to different states depending on the value of specific application state. Consider the following two dialog examples derived from a typical voice mail application interface:

```
Example 1:
    Computer:   Good Morning, You have 3 new messages.
    User:       Play back message number 1.
    Computer:   Message 1 ...
    User:       Skip remaining new messages.
    Computer:   You have 5 old messages.
    User:       Play back message 3.
    Computer:   Message 3 ...

Example 2:
    Computer:   Good Morning, You have no new messages.
                You have 1 old message.
    User:       Play message.
    Computer:   Message 1 ...
```

In the first example, the user has both new messages and old messages. The dialog from the computer should indicate that there are new messages and allow the user to access them. In the second example, the user has dialed in and has no new messages. Here the dialog must take a slightly different course, indicating that there are no new messages and that the user is only able to access old messages.

The application should be able to use this type of information dynamically to affect the flow of the dialog. Since the information is known only at run time, the language specification provides a mechanism for specifying dynamic control of dialog transitions.

3.4 Handling of User Feedback Within the Dialog

Accepting speech input to the application is only half of an interactive dialog. Providing voice, text, or graphical feedback to the user is a critical component of the dialog. A dialog specification language must provide sufficient mechanisms to allow the developer to easily include prompts and other feedback within the dialog.

Voice prompts can be specified for digital playback in numerous formats. Two of the most common formats for providing voice feedback are either digitized audio files or machine synthesized speech. Digitized audio files are recorded by the developer and maintained as separate files as part of the application. Whenever a new prompt is incorporated into the application, it is necessary to record an audio file containing that prompt. Synthesized speech, on the other hand, is usually generated from text strings (text-to-speech synthesis) or phoneme strings.

In the framework of the dialog states we can see that applications may require feedback to the user both upon entering a particular dialog state as well as during the transition out of a state. The dialog language will support prompting the user at both of these stages. Looking at our previous example:

```
(define-state WHICH_NUMBER
  (prompt "Which number please?")
  (input "home"
           (prompt "Dialing home")
         "office"
```

```
            (prompt "Dialing office")))
```

The **prompt** command which comes right after the state definition specifies the prompt to play upon entering the state. This prompt will be played before the recognizer is told to start listening for input. Additional prompts can be defined for each input by specifying the **prompt** command after the input. These prompts will be played if that particular input is recognized, during the transition to the next dialog state.

Sometimes it is necessary to generate a prompt which is dependent on information known only at runtime. For example, consider the specification for the **NEW_MESSAGES** state (continuing the voice mail example). If the application determines that the user has new messages the dialog will transition into the **NEW_MESSAGES** state. The prompt for this state will say "You have **N** new messages", where **N** is the number of new messages received today. Since the number of messages is only known at run time, it cannot be statically defined in the dialog specification. Instead, the language syntax allows variable strings to spliced into the prompt dynamically from the return values of function calls. Here is an example:

```
(define-state NEW_MESSAGES
  (prompt "You have " (num_messages()) " new messages.")
  (input "play message $MESSAGE_NUM"
         "reply message $MESSAGE_NUM"))
```

In this example, the function **num_messages()** will return a string containing the English word corresponding to the number of new messages, for example "three". This string is then dynamically spliced into the prompt string before it is sent off to the speech synthesizer.

Prompts can also be conditional, based on expressions evaluated at run time. This feature is discussed in further detail in section 3.5.1.

3.5 Assigning Actions and Prompts to State Transitions

Each transition in the dialog specification can be assigned any number of actions or prompts to be executed each time the state machine traverses that arc. Actions can be

any routine defined by the user and referenced in the dialog specification. The dialog state machine will call the user routines at the appropriate points in the dialog.

Prompts are similar to actions, but they are treated as a special case by the dialog specification language. Prompts define feedback which gets returned to the user, either in the form of text, digitized audio, or synthesized speech. Prompts can be assigned to any arc in the dialog so they can be issued to the user at the appropriate time in the dialog.

Figure 3.1 shows the inbound and outbound transition arcs for a state. Prompts and action functions can be assigned to either the inbound or outbound arcs of a state.

Prompts and actions assigned to the inbound transition arc of a state will be executed before the state waits for any input. Prompts assigned to the inbound arcs are most often used to query the user before the a response from the user is expected. Actions assigned to these arcs can be used to initialize parts of the application state when the dialog enters a particular state.

On the outbound arc, prompts and actions are executed only when a specified input is received. Both types of state inputs, speech and application events, can have prompts and actions attached to their outbound transition arcs. The actions can be used to control the application in any number of ways in response to a particular input. Outbound prompts are often used to provide a confirmation to the user of the input received.

3.5.1 Conditional Actions and Prompts

An additional feature is the ability to decide on a prompt or action based on the result of a function or expression evaluated at run time. Consider the following enhancement to the previous example:

```
(define-state NEW_MESSAGES
  (cond-prompt
     ("MSG == 0"  "You have no new messages.")
     ("MSG == 1"  "You have one new message.")
     (&default    "You have" (num_messages())
                  "new messages."))
  (input "play message $MESSAGE_NUM"
           (enable new_messages_p())
```

```
                "reply message $MESSAGE_NUM"
                  (enable new_messages_p()))))
```

Here the initial prompt for the **NEW_MESSAGES** state is decided at run time. The application variable **MSG** is tested to determine the number of new messages. If the value is 0, the prompt will indicate there are no new messages. If the value of **MSG** is 1, the prompt will indicate one message, otherwise the prompt will indicate the correct number of messages as returned by the **num_messages()** function. There are condition commands for actions and for prompts and they can be used interchangeably with the normal action and prompt commands.

This type of dynamic control is crucial in the dialog specification in order to build applications which can produce flexible interactive dialogs. Without this type of dynamic interaction between the dialog and the application, the dialog language would be useful for building only very simple applications with static user dialogs. Providing these features as part of the dialog specification results in a powerful application generation tool capable of building complex dialogs which are adaptive to numerous run time conditions.

3.6 Dynamic Expansion of Non-Terminals

When an action function is registered on a particular input the application developer knows that the function will only be called when that input has been received. When the inputs are relatively simple grammar rules usually no more information about the input received is needed by the application except the knowledge that the input was received. Since the dialog manager handles parsing the input to determine which action to call, generally the application does not need to do any more parsing on the input.

For more complex speech inputs, however, it is useful for the application to get particularly interesting pieces of the recognized input passed in as arguments to certain action functions. Consider the following slightly more complicated example of a dialog state specification.

```
(prompt "Who would you like to call")
(input "Call $NAME at $LOCATION")
```

In this example, the non-terminals **$NAME** and **$LOCATION** are specified elsewhere in the grammar by the application developer and expand into a list of possible names and telephone locations, respectively. When an action function is invoked from this particular input, it may not be enough for the function to simply know that the input was received. The application may need to know exactly which name and location were specified by the user.

The dialog specification supports the dynamic expansion of non-terminal grammar expressions, and can pass in to a function the substring of the recognition text which corresponded to a particular non-terminal. Such arguments are passed to action functions in the following manner:

```
(prompt "Who would you like to call")
(input "Call $NAME at $LOCATION"
         (action dial_number($1, $2)))
```

In this example, the special labels **$1** and **$2** refer to the first and second non-terminal expression specified on the input line, respectively. When the function **dial_number()** is called, those arguments will be expanded to the string of words which correspond to the part of the recognition string containing the name and location, respectively. For example, if the user says *"Call John Smith at the office"*, the action **dial_number("John Smith", "the office")** will be executed.

These special labels refer only to the non-terminals specified on the input line to which this function is attached. You can specify any **$n** as an argument, where **n** can go as high as the number of non-terminals specified on that input. The syntax of using numeric labels to reference the non-terminals, instead of symbolic names, was chosen in order to disambiguate the reference in the case where two or more non-terminals with the same name appear in one input, as in the following example:

```
(prompt "What reservation would you like to make")
(input "Reserve a flight from $CITY to $CITY"
         (action reservation_origin($1)
         (action reservation_destination($2)))
```

In this example, the non-terminal **$CITY** occurs twice in the input. For example, the user might say *"Reserve a flight from Boston to New York"*. The application must be able to distinguish between the two cities given by the user. Using numeric labels each occurrence of the non-terminal can be unambiguously referenced and passed as an individual argument to an action function, as shown in the example.

3.7 Global State

The dialog specification language supports a feature known as the global state. A dialog can have a global state defined in addition to all the other states in the dialog. The global state is a state which is always active in the dialog, and can be thought of as being "superimposed" over each of the other states. While the global state can never be the *current* state in the dialog, it is active at all times along with the current state.

Using a special definition, the global state can be defined with the same components as a normal state. The global state can have actions and prompts defined on its inbound and outbound transitions, and can include both speech and application event inputs.

The global state is used to define speech inputs and application events which should always be active in the dialog. For example, perhaps the input "help" is always allowed in the dialog and plays a general help message to the user. Instead of defining the input "help" in each state definition, it can be defined in the global state and it will remain active throughout the dialog.

Similarly, application events can be defined globally as well. For example, the event **RESET** might need to be globally defined in the dialog to handle events occurring in the application which would require the dialog to be reset to the top level. Specifying the event in the global state allows it to be active at all times.

When an input is received by the dialog manager, it checks to see if the input is defined in the current state. If so, the dialog chooses the transition defined in the current state. If the input is not defined in the current state, the dialog manager checks the global state. If the input appears in the global state then the dialog chooses the transition defined there. For application events, the definitions in the current state always take precedence over definitions in the global state. Therefore, if the same event is defined in the current

state and in the global state, the transition defined in the current state is chosen. For speech inputs, however, if the same input is defined in the current state and in the global state, the choice of transitions may be arbitrary. The reason has to do with the way the recognition grammar is constructed, and the probabilistic method the recognizer uses to choose a path through the grammar. For this reason, care should be taken to specify global speech inputs which are unique in the dialog.

A global state definition for the two inputs mentioned above might look as follows:

```
(define-global-state
  (input "help"
         (prompt-file help.wav))
  (event "RESET"
         (next TOP)))
```

These global inputs are active in the dialog at all times, concurrent with the inputs of the current state. Prompts and actions which are defined on the inbound arc of the global state are executed before the prompts and actions on the current state's inbound arc.

The global state definition is merely a convenience of the language. All the functionality provided by the global state definition could also be achieved by copying the definitions of the global state into each of the other states in the dialog. The single global definition simply makes it easier to specify global aspects of the dialog.

Chapter 4

Implementation

The implementation of this thesis project consists of two major components. The *dialog specification parser* component, also referred to as the *dialog compiler*, compiles the dialog specification into generated application code and recognizer configuration files. The *dialog manager* component controls the flow of the dialog state machine at run time. Calls to the dialog manager are embedded into the top level application to control the flow of the dialog.

4.1 Dialog Compiler

The dialog compiler parses the dialog specification and compiles it into application code and a HARK recognition grammar file. It is responsible for generating the application which implements the spoken dialog specified by the user. The dialog compiler itself is composed of two segments — the language parser and the code generator. Each of these segments is discussed in further detail in the following two sections.

4.1.1 Language Parser

The language parser is the segment of the dialog compiler which reads in the dialog specification and parses it into a series of language rules. The language parser is built using the Unix utility called *yacc* [Joh78]. *Yacc* is a tool which, given a BNF style grammar for a language, will construct a parser which is capable of parsing and analyzing

the specific syntax of the language.

Before the parser can analyze a language specification the input must be tokenized; each lexical component must be separated and identified. The tokenizer, or lexical analyzer, for the dialog specification language is built with the Unix utility *lex* [LS78]. Given a list of tokens, *lex* builds a tokenizer for the language which serves as a front-end to the *yacc* generated parser. *Yacc* and *lex* are designed to be used together to construct a complete language syntax parser.

The constructed parser is actually a push-down automaton, or stack machine. As the parser analyzes its input, it traverses through language syntax states and executes user-definable actions which have been attached to various rules in the language grammar [SF85]. These user-definable functions which are executed during the parsing phase of compilation form the basis for the code generation segment of the compiler, discussed in the next section.

The actual BNF grammar for the dialog specification language, used by *yacc* to build the language parser, is a bit tedious to follow and not particularly helpful to this discussion, however it has been included in Appendix B for the interested reader.

4.1.2 Code Generation

As the dialog language input is being parsed, the code generation segment of the compiler assembles the output application code and the recognition grammar. In all, a recognition grammar and five application source files are output from the code generation phase of the compiler.

Generating the Recognition Grammar

The grammar which is generated by the dialog compiler is carefully constructed and annotated with "tags" and "labels" to assist the application code in manipulating the grammar during run time and in parsing the recognition output. All of the speech inputs specified for a particular state are placed in their own subgrammar, corresponding only to that state. The top level of the grammar contains each of these state subgrammars configured in parallel with each other; figure 4.1 shows the top level construction of the generated grammar. Building a separate subgrammar for each state allows the application

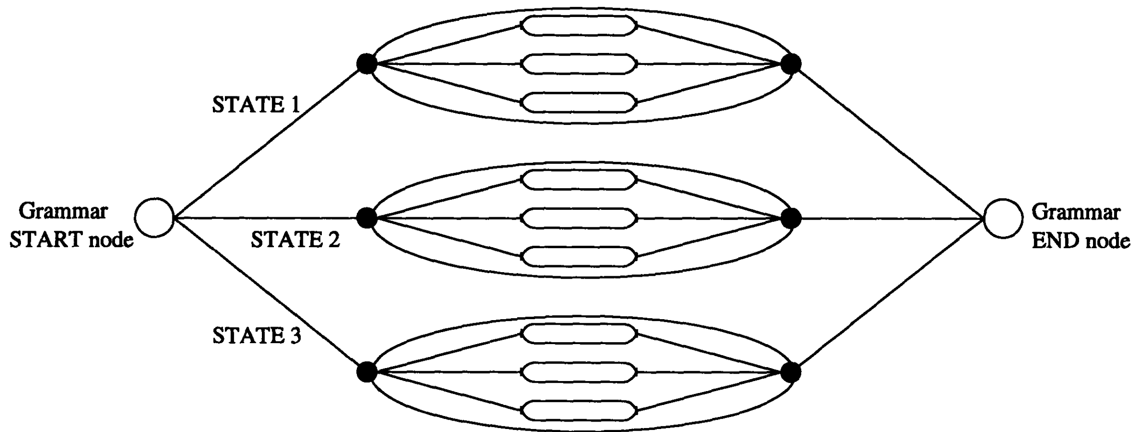


Figure 4.1: Top Level Structure of the Generated Grammar

to activate only the particular subgrammar which corresponds to the state that the dialog is currently in. All other subgrammars are switched off in order to improve the recognition accuracy. When the dialog makes a transition to a new state, the application can switch off the previous subgrammar and switch on the new one. The subgrammars can be switched on and off by labeling the arcs entering each of the subgrammars from the top level.

Within each state subgrammar, the inputs for that state are again placed in parallel with each other. For any given state, all the inputs specified in the dialog specification are considered equally likely when that state is active. In order to support dynamic enabling and disabling of particular inputs at run time, the arc entering each input is also given a unique label. During run time the application can switch on and off various inputs within a state depending on how those inputs were specified in the dialog specification. Figure 4.2 shows a typical state subgrammar with each of the arcs entering the specific inputs labeled.

Labeling arcs in the grammar at compile time provides the application with a mechanism for manipulating those arcs at run time. Specifically, the application can change the *weight* or *penalty* of a particular labeled arc at run time. Assigning an arc the weight of *zero* effectively switches off that portion of the grammar. No recognition theories will be expanded past an arc whose weight is zero, and this portion of the grammar will now be considered inactive. Similarly, if the weight of an arc is set to *one*, then that

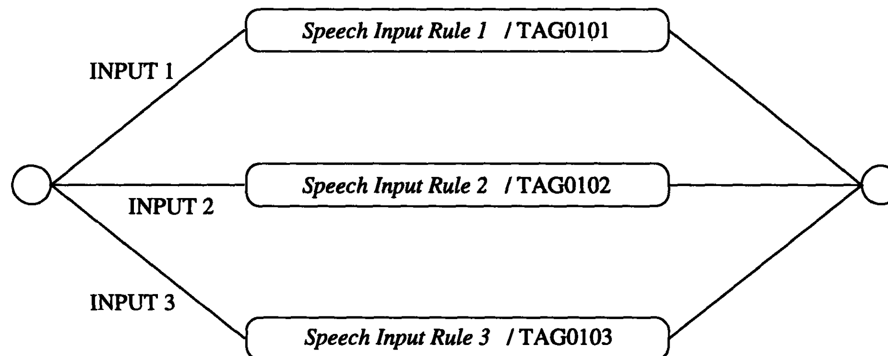


Figure 4.2: Single State Subgrammar

portion of the grammar becomes active. The weight of an arc can actually be set to any value between 0 and 1, which can be used to make certain subgrammars more likely than others, however the dialog compiler does not use this capability, and assumes all active subgrammars are equally likely.

Grammar labels are compiled into the grammar and can be used to reference particular subgrammars and inputs in the grammar. They are not returned by the recognizer, however, as part of the recognition result, and therefore cannot be used by the application to aid in parsing the recognition output. Instead, the dialog compiler uses a mechanism called *tags* to assist the generated application in parsing the result returned from the recognizer. The dialog compiler places a special tag at the end of every input in the grammar. This tag is returned by the recognizer as part of the recognition result.

When the dialog manager is informed that a recognition result has been received, the actual string returned by the recognizer as the recognition result need not be parsed for the dialog manager to execute the appropriate actions for that input. The dialog manager only needs to look at the special tag placed in the grammar for that input. That tag corresponds to an entry in a table of callback functions which was constructed by the dialog compiler during the compilation phase. The table contains the names of generated functions which will execute all the specified actions for that input and transition the dialog to the appropriate next dialog state. Using this mechanism, the dialog compiler can effectively leave notes for itself, in the form of tags, in the grammar. These notes are then used to reference particular functions listed in the table which is also built by the dialog compiler. This alleviates the need for the dialog compiler to store in a table or

State	Initialize Function	Speech Input Table			Event Input Table		
1	init01	Tag0101	action0101	next0101	Event0101	actionE101	nextE0101
		Tag0102	action0102	next0102	Event0102	actionE102	nextE0102
		Tag0103	action0103	next0103	Event0103	actionE103	nextE0103
2	init02	Tag0201	action0201	next0201	Event0201	actionE201	nextE0201
		Tag0202	action0202	next0202	Event0202	actionE202	nextE0202
		Tag0203	action0203	next0203	Event0203	actionE203	nextE0203
3	init03	Tag0301	action0301	next0301	Event0301	actionE301	nextE0301
		Tag0302	action0302	next0302	Event0302	actionE302	nextE0302
		Tag0303	action0303	next0303	Event0303	actionE303	nextE0303

Figure 4.3: Dialog State Table

similar structure any information about the exact input strings which were built into the grammar. By using recognition tags the application keeps no information about exactly what words are expected back from the recognizer for a particular input. Figure 4.2 shows how tags are used to annotate input phrases in the grammar.

The recognition grammar generated by the dialog compiler is placed in a file called **`gen_grammar.hg`**. The format of the generated grammar is HARK Prototyper format, and is intended to be input directly to the HARK grammar compiler.

Generating the Application Code

The dialog specification is represented in the application by an internal state table coded in C. Each entry in the state table contains three components:

- Initialization Function
- Speech Input Table
- Application Event Input Table

Figure 4.3 shows the construction of a dialog state table.

The *initialization* function is the name of a function which gets called each time the dialog state machine enters that state. The initialization function is generated by the

dialog compiler, and contains all the actions and prompts specified on the inbound arc for that state.

The speech input table and the application event input table are each lookup tables which contain a list of symbolic tags. Under each tag in the table is the name of an *action* function and the name of a *next* function. The *action* function, which is generated by the dialog compiler, executes all the user specified actions for a particular input. One *action* function is generated for each input listed in the state. The *next* function, also generated by the compiler, contains the prompts specified on the outbound arc for a particular input, and also the name of the next state the dialog should move to when this input is received. One *next* function is also generated for each input listed in the state.

The symbolic tags in the input tables are used for looking up the correct functions to call when a particular input is received. The tags are slightly different for the speech input tables and for the application event input tables.

For speech input tables, the symbolic tags correspond to unique tags attached to each speech input line in the recognition grammar. The dialog compiler generates these unique tags as it constructs the recognition grammar, and places there tags in the speech input table. The recognizer returns the tag associated with each input along with the recognized word sequence. The dialog manager uses the tag to look up the recognized phrase in the speech input table for the current state. This determines which action functions to call and which state to advance the dialog to next.

In the application event input table, the symbolic tags are exactly the labels used in the dialog specification to identify the event. When the dialog manager is informed that an event has occurred, it looks up the event label in the application event input table to determine which action functions to call and also which state to advance the dialog to next.

Figure 4.4 shows the output files which are generated by the dialog compiler. The dialog state table is built by the compiler and placed in a file called **state_table.h**. The speech input tables and application event input tables for each state, referenced in the dialog state table, are placed in a file called **input_table.h**.

All of the initialization functions, action functions, and next functions for each state in the dialog specification are generated by the compiler in a file called **dialog_gen.c**, with an associated header file called **dialog_gen.h**. Finally, the global definitions

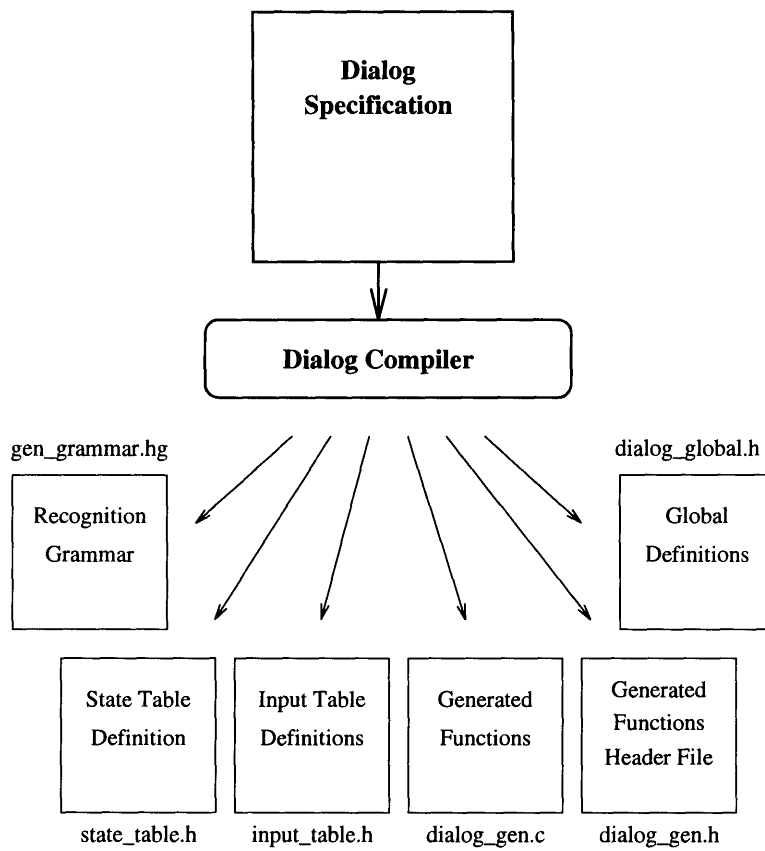


Figure 4.4: Compiler Generated Output Files

defining symbolic constants for each of the state names are placed in a file called **dialog_global.h**.

4.2 Dialog Manager

The dialog manager is responsible for controlling the dialog state machine at run time. The dialog manager uses the state table generated by the dialog compiler to determine which actions to execute and which state transitions to make when input is received.

The dialog manager maintains the current dialog state. When input is received by the application the dialog manager looks up the symbolic tag of the input in the appropriate input table and executes the functions listed. The current state is then updated to reflect the dialog transition.

Upon entering each state, the dialog manager executes the initialization function for the state. This function includes the actions and prompts which were specified on the inbound arc for that state. These actions and prompts are executed before the state begins waiting for input.

Figure 3.2 shows the flow of control in the dialog manager as it processes a dialog state.

There are three functions within the dialog manager which are accessed by the application to drive the dialog state machine. The dialog manager does not control the main loop of an application, therefore it exports three subroutines which are called by the application at appropriate points in the application main loop to inform the dialog manager of key events.

The three subroutines exported by the dialog manager for access by the application are listed below.

- **activate_current_dialog_state()**
- **process_dialog(HFI_EVENT hevent)**
- **trigger_dialog_event(char *EventTag)**

These routines are described in more detail in the next section.

4.3 Speech Application

The top level speech application is constructed by the application developer and integrated with the dialog manager using the three subroutines listed above. The application controls the setup and interaction with the speech recognizer, and with all other input sources in the system. An event driven application loop, with asynchronous communication to and from the various input sources, is the type of top level application used for this thesis project to demonstrate and test the dialog manager. The dialog manager subroutines are general, though, and can be integrated with many other types of top level applications, depending on the style and methodology of the application developer.

Since the dialog manager does not handle the interaction with the speech recognizer or other sources in the application, nor does it control the application's main loop, the dialog manager must be called at certain points in the application in order for the dialog state logic to be processed correctly. The three subroutines mentioned above provide the interface between the application main loop and the dialog manager. The following sections detail the specific points in the application where the dialog manager must be called.

4.3.1 Activating the Current Dialog State

In order to activate the current dialog, which entails executing all the initialization functions and prompts specified on the arc entering the state, the dialog manager must be called just before the application tells the recognizer to start listening for input.

The **`activate_current_dialog_state()`** routine is called from the main loop of the application immediately before the application sends the command to the recognizer to start listening for speech. When this routine is called, the dialog manager looks up the initialization function for the current state and executes it. The initialization function contains the actions and prompts which were specified on the inbound arc for the state.

4.3.2 Registering Speech Input

Since the application is handling the communication with the speech recognizer, the application will receive the results of the recognition when the user speaks an utterance.

The application does not necessarily need to look at the recognition result, it simply needs to pass it along to the dialog manager.

The **`process_dialog(HFI_EVENT hevent)`** routine is used by the application to register the speech input with the dialog manager. The argument **`hevent`** passes to the routine a special identifier used by the HARK recognizer to identify an event returned by the recognizer. Using this event identifier, the dialog manager can ask the HARK recognizer interface library information pertaining to the event.

If the event passed to **`process_dialog`** is a **`RECOGNITION_EVENT`**, the dialog manager gets the recognition result and looks up the special tag returned with the word sequence in the speech input table. The dialog manager executes the appropriate action and next functions for that input and returns control back to the main loop.

If the event is a **`RECOGNITION_FAILED_EVENT`** then the dialog manager queries the HARK interface library for the reason why recognition failed. There are two reasons why the recognition may have failed.

1. No speech / timed out
2. Speech rejected / no valid trace though the grammar

The first reason indicates that no speech was received and the recognizer timed out. In this case the dialog manager looks up the special tag **%TIMEOUT** in the application event input table. This entry in the table is generated by the compiler if the specification contains a **%TIMEOUT** event, indicating which actions to perform if the recognizer times out.

If the second reason is given for recognition failure, the dialog manager looks up the special tag **%REJECTED**. This tag is placed in the application event input table by the compiler if the state specification includes a **%REJECTED** event listed, indicating which actions to perform if the recognizer gets a rejected recognition response.

4.3.3 Registering Application Event Inputs

Application events are registered with the dialog manager in a manner similar to speech inputs. Application events can correspond to any input source in the application. Since

the application handles all interaction with these input sources the dialog manager only needs to know when the events occur, and not how the application got the event.

The **trigger_dialog_event(char *EventTag)** routine is used by the application to register application events with the dialog manager. The argument **EventTag** is a string corresponding to the label used to identify the event in the dialog specification. The dialog manager looks up the event label in the application event input table for the current state and executes the appropriate action and next functions for that event. Control of the application is then returned back to the main loop.

4.4 User Supplied Library

The one remaining segment of the application which has not yet been discussed is the library of user-defined functions provided by application developer. This library contains definitions for the user functions referenced in the dialog specification. The dialog manager will execute these functions at the appropriate point in the dialog based on their specification within the dialog state definitions. The user simply needs to supply these functions in a file called **user.c** so the dialog manager can find them. This library is linked in with the generated application code to build the executable speech application.

Chapter 5

Using the Language

This chapter presents a *User's Guide* to the dialog specification language. It provides a detailed description of the valid commands supported by the language and the proper syntax for using them.

5.1 Dialog Definition

The first definition in the dialog specification is the definition of the dialog itself. The dialog definition command defines the scope within which all subsequent dialog state definitions are contained. The definitions within the scope of the dialog definition form the dialog specification. Any definitions outside the scope of the dialog definition are not considered part of the dialog, and will either be ignored or rejected by the dialog compiler.

A dialog is defined using the **define-dialog** command, and consists of a *dialog-name* followed by a list of state definitions. The *dialog-name* is used as a reference name for the dialog, and the dialog state definitions, discussed in the next section, comprise the main part of the dialog specification. Figure 5.1 shows the format and syntax of the dialog definition.

The dialog definition command is specified in the following way:

(**define-dialog** *dialog-name*
state definition

```
(define-dialog  dialog-name
   (define-state   state-name
         -----
         -----
         -----            )

   (define-state   state-name
         -----
         -----
         -----            )


 )
```

Figure 5.1: Dialog Definition Command

state definition ...)

The *dialog-name* should be an alpha-numeric symbol containing no embedded whitespace, however underscores are allowed. To improve readability, a convention of using only uppercase letters for the dialog name is suggested.

5.2 State Definition

The **define-state** command is used to specify a state definition. Similar to the dialog definition command, the state definition command consists of a *state-name* followed by a list of internal state definitions.

States can be defined in the dialog in any order. It is not necessary for the definition of a state to appear in the specification before a reference is made to it. A state with the name **TOP** must be defined somewhere in the dialog to designate the state where the dialog begins.

Each state in the dialog represents a particular point in the interaction between the user and the computer. The specification of a dialog state, therefore, should completely specify the components of this interaction. These components include how the user is prompted, what user responses are valid, what actions to take for each response, and

```
(define-dialog  dialog-name
   (define-state   state-name
       (prompt  ...  )
       (action  ...  )

       (define-input
              -----
              -----
              -----          )
       (define-event
              -----
              -----
              -----          )  )

 )
```

Figure 5.2: State Definition Command

which state the dialog should advance to next. Figure 5.2 shows the overall structure and format of the **define-state** command.

The state definition command is specified in the following way:

(**define-state** *state-name*
state component
state component ...)

The components listed in the state definition can be any number of the following items:

- Prompts
- Actions
- Speech Inputs
- Application Events
- State Transitions

As discussed in Chapter 3, the dialog language supports the specification of both speech and non-speech inputs within each state. In this way an application can be

specified which accepts speech input as well as other forms of input, including keyboard, mouse, and touchtone. Non-speech inputs are defined in the dialog language as application events.

Each of these components will be discussed in more detail in the following sections.

5.3 Prompts

Prompts are the used in a dialog to generate feedback to the user, either in the form of voice responses, text messages. or both. Three types of prompts are supported by the dialog specification language.

- Text prompts
- Digitized voice prompts
- Synthesized speech prompts

Since each of the prompt types are handled differently by the dialog compiler as the application code is generated, there are three different prompt commands, corresponding to each of the different types of prompts. The supported prompt commands are listed below:

(prompt-text *"text string"*)
(prompt-file *digitized-speech-file*)
(prompt-synth *"speech synthesis string"*)

In the overall dialog specification, prompts can be included in two different places. They can be included in the beginning part of the state definition to define prompts which are to be presented to the user before the user begins speaking. These are the prompts which get assigned to the inbound arc of the state and are useful for prompting the user for information before the recognizer starts listening for input.

They can also be included as part of the definition for particular speech inputs or application events to specify feedback to the user once that input or event has been received. These types of prompts are assigned to the outbound transition arcs of a state

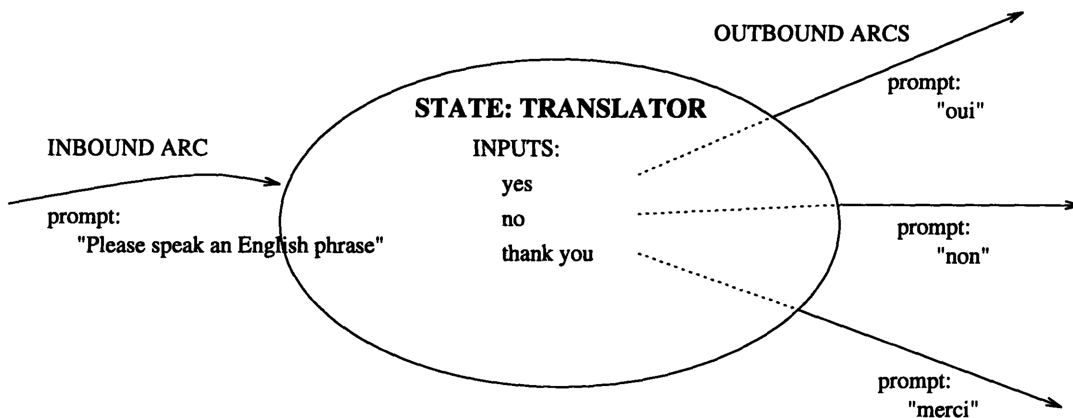


Figure 5.3: Dialog State Showing Transition Prompts

and are used for confirmation or other feedback which occurs after the user has said something. The three different prompt commands can each be used in either of the two places in the dialog.

Consider the following example of a simple English to French translator which shows how prompts can be defined on both the inbound and outbound transition arcs of a state.

```
(define-state TRANSLATOR
  (prompt-text "Please speak an English phrase")
  (input "yes"
           (prompt-text "oui")
         "no"
           (prompt-text "non")
         "thank you"
           (prompt-text "merci")))
```

Figure 5.3 shows a representation of the **TRANSLATOR** state. The initial prompt is assigned to the inbound arc, and the input prompts are assigned to the appropriate outbound arcs.

Often it is desirable to return feedback to the user in more than one form. For example, some applications might want to print a text prompt to the screen while digitized speech is being played from the speaker. Since the three prompting commands can be mixed

and used interchangeably it is possible to specify both commands one after the other. This will have the effect of providing both types of feedback simultaneously.

Multiple prompts of the same type can be specified using one prompt command. Each command will accept a single prompt or a list of prompts, separated by whitespace. This allows multiple prompt strings or speech files to be issued using a single prompt command. For example, both of the following commands are valid:

(**prompt-file** welcome.wav menu.wav)
(**prompt-text** "Welcome" "Please speak your request")

Prompts are always issued to the user in the same order they are specified in the dialog specification. Functionally, it makes no difference whether multiple prompts are specified in a single prompt command or multiple commands are used instead, it is simply a matter of style.

The dialog manager uses the HARK speech recognizer to play out the audio prompts and this is critical to the synchronization of prompts and speech input. The HARK recognizer is designed to start listening for speech only after it has completed playing out all pending audio prompts. This prevents the recognizer from listening to the prompt output and falsely recognizing speech. The dialog manager automatically uses HARK for all the audio playback. If a different mechanism is required for audio playback then the timing issues need to be resolved so that the audio output device and the speech recognizer are synchronized with each other.

5.4 Speech Input

Speech inputs are specified in the dialog specification using the **define-input** command. This command is specified in the following way:

(**define-input** *"speech-input-rule" input components*
"speech-input-rule" input components
"speech-input-rule" input components ...)

The *speech-input-rule* follows the same syntax as the HARK Prototyper grammar rule. The *input components* can include any number of actions and prompts which are

associated with that input, and a dialog transition specification for each speech input. The **define-input** command will accept multiple inputs listed under the same command as shown above. More discussion on the input components will be provided in the next sections.

Since the speech input rule can be any arbitrary Prototyper grammar rule, it must be surrounded by quotes in the dialog specification. The dialog compiler will embed the speech rule directly into the recognition grammar during the compilation of the dialog specification.

Here is an example of a HARK Prototyper definition which defines a particular subgrammar and contains five grammar rules:

```
$TELEPHONE_NUMBERS : home
                   | office
                   | fax machine
                   | car phone
                   | pager
                   ;
```

Using the dialog specification language, those input rules would be specified in the following format:

```
(define-input "home"
              "office"
              "fax machine"
              "car phone"
              "pager"
)
```

Each input is specified within quotes and on a separate line. The vertical bar notation is not necessary in the dialog specification to separate inputs. The dialog input line can be any valid Prototyper rule, and the rule can be as complex as desired. Care should be taken, however, if using the vertical bar operator within an input expression. Speech input rules which contain the vertical bar notation to define alternate paths in the grammar

must be properly parenthesized so as to insure that the input line will be expanded to have only a single endpoint in the recognition grammar. For example, the input line

"call | dial"

may cause problems for the dialog manager because the rule can be treated as two separate expressions within the grammar. By surrounding the expression with parenthesis the input rule becomes perfectly valid in the dialog specification, as shown.

"(call | dial)"

Other prototyper constructs may be used freely as part of the input specification. This includes using non-terminals defined elsewhere, using optional square bracket notation, and using the * and + notations to denote grammar loops. The following are examples of valid input specification lines.

"[(call | dial)] $PHONE_NUM"

"buy $STOCK at $PRICE"

In these examples, the non-terminals **$PHONE_NUM**, **$STOCK**, and **$PRICE** can be defined elsewhere by the user.

The list of input components following each input in the specification can be used to define a number of items associated with that input. The particular components which can be specified for an input are:

- predicate functions to enable/disable inputs dynamically
- action functions to execute when input is received
- next dialog state
- prompts to issue when input is received

Each of these components is specified directly below the input to which it refers. The following sections describe in more detail the exact syntax of each of the available input components.

5.4.1 Dynamic Activation of Speech Inputs

Dynamic control of inputs can be accomplished in the dialog specification by using the **enable** command as an option to any of the inputs. The syntax of this command is as

follows:

(enable *predicate-function*)

If the enable option is present after an input, the dialog manager will call the *predicate-function* each time that state becomes active. If the function returns true (1), the input it refers to will remain active at that particular point in the dialog. If the function returns false (0), then the input will be switched off in the grammar only for the current pass through the state, and the recognizer will not consider the input valid.

The enable option causes the *predicate-function* to be called *each time* the dialog reaches that state. Therefore the predicate function, provided by the application developer, can be written to test some application state each time through the dialog to determine if an input should be active at that time. Any input can have an enable option attached to it, and there is no limit to the number of inputs in each state which can have enable options. If more than one enable command is specified for any input, then only the *last one* specified has any effect; all others are ignored. If no enable option is specified for an input then that input is always active.

The example given below shows how the dialog can take advantage of information within the application to constrain the grammar dynamically at run time. The specification for a state where some of the inputs are dynamically enabled might look as follows:

```
(prompt-text "Which number please")
(input "home"
        "office"
        "fax machine"
        "car phone"
          (enable car_phone_p())
        "pager"
          (enable pager_p())
)
```

The predicate functions **car_phone_p()** and **pager_p()** should be written to test internal application state and return true if the respective speech inputs should remain active during that pass through the state. The dialog manager will test these functions each time the dialog reaches the state and activate those inputs only when appropriate.

5.5 Application Events

Application events are registered with a particular state in the dialog using the **define-event** command. This command is identical in every way to the **define-input** command, except that instead of specifying speech input, the application developer provides a unique event label. The event label is later used by the application to inform the dialog manager when the event has occurred. The same input components can be specified for application events as for speech inputs. The only option which is not available for events is the **enable** command, simply because this command only applies to speech input.

The event definition command is specified in the following way:

(**define-event** *"event-label" input components*
"event-label" input components
"event-label" input components ...)

The *event-label* can be any alphanumeric symbol, containing any amount of whitespace, underscores, or other special characters, and should be enclosed in quotes as shown above. For clarity and consistency, a convention of using uppercase alphanumerics with no whitespace is recommended for event labels. Since labels are treated as strings they will not conflict with any state names in the dialog specification or with the dialog name.

Any number of events can be specified using a single **define-event** command, much the same as the **define-input** command.

5.5.1 Special Reserved Event Labels

The are a few event labels which are reserved by the dialog manager and have special meaning. The following is a list of these reserved event labels and how they are interpreted by the dialog manager.

1. **%TIMEOUT** – This special label is used to define what should be done in the dialog in the event that the speech recognizer times out. This event occurs if the user is prompted for speech input, the recognizer starts to listen, and the user says nothing within a specified amount of time. Often the appropriate way for the dialog

to handle a recognition timeout is to reprompt the user, or to automatically play a help message or list of choices.

This example shows how the **%TIMEOUT** event can be used to have the application play a list of choices for the user if they time out, and then reprompt for their request.

```
(prompt-text "Please speak your request")
(input "account balances"
         (next BALANCES)
       "mortgage rates"
         (next MORTGAGE)
       "cd rates"
         (next CD))
(event "%TIMEOUT"
         (prompt-text "You can get information
                       on your account balances,
                       mortgage rates or cd rates"))
```

The recognition timeout event will automatically be detected by the dialog manager when it is informed of the outcome of the recognition.

2. **%REJECTION** – This special label is used to define what should be done in the dialog if the speech recognizer cannot decode a particular utterance. This event will occur in a number of different situations depending on exactly how the speech grammar is constructed. It means that the user spoke an utterance that the recognizer could not interpret or match with any of the inputs in the grammar – either because the utterance was completely out of the valid set of utterances, or because the recognizer simply could not understand the utterance. When such an event occurs it is generally appropriate to inform the user that the utterance was not understood and to prompt them to repeat it.

   Let's add a handler for rejection to the previous example. If the person speaks an utterance which is not understood by the recognizer we will inform them to speak more clearly and try again.

```
(prompt-text "Please speak your request")
(input "account balances"
         (next BALANCES)
       "mortgage rates"
         (next MORTGAGE)
       "cd rates"
         (next CD))
(event "%TIMEOUT"
         (prompt-text "You can get information
                       on your account balances,
                       mortgage rates or cd rates")
       "%REJECTION"
         (prompt-text "Sorry, I did not understand
                       you.  Please speak clearly
                       and try again."))
```

3. **%NULL** – This label is used only in special cases where there are **no** real speech inputs or application events which are to occur in a particular state. Sometimes, to help in the organization of a dialog into states, it is desirable to have a state which waits for no inputs, but simply executes some actions or prompts immediately, and then continues to the next state.

   If the %NULL event is defined anywhere in a state, then **all** speech inputs and application events in that state will be ignored. The only options which will be executed will be those attached to the %NULL event. For this reason, any state which has a %NULL event defined should have no other speech inputs or application events defined.

   The following example shows how the %NULL event can be used to define a state which simply plays a welcome prompt to the user, and waits for no input.

```
(define-state TOP
  (prompt-text "Welcome to the AnyTown Bank")
  (event "%NULL"
```

```
          (next MAIN_MENU)))

(define-state MAIN_MENU
  (prompt-text "Please speak your request")
  (input
          ...
  ))
```

Organizing the states in this manner allows the dialog to return to the MAIN_MENU state from other points in the dialog without having the user hear the welcome message over again. The TOP state is only traversed once per caller.

It is possible to define any actions, prompts, or state transitions following a %NULL event. They will be executed according to the same rules as any other application event or speech input, the difference being that the dialog manager will not wait for any input before executing those options. The dialog will transition directly to the next state and, if it is not a %NULL state as well, will wait for input there.

5.6 Action Functions

One of the most important features of the dialog specification language is that it allows actions associated with a particular input to be defined lexically adjacent to the definition of the input itself. This not only makes it easier to design the application dialog, but also makes it easier to visualize the flow of control in the application with respect to dialog states.

Actions for particular inputs, either speech inputs or application events, are specified in the dialog using the **action** command. Action commands should appear in the dialog specification after the particular input they are associated with, as part of the input components.

The action command is specified as follows:

(**action** *action-function*)

The *action-function* refers to a user defined routine will be called by the dialog manager each time the appropriate input is received. This provides the application

developer a mechanism to directly register a function to be called on a particular input, without requiring that the application developer write any code to parse or otherwise check the input. If a function is registered on a particular input it will only be called when that input is received.

The following example illustrates the use of the action command. Each input in the definition has an action function registered to dial the particular number requested by the user.

```
(prompt-text "Which number please")
(input "home"
           (action dial_home())
         "office"
           (action dial_office())
         "fax machine"
           (action dial_fax())
         "car phone"
           (action dial_car())
         "pager"
           (action dial_pager()))
```

Multiple action commands may be specified under any given input and they will be executed in the same order in which they are specified.

5.6.1 Conditional Actions

The conditional action command allows the specification of actions which are dependent on run time application state. Conditional actions are specified with the **cond-action** command in the following way:

(cond-action *("C-expr" action-function)*
("C-expr" action-function)
("C-expr" action-function) ...
(&default default-action-function))

The conditional action command works much the same as the *cond* function in Lisp. Each *C-expr* is evaluated in order until one evaluates to true. The *action-function* associated with that expression is executed, and no other C-expressions are evaluated. If no *C-expr* in the list evaluates to true, then the *action-function* associated with the special **&default** tag will be executed. If no default is given, and no expression returns true, then the command does nothing.

The conditional action command is specified along with the options to any speech input or application event. Any number of **cond-action** commands can be specified for a given input, intermixed with any number of regular **action** commands. All actions will still be executed in the proper order, whether or not they were part of a conditional command.

The following example shows how the **cond-action** command might be used in a dialog specification.

```
(prompt-text "Please enter your 4 digit id")
(input "$4_DIGIT_NUM"
          (action verify_id($1))
          (cond-action
            ("id_found_p()" lookup_info($1))
            (&default register_new_user($1)))
          (next MAIN_MENU))
```

In the example, the predicate `id_found_p()` is tested as part of the conditional statement, after the function `verify_id()` is called from the previous action command. If the predicate returns true then the user's id has been located in the database and the user is verified. If the predicate returns false, the application calls a function to register the new user.

The *C-expr* can be any valid C-language expression which evaluates to a boolean value. It can refer to any internal application functions or variables which are provided by the application developer and which will be called by the dialog manager to test the condition.

5.7 Next State

Once a particular input has been received and some action has been taken the dialog should advance to the next logical state in the dialog. The dialog specification language was developed especially to make such dialog state transitions very natural to specify.

The command used to specify the state transition from a particular input is the **next** command. It is specified as one of the input components for either speech inputs or application events in the following way:

(**next** *next-state*)

This defines the dialog transition to *next-state* when the particular input has been received. The state specified as *next-state* **must** be the name of a state which is defined somewhere else in the dialog specification. State names are case-sensitive, and a convention of using uppercase alphanumerics for state names is suggested. States need not be defined before they are referenced; the definition of a particular state can appear anywhere in the dialog specification. If *next-state* is not defined in the dialog then an error will result.

The **next** command may be specified anywhere in the list of options for an input, however the transition to the next state always happens after all the prompts and actions for that input have been executed. Transitioning to the next state is always the last thing to occur from any input.

Following is an example illustrating the use of the **next** command in a state definition.

```
(define-state TOP
  (prompt-text "Would you like directions"
               "or information on ticket prices")
  (input "directions [please]"
           (next DIRECTIONS)
         "ticket prices [please]"
           (next TICKETS)))
```

Only **one** next command is permitted for each input. If multiple next commands are attached to a particular input then only the *last one* specified will be considered. The others will be ignored. If no next command is given for an input then the default is for

the dialog to stay in the same state. If the dialog should loop back to the same state after an input is received then no next command need be specified for that input.

5.7.1 Conditional Next

The conditional next command provides the application with the ability to dynamically control state transitions at run time. Transitions need not be statically defined in the dialog. A transition can be dependent on internal application state which is tested each time through the state.

The conditional next command is called **cond-next** and has almost the identical syntax as the conditional action.

(**cond-next** *("C-expr" next-state)*
("C-expr" next-state)
("C-expr" next-state) ...
(&default default-next-state))

As with the conditional action, the **cond-next** command will test each *C-expr* in succession until one evaluates to true. The dialog will then transition to the state associated with that expression. If no expression returns true, then the default state, defined using the special **&default** tag, will be chosen. If no expression returns true and no default is specified then the dialog will remain in the current state.

Only one **next** or **cond-next** command may be specified for any speech input or application event. If multiple next commands are specified, then only the *last one* will have any effect; the others will be ignored.

The following example shows how the conditional next command can be used to direct the dialog transitions based on internal application state.

```
(prompt-text "Please speak your name and password")
(input "user $USER password $PASSWD"
         (action verify_user($1, $2))
         (cond-next
           ("password_ok()" MAIN_MENU)
           (&default INVALID_USER)))
```

Again, the expression used for *C-expr* can be any valid C language expression which evaluates to a boolean value. The functions referred to in the expression must be provided by the application developer and will be called by the dialog manager to test the conditional.

5.8 Summary of Input Components

Following is a summary of the input components which have been discussed in the preceding sections and some information on how they interact with each other.

1. Enable function, specified with the **enable** command, is tested before the recognizer starts listening for input in a given state. If the function returns false, the input to which the enable is attached will be disabled for that pass through the state. Otherwise the input is enabled. The default for any input without an enable function specified is for the input to remain active.

2. Action functions, specified with the **action** command, are executed immediately after the input is received. All action functions specified for an input will be executed in the order they are specified.

3. Next State, specified with the **next** command, defines the next state the dialog will transition to upon receiving the input. Only a single next command is allowed per input, and if none is given the dialog will remain in the same state. The transition is always the last thing to happen after an input is received, no matter where the next command appears in the list of options.

4. Prompts, specified with any of the three **prompt** commands, define feedback which should be presented to the user after receiving an input. Multiple prompts can be defined and will be issued in the order they are specified. All prompts, however, will be issued only after the action commands for that input are executed, no matter where they are specified in the list of options. It is not possible to interleave prompts with action functions using the dialog specification language. All actions will be invoked before any prompts are issued.

Chapter 6

Sample Application: A Voice Dialer

It is instructive at this point to present a complete application which was designed and developed entirely using the dialog specification language. This sample application is a *Voice Telephone Dialer*, which allows a user to speak a telephone number to dial, or speak a predefined name to reach some registered service.

6.1 Description of the Application

The application implements a voice dialing system. From the main menu, the user is allowed to say one of the following functions to dial out:

- **digit dial** *phone number*
- **speed dial** *name of service*

The application will confirm which number is being dialed and then place the call. The interface allows the user to speak the entire command from the main menu, or just say the keywords **digit dial** or **speed dial** and the system will prompt them for the remaining information.

Before getting to the main menu, the system might require the user to authenticate themselves by speaking a unique 4-digit PIN number. The system will look up the PIN number and check that it corresponds to a valid user. If the PIN number is not verified the user will not be allowed access to the system.

Since this is an application we would like first time users to be able to use without much advance instruction or training, it is important that the application provide appropriate context dependent help at each level of the dialog. Context dependent help indicates to the user what functions are available and what the user is allowed to say at each point in the dialog.

A typical dialog between the application and the user might look something like the following.

```
Computer: Welcome to the Voice Dialer.
          Please enter your 4-digit PIN number.
User: 1234
Computer: How may I help you?
User: help
Computer: The functions available from the main menu are
          digit dial and speed dial.
User: digit dial 555 1212
Computer: Dialing 555 1212
```

6.2 Designing the Dialog

The dialog specification language provides an intuitive interface to express an application in terms of dialog states. Breaking the problem down into states makes it easier to think about the flow of the dialog, and about which features should be available to the user at each stage.

Figure 6.1 shows what a simple 2 state model of the dialog might look like. In this example, the first state, **GET_ID**, is used to prompt the user for their PIN number. The dialog will loop back into that state until a valid PIN number is spoken. If the PIN has been verified, the dialog advances to state **MAIN_MENU** from where the user can access the available features of the system.

Since this simple model has only two dialog states, all of the options to the user must be made available in one of these two states. The **GET_ID** state serves a very

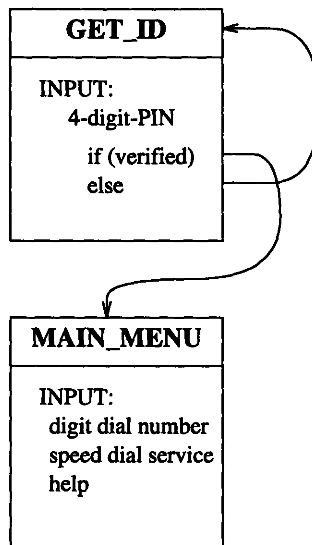


Figure 6.1: Simple Two State Dialog

specific purpose, to prompt for and verify the user's PIN number. The **MAIN_MENU** state provides the remaining system features.

From the main menu the following spoken commands are allowed:

- **digit dial** *phone number*
- **speed dial** *name of service*
- **help** *command name*
- **help**

Speaking one of the dial commands instructs the system to call the specified number for you. Speaking the help command plays the appropriate help file and returns back to the main menu.

Now, let's enhance the simple 2 state application to add a little more flexibility to the system. It might be desirable to allow the user to say only the keywords **digit dial** or **speed dial** and have the system prompt them for the remainder of the information. Figure 6.2 shows a more advanced 4 state version of our application.

In addition to the initial two states, there are now two additional states in the system — a state for digit dialing and a state for speed dialing. If, for example, from the main

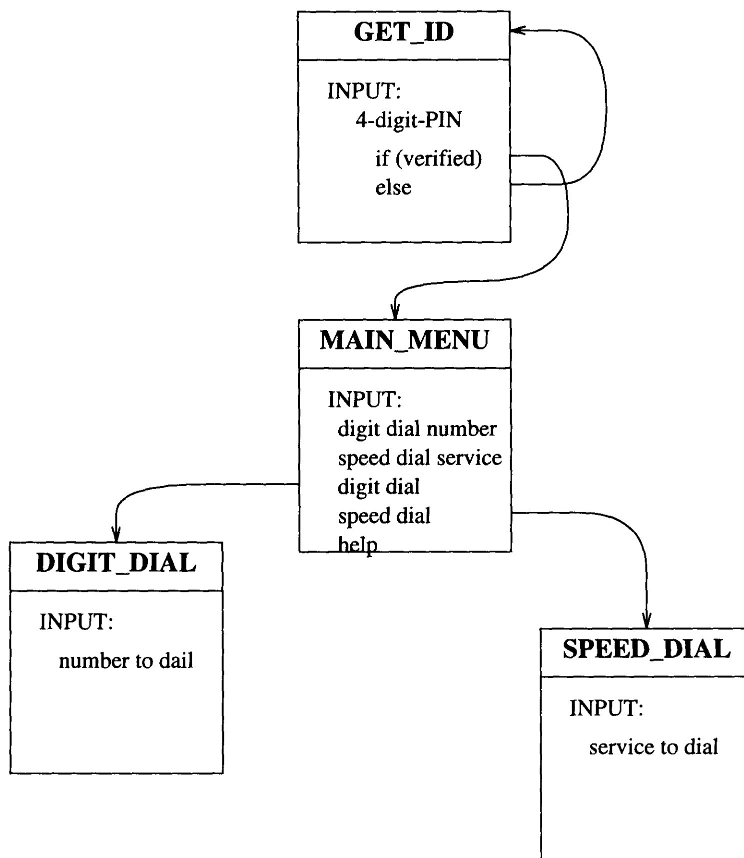


Figure 6.2: Four State Dialog

menu the user says **digit dial**, the dialog will transition to the **DIGIT_DIAL** state, which prompts the user for the number to call.

From the main menu, the allowable commands are as follows. The two new commands show the states where the dialog will transition to if those commands are spoken.

- **digit dial** *phone number*
- **speed dial** *name of service*
- **digit dial** : next state → DIGIT_DIAL
- **speed dial** : next state → SPEED_DIAL
- **help** *command name*
- **help**

We have reached a point in the discussion of the application design where it would be helpful to start expressing the dialog flow in the syntax of the dialog specification language. The next section builds the dialog specification for this application, explaining in detail each of the necessary steps, until the entire system has been specified.

6.3 Implementing the Dialog Specification

Let's start by building the dialog state definitions for the states designed in the previous section. The first state is the **GET_ID** state. The definition for this state might begin as follows:

```
(define-state GET_ID
  (prompt-file say-id.wav)
  (prompt-text "Please speak your 4-digit PIN.")
```

The first line defines the name of the state, **GET_ID**. Following the state definition we have two commands to issue the initial prompts in this state. The **prompt-file** command specifies the name of a pre-recorded audio file that we want the dialog manager to

play out at this point in the dialog. Presumably, the file specified here, **say-id.wav**, contains the digitized form of the text prompt specified in the command directly below. If the user will not be looking at a screen when using this application the **prompt-text** command may not even be necessary. In our sample application, however, we will always output both digitized speech prompts along with text prompts. Often the text prompts are useful for monitoring the application while running, or for debugging the dialog. It is the responsibility of the application developer to insure that each digitized prompt file contains a recorded version of the text prompt associated with it.

The prompts specified above will be issued to the user each time the dialog enters the **GET_ID** state. Once the prompts are played, the dialog needs to know what speech inputs to listen for. Continuing with the state definition, we add the definitions for valid speech inputs in this state:

```
(input
  "$PIN_4DIG"
    (action set_user_id($1))
    (cond-next
      ("verify_user_id()" MAIN_MENU)
      (&default GET_ID))
  "help"
    (prompt-file help-in-get-id.wav)
    (prompt-text "You are being asked to enter your
                  4-digit Personal ID Number.")
    (next GET_ID))
```

The **GET_ID** state allows only two possible speech inputs, either a *4-digit PIN number* or the word *help*. Let's carefully go over the specification for each of these inputs. The first input specification looks like this:

"$PIN_4DIG"

This refers to a non-terminal expression defined elsewhere in the grammar which expresses the subgrammar for a 4-digit PIN number. Remember that input specification lines can be any valid HARK Prototyper grammar specification. Here we are referencing the non-terminal expression **$PIN_4DIG**. This non-terminal should be defined by the

application developer in the user supplied grammar library. We will see where this definition is included later.

The next line in the dialog specification registers a function to be called when the PIN number input is received.

```
(action set_user_id($1))
```

The function **set_user_id** stores the user id number into an internal application variable for later access. This function is provided by the application developer in a special C file called **user.c**. The dialog manager knows to look in that file to find any user defined functions referenced in the dialog specification. The argument passed to the function, specified as **$1**, refers to the part of the recognition result which corresponds to the first non-terminal in the input line. The **$1** notation is a place-holder. When the input **$PIN_4DIG** is received by the dialog manager, its string value will be expanded and passed as an argument to **set_user_id**. For example, if the user speaks the id "1234", the action function will be invoked with the following argument.

```
(action set_user_id("ONE TWO THREE FOUR"))
```

The function is able to translate the string into a digit sequence and look up the PIN number in its database.

The next line in the dialog specification defines the dialog transition from this state. The dialog will advance to the specified next state after the actions for the input have been invoked. In this case, a conditional next statement is used to direct the dialog transition. The dialog should transition to one state if the PIN number is valid, and another if the PIN number is not valid.

```
(cond-next
   ("verify_user_id()" MAIN_MENU)
   (&default GET_ID))
```

This conditional next statement specifies two possible transitions. If the function **verify_user_id** returns true, or the integer value 1, the dialog will continue to the state MAIN_MENU. If the predicate function returns false, the default definition is for the dialog to remain in the state GET_ID. This is noted by the special label **&default**, which indicates what to do if none of the previous conditions return true.

The conditional next command specifies exactly the dialog logic we want at this point. If the user enters a valid PIN number we want them to gain access to the main menu. If they do not enter a valid number then we want to continue to reprompt for the PIN number until a valid one is given.

The second input to this state is a little more straightforward. The "help" input line specifies that the user is allowed to speak the word "help" within this state to get more information. If the help input is received, the dialog manager will play out the help audio file, along with its corresponding text, and then return back to the GET_ID state. The next state specification

```
(next GET_ID)
```

tells the dialog to stay in the GET_ID state. Once the user has listened to the help message, the dialog loops back into the same state and reprompts the user to enter their PIN number. In this case, the next command could have been omitted since the next dialog state is the same as the one the dialog is currently in. For any input, if the next command is omitted, the default is for the dialog to remain in the same state it is currently in.

The next state definition in the dialog is the definition for the MAIN_MENU state. This state should include the inputs which allow the user to access the dialing features of the system. The definition of the MAIN_MENU state begins with a prompt, asking the user to speak their request.

```
(define-state MAIN_MENU
  (prompt-file main-prompt.wav)
  (prompt-text "Please speak your request")
```

The beginning of this state definition is much like the previous state definition. The name of the state is defined as **MAIN_MENU** and the first two commands in the definition specify the initial prompt, in both text and digitized speech formats. The prompt *"Please speak your request"* will be issued to the user each time the dialog enters the MAIN_MENU state.

The input specification for the main menu is partially shown below:

```
  (input "digit dial $TEL_7DIG"
```

```
            (action play_digit_response(PLAY_DIAL, $1))
            (next TRANSFER_OUT)
          "digit dial"
            (next DIGIT_DIAL)
          "speed dial pizza"
            (prompt-file x-pizza.wav)
            (prompt "Ringing Pizza Hut.")
            (next TRANSFER_OUT)
          "speed dial taxi"
            (prompt-file x-taxi.wav)
            (prompt "Ringing Yellow Cab.")
            (next TRANSFER_OUT)
          "speed dial"
            (next SPEED_DIAL)
          "help"
            (prompt-file help-main.wav))
```

The input specification defines the possible spoken commands available to the user from the main menu, as described in the previous section. In addition to the speech input specification, the main menu contains the following event input specification as well:

```
(event "%TIMEOUT"
            (prompt-file help-main.wav)
        "RESET"
            (next TOP)
        "%REJECTION"
            (prompt-file help-main.wav))
```

The event specification for this state defines three possible events. The **%TIMEOUT** event indicates what should be done if the user does not speak when prompted. In this case the timeout event triggers a help prompt to be played out. The absence of a dialog transition specification implies that the dialog should return back to the same state after the prompt is played. The **%REJECTION** event specifies what to do if the user speaks

a command which is not understood by the recognizer. In this case the rejection event also triggers the playing of a help prompt, and the dialog returns back to the main menu state. Finally, the **RESET** state is a user defined application event label used to reset the dialog back to the top level state. This event may be triggered by the application when, for example, the user hangs up on the system or is connected to another number.

The remaining state definitions follow the same format as the ones previously described. Appendix A contains a complete listing of the dialog specification for the voice dialer application. Following the dialog specification listing are listings of the six output files which were generated by the dialog compiler when the voice dialer specification was compiled.

Chapter 7

Future Directions

This chapter describes some of the enhancements which can be made to this system, which, due to time and resource limitations, were not included within the scope of this thesis. It also outlines some future directions for additional work in this field.

7.1 Graphical User Interface

Many commercially available application generation systems provide some type of graphical user interface to their system to improve the usability of the tool. Graphical interfaces aid the application developer by providing a means to visualize the application control flow, interdependencies among components, and a variety of other aspects of the application.

The application development process using the dialog specification language would benefit from the addition of a graphical user interface (GUI). One aspect of the graphical interface might be a graphical state editor. Instead of specifying the dialog specification using the current text syntax, a GUI built over the language might allow the developer to draw the dialog state diagram using boxes to represent states and lines to represent transitions between states. The pictorial representation of the dialog states could be interpreted automatically into the proper text syntax for the language. The developer would be able to click the mouse on a particular state and edit the inputs and actions for that state.

7.2 Interactive Development Environment

A more complete interactive development environment could be provided that would offer additional services to the developer to aid in the design and development of spoken dialog systems. Two such services which would enhance the development environment are a dialog simulator, and a voice prompt manager.

7.2.1 Dialog Simulation

The dialog specification language provides a convenient syntax for specifying the flow of a spoken dialog application. Often, however, a dialog design might have to go through several iterations before it reaches its final form. Many human factors issues come into play in the design of a dialog which often cannot be anticipated during the initial development cycle of the application.

A dialog simulator is a utility that would take a dialog specification and simulate the flow of the dialog, without executing any of the application functions. The dialog simulator may not even require interfacing with a speech recognizer; initial dialog simulations could be done entirely using text instead of voice inputs, although for realistic simulations voice input would be preferred. This would allow the developer to iterate over the design of the dialog, getting a real sense of the "look and feel" of the dialog, without having to construct the entire application. The prompts to the user might also be text instead of voice, so that the dialog could be refined to its final state before any of the voice prompts were recorded. Depending on the size of the system, recording digitized prompts can be a large task in itself, and therefore finalizing the design of the dialog prior to recording prompts can result in a substantial time savings.

7.2.2 Prompt Manager

A complex interactive dialog application may require a fair number of recorded voice prompts to use for feedback at various points in the dialog. The command syntax of the dialog specification language supports playing back recorded voice prompts. However, the application developer must provide the name of the prompt file in the specification so the dialog manager knows which prompt to play.

When designing a dialog, and even when simulating a dialog under development, it is often easier to deal with the prompts as text, rather than as file names. Embedding file names into the specification can make it difficult to iterate through the dialog design because often the file names alone do not provide enough detail as to the exact words recorded in the file.

A helpful utility is one which could manage the recording of the voice prompts in the system. The developer would specify all the prompts as text. Before compiling the dialog specification, the prompt manager could replace the text prompts with automatically generated file names. The prompt manager would then prepare a table mapping file names to text transcriptions, and could easily manage the collection of each of the voice prompts.

Prototype dialogs could be developed using only text prompts. Once the dialog was in its final form, the prompt manager would convert the text prompts to digitized files, insuring that all the necessary prompts were recorded.

7.3 Parameterized States

A powerful enhancement to the dialog specification language would be the ability to define parameterized, or generic states. A parameterized state is one in which not all the definitions are constant. For example, instead of having fixed definitions for all the transitions from a state, some may be defined as variables. The value of the variables are passed into the state from the state it was called from.

Parameterized states can be thought of more like subroutine calls rather than static transitions. When a parameterized state is entered, the state which called it must pass in values for the undefined variables in the state.

Consider the following example of a confirmation state. The confirmation state prompts the user to say yes or no, and then proceeds to the next state accordingly. The static definition for such a state may look as follows:

```
(define-state CONFIRM
  (prompt "Is this correct?")
  (input "yes"
```

```
            (next CONFIRM_YES)
          "no"
            (next CONFIRM_NO)))
```

This state may be useful in the dialog in several different places, for example to confirm a phone number entry, or a date, or any number of things which require confirmation. If the transition states **CONFIRM_YES** and **CONFIRM_NO** are fixed then we cannot call this state from different places in the dialog. A separate copy of the state needs to be defined for each place in the dialog where confirmation is required. However, if the transition states were allowed to be variable the dialog specification could access this state from various places in the dialog and maintain the correct context of the dialog.

A reference to a parameterized state could directly include the assignments for all variables in the state. A call to such a state might look as follows:

```
(next CONFIRM
    (CONFIRM_YES = MAIN_MENU,
     CONFIRM_NO  = RETRY))
```

This reference indicates that if the **CONFIRM** state is called from this point then if the user says yes the dialog should progress to the main menu. If the user answers no the dialog goes to a retry state. From other parts in the dialog these transitions may be defined differently.

In addition to parameterizing the transitions from a state, it is also possible to support parameterized actions and prompts as well.

7.4 Talk Ahead

"Talk ahead" is an advanced dialog feature which permits an experienced user of a system to speak a command or a series of commands ahead of the prompts. In a typical IVR application, for example, the system can contain many levels of voice prompted menus which the user must traverse in order to reach the desired information.

Figure 7.1 shows an example of a simple menu structure for an IVR application designed for phone banking. From the main menu the user is given a list of broad

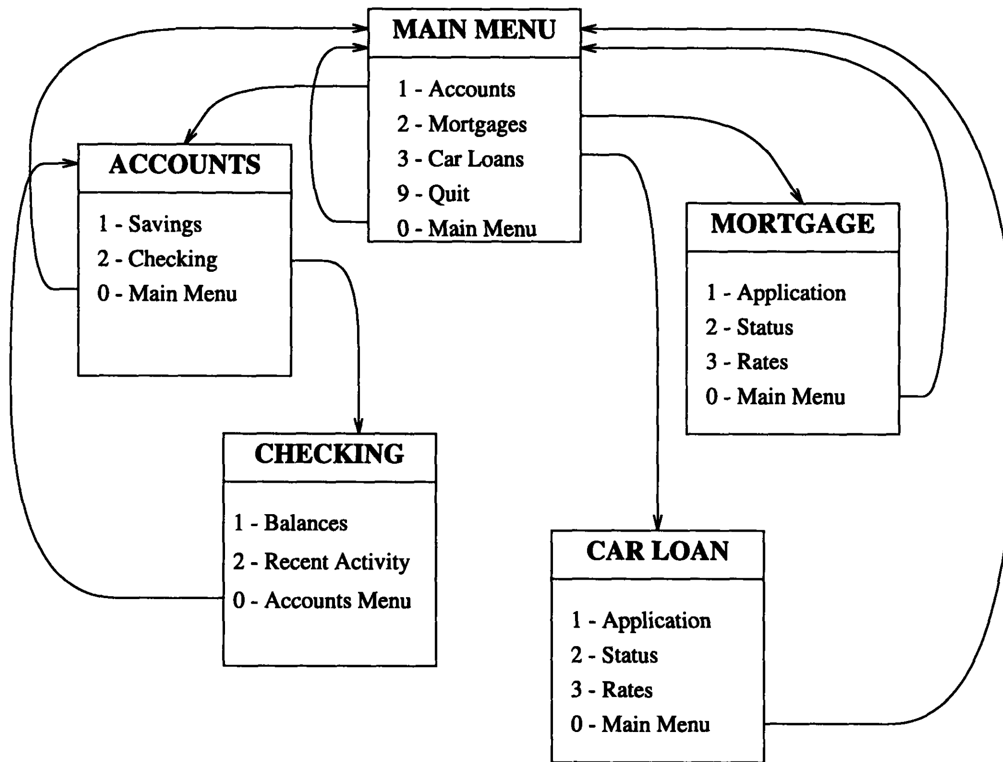


Figure 7.1: Banking IVR Application

categories to choose from — such as "accounts", "mortgages", or "car loans". When the user chooses one of those categories the dialog advances to another state and presents the user with a more specific list of choices. For example, if the user chose "accounts" from the main menu, the next menu asks the user to choose "savings"or "checking". From there the user may go through yet another menu asking "balances" or "recent activity".

In a touchtone based IVR application, where the user presses telephone touchtone keys to select choices, it is often possible for an experienced user to enter an entire sequence of numbers quickly before waiting for the prompts. An experienced user, for example, may know that the sequence [1 - 2 - 1] is the right sequence to get to checking account balances. If users access the same information frequently they can remember the appropriate sequence of key presses and not have to wait to hear the prompts.

In a speech based IVR system, however, it is more difficult to provide this capability to the user. The speech recognizer is based on a phrase structured grammar which is modified dynamically to reflect only the valid inputs of the current dialog state. This

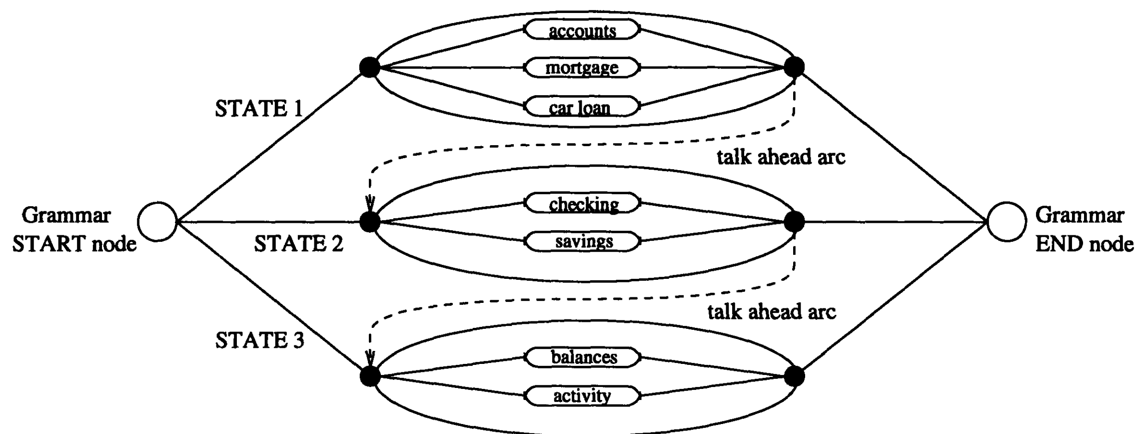


Figure 7.2: Grammar with Talk Ahead Enabled

means that the recognizer will only accept choices from the current menu. When the dialog explicitly changes state, the recognizer is then reconfigured to accept speech input from the next menu.

A "talk ahead" feature for this system allows the user to speak "account checking balances" as one command from the main menu, and traverses through the dialog states as if each command were given as a separate utterance. The experienced user can give a single command from the main menu and reach the desired information without having to listen to three levels of menu prompts.

Implementing such a feature into the dialog specification language requires a special capability of the speech recognizer which allows any two arbitrary nodes in the grammar to be dynamically "wired together". Using this capability, the dialog compiler would be able to "chain" several states together in the grammar, effectively allowing the inputs for those states to be chained together in a series as one spoken utterance.

Figure 7.2 shows an example of the top level grammar for such an application with the talk ahead grammar wires installed. From the main menu, only STATE 1 is considered active, however the dashed line in the grammar, or wire, provides an alternate path from the end of STATE 1 to the beginning of STATE 2. If the user chains commands from the two states together, the recognizer will follow the wired path through the grammar. Likewise, the wire from STATE 2 to STATE 3 allows those inputs to be chained together as well.

In the dialog specification, an additional command in the definition of a state would indicate to the dialog compiler that two states are to be "wired" together. The syntax for such a feature might look as follows:

```
(define-state MAIN_MENU
  (enable-talk-ahead ACCOUNTS)
  (input "accounts"
           (next ACCOUNTS)
         "mortgages"
           (next MORTGAGES)
         "car loans"
           (next CAR_LOANS)))

(define-state ACCOUNTS
  (enable-talk-ahead CHECKING SAVINGS)
  (input "checking"
           (next CHECKING)
         "savings"
           (next SAVINGS)))
```

7.5 Integrating Speech into a Graphical User Interface

There are many GUI building toolkits available which provide a fast means of implementing graphical interfaces for applications. These GUI builders often allow the developer to draw a representation of the GUI as it should be presented to the user. The graphical representation of the GUI is then interpreted by the GUI builder which generates the appropriate application code to implement the specified graphical interface.

The paradigm for specifying a graphical interface usually involves associating callback functions with various application events. Consider for example, a graphical dialog box containing two buttons, **ok** and **cancel**. The developer can associate a different callback function to each of the button events. When the user clicks the mouse on one of the two buttons, a button event is generated which causes the appropriate callback function to be

invoked.

The dialog specification language uses a similar paradigm for specifying a speech interface for an application. Action functions, like callbacks, are registered with specific speech inputs in the dialog specification and are invoked when that speech input is received.

Since the dialog specification already supports application event inputs as part of the dialog along with speech inputs, it would require no extension of the language to be able to specify a fully speech enabled graphical interface using the dialog specification language. Graphical application events can be listed within a state definition along with corresponding speech inputs. This would allow the user to then either click the mouse to choose an option, or to speak the option. Both methods would have the same effect on the flow of the application.

The dialog specification for the simple confirmation dialog box discussed above might look as follows:

```
(define-state CONFIRM_BOX
  (prompt "Please confirm or cancel selection")
  (input "ok"
           (action do_choice())
           (next CHOICE_CONFIRMED)
         "cancel"
           (action cancel_choice())
           (next CHOICE_CANCELED))
  (event "OK_BUTTON_EVENT"
           (action do_choice())
           (next CHOICE_CONFIRMED)
         "CANCEL_BUTTON_EVENT"
           (action cancel_choice())
           (next CHOICE_CANCELED)))
```

Both the speech input “ok” and the ok-button event have the same registered callback and the same state transition. The application will respond in the same way whether the user selects a choice with voice or with the mouse.

The capability of enabling a graphical interface with speech input using an integrated specification language is a powerful enhancement to current GUI builder tools. Speech input in a graphical interface can be used to select fields on a form, choose options from a list, or even input text into a text window, for example to select a filename.

Appendix A

Voice Dialer Application Source

This appendix includes the full dialog specification for the voice dialer sample application presented in chapter 6. Following the listing of the dialog specification are listings of each of the files generated by the dialog compiler from this specification. Six files are generated by the compiler — the first one listed is the HARK recognition grammar file, the remaining files are C application source and header files which implement the spoken dialog application defined by the dialog specification.

voice_dialer.dlg

```
(define-dialog VOICE_DIALER

	(define-state TOP
		(prompt-file welcome.wav)
		(prompt "Welcome to the Voice Calling Demo")
		(define-event "%NULL"
				(next GET_ID)))

	(define-state GET_ID
		(prompt-file say-id.wav)
		(prompt "Please speak your 4-digit id")
		(define-input "$PIN_4DIG"
				(action set_user_id($1))
				(cond-next
				    ("verify_user_id()" MAIN_MENU)
				    (&default REPROMPT_ID))
			       "help"
				(prompt-file help-in-get-id.wav)
				(next GET_ID))
		(define-event "%TIMEOUT"
				(prompt-file help-in-get-id.wav)
			       "RESET"
				(next TOP)))

	(define-state REPROMPT_ID
		(prompt-file not-valid-id.wav)
		(prompt "Sorry, that id number is not valid")
		(define-event "%NULL"
				(next GET_ID)))

	(define-state MAIN_MENU
		(prompt-file main-prompt.wav)
		(prompt "How may we help you")
		(define-input "digit dial $TEL_7DIG"
				(action play_digit_response(PLAY_DIAL, $1))
				(action wait_for_touchtone())
				(next TRANSFER_OUT)
			       "digit dial"
				(next DIGIT_DIAL)
			       "speed dial pizza"
				(prompt-file x-pizza.wav)
				(prompt "Ringing Pizza Hut.")
				(action wait_for_touchtone())
				(next TRANSFER_OUT)
			       "speed dial taxi"
				(prompt-file x-taxi.wav)
				(prompt "Ringing Yellow Cab.")
				(action wait_for_touchtone())
				(next TRANSFER_OUT)
			       "speed dial rental car"
				(prompt-file x-rental.wav)
				(prompt "Ringing Hertz Rental Cars.")
				(action wait_for_touchtone())
				(next TRANSFER_OUT)
			       "speed dial road service"
				(prompt-file x-garage.wav)
				(prompt "Ringing Jerry's Garage.")
				(action wait_for_touchtone())
				(next TRANSFER_OUT)
			       "speed dial police"
```

```
                            (prompt-file x-police.wav)
                            (prompt "Ringing the Police.")
                            (action wait_for_touchtone())
                            (next TRANSFER_OUT)
                          "speed dial hospital"
                            (prompt-file x-hospital.wav)
                            (prompt "Ringing the Hospital.")
                            (action wait_for_touchtone())
                            (next TRANSFER_OUT)
                          "speed dial"
                            (next SPEED_DIAL)
                          "help"
                            (prompt-file help-main.wav)
                          "menu"
                            (prompt-file help-menu.wav)
                          "help digit dial"
                            (prompt-file help-digit-dial.wav)
                          "help speed dial"
                            (prompt-file help-speed-dial.wav)
                            (prompt-file help-speed-dial-list.wav))
        (define-event "%TIMEOUT"
                            (prompt-file help-main.wav)
                          "RESET"
                            (next TOP)
                          "%REJECTION"
                            (prompt-file help-main.wav)))

(define-state TRANSFER_OUT
        (define-event "TOUCHTONE"
                            (next MAIN_MENU)
                          "RESET"
                            (next TOP)))

(define-state DIGIT_DIAL
        (prompt-file number.wav)
        (prompt "Number please")
        (define-input "$TEL_7DIG"
                            (action play_digit_response(PLAY_DIAL, $1))
                            (action wait_for_touchtone())
                            (next TRANSFER_OUT)
                          "help"
                            (prompt-file help-in-digit-dial.wav)
                          "cancel"
                            (prompt-file cancelled.wav)
                            (next MAIN_MENU))
        (define-event "%TIMEOUT"
                            (prompt-file help-in-digit-dial.wav)
                          "RESET"
                            (next TOP)
                          "%REJECTION"
                            (prompt-file try-again.wav)
                            (prompt "Please try again")))

(define-state SPEED_DIAL
        (prompt-file names-short-1.wav)
        (prompt "Name please")
        (define-input "pizza"
                            (prompt-file x-pizza.wav)
                            (action wait_for_touchtone())
                            (next TRANSFER_OUT)
                          "taxi"
                            (prompt-file x-taxi.wav)
```

```
                        (action wait_for_touchtone())
                        (next TRANSFER_OUT)
                      "rental car"
                        (prompt-file x-rental.wav)
                        (action wait_for_touchtone())
                        (next TRANSFER_OUT)
                      "road service"
                        (prompt-file x-garage.wav)
                        (action wait_for_touchtone())
                        (next TRANSFER_OUT)
                      "police"
                        (prompt-file x-police.wav)
                        (action wait_for_touchtone())
                        (next TRANSFER_OUT)
                      "hospital"
                        (prompt-file x-hospital.wav)
                        (action wait_for_touchtone())
                        (next TRANSFER_OUT)
                      "help"
                        (prompt-file help-in-speed-dial.wav)
                      "list"
                        (prompt-file help-speed-dial-list.wav)
                      "cancel"
                        (prompt-file cancelled.wav)
                        (next MAIN_MENU))
        (define-event "%TIMEOUT"
                        (prompt-file help-in-speed-dial.wav)
                      "RESET"
                        (next TOP)
                      "%REJECTION"
                        (prompt-file try-again.wav)
                        (prompt "Please try again")))
)
```

gen_grammar.hg

```
# Application Grammar - Machine Generated #
<START> $TOP;
$STATE00: @NULL;
$STATE01: @INPUT0100 $_NT010001 _SILENCE_/TAG0100
        | @INPUT0101 help _SILENCE_/TAG0101
        ;
$STATE02: @NULL;
$STATE03: @INPUT0300 digit dial $_NT030001 _SILENCE_/TAG0300
        | @INPUT0301 digit dial _SILENCE_/TAG0301
        | @INPUT0302 speed dial pizza _SILENCE_/TAG0302
        | @INPUT0303 speed dial taxi _SILENCE_/TAG0303
        | @INPUT0304 speed dial rental car _SILENCE_/TAG0304
        | @INPUT0305 speed dial road service _SILENCE_/TAG0305
        | @INPUT0306 speed dial police _SILENCE_/TAG0306
        | @INPUT0307 speed dial hospital _SILENCE_/TAG0307
        | @INPUT0308 speed dial _SILENCE_/TAG0308
        | @INPUT0309 help _SILENCE_/TAG0309
        | @INPUT0310 menu _SILENCE_/TAG0310
        | @INPUT0311 help digit dial _SILENCE_/TAG0311
        | @INPUT0312 help speed dial _SILENCE_/TAG0312
        ;
$STATE04: @NULL;
$STATE05: @INPUT0500 $_NT050001 _SILENCE_/TAG0500
        | @INPUT0501 help _SILENCE_/TAG0501
        | @INPUT0502 cancel _SILENCE_/TAG0502
        ;
$STATE06: @INPUT0600 pizza _SILENCE_/TAG0600
        | @INPUT0601 taxi _SILENCE_/TAG0601
        | @INPUT0602 rental car _SILENCE_/TAG0602
        | @INPUT0603 road service _SILENCE_/TAG0603
        | @INPUT0604 police _SILENCE_/TAG0604
        | @INPUT0605 hospital _SILENCE_/TAG0605
        | @INPUT0606 help _SILENCE_/TAG0606
        | @INPUT0607 list _SILENCE_/TAG0607
        | @INPUT0608 cancel _SILENCE_/TAG0608
        ;
$TOP    : @STATE00        $STATE00
        | @STATE01        $STATE01
        | @STATE02        $STATE02
        | @STATE03        $STATE03
        | @STATE04        $STATE04
        | @STATE05        $STATE05
        | @STATE06        $STATE06
        ;

# Non-Terminal Expansions
$_NT010001: $PIN_4DIG {:region=1};
$_NT030001: $TEL_7DIG {:region=1};
$_NT050001: $TEL_7DIG {:region=1};
```

dialog_global.h

```
/*** Global Definitions - Machine Generated ***/
#define TOP          0
#define GET_ID       1
#define REPROMPT_ID  2
#define MAIN_MENU    3
#define TRANSFER_OUT 4
#define DIGIT_DIAL   5
#define SPEED_DIAL   6

extern int CURRENT_STATE;

extern char NonTerminalWords[25][256];
```

state_table.h

```
/*** State Table Definitions - Machine Generated ***/
#include <stdio.h>
#include "dialog_global.h"
#include "dialog_gen.h"
#include "input_table.h"

struct StateTableType {
  int                 StateID;
  DialogCallbackProc  InitProc;
  EventType           *Inputs;
  EventType           *Events;
} StateTable[] = {
  {0, init00, InputTable00, EventTable00},
  {1, init01, InputTable01, EventTable01},
  {2, init02, InputTable02, EventTable02},
  {3, init03, InputTable03, EventTable03},
  {4, init04, InputTable04, EventTable04},
  {5, init05, InputTable05, EventTable05},
  {6, init06, InputTable06, EventTable06},
  {-1, NULL, NULL, NULL}
};
```

input_table.h

```
/*** Input Table Definitions - Machine Generated ***/
typedef void (*DialogCallbackProc) ();

typedef struct eType {
  char                  *LookupTag;
  DialogCallbackProc    ActionProc;
  DialogCallbackProc    NextProc;
} EventType;

EventType InputTable00[] = {
  {NULL, NULL, NULL}
};

EventType EventTable00[] = {
  {"%NULL", actionE0000, nextE0000},
  {NULL, NULL, NULL}
};

EventType InputTable01[] = {
  {"TAG0100", action0100, next0100},
  {"TAG0101", action0101, next0101},
  {NULL, NULL, NULL}
};

EventType EventTable01[] = {
  {"%TIMEOUT", actionE0100, nextE0100},
  {"RESET", actionE0101, nextE0101},
  {NULL, NULL, NULL}
};

EventType InputTable02[] = {
  {NULL, NULL, NULL}
};

EventType EventTable02[] = {
  {"%NULL", actionE0200, nextE0200},
  {NULL, NULL, NULL}
};

EventType InputTable03[] = {
  {"TAG0300", action0300, next0300},
  {"TAG0301", action0301, next0301},
  {"TAG0302", action0302, next0302},
  {"TAG0303", action0303, next0303},
  {"TAG0304", action0304, next0304},
  {"TAG0305", action0305, next0305},
  {"TAG0306", action0306, next0306},
  {"TAG0307", action0307, next0307},
  {"TAG0308", action0308, next0308},
  {"TAG0309", action0309, next0309},
  {"TAG0310", action0310, next0310},
  {"TAG0311", action0311, next0311},
  {"TAG0312", action0312, next0312},
  {NULL, NULL, NULL}
};

EventType EventTable03[] = {
  {"%TIMEOUT", actionE0300, nextE0300},
  {"RESET", actionE0301, nextE0301},
  {"%REJECTION", actionE0302, nextE0302},
```

```
  {NULL, NULL, NULL}
};

EventType InputTable04[] = {
  {NULL, NULL, NULL}
};

EventType EventTable04[] = {
  {"TOUCHTONE", actionE0400, nextE0400},
  {"RESET", actionE0401, nextE0401},
  {NULL, NULL, NULL}
};

EventType InputTable05[] = {
  {"TAG0500", action0500, next0500},
  {"TAG0501", action0501, next0501},
  {"TAG0502", action0502, next0502},
  {NULL, NULL, NULL}
};

EventType EventTable05[] = {
  {"%TIMEOUT", actionE0500, nextE0500},
  {"RESET", actionE0501, nextE0501},
  {"%REJECTION", actionE0502, nextE0502},
  {NULL, NULL, NULL}
};

EventType InputTable06[] = {
  {"TAG0600", action0600, next0600},
  {"TAG0601", action0601, next0601},
  {"TAG0602", action0602, next0602},
  {"TAG0603", action0603, next0603},
  {"TAG0604", action0604, next0604},
  {"TAG0605", action0605, next0605},
  {"TAG0606", action0606, next0606},
  {"TAG0607", action0607, next0607},
  {"TAG0608", action0608, next0608},
  {NULL, NULL, NULL}
};

EventType EventTable06[] = {
  {"%TIMEOUT", actionE0600, nextE0600},
  {"RESET", actionE0601, nextE0601},
  {"%REJECTION", actionE0602, nextE0602},
  {NULL, NULL, NULL}
};
```

dialog_gen.h

```
/*** Dialog Definitions - Machine Generated ***/
extern void actionE0000();
extern void nextE0000();
extern void init00();
extern void action0100();
extern void next0100();
extern void action0101();
extern void next0101();
extern void actionE0100();
extern void nextE0100();
extern void actionE0101();
extern void nextE0101();
extern void init01();
extern void actionE0200();
extern void nextE0200();
extern void init02();
extern void action0300();
extern void next0300();
extern void action0301();
extern void next0301();
extern void action0302();
extern void next0302();
extern void action0303();
extern void next0303();
extern void action0304();
extern void next0304();
extern void action0305();
extern void next0305();
extern void action0306();
extern void next0306();
extern void action0307();
extern void next0307();
extern void action0308();
extern void next0308();
extern void action0309();
extern void next0309();
extern void action0310();
extern void next0310();
extern void action0311();
extern void next0311();
extern void action0312();
extern void next0312();
extern void actionE0300();
extern void nextE0300();
extern void actionE0301();
extern void nextE0301();
extern void actionE0302();
extern void nextE0302();
extern void init03();
extern void actionE0400();
extern void nextE0400();
extern void actionE0401();
extern void nextE0401();
extern void init04();
extern void action0500();
extern void next0500();
extern void action0501();
extern void next0501();
extern void action0502();
extern void next0502();
```

```
extern void actionE0500();
extern void nextE0500();
extern void actionE0501();
extern void nextE0501();
extern void actionE0502();
extern void nextE0502();
extern void init05();
extern void action0600();
extern void next0600();
extern void action0601();
extern void next0601();
extern void action0602();
extern void next0602();
extern void action0603();
extern void next0603();
extern void action0604();
extern void next0604();
extern void action0605();
extern void next0605();
extern void action0606();
extern void next0606();
extern void action0607();
extern void next0607();
extern void action0608();
extern void next0608();
extern void actionE0600();
extern void nextE0600();
extern void actionE0601();
extern void nextE0601();
extern void actionE0602();
extern void nextE0602();
extern void init06();
extern void init_grammar();
```

dialog_gen.c

```
/*** Dialog Routines - Machine Generated ***/
#include "dialog_global.h"

#include "user.h"

void actionE0000()
{
}
void nextE0000()
{
grammar_disable(state_arc(CURRENT_STATE));
CURRENT_STATE = GET_ID;
}
void init00()
{
grammar_enable(state_arc(CURRENT_STATE));
play_wavfile("welcome.wav");
printf("Welcome to the Voice Calling Demo\n");
}
void action0100()
{
set_user_id(NonTerminalWords[1]);
}
void next0100()
{
grammar_disable(state_arc(CURRENT_STATE));
if (verify_user_id())
        CURRENT_STATE = MAIN_MENU;
else    CURRENT_STATE = REPROMPT_ID;
}
void action0101()
{
}
void next0101()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("help-in-get-id.wav");
CURRENT_STATE = GET_ID;
}
void actionE0100()
{
}
void nextE0100()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("help-in-get-id.wav");
}
void actionE0101()
{
}
void nextE0101()
{
grammar_disable(state_arc(CURRENT_STATE));
CURRENT_STATE = TOP;
}
void init01()
{
grammar_enable(state_arc(CURRENT_STATE));
play_wavfile("say-id.wav");
printf("Please speak your 4-digit id\n");
```

```
}
void actionE0200()
{
}
void nextE0200()
{
grammar_disable(state_arc(CURRENT_STATE));
CURRENT_STATE = GET_ID;
}
void init02()
{
grammar_enable(state_arc(CURRENT_STATE));
play_wavfile("not-valid-id.wav");
printf("Sorry, that id number is not valid\n");
}
void action0300()
{
play_digit_response(PLAY_DIAL, NonTerminalWords[1]);
wait_for_touchtone();
}
void next0300()
{
grammar_disable(state_arc(CURRENT_STATE));
CURRENT_STATE = TRANSFER_OUT;
}
void action0301()
{
}
void next0301()
{
grammar_disable(state_arc(CURRENT_STATE));
CURRENT_STATE = DIGIT_DIAL;
}
void action0302()
{
wait_for_touchtone();
}
void next0302()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("x-pizza.wav");
printf("Ringing Pizza Hut.\n");
CURRENT_STATE = TRANSFER_OUT;
}
void action0303()
{
wait_for_touchtone();
}
void next0303()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("x-taxi.wav");
printf("Ringing Yellow Cab.\n");
CURRENT_STATE = TRANSFER_OUT;
}
void action0304()
{
wait_for_touchtone();
}
void next0304()
{
grammar_disable(state_arc(CURRENT_STATE));
```

```
play_wavfile("x-rental.wav");
printf("Ringing Hertz Rental Cars.\n");
CURRENT_STATE = TRANSFER_OUT;
}
void action0305()
{
wait_for_touchtone();
}
void next0305()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("x-garage.wav");
printf("Ringing Jerry's Garage.\n");
CURRENT_STATE = TRANSFER_OUT;
}
void action0306()
{
wait_for_touchtone();
}
void next0306()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("x-police.wav");
printf("Ringing the Police.\n");
CURRENT_STATE = TRANSFER_OUT;
}
void action0307()
{
wait_for_touchtone();
}
void next0307()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("x-hospital.wav");
printf("Ringing the Hospital.\n");
CURRENT_STATE = TRANSFER_OUT;
}
void action0308()
{
}
void next0308()
{
grammar_disable(state_arc(CURRENT_STATE));
CURRENT_STATE = SPEED_DIAL;
}
void action0309()
{
}
void next0309()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("help-main.wav");
}
void action0310()
{
}
void next0310()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("help-menu.wav");
}
void action0311()
```

```
{
}
void next0311()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("help-digit-dial.wav");
}
void action0312()
{
}
void next0312()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("help-speed-dial.wav");
play_wavfile("help-speed-dial-list.wav");
}
void actionE0300()
{
}
void nextE0300()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("help-main.wav");
}
void actionE0301()
{
}
void nextE0301()
{
grammar_disable(state_arc(CURRENT_STATE));
CURRENT_STATE = TOP;
}
void actionE0302()
{
}
void nextE0302()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("help-main.wav");
}
void init03()
{
grammar_enable(state_arc(CURRENT_STATE));
play_wavfile("main-prompt.wav");
printf("How may we help you\n");
}
void actionE0400()
{
}
void nextE0400()
{
grammar_disable(state_arc(CURRENT_STATE));
CURRENT_STATE = MAIN_MENU;
}
void actionE0401()
{
}
void nextE0401()
{
grammar_disable(state_arc(CURRENT_STATE));
CURRENT_STATE = TOP;
}
```

```
void init04()
{
grammar_enable(state_arc(CURRENT_STATE));
}
void action0500()
{
play_digit_response(PLAY_DIAL, NonTerminalWords[1]);
wait_for_touchtone();
}
void next0500()
{
grammar_disable(state_arc(CURRENT_STATE));
CURRENT_STATE = TRANSFER_OUT;
}
void action0501()
{
}
void next0501()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("help-in-digit-dial.wav");
}
void action0502()
{
}
void next0502()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("cancelled.wav");
CURRENT_STATE = MAIN_MENU;
}
void actionE0500()
{
}
void nextE0500()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("help-in-digit-dial.wav");
}
void actionE0501()
{
}
void nextE0501()
{
grammar_disable(state_arc(CURRENT_STATE));
CURRENT_STATE = TOP;
}
void actionE0502()
{
}
void nextE0502()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("try-again.wav");
printf("Please try again\n");
}
void init05()
{
grammar_enable(state_arc(CURRENT_STATE));
play_wavfile("number.wav");
printf("Number please\n");
}
```

```
void action0600()
{
wait_for_touchtone();
}
void next0600()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("x-pizza.wav");
CURRENT_STATE = TRANSFER_OUT;
}
void action0601()
{
wait_for_touchtone();
}
void next0601()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("x-taxi.wav");
CURRENT_STATE = TRANSFER_OUT;
}
void action0602()
{
wait_for_touchtone();
}
void next0602()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("x-rental.wav");
CURRENT_STATE = TRANSFER_OUT;
}
void action0603()
{
wait_for_touchtone();
}
void next0603()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("x-garage.wav");
CURRENT_STATE = TRANSFER_OUT;
}
void action0604()
{
wait_for_touchtone();
}
void next0604()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("x-police.wav");
CURRENT_STATE = TRANSFER_OUT;
}
void action0605()
{
wait_for_touchtone();
}
void next0605()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("x-hospital.wav");
CURRENT_STATE = TRANSFER_OUT;
}
void action0606()
{
```

```
}
void next0606()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("help-in-speed-dial.wav");
}
void action0607()
{
}
void next0607()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("help-speed-dial-list.wav");
}
void action0608()
{
}
void next0608()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("cancelled.wav");
CURRENT_STATE = MAIN_MENU;
}
void actionE0600()
{
}
void nextE0600()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("help-in-speed-dial.wav");
}
void actionE0601()
{
}
void nextE0601()
{
grammar_disable(state_arc(CURRENT_STATE));
CURRENT_STATE = TOP;
}
void actionE0602()
{
}
void nextE0602()
{
grammar_disable(state_arc(CURRENT_STATE));
play_wavfile("try-again.wav");
printf("Please try again\n");
}
void init06()
{
grammar_enable(state_arc(CURRENT_STATE));
play_wavfile("names-short-1.wav");
printf("Name please\n");
}
void init_grammar()
{
grammar_disable(state_arc(0));
grammar_disable(state_arc(1));
grammar_disable(state_arc(2));
grammar_disable(state_arc(3));
grammar_disable(state_arc(4));
grammar_disable(state_arc(5));
```

```
grammar_disable(state_arc(6));
}
```

Appendix B

BNF Grammar for the Dialog Specification Language

This appendix includes the BNF grammar which was used to build the language parser for the dialog specification language. The language parser was constructed using the Unix utility *yacc* and this grammar specification.

```
%{
/* ************************************************************ */
/* dialog_parser.y --
        parser for dialog specification language.
*/
/* ************************************************************ */

%}
%union {        /* stack type */
        int        tok;
        char       sym[128];
}

%token <tok> TOK_ERROR
%token <tok> TOK_DIALOG
%token <tok> TOK_STATE, TOK_GLOBAL_STATE
%token <tok> TOK_PROMPT_TEXT, TOK_PROMPT_FILE
%token <tok> TOK_INPUT, TOK_EVENT
%token <tok> TOK_NEXT, TOK_ACTION
%token <tok> TOK_COND_NEXT, TOK_COND_ACTION
%token <tok> TOK_ENABLE
%token <tok> TOK_LPAREN, TOK_RPAREN
%token <tok> TOK_COMMA, TOK_PERIOD
%token <tok> TOK_DEFAULT
%token <tok> TOK_NAME, TOK_TAG
%token <tok> TOK_INT, TOK_REAL
%token <sym> TOK_STR, TOK_SYM
%token <tok> TOK_NONT
```

```
%type  <sym> test_exp
%type  <sym> function prompt wavfile
%type  <sym> arg expr
%type  <sym> string symbol

%start        dialog

%%

dialog  : TOK_DIALOG symbol states TOK_RPAREN
                                { GenDialog($2); }
        ;

states  : state states
        |
        ;

state   : TOK_STATE symbol items TOK_RPAREN
                                { GenState($2); }
        | TOK_GLOBAL_STATE items TOK_RPAREN
                                { GenGlobalState(); }
        ;

items   : item items
        |
        ;

item    : TOK_PROMPT_TEXT pretextprompts TOK_RPAREN
        | TOK_PROMPT_FILE prefileprompts TOK_RPAREN
        | TOK_INPUT inputs TOK_RPAREN
        | TOK_EVENT events TOK_RPAREN
        ;

pretextprompts: pretextprompts prompt
                                { GenPrePrompt(PROMPT_TEXT, $2); }
        |
        ;

posttextprompts: posttextprompts prompt
                                { GenPostPrompt(PROMPT_TEXT, $2); }
        |
        ;

prefileprompts: prefileprompts prompt
                                { GenPrePrompt(PROMPT_FILE, $2); }
        |
        ;

postfileprompts: postfileprompts prompt
                                { GenPostPrompt(PROMPT_FILE, $2); }
        |
        ;

prompt  : string                { strcpy($$, $1); EMBED_FUNCTION = 0; }
        | wavfile                { strcpy($$, $1); EMBED_FUNCTION = 0; }
        | TOK_LPAREN function TOK_RPAREN
                                { strcpy($$, $2); EMBED_FUNCTION = 1; }
        ;

wavfile : symbol                { strcpy($$, $1); }
        | symbol TOK_PERIOD symbol          { strcpy($$, $1);
                                          strcpy($$+strlen($$), ".");
```

```
                                        strcpy($$+strlen($$), $3); }

inputs  : input inputs
        |
        ;

input   : string options          { GenInput($1); }
        ;

events  : event events
        |
        ;

event   : string options          { GenEvent($1); }
        ;

options : option options
        |
        ;

option  : TOK_ENABLE enable_expr TOK_RPAREN
        | TOK_ACTION action TOK_RPAREN
        | TOK_NEXT next TOK_RPAREN
        | TOK_PROMPT_TEXT posttextprompts TOK_RPAREN
        | TOK_PROMPT_FILE postfileprompts TOK_RPAREN
        | TOK_COND_ACTION actioncond TOK_RPAREN
                                        { GenActionCondClose(); }
        | TOK_COND_NEXT nextcond TOK_RPAREN
                                        { GenNextCondClose(); }
        ;

enable_expr: expr                 { GenEnable($1); }
        ;

action  : actionfnc
        ;

actionfnc: function                 { GenAction($1); }
        ;

actioncond: actioncond actioncondex
        |
        ;

actioncondex: TOK_LPAREN expr function TOK_RPAREN
                        { GenCondAction($2, $3); }
        ;

function: symbol TOK_LPAREN args TOK_RPAREN
                            { strcpy($$, $1); }
        ;

args    : args TOK_COMMA arg        { ArgAppend($3); }
        | arg                 { ArgAppend($1); }
        |
        ;

arg     : symbol        { strcpy($$, yytext); }
        | TOK_NONT        { strcpy($$, yytext); }
        ;

next    : symbol        { GenNext($1); }
```

```
        ;

nextcond: nextcond nextcondex
        |
        ;

nextcondex: TOK_LPAREN expr symbol TOK_RPAREN
                        { GenCondNext($2, $3); }
        ;

expr    : string        { strcpy($$, $1); }
        | TOK_DEFAULT       { strcpy($$, yytext); }
        ;

symbol  : TOK_SYM        { strcpy($$, yytext); }
        ;

string  : TOK_STR        { yytext [strlen (yytext) - 1] = 0;
                        strcpy($$, yytext + 1);}
        ;

%%
```

Bibliography

[BBN93] BBN Systems and Technologies, Cambridge, MA. *HARK Prototyper User's Guide*, March 1993.

[BBN94] BBN Systems and Technologies, Cambridge, MA. *HARK Recognizer System Integrator's Guide*, January 1994.

[BN93] Christopher Baber and Janet M. Noyes, editors. *Interactive Speech Technology: Human Factors Issues in the Application of Speech Input/Output to Computers*. Taylor and Francis, 1993.

[Joh78] S. C. Johnson. *Yacc: Yet Another Compiler-Compiler*, 1978.

[LS78] M. E. Lesk and E. Schmidt. *Lex: A Lexical Analyzer Generator*, 1978.

[LS93] Eric Ly and Chris Schmandt. Chatter: A conversational learning speech interface. Technical report, Speech Research Group, MIT Media Laboratory, 1993.

[Luh91] Rick Luhmann. IVR trends. *Teleconnect*, 9(9):106–112, September 1991.

[Nie93] Jacob Nielsen. Noncommand user interfaces. *Communications of the ACM*, 36(4):82–100, April 1993.

[Pap93] Bruce Papazian. Management of errors in spoken language system dialogues: A literature review. Technical report, Bolt, Beranek, and Newman, Inc., 1993.

[Per87] J.F. Perkins. Application software tools in speech technology. In *Official Proceedings of SPEECH TECH '87*, pages 78–80, April 1987.

[RE89] Teresa L. Roberts and George Engelbeck. The effects of device technology on the usability of advanced telephone functions. In *Proceedings of the ACM SIGCHI '89*, pages 331–337, 1989.

[Ren92] The Renaissance Group, Inc., San Jose, CA. *Arch Technical Reference Manual*, December 1992.

[Rol89] Rolm, Santa Clara, CA. *PhoneMail Release 4: Implementation and Administation Guide to Basic Features*, June 1989.

[SF85] Axel T. Schreiner and George H. Friedman. *Introduction to Compiler Construction with UNIX*. Prentice Hall, Inc., 1985.

[TWW93] Mike Tate, Rebecca Webster, and Richard Weeks. Evaluation and prototyping of dialogues for voice applications. In *Interactive Speech Technology: Human Factors Issues in the Application of Speech Input/Output to Computers*, pages 157–165. Taylor and Francis, 1993.

[VP85] R.S. Van Peursem. Do's and dont's of interactive voice dialog design. In *Official Proceedings of SPEECH TECH '85*, pages 48–56, April 1985.

[WME93] Cathleen Wharton, Monica Marics, and George Engelbeck. Speech recognition vocabulary scoping for automated call routing. In *Proceedings of the Human Factors and Ergonomics Society*, pages 240–243, 1993.