A SURVEY OF DIALOG MANAGEMENT WITH APPLICATIONS IN RAVENCALENDAR

A Thesis Presented by

William Joseph Lahti Jr.

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Computer Science

Stony Brook University

December 2008

Stony Brook University

The Graduate School

William Joseph Lahti Jr.

We, the thesis committee for the above candidate for the Master of Science degree,
hereby recommend acceptance of this thesis.

Amanda Stent – Thesis Advisor
Associate Professor, Computer Science

Steven Skiena
Professor, Computer Science

David Warren
Professor, Computer Science

This thesis is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

Abstract of the Thesis

A Survey of Dialog Management With Applications in RavenCalendar

by

William Joseph Lahti Jr.

Master of Science

in

Computer Science

Stony Brook University

2008

A crucial component of any spoken dialog system is the dialog manager (DM), which controls the system's end of a conversation. Many approaches to implementing a DM have been developed. For a given domain, some approaches will be more appropriate than others. In this thesis, I describe various approaches to dialog management, various systems for calendars, the development of a DM for a calendar system, and finally suggest future calendar DM implementations.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A interesting domain to work with in dialog management is calendar systems, something people commonly use. Many spoken calendar systems have been developed, in addition to the many non-spoken calendar products currently available. In the case of a calendar system, you can try to make the process of manipulating appointments as smooth as possible. Work can also be done on different ways of representing information. One example of a spoken calendar system is MIPAD[24], a system developed by Microsoft designed to provide the functionality of a Personal Digital Assistant (PDA), including a calendar.

A fundamental component to a spoken dialog system is the dialog manager (DM). This component manages the conversation between the user and the computer. A dialog manager determines what action should be taken by the system given input. Input arrives at a dialog manager via some other system component, often a speech recognizer. Their output can come in the form of speech and/or some other action. For instance, telling a calendar system to ”delete the appointment at noon tomorrow” would result in the DM instructing other system components to delete the appointment matching the given parameters, and reporting on the operation’s success or failure.

Dialog management research has several goals. There is an interest in how to represent conversations. Efficiency in terms of minimizing the amount of time it takes to accomplish a task, is also a goal.

There are numerous ways to implement a dialog manager. One amounts to treating a conversation like a finite-state automaton. Another focuses on the ability to summarize

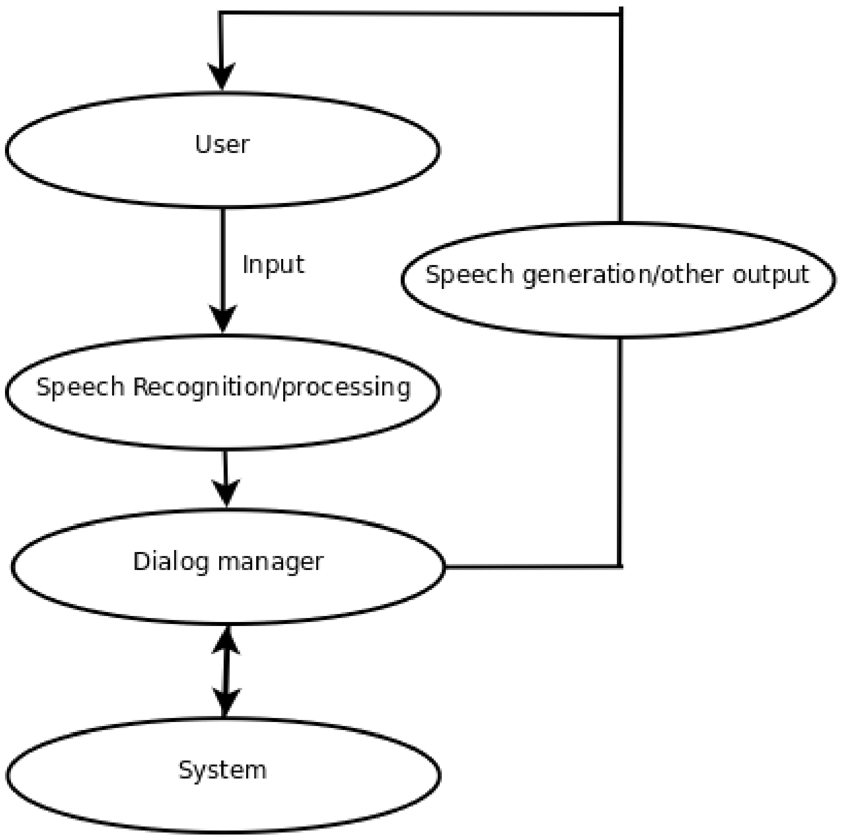


Figure 1.1: The basic architecture of a spoken dialog system. The dialog manager is the agent the user and system communicate to each other through.

large amounts of information, typically to expedite conversations with the goal of something like identifying a good restaurant to eat at. A more simplistic DM, targeting Web applications, considers conversations to be a series of menus and forms.

First, I will discuss many methods of dialog management, with some sample implementations.

I will then describe various spoken and non-spoken calendar systems, and the implementation of RavenCalendar [20], a calendar system developed at Stony Brook University. The work was on the dialog manager. I will then describe work done to improve upon the system.

Finally, I will discuss approaches to building a calendar system using styles of dialog management other than the one used, with some approaches being much more viable than others.

Chapter 2

Dialog Management Issues & Applications

2.1 Calendar interfaces

Multiple products exist that provide a calendar in electronic format. Some are designed for meetings to be organized, and some are designed to provide shared calendars. They also provide various means of inputting/displaying appointments.

In general, appointments have attributes, including a title, description, location, and start/end times/dates. There is often a class of appointment known as an all-day appointment, which has dates, but not times. Examples of its use could include a trip, or some annual event, such as a holiday or anniversary. Multi-day appointments are also possible, and they can be all-day or standard appointments that span multiple days.

Microsoft Outlook is a commercial product that offers full personal information manager (PIM) functionality, including an email client, contact manager, to-do list, and a calendar. Appointments can have attendees, to facilitate the coordination of a meeting. It is also possible to set your status during an appointment, whether or not it is all-day, to a setting like ’free’, ’busy’, or ’out-of-town’.

Several views for calendars are offered. The screen can display a day’s appointments by day, work week, 7-day week, or month.

Lotus Notes provides calendar functionality. Calendars may be visible to and sometimes editable by one or many users. They are held on a server, and are accessible through a web interface or client program.

A spoken dialog system [6] exists to manage a calendar over the telephone. The system is intended to cooperatively work with a user on a calendar.

The design of the system takes into account the fact that the speech recognizer will make errors. As a result, the system takes measures to prevent a breakdown in communication between user and system by doing things such as confirming appointments. Confirmation is also requested if the confidence score of the speech recognizer is very low, or if the answer is unlikely or too "destructive". Semantic frames are filled using the Phoenix semantic parser (to be discussed later).

2.2 Summaries of information

A challenge in dialog management on systems involving a large amount of information is the generation of summary information. Experiments [15] have been done to determine a good method of communicating summaries of large amounts of information to users.

They use a system to provide information on 596 restaurants in London. They have as many as 19 attributes, such as price range, quality, and the type of food (i.e. Indian, Chinese).

They explore two methods of ranking the attributes. One creates a model for each user, and the other chooses attributes whose top 4 values combine to at least 80% of the collection. No summary is generated for cases where the 80% threshold is not satisfied. For instance, the top 4 neighborhoods of a particular type of restaurant could account for about 80% of that type of restaurant. Their algorithm discounts locations with "unknown" as the value for the attribute under consideration.

They utilize subset clustering to better describe the often large amount of in-focus data. It typically requires at least 3 attributes to get cluster sizes of a small size. Their system used 2 parameters, so the clusters would remain large enough to be worthy of summarizing. Clusters were then ranked using a user model, if available, otherwise the cluster size was used.

They ran user experiments involving 15 users and 8 tasks within the domain of London restaurants. Four tasks involved data sets of over 100 restaurants, and the other four involved smaller sets. The subjects were presented with the tasks as well as four candidate responses. They were asked to score each response's usefulness in finding a restaurant from 1 to 5, as well as naming a best and worst response. They could also provide free text feedback about the responses.

They found that the users preferred small datasets. Users also first preferred summaries with no user modeling or association rules, and then ones that had both. They also spoke of a preference for a general overview, as well as summaries taking the neighborhood of a restaurant into account.

For a calendar system such as ours, it could be useful to efficiently summarizes somebody's appointments, or some attribute about them.

2.3 Side effects

Side effects are an issue in dialog systems. Dialog managers both access and update the current state of the conversation, and the system's state. However, a system's data and state is typically scattered across many modules. Side effects therefore affect the intermodular operation of a system, and can introduce testing and debugging challenges.

An architecture [16] was developed to overcome this issue.

The architecture was implemented in a system that serves as a procedure browser for astronauts aboard the International Space Station. It is meant to talk an astronaut through a procedure, a role typically fulfilled by a second astronaut. The system offers the ability to move between steps, both sequentially and non-sequentially, as well as working with voice notes and alarms. The system also supports a GUI interface.

The dialog manager must also contend with the background noise level of the space station, which can increase the rate of recognition errors. It must also have a mechanism to confirm the successful completion of each step before advancing to the next, due to the potentially lethal consequences of an error.

The DM uses a framework with the central concepts of dialog actions (output), dialog moves (input), and information state. A move results in a new state and set of actions.

Most moves correspond to a system command. The information state is a vector whose contents include current dialog status and information for the task. By keeping input and output self-contained, and by completely representing the current task state, side effects are minimized. Side effects caused by the requests for environmental examination are eliminated by breaking that action into 2 steps for the DM. The first one outputs a state reflecting that it is waiting for the result of the query, and the second one outputs a state containing the result.

2.4 Error recovery

A key part of any spoken dialog system is its ability to quickly identify errors and smoothly recover from them. One way to approach this is to focus on creating useful error messages based on what was understood.

Hockey et. al [7] developed the Targeted Help extension to WITAS to provide help messages to users targeted to their utterance. Specifically, this system covers utterances that weren't understood by the speech recognizer, which is grammar-based.

The new system uses a statistical language model for utternaces out of the grammar's coverage area. Utterances are examined by both the primary recognizer and the secondary recognizer for Targeted Help.

The system is used in 2 scenarios. One is if the primary recognizer rejects the input, and the secondary recognizer has a parsable hypothesis, in which case the system continues using that one. That scenario is rare. The second scenario is like the first, except the hypothesis isn't parseable. In that case, an error message is generated by the agent. The message can contain information on what was heard, what wasn't understood, and what the user could say instead. If both recognizers failed, a default error message would be produced.

They can identify errors caused by words being left off of either end of the utterance, words said that are in the language model, but not the grammar's volcabulary, and cases of words in the grammar being used in ways not in the grammar.

They found their system to be useful. In trials, fewer users gave up on the system, and tasks were accomplished faster.

Chapter 3

Dialog Management Strategies

Dialog managers come in many varieties. Fundamental differences between systems often involve how they represent data, or how they decide what to do. These differences can make a system more appropriate for one domain and less appropriate for others. For instance, a system capable only of moving through menus and answering simple question might be fine for an automated system to order a pizza, but not for a system which controls a helicopter.

Simpler DMs use very rigid methods to reason about what it should do. They are limited to menus and form filling, and they lean toward system initiative, meaning that the dialog system, as opposed to the user, leads the discussion. Some systems bring in probability and machine learning to enhance the accuracy of these systems' recognition/interpretation of an utterance. Other DMs perform planning, and others express the task in many small modules. Some systems use artificial intelligence techniques heavily to perform complex tasks at the command of a user.

There are also many different ways to represent data. A number of systems store data in a tree-like fashion. Others think of data as *concepts* within object-like structures.

In this chapter, dialog management styles are surveyed, compared, and contrasted, especially in the domain of a calendar system. Calendar systems are described. In addition, common issues in dialog management are discussed, along with efforts to overcome them.

DM type	Data representation	Reasoning method	Example system
Template-based	Form	Transitions based on input from the user	Systems built using the VoiceXML[13] framework
Finite-state systems	Form	Transitions based on most recent input from user along a predetermined path	Scaled-down directory assistance system[14]
Probabilistic	Form	State transitions with probability distributions - built manually or with machine learning	Sample ATIS taks system in [12]
DMs for semi-autonomous systems	Utternaces reduced to a common form	Planning	Personal Satellite Assisstant[17]
Plan-based systems	Trees	Planning	Flight reservation system[22]
Task-based systems	Trees	’Scripts’ collecting all pieces of a task	CMU Communicator[18]
Agenda-based systems	Structures for the dialog’s output, and how a user may interact with it	Planning based on an ordered list of handlers for ’miniture dialog centerings’	[23]
Agent-based systems	Concepts representing objects and their contents	Agents claim focus based on triggers and the current dialog state	RavenCalendar[20]

Table 3.1: Table 1: Types of dialog managers, basic data units, and example systems.

3.1 Template-based

VoiceXML [13] is a very simple language for template-based dialog management intended for Internet applications. It allows input through menus and forms.

Input is done to fill fields of different types. It comes with some built-in types, but also offers the ability to define other types through the use of grammars. For instance, there are builtin grammars for digits and phone numbers, but you could also define a grammar allowing a user to specify different toppings on a pizza. The architecture supports the ability to fill fields on a form in variable order.

The language provides facilities to run code at any point during the dialog. This is one feature which allows VoiceXML to interface with data systems. There is also a tag to do input validation given certain conditions, useful when fields can be filled in variable order in mixed-initiative dialog systems. It also provides the ability to include help and error messages.

Information collected from the user is typically held only in the fields local to that part of the conversation. Conversational history isn't held at all.

VoiceXML doesn't seem appropriate for user-initiative dialog systems, as our calendar system can be, at moments. The grammars, as written, tend to expect fairly specific responses, which makes it hard to answer the question, "How can I help you?" While it would probably be possible to implement a system, the interaction wouldn't be as natural as we intend it to be. VoiceXML does have the advantage of allowing rapid development, as well as development by people with a limited background in computer science.

3.2 Finite-state transitions

McTear [14] describes dialog management using finite state transitions, where a dialog system is represented through what is essentially a finite-state automaton. There is no history of the conversation maintained. The system asks questions in a predetermined order. This employed system-initiative dialog.

Among the given example systems is a questionnaire. It is an example of how well a state transition system works with something as structured as this. It can ask questions in a fixed order, and determine what questions should be asked based on prior responses.

For something as complex as our calendar system, something like this is completely inappropriate. As the author says, these systems are not very flexible. It tries to fill a form in a rigid order, unable even to accommodate extra information the user provides along with their response. Recovery mechanisms, as well as any sort of facility for flexibility causes the number of transitions to quickly become unwieldy. These systems are useful for simple, small, menu-based systems.

3.3 Probabilistic Dialog Modelling

Bayesian networks [10] can be used to model dialog using probabilities, as opposed to simple true/false judgements. In addition to dealing with dialog moves, the network also resolves anaphora, the relation between a pronoun/proverb and the word it was used in place of. The probability tables for the dialog moves were hand-constructed, based on experience from building rule-based systems.

The anaphora resolution's probability distributions are dynamically generated. They use several factors, including the recency of a reference, the manner in which something was referenced, and the prominence of potential referents in the dialog.

Others [11] have experimented with machine learning techniques for dialog moves.

3.4 Markov Decision Processes

Systems [12] have been developed using Markov decision processes to improve upon the dialog strategy, which is the "next move" in a dialog. They consider dialog as a finite-state machine, and the dialog strategy determines the next transition. Machine learning is required, as it is often the case that the space state is far too large to compute and represent the optimal strategy in the memory of a computer, if it is even possible to compute it. They combine supervised and reinforcement learning techniques to determine the best transition. Their goal is to minimize the transaction cost, which is a measure of efficiency. It includes factors such as misrecognition, and the time it takes to query the database.

Supervised learning, involving training the system on a corpus, isn't an effective means of determining dialog strategy, the next action to be taken in the current state, as small deviations in strategy from the user could cause significant changes to the distribution of strategy paths, compared to the one generated by the training process. This is related to the fundamental flaw in that supervised learning assumes that both the training and testing data come from the same source, and that there won't be deviations. In addition, the drawing of transitions is not independent, as the strategy the system is trying to determine is also the source of the distributions.

Reinforcement learning, in which probabilities of transitioning to various states are learned and adjusted based on past experience, isn't appropriate as a sole solution either. For it to be effective, a large number of dialog sessions are required to work out a good distribution. They implemented reinforcement learning through a Monte Carlo simulation of a user.

They place probabilities on all of the possible transitions from the current state. The probability depends upon the expected inputs. For instance, in a dialog that asks for a month, it places a probability of zero on everything but the 12 transitions corresponding to the months. This assumes that the user will, in fact, provide a month.

They use an airline information retrieval system as their example system. Their goal is to create a mixed-initiative system in which the complex strategies manually created by others can be learned. They first demonstrate that a user-initiative system isn't useful, as it could end up reciting information for every flight in its database, or if there are no flights returned in the query, offer no alternatives at all.

For the purposes of the model, a single action in which user interaction is involved consists of: prompting the user, collecting and recognizing the speech, and finally understanding the speech. Actions taken by the dialog manager include asking the user to further constrain their query, or asking the user to relax a constraint on their query. The attributes of a flight are the start/end locations/times, and the airline.

They ran their reinforcement training for 100 *epochs*. An epoch consists of dialogs starting from all states discovered already, exploring every path possible. The first epoch taught the DM to immediately terminate the dialog, resulting in a very unhappy user, and thus a high cost function value resulted. The next strategy yielded the user-initiative system described previously, and was discovered on the next epoch. The eighth epoch taught the system to ask the user to constrain their parameters if too much information was returned by the original query. The tenth epoch taught the system to have the user relax their constraints if no information was returned, and the 23rd epoch introduced the final optimizing feature, in which the system learned not to query the database until it had enough information from the user.

For the purposes of a calendar system, this isn't much different than if we were to try and implement it using a finite-state machine. It would still be a rigid interaction, and we'd still require all of the states (too many of them) that we would with the finite-state system.

3.5 Partially Observable Markov Decision Processes

The hidden information state (HIS) model [25], a type of partially observable Markov decision process (POMDP) was developed to address some of the shortcomings with Markov Decision Processes. This model provides a means of explicitly representing uncertainty.

Internally, the dialog manager must remember the user's last action, the user's goal, and some history of the dialog. In a POMDP, the system is always in an unknown state. Because the state is unknown, a belief state exists. The belief state is a distribution across all states. Based on this state and an action, a new state is chosen. The observed action is a user's utterance, as approximated by a speech recognizer. The recognizer provides possible actions and their probabilities.

The HIS, which uses state partitioning to make belief monitoring feasible. The states are in a single partition initially, and the partition is split at each step, as the user's goal becomes more clear, and more input is given towards that goal. The goals are represented with a forest of trees, initially all in a single partition. Each action is measured against the partitioned goals. If there is no match, the tree is expanded, creating another partition in the process. The probability from the parent node is distributed to its children according to ontology rules. For example, "drink" could be split into things such as "water" with probability 0.3, "soda" with probability 0.5, and "juice" with probability 0.2.

When the user takes an action, the goal partitions are evaluated, as described before. The act(s) are bound to partitions, a belief state is constructed, and the nearest belief point is found. The appropriate action is taken, and the process repeats until the goal is achieved.

3.6 Construct algebra

Abella & Gorin [1] describe a dialog manager which uses object-oriented properties to store data and in some cases, determine the course of action.

The system's knowledge is stored in object form, and uses inheritance to determine what to ask the user for. They represent knowledge with a hierarchy of constructs. Construct algebra is a set of relations and operations on a set of constructs. A construct would be a ¡head, body¿ pair in which the body could be a set of constructs. They define a set of relations between constructs, as well as union and projection operations.

Dialog motivators determine what action the dialog manager needs to take.

The dialog manager iterates over the dialog motivators.

They apply this dialog manager to two systems. One system accesses a large personnel database, allowing users to get information about people, or to place a call to them. The other system is designed to answer the open-ended question "How may I help you?"

For a calendar system, however, construct algebra doesn't seem appropriate. One of its shortcomings is its hierarchical nature, whereas a calendar is not. Its disambiguation motivator is potentially useful, but the system's set operations aren't useful for the tasks of making and deleting appointments.

3.7 Dialog management for semi-autonomous systems

Dialog managers have been developed to interface with semi-autonomous systems, such as robots. Several paradigms for turning input into action have been developed for such applications.

An early system [21] for controlling a simulated robot was based on representing knowledge as a collection of logical statements.

Rayner et. al. [17] argue that logical statements are not a good way to represent the interpretation of a user's commands. For instance, logical statements do not specify any order. They also mention that scripting languages do not always assume a command was successful.

Their dialog manager is designed with the assumption that errors will be made during the process of recognizing speech, followed by the process of making a program out of it. They incorporate the concept of multiple streams present throughout UNIX systems and scripting languages. The streams are `stdin`, `stdout`, and `stderr`. In their system, each step produces what they call meta-outputs, which provide information about how the translation was performed.

The DM first normalizes the utterance in such a way that synonymous statements become the same statement. They call the original statement a *linguistic level representation* and the converted form a *discourse level representation*. The system then resolves references to previous commands, such as when the user gives a command to do something to/involving an object, and uses the word ”it” as the subject of the next command. The subject would be resolved to the previous subject. From there, the input is translated into its final form, something similar to a script for the UNIX C Shell. The plan is then optimized. Finally, the plan is evaluated to determine what the results would be if carried out. The dialog manager can compare the different interpretations of the command, and can also compare efficiency.

However, the process of evaluating the plan doesn't always end with the command being carried out. Among the things the dialog manager can look for is the case that a given command was invalid, such as a request to open a door that is already open.

Another example of a command-and-control system is WITAS [8]. The system is intended to control an unmanned helicopter, known as a UAV. Their overall architecture is hub-and-spoke based.

Their dialog uses an *information state* to track the context. *Dialog moves*, actions taken by either the user or the UAV. The information state includes an *IR stack*, containing issues raised during the conversation, and a *UAV agenda*, containing the issues the UAV has *not* raised yet. A dialog move can move an issue from the agenda to the stack. Information is also kept on what objects have been referenced, in the *salience list*.

Utterances from the user are parsed into acts, such as commands or declarations.

Our calendar system isn't intended to have any sort of autonomy, so any system like these isn't appropriate for our purposes. We collect input and execute *a* command, meaning that any sort of scripting is unneeded.

3.8 Task-based

Rudnicky et. al. [18] implement a task-based dialog system. In such a system, operations involving complex data structures are broken into smaller tasks. For a task to be complete, both parties must agree on a result. They use two main data structures: the product, which holds the result of the dialog, and the schema, which details how the user may interact with the product. The dialog's behavior is based on a combination of the two. The collection of schema for a task is called the script.

They use this design in a system designed to create complex travel itineraries. The itinerary is represented hierarchically. Trips are defined, followed by details about them. For instance, you could say, "I'd like to fly to New York tomorrow, then I need to go to Montreal." This would create a tree for the trip, with a subtree for the Montreal leg of the trip.

The system is phone-based, isn't turn-based, and allows barge-in. The dialog manager's script doesn't specify a fixed order for the steps to perform a task; the agenda generates that. The system uses what they call a target data structure to store the result of the dialog.

The script-based system had the flaw of being hard to navigate. [23] While it was possible to go back to previously data fields, the system was still too constrained by the scripts to navigate around the data well. In addition, the type of data that can be entered with this system is static, as the scripts are essentially just filling forms in.

As far as a calendar system is concerned, navigation is vital. Creating a hierarchical tree doesn't seem like a good way to represent calendar data either. This system doesn't seem to have a logical place for multimodality, as our system will offer.

3.9 Agenda-based

Xu & Rudnicky [23] describe a agenda-based dialog management system. This is an improved version of the original script-based approach, in that it is more flexible. It is now possible to dynamically define the data structures, which avoids the issue of a limited range of information types the user could provide the system with. It is not known what type of a trip somebody might take, though the building blocks of a trip are known.

They now use an agenda in place of the script. They continue to use a tree to represent the product. They define subtrees that can be attached to the main tree through user input, such as specifying flight as the mode of travel to a location.

This system, like their previous one, is intended to make travel reservations. Their studies say that when humans interact with travel agents, they treat the task of making reservations as a series of topics that are discussed in full before moving on to another topic. If a topic were to be revisited, that action would be explicitly stated in the conversation.

Lemon et al. [9] describe a system for collaborative dialog to manage concurrent tasks. Their system is used to control an unmanned helicopter. The input is multimodal, as the user can do things such as specify location by clicking on a map.

The state of the dialog is based on a dialog move tree. Its nodes are dialog move classes, which are tags of every incoming logical form. A logical form is generated from the process of parsing the input.

Our calendar system is not going to be handling collaborative tasks. In addition, the ability to do tasks concurrently is not needed for our system, as the system completes tasks on demand, instead of attempting a mission similar to the ones their system describe.

3.10 Plan-based dialog management

Plan-based dialog management, as implemented by Wu et al. [22], is an offshoot of task-based dialog management. It is designed to handle multiple topics, as well as changing topics. It is also able to adapt to the requirements previously specified by the user. It provides mixed-initiative dialog. It is based on breaking a specific task into goals and plans, which are carried out by exercising dialog control. The underlying idea is that utterances attempt to accomplish a goal, and the computer's job is to model the goals and adjust the dialog appropriately.

In this system, the topics of a task are structured as a topic tree, and all of the trees form the topic forest. A topic tree structure contains the information about a given topic, and consists of three types of nodes. The root of the tree represents what the topic is. The midnodes represent logical relationships of its children. There are also special midnodes denoting that their children serve as primary, secondary, or additional properties about the task. Leaf nodes store information items about the task. Primary properties are the essential information, secondary properties are details, and additional properties are optional to the point that the system won't mention them if the user doesn't. Each node contains a flag representing the completeness and validity of its information (leaf nodes) or that of its children (root/midnodes). There is a response generation function associated with every node. There is a shared information index, which holds pieces of data pointed to by leaf nodes from multiple tasks which require a common piece of information, such as the destination of a trip.

The action to take from or with the dialog is determined by a reasoning engine. It is aware of the current status of the dialog, and its parameters are the topic of the utterance and the semantic frame produced by the semantic analysis module from the speech recognition. The frame contains slots representing data that will be put in a leaf node. Information can be appended or replaced in these nodes as more information becomes available.

They use a flight reservation system to demonstrate the concept. They provide an example in which the user specifies a destination city. The system asks for a flight date, and then asks for a departure time. This time, the user specifies "After 10:00", causing the system to enumerate the times, as the generated query returned multiple results. Later, the user changes the topic, by asking what the local time is in the destination upon arrival from a particular flight. The user changes the topic back to the first one by saying that they want that flight.

Another example of such a system is Catizone et. al[5].

This style of system gets closer to what we're looking for. We seek mixed-initiative dialog for our system. Our system will also need to be able to handle changing topics. In addition, it will be useful to be able to fill more than one slot at a time. However, more modularity would be useful.

3.11 Agent-based

Agent-based dialog management is another style. Our system[20], as well as the ones it is based on, use this style of dialog management. These systems and this style of dialog management are described in detail in the next chapter.

Chapter 4

The RavenCalendar system

I will first describe the components upon which RavenCalendar is based before describing the system itself.

The system's architecture is based on the Olympus [3] and RavenClaw [4] architectures developed at CMU. It allows dialog structure to be defined in an object-oriented manner. The dialog manager is agent-based. The dialog is defined as a graph, with each node being a small component with pre- and postconditions. The edges define the flow. This allows for components to be reused as much as possible. The Olympus architecture currently provides the speech recognition and generation, as well as text-to-speech, though we intend to replace it over time.

As a whole, the system is hub-and-spoke based, using the Galaxy architecture [19]. The hub serves as the point bridging all components of the system, such as the dialog manager, Google Calendar interface, speech recognizer, and TTS.

RavenCalendar is a system that provides a natural language interface to the popular Google Calendar and Google Maps applications. It allows users to make, edit, and remove appointments. In addition to dates and times, appointments may contain descriptions and locations.

The system is multimodal. Users may speak or type commands in natural language. They may specify locations through mouse clicks within Google Maps, or through natural language. An XML file defining locations on a map can be preloaded to help the user

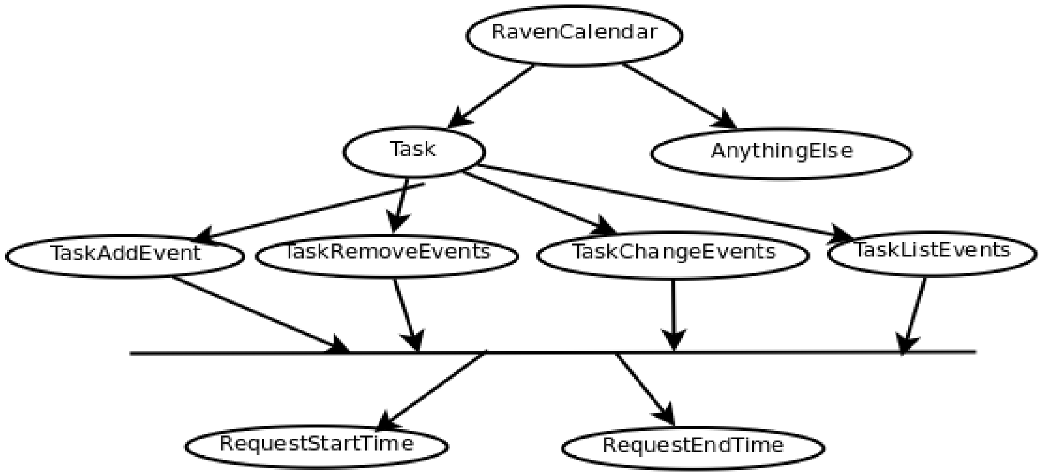


Figure 4.1: A small sample of the structure of RavenCalendar's agents. Agents have subagents, and agents on different branches can share subagents, such as the ones for requesting a time from the user.

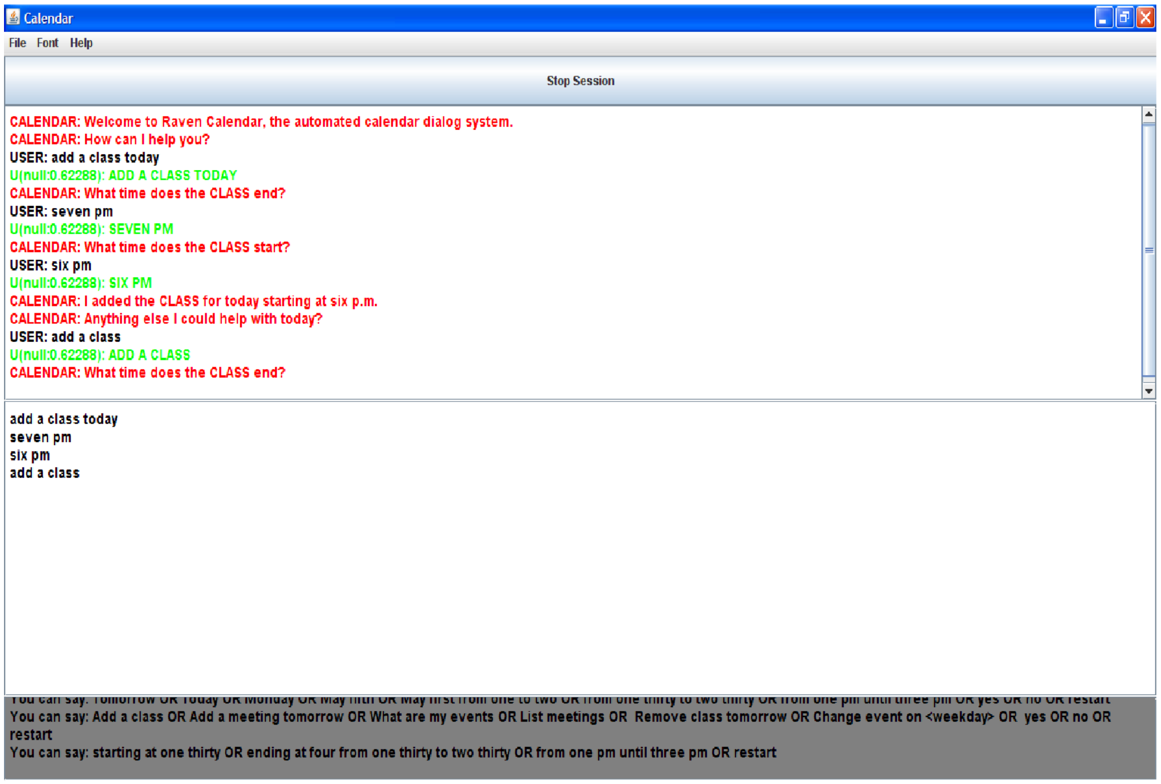
choose a location. They may also interact directly with Google Calendar, without interacting with RavenCalendar.

It allows users to enter appointments in steps, or all at once. For instance, you may say "Add an appointment tomorrow from one to two in the gym," or say "add an appointment," and then be asked for the details one at a time. You may also add appointments with some of the details in the initial utterance, with others asked afterward.

4.1 Google Calendar

This system serves as a multimodal interface to two tools from Google, though Google Calendar is the primary focus.

Google Calendar offers the ability to add and remove shared calendars, and all of the appointments that come with them, from your personal calendar. For instance, you may choose to add a calendar containing all of the dates and times of games for a particular sports team.



```
CALENDAR: Welcome to Raven Calendar, the automated calendar dialog system.
CALENDAR: How can I help you?
USER: add a class today
U(null:0.62288): ADD A CLASS TODAY
CALENDAR: What time does the CLASS end?
USER: seven pm
U(null:0.62288): SEVEN PM
CALENDAR: What time does the CLASS start?
USER: six pm
U(null:0.62289): SIX PM
CALENDAR: I added the CLASS for today starting at six p.m.
CALENDAR: Anything else I could help with today?
USER: add a class
U(null:0.62288): ADD A CLASS
CALENDAR: What time does the CLASS end?
```

```
add a class today
seven pm
six pm
add a class
```

```
You can say: Tomorrow OR Today OR Monday OR May fifth OR May first from one to two OR from one thirty to two thirty OR from one pm until three pm OR yes OR no OR restart
You can say: Add a class OR Add a meeting tomorrow OR What are my events OR List meetings OR  Remove class tomorrow OR Change event on <weekday> OR  yes OR no OR
restart
You can say: starting at one thirty OR ending at four from one thirty to two thirty OR from one pm until three pm OR restart
```

Figure 4.2: A screenshot of the RavenCalendar dialog system. Users can communicate with the system in natural language through speech or text, with the same result.

4.2 Galaxy

Galaxy [19] provides an architecture for accessing online resources through spoken dialog. The architecture is client-server, with the user interacting with a GUI written in Java.

The hub-and-spoke architecture of Galaxy defines each component of a dialog system as an independent module. These modules are implemented as servers within a network. Interaction with the hub is controlled through scripting. The scripts are composed of a list of servers, as well as their location and supported operations, and a set of programs. The programs are a set of rules, corresponding operations and conditions under which the rules are fired. When a rule is fired, the appropriate input is sent to the appropriate server, which will eventually return some output. Communication is done using frames.

The hub maintains a master list of variables, as well a history of the discourse. They implement a single hub for multiple dialog sessions, with separate states for each session.

4.3 Olympus

Olympus [3] is an open-source framework designed to provide an interface for large-scale spoken language research projects. The project arose out of the need for one to use in academic research, as commercial efforts to develop systems are never available to the research community. It is composed of many completely modular components.

The intermodular communications are based on Galaxy. Speech recognition is done with Sphinx, though interfaces for later versions of the engine exist. In addition, many varieties of language models are supported. The language recognition is done with Phoenix. Confidence annotation is determined with Helios. Dialog management is done with RavenClaw (see next section). Rosetta does the language generation based on the semantic output from RavenClaw, and then are passed to the Kalliope speech synthesis module. Tools are available for comprehensive data logging, processing, and analysis to aid in the development of dialog systems.

4.4 RavenClaw

RavenClaw [4], [2] is a framework for dialog management. It is the default dialog manager used by the Olympus framework described in the previous subsection. It is split into two layers, the Dialog Task Specification (DTS) and the Dialog Engine (DE). The DTS contains the logic specific to the domain, and the DE controls the dialog by executing the DTS, as well as applying strategies that are universal to conversation, like turn-taking. This essentially separates the domain-specific and domain-independent pieces of dialog management, which makes this framework much more flexible with respect to the domains it can be used in.

The framework is well-suited to research, as it is completely transparent. Each component can generate thorough logs, allowing developers to easily determine the current state of the dialog, as well as what actions were taken by the system. In addition, the framework is available under an open-source license. This allows RavenClaw to be adapted for new applications, as well as furthering the potential for people to conduct research using it. The system as a whole is designed to be completely modular. For instance, it is possible to substitute one error handling mechanism for another.

The system is composed of a series of dialog agents. Each agent represents a task. Agents are broken into subagents, which follow from a natural decomposition of a task. Each agent has an execute routine, as well as a set of preconditions and/or triggers that can cause the agent to execute. They also have criteria for completion. The agents are arranged in a tree-like manner. Nonterminal nodes are called *agencies*, and the leaves are *fundamental dialog agents*.

RavenClaw has 4 types of fundamental agents, to request information, output, expect (get input without asking), and 'other', to perform domain-related tasks. Their execute routines carry out the agent's task. An agency's execute routine determines the execution order of the agents beneath it.

The structure of the dialog's current state is held in the dialog stack. The stack is not bound to the tree structure, and the overall tree structure of the system does not dictate a rigid path for the conversation to follow. As the user shifts the conversation, the dialog manger can push the appropriate agents onto the stack, provided that the preconditions/triggers for those agents have been met, thus breaking from the apparent tree structure. This also allows for the easy accommodation of subdialogs. For example, if somebody tells a calendar system to "Add an appointment," a subtask for the task of adding an appointment would be getting the start/end dates/times, which would be broken into the subtasks of getting each date/time.

The DM goes through execution and input phases. During an execution phase, the system executes the agent at the top of the stack. Request phases can cause input phases. Input phases are somewhat more complex.

In an input phase, the system takes several steps. The system first constructs an agenda of expectations. It is created by asking each agent on the dialog stack from the top down for their expectations. An expectation is a slot in the semantic grammar that an agent is seeking. For instance, an agent seeking the time that an event begins is going to have an expectation corresponding to the slot of a grammar corresponding to a time. So if you say, "the event begins at eight pm," that agent will be interested in "eight pm". With each agent on the stack will come an increase in the size of the expectation agenda.

After the input is taken in, the dialog manager attempts to match pieces of the input to corresponding expectation slot. The input is bound to concepts, as appropriate. This is done by passing through the agenda in top-down order. It is possible that multiple agents on the stack will be expecting the same slot to be filled. In this case, the agent closest to the top will be the one the input is bound to, as it is the most relevant to the current state of the conversation. During this process, agents below the top of the stack may have their criteria satisfied, causing agents not to execute. For instance, if the system doesn't know the day or time of an event, asks when an event starts, and the user says ”Monday at one pm,” the slot for day will be filled in addition to the start time, causing the agent for requesting the day of an event to never be executed.

During the conversation, historical information is within the context of the current action being performed, owing to the use of a stack of dialog agents.

Once the binding process is complete, the agents on the topic tree are given a chance to claim focus of the conversation. This is dependent upon their trigger conditions being met. Agents that claim focus are pushed onto the dialog stack. After this process is complete, the agent at the top of the stack executes.

Our system is one among multiple systems that have been developed using this dialog management architecture. Among the systems CMU developed with this architecture are one for scheduling conference rooms, and another to provide information about busing in the Pittsburgh area.

S: Welcome to Raven Calendar, the automated calendar dialog system.	Agents to setup the connection have already executed. The speech occurs when the /RavenCalendar/Welcome agent executes.
S: How can I help you?	At this point, the /RavenCalendar/Task agent, from which all tasks are derived, has taken over, and first running its HowMayIHelpYou agent.
U: Add a class today	The system first recognizes that a class is to be added. It is then able to fill the slot for date that exists within the event object.
S: What time does the class end?	Agents that ask for the rest of the required information have been added to the stack. This is the first one popped.
U: seven pm	
S: What time does the class start?	
U: six pm	
S: I added the class for today starting at six p.m.	Having all of the required information, a series of agents that create the event are processed.
S: Anything else I could help you with today?	

Figure 4.3: A conversation with the system.

Chapter 5

RavenCalendar System Development

I encountered issues during the development of the dialog manager. It doesn't always move properly between states. Among other problems, this can result in the dialog manager freezing, not knowing what to do next.

The system has undergone the fundamental change of replacing preconditions with triggers in some cases. They cause that agent to assert control when a given set of conditions are met. This was done for the dialog agents for adding, editing, and removing appointments.

I made changes to the system's method of determining which agents claim focus after one is completed. As it was implemented, after an agent completes, the system polls every agent registered within the system, asking if it wishes to claim focus. This introduces problems, as many branches will have identical subagents. For example, the agents for adding, modifying, and removing events have the same subagents for asking for date and time, which have the same preconditions and triggers. In some cases, this yielded an ambiguous focus shift error, often leading to the system taking the wrong path.

To attempt to amend this, the system was modified to poll only agents whose parent is either claiming focus or already on the execution stack. The latter was necessary, as if only the former were implemented, the dialog manager would break, as the root agent wouldn't claim focus, preventing any other agents from being polled. However, this was not completely successful. The fact that agents were being polled only if their immediate parent met the criteria precluded possible legitimate claims by their descendants.

I later found that the cause of the problem was how the system handled an ambiguous focus shift. All claiming agents were being put on the execution stack. I modified the system so that only one agent would be put on the stack. The system now chooses the agent that is closest to the top. This resolves the issue of multiple date/time agents triggering when the user says "Add a class," as the topmost agent for adding an event is also triggered, and is chosen.

5.1 Testing methods

The system's functionality was exhaustively tested before and after modifications.

	Appointment	Class	Concert	Game	Trip/ conference	Class that meets weekly
Add						
Add long						
Add date						
Add time						
Add start and end times						
Add location						
Add description						
Modify						
Modify long						
Modify date						
Modify time						

Modify start and end times						
Modify location						
Modify description						
Remove						
Remove by type						
Remove date						
Remove by time						
Remove by location						
Remove by description						
List						
List by date						
List by location						
List by type						

Table 5.1: The testing matrix for the dialog manager. ”Long” refers to saying everything needed to create the appointment.

Chapter 6

Discussion

Other styles of dialog management could be viable for a calendar system.

6.1 Finite-state model

A finite-state approach could be used. The initial state could represent “How can I help you?”, and there would be transitions to each basic operation of the calendar.

These states would be abstractions of the actual operation. Within these operation states, you could have states defining the order in which that operation’s questions are asked. The abstract states could have slots for the data required for the task, with more than one possible being filled in a single utterance (typically the one that invoked that state).

One example is ”Add an appointment at one pm.” This would invoke ”add event” and fill the ”start time”, and possibly ”start date” at once, with ”end time” still empty. The states defining the questions to be asked could have transitions to bypass them if their slot is full already, though this starts to break away from a finite-state automaton for lower-level system functions, preserving it only at the higher, abstract levels.

This model naturally leads to modelling the data as objects which are filled out like forms. The history of the dialog is unneeded, as the path to a state is irrelevant.

Users’ most common actions could be learned by a system, to aid in resolving ambiguity.

6.2 Dialog manager development

Dialog management development is still a developing field. There isn't yet a dialog management style or architecture that fits most domains well.

A number of styles discussed here use techniques that originated in the AI field, such as hidden Markov models, in addition to the machine learning techniques used in some of the systems.

Chapter 7

Future work

The RavenCalendar system can be taken in many directions.

For instance, the RavenClaw developers suggest in their paper [4] that their architecture's means of determining the order in which agents could be executed could be changed from its current left-to-right order. User models of calendar usage could be developed to try and predict common user actions, and favor the appropriate agents.

Bibliography

[1] Alicia Abella and Allen L. Gorin. Construct algebra: Analytical dialog management. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, pages 191–199, 1999.

[2] Dan Bohus. *Error Awareness and Recovery in Conversational Spoken Language Interfaces*. PhD thesis, Carnegie Mellon University, May 2007.

[3] Dan Bohus, Antoine Raux, Thomas K. Harris, Maxine Eskenazi, and Alexander I. Rudnicky. Olympus: an open-source framework for conversational spoken language interface research. In *Proceedings of HLT-2007*, 2007.

[4] Dan Bohus and Alexander I. Rudnicky. Ravenclaw: Dialog management using hierarchial task decomposition and an expectation agenda. In *Eigth European Conference on Speech Communication and Technology*, 2003.

[5] R. Catizone, A. Setzer, and Y. Wilks. State of the art in dialogue management. Technical report, September 2002.

[6] M. Fanty, S. Sutton, D. G. Novick, and R. Cole. Automated appointment scheduling. In *ESCA Workshop on Spoken Dialogue Systems*, Vigsø, Denmark, 1995.

[7] Beth Ann Hockey, Oliver Lemon, Ellen Campana, Laura Hiatt, Gregory Aist, James Hieronymus, Alexander Gruenstein, and John Dowding. Targeted help for spoken dialogue systems: intelligent feedback improves naive users' performance. In *Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics - Volume 1*, 2003.

[8] Oliver Lemon, Anne Bracy, Alexander Gruenstein, and Stanley Peters. The witas multi-modal dialogue system i. In *Eurospeech 2001*, 2001.

[9] Oliver Lemon, Alexander Gruenstein, Alexis Battle, and Stanley Peters. Multi-tasking and collaborative activities in dialogue systems. In *Proceedings of the Third SIGdial Workshop on Discourse and Dialogue*, pages 113–124, July 2002.

[10] Oliver Lemon, Prashant Parikh, and Stanley Peters. Probabilistic dialogue modelling. In *Proceedings of the Third SIGdial Workshop on Discourse and Dialogue*, pages 125–128, July 2002.

[11] Piroska Lendvai, Antal van den Bosch, and Emiel Krahmer. Machine learning for shallow interpretation of user utterances in spoken dialogue systems. In *EACL proceedings*, 2003.

[12] Esther Levin, Roberto Pieraccini, and Wieland Eckert. A stochastic model of human-machine interaction for learning dialogstrategies. *IEEE Transactions on Speech and Audio Processing*, 8(1):11–23, January 2000.

[13] Bruce Lucas. Voicexml for web-based distributed conversational applications. *Commun. ACM*, 43(9):53–57, 2000.

[14] Michael McTear. Modelling spoken dialogues with state transition diagrams: Experiences with the cslu toolkit. In *Proceedings of the Fifth International Conference on Spoken Language Processing*, 1998.

[15] Joseph Polifroni and Marilyn Walker. An analysis of automatic content selection algorithms for spoken dialogue system summaries. In *Proceedings of the 2006 IEEE Spoken Language Technology Workshop*, December 2006.

[16] Manny Rayner and Beth Ann Hockey. Side effect free dialogue management in a voice enabled procedure browser. In *Proceedings of INTERSPEECH 2004*, 2004.

[17] Manny Rayner, Beth Ann Hockey, and Frankie James. A compact architecture for dialogue management based on scripts and meta-outputs. In *Proceedings of Applied Natural Language Processing (ANLP)*, 2000.

[18] Alexander I. Rudnicky, E. Thayer, P. Constantinides, C. Tchou, R. Shern, K. Lenzo, Wei Xu, and A. Oh. Creating natural dialogs in the carnegie mellon communicator system. In *Sixth European Conference on Speech Communication and Technology*, 1999.

[19] Stephanie Seneff, Ed Hurley, Raymond Lau, Christine Pao, Phillip Schmid, and Victor Zue. Galaxy-ii: A reference architecture for conversational system development. In *Proceedings of the Fifth International Conference on Spoken Language Processing*, 1998.

[20] Svetlana Stenchikova, Basia Mucha, Sarah Hoffman, and Amanda Stent. Ravencalendar: A multimodal dialog system for managing a personal calendar. In *2007 NAACL HLT Demonstration Program*, 2007.

[21] Terry Winograd. *A Procedural Model of Language Understanding*, chapter 3, pages 114–151. W. H. Freeman and Company, 1973.

[22] Xiaojun Wu, Fang Zheng, and Mingxing Xu. Topic forest: A plan-based dialog management structure. In *Proceedings of the 2001 IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2001.

[23] Wei Xu and Alexander I. Rudnicky. Task-based dialog management using an agenda. In *ANLP/NAACL Workshop on Conversational Systems*, pages 42–47, 2000.

[24] Alex Acero C. Chelba Li Deng D. Duchene Joshua Goodman Hsiao-Wen Hon D. Jacoby L. Jiang R. Loynd M. Mahajan P. Mau S. Meredith S. Mughal S. Neto M. Plumpe Kuansan Wang Y. Wang Xuedong Huang. Mipad: A next generation pda prototype. In *ICSLP-2000*, 2001.

[25] Steve Young, Jost Schatzmann, Karl Weilhammer, and Hui Ye. The hidden information state approach to dialog management. In *Proceedings of ICASSP 2007*, 2007.