A Natural Human-Computer Interface for Controlling Wheeled Robotic Vehicles

Frans Flippo

26th August 2003

Abstract

Robots are used increasingly to execute dangerous tasks and military missions. Autonomous robots are the warriors of the future, executing missions without requiring continuous supervision.

Multimodal interfaces are the interfaces of the future in which speech, gestures, gaze, and other modalities are combined to provide a natural way for humans to communicate with machines.

In this thesis I present a multimodal interface that was built to control and task wheeled robotic vehicles for military missions using spoken language, keyboard, mouse, touch, and gaze inputs, which can be used simultaneously. I developed a framework for multimodal command and control applications in which a novel fusion technique is used to combine these inputs. This framework was used to add multimodal interaction to Flatscape, a military mission planning and execution tool. With Flatscape and this new multimodal interface, natural interaction methods can be used to control robots directly or assign missions for them to execute autonomously.

Acknowledgements

The research work for this thesis was done at the Center for Advanced Information Processing at Rutgers University in Piscataway, NJ, USA. I want to thank Dr. Flanagan for inviting me to spend what has become a challenging and enriching year at CAIP. Thanks also to my supervisors at CAIP: to Dr. Marsic, who provided valuable pointers and guidance; and to Allen Krebs, for his ideas, his help with Flatscape and DISCIPLE, and for being good company.

A special thanks to my supervisor at Delft University, Dr. Leon Rothkrantz, who provided me with the opportunity to do my thesis work abroad, helped me arrange funding, and guided me in writing this thesis.

I also want to thank my parents, family, and friends in Holland for their support and faith in me leaving home to study abroad for a year. To all the friends I have made in New Jersey: you have made this an unforgettable year, I love you all. Finally, eternal thanks to my God and Savior, Jesus Christ. I owe this life to you.

I acknowledge the financial support of the Stichting Universiteitsfonds Delft, STIR, the Delft University ITS department, and sponsoring from CAIP.

This research at CAIP is supported by US Army CECOM Contract No. DAAB07-02-C-P301, a grant from the New Jersey Commission on Science and Technology, and by the Center for Advanced Information Processing (CAIP) and its corporate affiliates.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
  - 1.1.1 Multimodal Interfaces . . . 1
  - 1.1.2 Robots . . . 1
  - 1.1.3 Controlling robots through a multimodal interface . . . 2
- 1.2 Project Description . . . 2
  - 1.2.1 Background . . . 2
  - 1.2.2 Problem Description . . . 3
    - Non-functional requirements . . . 3
  - 1.2.3 Approach . . . 4
  - 1.2.4 Overview . . . 4

2 Dialog systems **5**
- 2.1 Data Flow in a Dialog System . . . 6
- 2.2 Speech Recognition . . . 7
  - 2.2.1 A brief look at the theory . . . 8
  - 2.2.2 Products . . . 9
- 2.3 Parsing . . . 10
  - 2.3.1 Parsers and Multimodal Systems . . . 11
  - 2.3.2 Products . . . 11
- 2.4 Dialog Management . . . 12
  - 2.4.1 Products . . . 13
- 2.5 Natural Language Generation . . . 14
  - 2.5.1 Products . . . 14
- 2.6 Speech Synthesis . . . 15
  - 2.6.1 Speech Markup . . . 15
  - 2.6.2 Speech Synthesis Techniques . . . 16
  - 2.6.3 Products . . . 17
- 2.7 Fusion . . . 17
- 2.8 Fission . . . 18
  - 2.8.1 Products . . . 18
- 2.9 Multi-agent Architectures . . . 18
  - 2.9.1 Galaxy Communicator . . . 19
  - 2.9.2 Open Agent Architecture (OAA) . . . 21
  - 2.9.3 Adaptive Agent Architecture (AAA) . . . 22
  - 2.9.4 JADE . . . 22
- 2.10 Multimodal Systems . . . 23
  - 2.10.1 Advantages of Multimodal . . . 24

2.10.2 Multimodal Devices 25
Gaze Tracker 25
Tactile Glove 26
Touch Screen 26
Microphone Array 26
2.10.3 Current Multimodal Systems 26
2.10.4 From Unimodal to Multimodal 27
2.10.5 Reuse of Programming Code in Multimodal Dialog Systems 28
2.10.6 The Future of Multimodal Interfaces 29

3 Design of the Multimodal Framework 31
3.1 Developing a New Multimodal Interface 31
3.1.1 Rationale 31
3.1.2 Design Goals 32
3.1.3 Interaction Style 32
3.1.4 Development Procedure 32
3.2 Object Oriented Frameworks 33
3.3 A Multimodal Framework 34
3.3.1 Design Goals 34
3.3.2 Approach 35
3.4 Architecture 36
3.5 Components 37
3.5.1 A Common Infrastructure: Communicator 37
Communicator and system robustness 38
3.6 New vs. Off-the-Shelf Components 38
3.7 Off-the-Shelf Components 39
3.7.1 Speech Recognizer 39
3.7.2 Parser 40
3.7.3 Speech Synthesizer 40
3.8 New Components 40
3.8.1 Reusability in Components 40
3.8.2 Fusion 41
The Fusion Process 41
The Fusion Design 44
The Fusion Manager 44
Services Provided by the Fusion Manager 47
3.8.3 Dialog Manager 48
Dialog Manager Tasks 48
3.8.4 Dialog Manager Limitations 49
3.8.5 Abstractions and Assumptions 50
Application Abstractions and Assumptions 51
Fusion Abstractions and Assumptions 51
Dialog Management Abstractions and Assumptions 52
3.8.6 Fission 52
3.8.7 Natural Language Generation 53

4 Implementation of the Multimodal Framework 55
4.1 Choice of Language . . . 55
4.2 Programming with Communicator . . . 56
4.2.1 Configuring the Hub . . . 56
4.3 Speech Recognizer . . . 57
4.4 Parser . . . 60
4.4.1 Phoenix frame representation . . . 60
4.5 Fusion and Dialog Management . . . 61
4.5.1 Fusion Resources . . . 62
4.6 Fusion Manager . . . 64
4.6.1 Resolving Contradictory Inputs . . . 65
4.6.2 Fusion Interfaces . . . 65
Resolver . . . 66
ContextProvider . . . 68
ContextData . . . 69
LocatableObjectStore . . . 69
CoordinateTransform . . . 71
4.7 Dialog Manager . . . 71
4.7.1 Dialog History . . . 71
4.8 Natural Language Generation . . . 72

5 Building the Robot Control Application 73
5.1 The Robot . . . 73
5.2 Design . . . 73
5.2.1 The Robot Model . . . 75
5.2.2 Design of the Multimodal Interface . . . 78
5.2.3 Robot Commands . . . 78
5.2.4 Writing a Grammar . . . 79
5.2.5 Developing New Resolving Agents . . . 80
`DistanceResolver` . . . 81
`SpeedResolver` . . . 81
5.2.6 Writing an API Class . . . 82
5.2.7 Writing the Fusion Manager Configuration File . . . 82
5.2.8 Modifying Robot-Side Code . . . 85
New Commands . . . 85
New Mission Type . . . 86
JNI Troubles . . . 87
5.2.9 Miscellaneous Tasks . . . 88
Mapping . . . 88
DISCIPLE V3 . . . 89
Video Transmission . . . 89

6 Evaluation 91
6.1 Evaluation of Project Approach . . . 91
6.2 Reusability . . . 92
6.3 Response Times . . . 92
6.4 Conclusions . . . 93
6.5 Future Work . . . 93

A Publication for the Fifth International Conference on Multimodal Interfaces (ICMI-PUI'03) **95**

B Glossary **105**

C Grammar **107**
C.1 Frames 107
C.2 BNF Grammars 111

D Fusion and Dialog Manager Configuration **135**

E Online Resources **155**

Bibliography **156**

List of Tables

2.1 Overview of how humans and computer sense and generate sensory information 24

5.1 The robot data model; the last two columns indicate who "controls" a property. A check in the **F** column means the Flatscape client (or another client) sets the property to request a change on the robot. A check in the **R** column indicates that the robot will update that property to synchronize the model with the robot's state 76
5.2 The resolver instances used in the robot interface 83

6.1 Implementation effort for the situation map tool 92

List of Figures

1.1 A bomb disposal robot inspects a potential suicide bomber for explosives in Israel 2

2.1 An example of a directed dialog 5
2.2 A mixed initiative dialog 6
2.3 Data flow in a typical dialog system 6
2.4 An example of a 'flight' frame 13
2.5 An example Galaxy-based system 20
2.6 An example AAA system with a broker team consisting of 3 members, and two client agents 22
2.7 FIPA reference model of an Agent Platform [26] 23

3.1 Data flow in a multimodal system 36
3.2 The layout of the framework's servers in the Communicator infrastructure 38
3.3 Abstracting away from the speech recognizer implementation . . 39
3.4 Different alignment cases for speech and mouse 42
3.5 The design of the fusion process 44
3.6 An example of how a slot is filled by resolving agents 46
3.7 The `ObjectLocator` and related classes 47
3.8 Hot spots in the framework for application implementation details 51
3.9 Natural Language Generation 53

4.1 A class diagram of the `galaxy.server-MainServer` and `galaxy.-server.Server` classes, and their subclasses in the framework 56
4.2 A UML class diagram of the speech recognizer classes 57
4.3 A UML activity diagram showing the steps from parser output to dialog manager 62
4.4 A UML Sequence diagram showing how resources are created . . 63
4.5 The `Resolver` class hierarchy and related classes 66
4.6 An example of a **spelling** parse tree node 67
4.7 An example of a **coordinate** parse tree node 67
4.8 The `ContextProvider` class hierarchy and related classes . . . 68
4.9 A stack of three context providers: eye tracker coordinates are clustered and buffered in a `ContextDataList` 69
4.10 The `ContextData` class hierarchy 70
4.11 An example dialog history 71
4.12 A UML class diagram of the natural language generation component 72

5.1 The Pioneer 2-AT all-terrain robot (pictured here with the gripper accessory, which CAIP's model does not have) 73
5.2 Flatscape with panels for monitoring and controlling the robot . 74
5.3 Overview of the DISCIPLE system architecture (from [39]) . . . 75
5.4 Model of the various modules used in the robot system and how they interact 77
5.5 Design of the multimodal robot client 78
5.6 An example of a `distance` parse tree node 80
5.7 The `Number` abstract class 81
5.8 The `Conversion` class 81
5.9 An example of a `speed` parse tree node 82
5.10 Diagram of the robot API class and related classes 83
5.11 A UML Sequence Diagram for a **`setDistance`** command 86
5.12 The map used by the robot for localization and path planning . . 88

6.1 Response times for five types of speech acts 93

Listings

2.1 An example of a BNF grammar 10
3.1 A universal framework 34
4.1 A fragment of the server definitions XML file 58
4.2 The configuration file that is created for the process monitor on Unix when the code in Listing 4.1 is transformed 59
4.3 The hub script that is generated when 4.1 is transformed 59
4.4 A sample frame declaration 64
5.1 The frame definition of the **move** frame 79
5.2 The **contextproviders** section from the fusion configuration file for the robot control application 84

Chapter 1

Introduction

This thesis describes the multimodal interface that was created at the Center for Advanced Information Processing at Rutgers University, New Jersey to control an all-terrain wheeled robotic vehicle. This chapter gives an overview of the reasons for implementing this interface, as well as the formal description of the project and the requirements that were determined. The last sections gives an overview of the structure of this thesis.

1.1 Motivation

1.1.1 Multimodal Interfaces

Science fiction films and series such as *2001: A Space Odyssey* and *Star Trek* show computers that can flawlessly understand human speech and answer complex questions with natural language. Research efforts over the past thirty years have brought this idea closer to reality, but the pictures portrayed in science fiction movies still remain just that: fiction.

Unreliable speech recognition, especially in noisy environments, and the need for more natural interaction with computers using not just speech but also other modes of communication that humans are familiar with, such as pointing, gaze, facial expressions, and tone of voice, have motivated development of multimodal systems: systems that combine different *modalities* to create a more robust and natural interaction between man and machine. By combining modalities into a multimodal system, we are able to come much closer to the ideal of the computer understanding what it is we want it to do.

1.1.2 Robots

Robots have proved to be very helpful tools in many types of work. In industry, robots relieve humans from dangerous and repetitive tasks. In military applications, robots can be used to investigate areas that may contain mines or enemy units without having to risk the life of a human soldier. Figure 1.1 shows how a robot is used by Israeli security to inspect a potential suicide bomber. In space, robots can explore planets and moons that humans could not survive on, due to climate or lack of oxygen. Autonomous robots can also perform these tasks

without continuous supervision, thereby relieving its operator to do other things until the robot notifies him/her that it has completed its task.

1.1.3 Controlling robots through a multimodal interface

Multimodal interfaces provide the tools to be able to naturally control robots for all kinds of tasks. The paradigm is that of remote voice-based control, but extended with visual and tactile communication. The ideal goal is for a robot to be able to replace a human in the most transparent way: the robot understands the commands it is given and reports back in the way a human would, using natural language.

Figure 1.1: A bomb disposal robot inspects a potential suicide bomber for explosives in Israel, about 12 miles from Haifa. Source: `http://news.bbc.co.uk/1/hi/world/middle_east/1976341.stm`

The goal of this project was to enable communication with a military robot through a multimodal interface. The robot is used for a variety of tasks, such as reconnaissance, securing an area, or fighting. Any human-computer interface in a military setting should aim to provide the speed and accuracy human-human communication would have. For quick development, existing off-the-shelf technology is to be used as much as possible. For parts of the system where this is not available, reusable components should be developed so that future development efforts *will* have access to off-the-shelf code.

In summary, the motivation for this work is two-fold. First, the desire to have a natural interface to control a wheeled robotic vehicle for military purposes. Second, the need for a reusable set of components for developing multimodal control systems.

1.2 Project Description

1.2.1 Background

The research done for this thesis is part of a project for CECOM, part of the US Army. The project's goal is to develop the next generation warfare technology, involving wireless communication, advanced user interface technology such as multimodal interfaces, and the use of robotic vehicles.

Robots are an excellent tool for high-risk operations such as mine finding or reconnaissance in enemy territory; losing a robot is much less severe than losing a soldier. To be able to use the robots to their full extent, they must possess some intelligence, or autonomous behavior. Implementing this was the research topic of another student here at CAIP [27].

An interface paradigm different from the traditional WIMP one is desirable in a military setting. Replacing mouse-keyboard interfaces with ones that use other modalities such as speech and gaze is useful for the following reasons:

- Military commanders will need to be able to work hands-free as they will need their hands for other tasks.

- War situations require quick decisions and quick man-machine communication; communication in speech-based and multimodal applications is potentially much quicker than for WIMP interfaces, especially in spatial tasks such as mission planning.

- Natural interfaces fit better into the military organization. Currently a human operator accepts commands from a higher ranked officer and dispatches them to the computer by typing. Direct communication with a computer by the higher officer will require little change to the style of operation, while freeing the computer operator to do other tasks.

1.2.2 Problem Description

The goal of the research described in this paper was to design and implement a natural interface for controlling wheeled robotic vehicles. These robots can be tele-operated or work autonomously. In the autonomous mode, a mission is set up that the robot is to later perform without any other user intervention. In tele-operated mode, the robot can be directly manipulated. The user interface used to control the robot should use natural language processing and possibly other modalities, such as gaze and gesture.

Non-functional requirements

In addition to the functional requirements described before, the following non-functional requirements were also deemed desirable:

- **Use of existing systems:** Existing systems, such as Galaxy Communicator, were to be looked into to see if they could be used.

- **Reusability:**The resulting system should be easily adaptable to other domains and applications.

- **Efficiency:** Direct manipulation using multimodal techniques requires a response time in the order of hundreds of milliseconds. The natural interface should therefore be fast.

- **Modularity:** Good software engineering practice prescribes that systems should be modular, that is, the dependencies between separate parts of the system should be minimal, making it possible to take one part out and replace it with another. Modularity generally increases reusability as well.

- **Interface with previously developed robot control software:** The software that is created builds on existing work done at CAIP on robot control and collaborative mission planning. The DISCIPLE [39] framework is used to communicate between the different team members – both human and robotic. Flatscape, a collaborative mission planning and situation map tool is used as the GUI framework in which the robot control must be integrated. Previous work done with the robot both for tele-operation and mission planning [27] can be used.

1.2.3 Approach

Since it was desirable to use existing technologies and methods as much as possible, an extensive literature survey was done investigating existing conversational systems, both unimodal and multimodal. Based on this survey, a selection was made of existing architectures and components that would be usable in the multimodal interface that was to be created. Furthermore, designs for new components were partly based on ideas from the papers found in this survey.

To ensure a proper system design, I decided to first create a generic framework for multimodal interfaces. The specific interface for this project could then be built on this framework, thereby separating application-specific code from generic code for multimodal interfaces, improving modularity and extensibility of the resulting interface.

1.2.4 Overview

This thesis will cover the research done for this project and describe the design and implementation of a multimodal interface for the robot control system. The next chapter describes the architecture of state-of-the-art dialog systems followed by a detailed look at each of the components. Based on this survey, I will describe in Chapter 3 the design of a generic multimodal framework. The implementation of this framework is described in detail in chapter 4. Chapter 5 will describe the design and implementation of a multimodal interface to control a robot using CAIP's mission planning and execution tool Flatscape and the DISCIPLE collaboration framework. This interface built on the framework described in the preceding two chapters. An evaluation of the system will be given in Chapter 6, followed by conclusions and suggestions for future work.

Chapter 2

Dialog systems

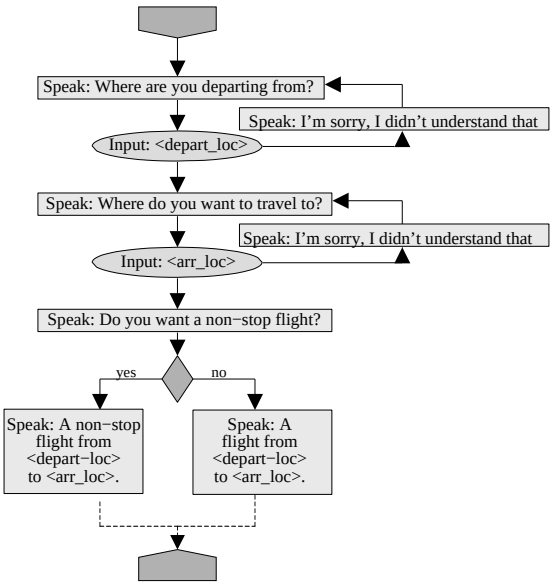

Figure 2.1: An example of a directed dialog

Spoken dialog systems are systems in which spoken language is used as the main form of interaction between human and computer. Three types of dialog systems can be distinguished based on the locus of control. A dialog system that follows a fixed, predefined flow of dialog determined by the system is called system-initiative, or system-led. This is usually implemented with a finite-state network, and therefore these systems are also called state-based or directed systems. An example is shown in Figure 2.1.

User-initiative systems are led by the user, who asks questions to obtain some information from the computer [44].

In mixed initiative systems, control is not fixed. Rather, the dialog system tries to fill a 'frame' with information given by the user, providing prompts as to what information is useful, and possibly verifying input that the user has given. Although the computer can ask questions to obtain the most useful piece of information at a point in the dialog, the user can choose to provide a different piece of information, or multiple pieces of information. An example is given in Figure 2.2. In the first turn of the dialog, the computer asks for a piece of information (the departure location) and the user provides not just that, but also the arrival location, which the system also registers. The second dialog turn shows the user not answering the system's question but instead providing another piece of information. Mixed initiative dialogs end when a frame has been filled. A frame can be filled in different ways: in the previous example,

Computer:	Welcome to the flight booking system. Where are you flying from?
User:	I want to fly from JFK to San Francisco.
C:	What time do you want to leave?
U:	Uhm, I want to get there at 3 pm.
C:	I have a United Flight leaving from JFK at 12:10, a US Airways flight ...

Figure 2.2: A mixed initiative dialog

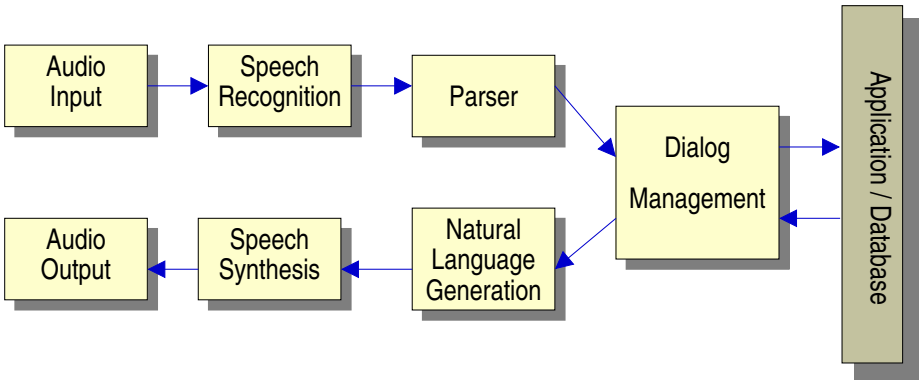

Figure 2.3: Data flow in a typical dialog system

either an arrival time or a departure time are a sufficient constraint and will satisfy one of the requirements for the frame being filled.

McTear [45] identifies another type of dialog system: agent-based. These systems go beyond cooperating with the user to fill a frame and attempt to solve a problem together with the user. The user and the system exchange knowledge and reason about their own actions and beliefs, as well as each other's input.

Smith [70] identifies four mixed-initiative "modes", depending on the computer's level of initiative: directive, suggestive, declarative, and passive. Mode switches can occur in a dialog in response to a user's request. For example, the computer may start out passive, but switch to a higher level of initiative when the user indicates he doesn't know what to do or even ask next.

2.1 Data Flow in a Dialog System

Typical dialog systems have a data flow as depicted in Figure 2.3. Data starts at a low level — samples from a digital audio converter obtained through a microphone or other audio input device — and is transformed to high level information in several steps — speech recognition and parsing.

The dialog manager takes decisions on what to do next based in this information. This may result in one or more actions such as retrieving data from a database, making a call to an application, and generating a spoken response or prompt for the user. Output from the dialog manager is then transformed back to low level data in several steps that mirror those of the incoming data: natural language generation, speech synthesis. The resulting output is sent to

a digital audio converter which converts it to audible sound that is emitted through speakers or headphones.

This chapter gives an overview of each of the components in a typical dialog system as shown in Figure 2.3. Each component's task within the dialog system is explained as well as how it relates to the other components. A brief description is given of the techniques and algorithms used to accomplish that task. A list of some available implementations concludes each section.

This survey is used to inventorize the available components of a dialog system and provide directions for implementing components that are not easily acquired.

2.2 Speech Recognition

Research on speech recognition can be traced back to Alexander Bell in the 1870's who wanted to create a machine that could visualize speech, the "phonautograph" [88]. This project failed, although it did lead him to invent the telephone.

The anecdotal first speech recognizer was a toy dog that, when his name "Rex" was called, would jump out of his dog house. The "speech recognizer" in this system was a circuit breaker that responded to frequencies of about 500 Hz (which were generated when someone shouted "Rex") and would break the circuit that powered the electromagnet that kept Rex in place.

Serious speech recognition research started in the 1970's, largely driven by the U.S. government under its Defense Advanced Research Projects Agency (DARPA) [52], notably the Speech Understanding Research that provided a total of $3 million of funding per year from 1971 to 1975 [55]. DARPA has made continuous small but consistent improvements in accuracy over time, until funding was all but dropped in the late 90's, stating that the speech recognition problem was essentially "solved".

The "holy grail" of speech recognition is to achieve 100% recognition. Anybody who has ever worked with a speech recognizer will know that this goal has not been achieved. 100% accuracy *can* be accomplished in very constrained cases. However, such recognizers are only useful in specific small tasks. Whether a speech recognizer is 'good' or 'bad' therefore depends on more than the accuracy. The following properties together define the quality of a speech recognizer:

1. the size of the vocabulary
2. whether training is needed to use the speech recognizer, that is, whether the recognizer is speaker-dependent
3. whether the user can speak naturally (continuous speech) or must speak each word separately (discrete speech)
4. the word error rate, that is, the percentage of words that are not recognized or are misrecognized
5. whether speech processing is done real time or offline
6. whether recognizer performance is independent of the speaker's gender and age

Considering all these properties, the ideal speech recognizer is a speaker independent, large vocabulary, continuous recognizer with a low word error rate. In practice a tradeoff has to be made. If high accuracy is desired, vocabulary must be kept small; if both a large vocabulary and high accuracy are desired, speaker-dependent training is unavoidable. Discrete speech (i.e. "speaking ... like ... this") used to be necessary in the early days of speech recognition to allow the speech recognizer to find word boundaries. Present-day speech recognizers can recognize continuous speech using vocabularies and statistical models.

2.2.1 A brief look at the theory

Speech recognition consists of a series of processing steps, including noise estimation and reduction, Fourier analysis, and a Viterbi search using hidden Markov models (HMMs). This results in several possible sequences of phonemes. Usually a statistical language model is used to determine the most likely sequence of words for this series of phonemes. In all, four (statistical) models are generally used: the *acoustic model*, the *pronunciation model*, *lexical model*, and *language model*. Mathematically, speech recognition can be represented as optimizing the following conditional probability:

$$\widehat{W} = \underset{W}{argmax}\ P(W|A) \tag{2.1}$$

Where A is an acoustic signal and W is a (sequence of) word(s). Using Bayes' rule, this can be rewritten as

$$\widehat{W} = \underset{W}{argmax}\ P(A|W)P(W) \tag{2.2}$$

which can be split up as

$$\widehat{W} = \underset{W}{argmax}\ P(A|U)P(U|W)P(W) \tag{2.3}$$

where

- $P(W)$ is the probability that a sequence of word W is/are spoken (determined by the language model)
- $P(U|W)$ is the probability that a pronounciation U is given to the word(s) W (pronounciation and lexical models)
- $P(A|U)$ is the probability that a sequence of phonemes A is observed for the pronounciation U (acoustic model)

For speaker-independent systems, the acoustic model is trained to generalize across a large amount of users, and the same model is used for each user. Speaker-dependent systems, on the other hand, maintain a separate acoustic model for each user that is optimized for his or her way of speaking. Pronounciation models are specific to a certain language, as are lexical models. Language models are also language-specific, but can be optimized for a certain domain to obtain better recognition in that domain. To accomplish this, a *corpus* of representative sentences is used to train the language model.

Early speech recognition used neural networks to match an incoming speech signal to a number of speech samples the network was previously trained on.

This works reasonable well for recognition of a small set of possible speech fragments, when they are spoken by the same person in more or less the same way, for example for recognition of the digits zero through nine, or for voice dialing on mobile phones.

A popular technique for speech recognition is the use of hidden Markov models (HMMs), which was introduced by Lenny Baum of Princeton University in the early 1970's [55]. A HMM models the probabilities of the user speaking a certain sound (or *phoneme*) given the frequency spectrum at a certain segment in speech and the previous phoneme. To find the most likely sequence of phonemes, a Viterbi search is used. Some recognizers mix neural networks and hidden Markov models [16, 93] to use the strengths of each and compensate for the weaknesses. For example, one weakness of hidden Markov models is the assumption that the current state transition is independent of the previous or future ones, for example, is not really valid for speech.

2.2.2 Products

Several commercial speech recognition products are available on the market, such as IBM ViaVoice, Dragon NaturallySpeaking, and SpeechWorks' OpenSpeech. These are all speaker-dependent systems, for high accuracy. Open source speech recognition products include CMU Sphinx, which is a speaker independent system generally using small vocabularies. Some universities have their own, internally developed, speech recognition software. Operating systems are also starting to integrate speech recognition, for example Apple's Mac OS X [5].

Besides the quality of the speech recognizer in terms of the word error rate and vocabulary size, criteria for determining which speech recognizer to use are:

- **API** — Does the speech recognizer interface well with our application? Does it support the programming language? Is the interface easy to understand and use?
- **Documentation** — Is the product and its API well-documented?
- **Price** — Is the speech recognizer affordable?
- **Setup** — Is the speech recognizer easy to set up and train (if needed)?

Sun has developed a cross-platform speech API for Java together with a group of major players such as Apple, AT&T, Dragon Systems, and IBM [73]. The JavaSpeech API, or JSAPI, provides access to all speech recognizers that support it, using a single interface. Microsoft has a similar interface in C++ for Windows called SAPI (Speech API).

From a multimodal point of view, a necessary function of a speech recognizer (and its API) is to provide word-level timestamps. Most current recognizers are able to provide the beginning and end time stamps for each word. Since speech is continuous, words don't have crisp beginnings and endings; the timestamps returned by the recognizer are therefore the ones which it has determined to be the most likely.

JSAPI can return timestamps, but doesn't require implementations to do so. IBM ViaVoice currently returns timestamps for each word for the best recognition result, but not for the N-best list (i.e. "–1" is returned instead).

2.3 Parsing

The output of a speech recognizer is a sequence of words, which have little meaning for a computer. The task of a parser is to organize and assign meaning to the speech recognizer output. Two types of parsing can be distinguished: *syntactic parsing* and *semantic parsing*. Syntactic parsing aims to determine the grammatical structure of the sentence, such as what type of sentence it is — a wh-question ("what...?", "where...?", etc.), a command, a yes/no question — and what its components are — subject, object, etc. No real meaning can be determined from a syntactic parse by itself. While syntactic parsing is sufficient for some systems, such as translation systems and systems that just store and retrieve knowledge without having to know what it means, a real dialog system needs to understand the meaning of what the user is saying.

Semantic parsing aims to categorize the parts of a sentence and the sentence as a whole, to thereby determine its meaning. Both types of parsing should be present in some form. Just categorizing the meaning of the words in the sentence does not give a complete analysis of its meaning, as the order of the words can be very relevant. Some parsers contain both elements in a single parser. [56] argues that a semantic parser that contains syntactic analysis as well is suboptimal.

Outside of spoken dialog systems, parsing is used in compilation, the process of converting program code in a higher level programming language into machine code. In this case, the source language is well-defined, known a priori, and completely unambiguous. The grammar used for this kind of parser is usually in BNF ("Backus-Naur Form") or a similar notation. Natural language parsing is different: there are many ways of saying the same thing, and the same sentence can mean different things depending on context. Also, speech recognition errors can cause parts of a sentence to be altered or dropped. Since some parts of a sentence don't contribute significantly to its meaning, humans can still understand what is said even if not every word in the sentence was heard correctly. A

```
article ::= THE | A;;
toggle ::= TURN | TOGGLE | SWITCH;;
turn_on ::= ON | ACTIVATE;;
turn_off ::= OFF | DEACTIVATE;;
living_room_lights ::= LIVING ROOM LIGHTS;;
bathroom_lights ::= BATHROOM LIGHTS;;
coffee_maker ::= COFFEE MAKER | COFFEE MACHINE;;
thing ::= living_room_lights | bathroom_lights | coffee_maker;;
turn_on_cmd ::= [ toggle ] turn_on [ article ] thing
              | [ toggle ] [ article ] thing turn_on;;
turn_off_cmd ::= [ toggle ] turn_off [ article ] thing
               | [ toggle ] [ article ] thing turn_off;;
```

Listing 2.1: An example of a BNF grammar

spoken dialog system is expected to do the same. Nonetheless, some simple dialog systems do use BNF grammars, such as the one shown in Listing 2.1. These systems either have a very large number of grammar rules to handle every way

a user might say something, or require the user to say commands in the exact wording that the system has been designed to recognize. This creates a very *un*natural user interface.

A much better solution is one where the fact that speech may contain disfluencies and that the speech recognizer might misrecognize some words is taken into account. This leads to a parser that uses a more flexible method of parsing that searches for the important parts in a sentence, known as *salient phrases*. The process of parsing given the possibility of recognition errors is called *robust parsing*. Robust parsers create a chart of all the grammatical phrases found in the input and determine the way in which these phrases can optimally cover the input sentence. For this reason robust parsers are also called *chart parsers*.

A very trivial, but sometimes functional, way of doing this is *word spotting*. With this technique, spoken text isn't actually parsed. Instead, the system looks for certain words. For example, a spoken language system for controlling household appliances might recognize the words and phrases 'on', 'off', 'living room lights', 'bathroom lights', 'coffee maker'. When the user speaks, the system will *spot* these terms and take action accordingly. For example, spotting the words 'living room lights' and 'on' would cause the system to turn on the living room lights. The user could say this in any of the following ways:

- Computer, could you turn on the living room lights?
- Switch the living room lights on, please.
- Living room lights on.

Even though this system is simple, it creates a natural experience for the user, who does not have to memorize computer commands to use it.

2.3.1 Parsers and Multimodal Systems

Parsers have changed very little in the past five years. Changes are needed to use current parsers in multimodal systems. Since timing information and confidence scores are not needed in unimodal applications, most parsers don't output these, even if they are provided by the speech recognizer input. Multimodal interfaces, however, can benefit greatly from this information. Timing information is used to synchronize speech with other modalities. Confidence scores are used along with confidence scores from other modalities to disambiguate and find what the user most likely intended to convey.

2.3.2 Products

Parsers usually have a clear separation between the actual parser code and the grammar. This makes them very reusable, since no program code needs to be changed. A new grammar is all that is needed to use the parser in a different application. The following parsers have been used in spoken language systems in the Communicator task:

- **CU Phoenix** — Phoenix [19] is a robust semantic parser that was developed at Colorado University. It parses text into frames, which consist of slots. Slots contain actual words or other slots, which results in a tree-like output. A BNF grammar is associated with each slot, but words can

be interjected between slots in the spoken text, which accommodates for disfluencies and different ways of articulating something. Phoenix is used in CU's Communicator system, as well as Carnegie Mellon's, and is open source.

- **TINA** — This is MIT's parser, used in all of their Galaxy systems [49]. Tina uses a combination of syntactic and semantic parsing, and can be trained on example sentences to build a probabilistic model that can improve parsing by choosing the most likely analysis. Tina uses strict parsing, but resorts to robust parsing if this fails [45]. Tina has recently been extended to support time stamps, to use it in MIT's multimodal systems [80]. MIT does not freely distribute Tina.

- **Gemini** — SRI's parser interleaves syntactic and semantic parsing [25]. Unification is used to match parts of a sentence to a grammar. Gemini has some advanced robust features such as being able to handle speech repairs [24].

- **JSAPI** — JavaSpeech-compliant speech recognizers support loading of grammars in "Java Speech Grammar Format", or JSGF [75]. JSGF looks like a mix of BNF and Java, and is essentially BNF. A developer can specify 'tokens' in the JSGF file that are output whenever a grammar rule is applied. This provides some abstraction from the actual language or wording used by a user. Nevertheless, as mentioned, the use of strict BNF makes it difficult to allow truly natural speech input, and for serious dialog systems, this can hardly be considered an option. JSAPI is just an interface, but it is implemented by IBM ViaVoice, the upcoming CMU Sphinx 4, and possibly other products.

Additionally many parsers are available that were written in interpreted languages such as Scheme, Lisp, and Prolog. Because they are interpreted, their performance will inevitably not be as good as that of parsers written in compiled languages.

2.4 Dialog Management

The dialog manager's task is keeping track of the dialog, using the user's input to update its internal representation of the conversation and generate a new question or a reply, possibly making queries or calls on a database or application.

As mentioned in the introduction, dialog managers can have a fixed, preset dialog flow based on a finite state model, or it can be more flexible and use 'forms' or 'frames' that are filled as the user delivers more information [32]. One could say that the first type of dialogs are 'procedural' where the latter are 'declarative'. Since finite state dialogs are too limited for truly natural human-computer interaction, we will focus on frame-based dialog managers.

When a dialog manager receives a parse from the parser, it will:

1. Resolve ambiguities in the parse tree and the information into a canonical form,

2. Combine this information with the current discourse frame,

3. Decide if the current frame has sufficient data,
4. Prompt for user input, get data from a database or application, and/or give user feedback.

There are many different ways of expressing the same piece of information in a frame. One of the dialog manager's tasks is to convert parts of the parse tree into a canonical form that can be used in a frame. For example, a date can be expressed absolutely — "August 25th, 2003" — or partially — "the 25th" — in which case the missing pieces of information can be derived from the context, e.g. the current month is August of 2003, or the conversation was on dates in July 2004; relative dates — "in three days", "next Monday" — are resolved in a similar way. The same goes for pronouns, for example:

flight	
airline	
depart-loc	JFK
arrive-loc	
depart-date	08/25/2003
arrive-date	
depart-time	
arrive-time	

Figure 2.4: An example of a 'flight' frame

Computer:	I have two flights available: a 7:30 United Airlines flight and a 9 o'clock Delta Airlines flight.
User:	Yeah, that one.

The phrase "that one" is ambiguous. But using some semantic rules, we can determine the user is referring to the 9 o'clock flight.

Some or all of a frame's slots or can be marked as required. When all required slots have been filled, the frame is considered complete and action can be taken by the dialog manager, such as querying a database or making an application call. Depending on the result of the query or call, user feedback may need to be given through the natural language generation component, possibly updating the dialog context (since this is the computer's turn in the dialog). If required slots have not yet been filled, the system can prompt the user for one of the empty, required slots.

Some systems separate parts or all of the context resolution in an autonomous component. For example, the CMU Communicator [15] has a date resolution component, which resolves phrases like "tomorrow", or "next Tuesday". MIT's Galaxy systems have a distinct context tracking server [28, 67].

2.4.1 Products

The dialog manager is the most application-specific part of a dialog system. Therefore, most dialog managers that are publicly available are yet ill-suited for use in a new system. MIT uses a separate discourse manager and dialog manager, both of which do manage to separate application-specific data from the application-dependent functions. Dialog manager data is stored in the form of rules [68]. MIT's dialog manager is, however, not publicly available. The same goes for the "Discourse" component, a context tracking server which is described in [28].

Some (graphical) finite-state dialog modeling tools are available, such as OGI's CSLU Toolkit, but as mentioned, finite-state dialogs are not powerful

enough for real-life applications.

2.5 Natural Language Generation

Natural language generation mirrors parsing. The goal of parsing is to determine the semantics of a sentence. The goal of natural language generation is to build a sentence given semantics. The most straightforward way of doing this is template-filling. With this method, there is a template for each class of responses the system can give. The template contains blanks that are filled in to create a specific response. For instance:

> ⟨**airline**⟩ flight ⟨**flight-no**⟩ with destination ⟨**arrive-loc**⟩ departs at ⟨**depart-time**⟩.

Filling in the four fields **airline**, **flight-no**, **arrive-loc** and **depart-time** will yield a valid English sentence. The disadvantage of this approach becomes obvious when many of the same class of sentences are generated. Humans will try to introduce some variation and leave out information that can be determined from the context. For example:

> United Airlines flight 843 with destination San Francisco departs at ten thirty. Flight 847 departs at noon.

In this example, information that is the same as in the previous sentence is not mentioned the second time. Natural language generation using the template approach cannot do this, and therefore the output will sound unnatural in this case. A possibility is to provide multiple templates for the same class of sentences. This will introduce some variation. However, introducing ellipsis (omitting words or phrases that are implied by the context) is far more complicated.

More advanced natural language generators use a multi-step process in which high level communicative *goals* are consecutively planned and realized as a spoken sentence [64]. This process is more complex as it builds a sentence out of fragments of generated text, rather than just filling in blanks. The system has to have knowledge of the grammatical structure of the language being used and know how to put pieces of text together to create a valid sentence that fulfills the communicative goals. For example, two sentences "Flight 843 departs at ten thirty" and "Flight 843 departs from gate 10" might be merged into "Flight 843 departs from gate 10 at ten thirty". Since there are multiple ways of doing this — the previous sentence could also have been generated as "Flight 843 departs at ten thirty from gate 10" — a truly natural system should use some form of heuristics to create the most "natural" version of the sentence. The system described in [64] can be trained to learn exactly that.

2.5.1 Products

MIT uses its **GENESIS** natural language generation component within the context of its Galaxy architecture. A new version was written in 2000 [7]. GENESIS is used by MIT to convert semantic frames into spoken output, but also to generate non-natural language such as SQL and HTML.

AT&T's **FERGUS** [6] system employs a hybrid syntactic/stochastic approach. The output quality has been determined to be better than when using just syntactic or stochastic models singularly [64].

ASTROGEN [22] is a freely available natural language generation system written in Prolog, but is still under development. It can do merging and pronominalization. Customizing ASTROGEN involves creating a Prolog file with definitions for the nouns and verbs to be used.

KPML [83] is a multilingual natural language generation and grammar creation tool running in a LISP environment. It is unclear if and how the natural language generation part can be integrated into an application. Although KPML is free, it requires a LISP environment that is not free.

2.6 Speech Synthesis

Simply put, speech synthesis, or *text-to-speech* is the inverse operation of speech recognition. Because the two are opposites, the types of challenges are very different for each. For speech recognition, the main challenge is to recognize spoken text in the face of enormous variability — even the same person can pronounce words in different ways depending on the context, a voice can change under influence of fatigue or differing air conditions, background noises can influence what is captured by a microphone, etc. With speech synthesis, however, the challenge is to *create* variability. To create natural-sounding speech, a speech synthesizer must know how to pronounce a word given the context, it should use the right intonation and pause at certain points in the sentence. Lack of these features in early speech synthesizers (and even some current ones) has resulted in the typical "computer voice" or "robot voice" which people believe to be the only way computers can talk: monotonously, at a constant speed, occasionally mispronouncing or misemphasizing words and with a voice that does not resemble a human voice at all. While fixing these problems might seem at first to just be the proverbial "icing on the cake", natural sounding speech is in fact essential for humans to be able to understand computers well. Mispronounced and misemphasized words can cause great confusion and can be fatiguing, just as listening to a non-native speaker.

2.6.1 Speech Markup

Humans can generally determine prosody from plain text, based on their understanding of the text and previous experience with reading. However, for computers it is difficult to do this, which has resulted in introduction of *speech markup languages*. These languages add information to plain text to aid the computer in speaking text. These languages include JSML [76, 77], SSML [78], and SIML [61]. These language attempt to structure text. It is therefore not entirely surprising that a number of them use XML, which is a structuring language. As an example, an SIML-annotated sentence might look like this:

The tags in most speech markup languages are low-level, indicating hints for volume, speed, and pitch. Providing higher level information, such as which parts of a text are subsentences, or which parts need emphasis, might be more desirable. This leaves implementation of how to derive actual speech characteristics from this high-level information to the speech synthesizer. This is

```
<u.pro>Flight <phrase rate=0.8> 8 5 4 </phrase>
<pause dur=50 durunit=ms> with destination
<phrase rate=0.8> Los Angeles </phrase>
departs at ten <pause dur=5 durunit=ms> forty </pause> </u.pro>
```

advantageous because the speech synthesizer can be considered the expert in generating waveforms from text, and choosing pitch, speech rate, and volume is part of that task. The natural language generator which generates the speech synthesizer's input, however, knows only about sentence structure and meaning, so requiring it to also generate pitch, speed, and volume information would violate the expert paradigm.

2.6.2 Speech Synthesis Techniques

The most common form of speech synthesis is concatenation. The speech synthesizer seeks out pre-recorded or generated audio samples, or *speech units*, that, together, will most closely articulate the desired sentence. Signal processing is done on the samples to attain the desired pitch and duration for each part. Finally the processed speech units are concatenated — seamlessly in the ideal case — and the resulting waveform can be send to the audio hardware.

In an alternate approach, no signal processing is done in the speech units. Instead, a large number of samples are recorded for each phoneme, with different prosodic characteristics. Upon synthesis, the system must select the sequence of phonemes that best approximate the target phoneme string. The possible phonemes can be considered as a state transition network, where the state occupancy cost is the difference between the target phoneme and the phoneme in that state. The optimal sequence can then be chosen with a Viterbi search [34].

In the most compact form, the speech units are used separate phonemes. That is, there is a sample for each of the 36 possible English phonemes (this number is debatable, see [8]) that the system can choose from. This output in this case does not sound very natural, because the transitions between phonemes will simply be the result of linear interpolation of the audio signals of the two phonemes. Better is to have a sample for each phoneme in the case of every phoneme that can follow it. This adds up to a theoretical $36 \times 35 = 1260$ *diphones* that would be needed. However, not every combination of phonemes is used in English, so the number in practice can be smaller. Multiple variations of each diphone may be available with different characteristics, though; the number of phonemes will generally be around 2000 [20].

For more natural sounding speech, a more application dependent speech synthesizer can be used with longer units of prerecorded speech, such as fixed prerecorded phrases ("New York") or prompts ("Welcome to the flight reservation system. How may I help you?"). This approach is called unit selection. While unit selection yields more natural sounding speech, quality among samples is often inconsistent and a broad range of samples are needed with different prosodic characteristics to truly achieve natural sounding speech. Selecting the speech units with the right prosody characteristics at run time is difficult [20].

2.6.3 Products

Festival [84] is a mature concatenative text-to-speech system that was developed at the Center for Speech Technology Research at the University of Edinburgh. Different voices are available (American and British English, Welsh, and Spanish) and new ones can easily be added on. Festival is written in C++ but also has a Scheme interface. Speech markup is done in Sable, an XML-based language developed by Bell Labs, Sun Microsystems, AT&T and the University of Edinburgh [10].

IBM **ViaVoice** supports speech synthesis through its interfaces, including JSAPI. JSML is not fully supported (yet), so it is not possible to give prosody hints. Because of this, the generated speech can sound unnatural.

CMU's Communicator travel planning system uses a modified version of Festival with prerecorded samples of many of the prompts and phrases used. Because of this, the speech sounds very natural.

2.7 Fusion

Multimodal systems combine data from multiple modalities. At some point in the system, the streams of data need to be merged into a single data representation. This process is referred to as *multimodal fusion*, or fusion for short.

Fusion can be done at several levels in a multimodal system, and fusion is labeled accordingly:

- *Early fusion* — Also called *feature-level fusion*, because signals are integrated at the feature level. This usually means concatenating the feature vectors from each modality and using statistical methods such as hidden Markov models or temporal neural networks [92] to classify the 'super-vector'. Classification is expected to be more accurate with features from different sources. Feature-level fusion is appropriate for multimodal streams that are synchronized and correlated [80, 60]. The most familiar example is fusing acoustic information from speech with images of the speaker's mouth, to create more accurate speech recognizers [36, 90, 91, 89]. The 'phonemes' generated by speaking and the 'visemes' that can be observed by watching lip movements are highly correlated. Hidden Markov models and neural networks need to be trained with real data, which can be a problem. Due of the relative novelty of multimodal interfaces, multimodal data is scarce and expensive [60]. Collection of audio-visual data for Dutch bimodal speech recognition is described in [91].

- *Late fusion* — Also called *semantic fusion*. Fusion is typically done after all the inputs are received and parsed. It occurs at a higher, semantic level. Late fusion allows fusion of inputs that are not synchronized, but overlapping or entirely disjoint. In addition, more complex relations are available because semantic relations between inputs can be used as opposed to the strictly numerical ones in feature-level fusion. Unimodal recognizers are used at the input level, which make use of training data that is readily available [60].

To fuse data from different sources, we need to convert them into a common meaning representation. Different choices can and have be made for this rep-

resentation. [69] uses spoken language. This means fusion can be done at the parse level. This seems elaborate, however, as language is first generated and subsequently parsed. [80] uses semantic frames to represent meaning. Fusion involves merging the slots in those frames. [41] uses a "slot filling" approach, where text is used as representation. Whether and how this text is structured in any way is not mentioned.

2.8 Fission

Fission is the opposite of fusion. The reason for using fusion is to create a more natural experience for the user by allowing him/her to use other methods of communication than just speech or just mouse, and aid the computer in understanding what the user wants by providing multiple modality streams that can disambiguate each other. Fission creates a more natural interface by distributing data over multiple output channels. Additionally the system's output can be better understood by the user, since outputs may be partially redundant. Cohen [17] shows that a talking face added to speech output can significantly increase understanding for the human listener when the speech is noisy.

I first found the term 'fission' in [12], but it seems to be becoming more popular. Another term used specifically for synchronizing an on-screen animated face with synthesized speech is "multimodal speech synthesis". This currently seems to be the most mature form of fission, with research being done at the Royal Institute of Technology in Stockholm, Sweden [23], and University of California at Santa Cruz [82].

2.8.1 Products

Baldi is the talking face developed at UCSC and is distributed as part of OGI's CSLU Toolkit. Baldi synchronizes synthesized speech output with an image of a talking face on the screen; the two output modalities are tightly coupled.

2.9 Multi-agent Architectures

Many dialog systems found in literature are built on top of some kind of agent-based architecture. These architectures supply a hub-and-spoke infrastructure where various components (or agents) of a dialog system communicate via a central routing agent or hub. The agents communicate by sending messages via this central hub. Messages are structured according to some high-level representation and are transported using TCP/IP.

The advantages of such an architecture are:

- **No programming language lock-in**: All dialog components need not be written in the same programming language, as long as they can communicate using the high-level representation agreed upon
- **Transparency in distribution and location of components**: Resource-intensive components can be moved to a different machine, distributing

the load and making the application more scalable. Because all communication is done via the central routing agent, components don't need to know the exact location of the receiver of their messages, since this information is kept at the routing agent.

- **Independence of components**: Since components run as separate processes, one can be restarted in case of failure or component upgrade without having to bring the entire system down.

Disadvantages with respect to alternatives such as (remote) procedure calls, DCOM or CORBA are:

- **Efficiency**: Potentially slower operations because of TCP/IP overhead and overhead of building and parsing messages
- **Provenness**: The multi-agent architectures described in the following sections have been used to build research prototypes, and although the resulting products have in some cases been distributed to various clients, the architecture has not been proven in the degree that CORBA or DCOM have been.
- **Ease-of-use**: multi-agent based systems require multiple concurrent processes to be started for the application to run. Even though, in the single-machine case, this task is usually taken care of by a 'process monitor' which will start all the individual processes, it can never be completely hidden from the user and therefore might be confusing.

Examples of multi-agent architectures are:

- Galaxy Communicator
- Open Agent Architecture (OAA)
- Adaptive Agent Architecture (AAA)
- Java Development Framework (JADE)

A common mistake is to think these architectures will provide a complete dialog system or multimodal system, when in fact all they do is provide a transport layer, or infrastructure, on which those types of applications can be built. Therefore, availability of components is one of they key points to consider when choosing an architecture.

The three aforementioned architectures will now be described and evaluated.

2.9.1 Galaxy Communicator

MIT's Galaxy Communicator provides a communication infrastructure set as the reference architecture for dialog applications by DARPA, and is used by MIT [67], Carnegie Mellon [15] and Colorado University [21] in their dialog systems. Galaxy is now being maintained by MITRE. MITRE describes Galaxy as "an open source distributed, message-based infrastructure optimized for dialogue system design" [50]. Galaxy is a modular hub-and-spoke architecture that uses semantic frames as the units of communication between the components, which are called "servers". The frames consist of a name that specifies the operation to

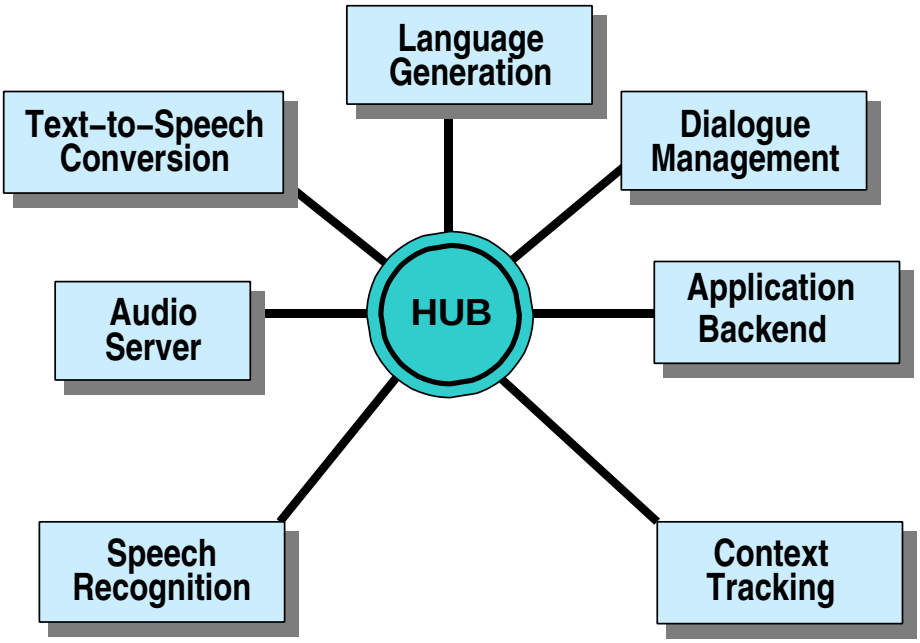

Figure 2.5: An example Galaxy-based system

carry out on a server, and a set of zero or more key-value pairs that parametrize the operation. Frames have a formatted text representation that looks as follows:

Destination can be omitted, in which case one of the servers that support the operation is chosen at random.

Communication between servers is done over TCP/IP and all frames are routed through a central hub. That is, servers don't communicate directly, but send messages to the hub, which routes it to the correct server based on the frame's name and possibly routing rules defined in the hub. An example system is shown in Figure 2.5. The hub is configured using a file that describes which servers that it can connect to, or that will connect to it. This file also lists the operations each server supports. This means that the number and type of servers in a Galaxy-based system is static. Routing rules can be specified in the configuration to reroute frames or changes their names and/or key-value pairs. This can be used to make two servers work together that use different operation or key names.

Whenever a message comes into the hub, it first determines whether it has an explicit destination. If so, the frame is routed to that server. If not, routing is done based on the operation ? the frame's name. The hub will first look for a script with this name. If found it is executed. If not, the hub looks for a server that is declared to be able to handle the operation the frame states. If one or more servers are found, one is picked at random. If none is found, the frame is discarded and a warning message sent to the sender.

For high-bandwith data transfer between two servers, the hub can set up a connection between the two, and then allow them to communicate outside of the hub through a back channel. This is called brokering (the hub is the broker in setting up the connection).

MITRE distributes an open source toolkit based on the Galaxy infrastructure that is eventually to provide a complete set of dialog system components and can potentially supply components that are usable in a multimodal system. However, the toolkit has by far not reached a complete state; the majority of current components are audio servers and speech recognizer wrappers.

Both Carnegie Mellon and Colorado University have built systems on Galaxy that are publicly available. Some Galaxy-compliant components, specifically the speech recognizers CMU's Sphinx-II and CU's Sonic and the CU Phoenix parser used in these systems can be used to build others. A large part of their systems, such as the dialog manager and natural language generation, is domain dependent, however, and these components cannot be easily used in other domains (i.e. other than the travel domain).

Galaxy was written in C, but bindings are available to other languages, such as Java and Lisp, so that Hub-compliant servers can be written in a variety of languages, and existing software can easily be made to work within the Galaxy infrastructure.

In short, pro's of Galaxy are broad support and proven technology, a con is the fact that configuration of servers is static.

2.9.2 Open Agent Architecture (OAA)

OAA is SRI International's distributed software architecture and is used extensively in their own projects, as well as — interestingly — in DARPA's BioSPICE project [72]. (This is interesting because Galaxy is DARPA's reference architecture for dialog systems. BioSPICE, however, is not related to dialog systems). OAA seems to have broader application — distributed problem solving in general — than Galaxy, which is used strictly for dialog applications.

OAA is in many ways very similar to Galaxy. However, the terminology used is that of the field of artificial intelligence, with emphasis on problem solving. Each entity — the hub and each of its clients — is called an agent, and agents work together with the user they to reach a goal. The hub is called the facilitator, the other agents are called client agents or clients. Communication is done using a specialized language ICL (Inter-agent Communication Language).

ICL is used to send events to agents. Events have a name and parameters, much like a function call. One special type of event is the **`ev_post_solve`** event, which is equivalent to an operation frame in Galaxy. The facilitator uses unification (from lambda calculus and logical programming languages) to match an event with an agent's capability, or solvable. Just as in Galaxy, an explicit agent can be specified to handle the solvable, but in general this will be left for the facilitator to decide, reducing hard-coded dependencies between agents [42].

[42] describes the intent to implement direct communication between servers, similar to brokering support in Galaxy. Since OAA has been improved since [42] was written, this may already have been implemented at this time.

We can conclude that OAA provides an architecture that supports better abstraction from agent topology than Galaxy, but is aimed more at agent-based systems in general and therefore has less publicly available dialog-related components.

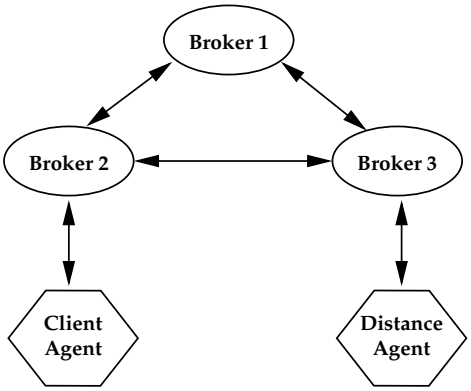

Figure 2.6: An example AAA system with a broker team consisting of 3 members, and two client agents

2.9.3 Adaptive Agent Architecture (AAA)

A key point of failure in both Galaxy and OAA is the central hub or facilitator. This is also the bottleneck for communication. The Adaptive Agent Architecture (AAA) features a team of facilitators, or brokers, instead of just one. The agents in this broker team have a mutual goal to maintain the broker team, that is, to ensure that a minimum number of brokers is always present in the team. When a broker dies, and the number of team members drops below the minimum, the broker team assumes a joint goal to restore the number of members. Client agents are capable of spawning brokers at the request of a broker. This is used to restore a broker team to its original configuration when one of its members fails. When an agent loses its connection to the broker team, either because it fails or because its connecting broker fails, the broker teams assumes a goal to reconnect to the agent.

An example AAA system is shown in Figure 2.6. A broker team consisting of three facilitators serves two clients.

AAA is backwards compatible with OAA, and therefore any OAA system should run using the AAA facilitator. Not all AAA-specific features have been fully implemented yet [58].

An advantage of AAA is greater robustness in the face of facilitator failure. However, since AAA is not complete, very few AAA-compliant agents are available, and although AAA is OAA-compatible, OAA dialog components are also scarce, as stated in the previous section.

2.9.4 JADE

JADE is a software framework that simplifies development of agent applications that comply with the FIPA specifications for interoperable intelligent multi-agent systems [26]. FIPA [29] aims to create a complete specification for multi-agent systems, specifying not just the communication *language* used in the messages that are passed between agents, as is done in the agent systems discussed in the preceding sections, but also the *semantics* of these messages. It is a pure interface specification, implementation details are entirely left up to the implementor. This includes the action transport method used. FIPA only

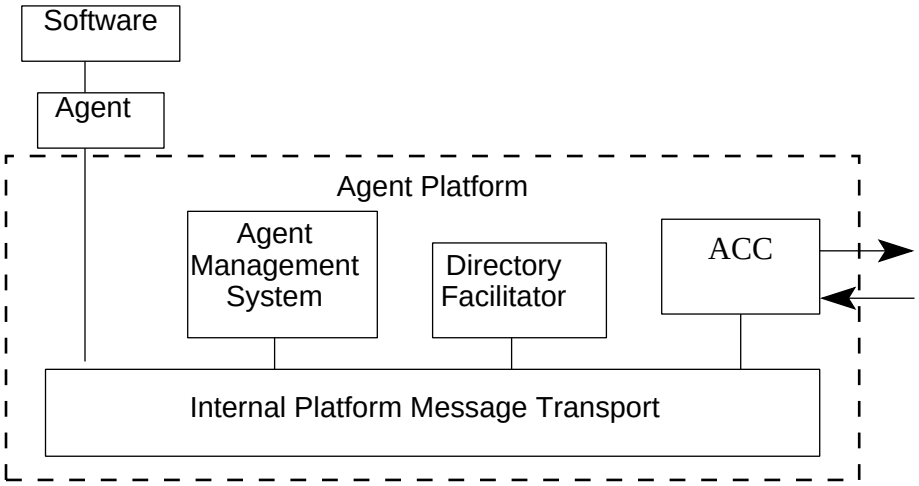

Figure 2.7: FIPA reference model of an Agent Platform [26]

dictates that messages are to be transported at plain text. The FIPA reference model is shown in Figure 2.7.

Agents can use an agent directory service to find out about other agents in the system, or, on a higher level, use a service directory to find out which services are available and which agents provide them. The message structure is written in an agent communication language (ACL), its content is expressed in a content language. Messages are transported as text, although exceptions can be made for low-bandwith links, such as wireless links [30]. As OAA and AAA, FIPA and JADE seems to be a general-purpose agent specification and architecture. I have not found attempts to use JADE for conversational systems.

2.10 Multimodal Systems

To most people outside of the field of human-computer interfaces, *multimodal* is a completely new term. To those in the field, it has become the buzz word that represents the future of computers and the solution to the gap that has always existed between man and machine: the enormous differences between communicating with another human and communicating with a computer. Because of its buzz, multimodal has been used increasingly both appropriately and inappropriately. Therefore it is useful to define what we mean by it. [12] dedicates more than four pages to defining the term. We use some of that information to give a slightly shorter summary here and refer the reader to [12] for their full description of the term.

The word multimodal is built from two words: *multi* and *mode*. By mode we mean a physical sense that is used in communication. The five human senses are sight, sound, touch, taste, and smell. The latter two are not pervasive in human-human communication, and completely absent in human-machine communication, with the exception of computer malfunction, where smell may indicate the burning of one of the components. Previous experiments with smell television ("smell-o-vision") have been unsuccessful [63] and their application

Table 2.1: Overview of how humans and computer sense and generate sensory information

sense	human		computer	
	input	output	input	output
sight – *visual*	eyes	hands (gesture); face (facial expressions)	camera (face recognition, gesture recognition); eye/gaze tracker	screen/terminal
sound – *auditory*	ears	mouth (speech and other sounds); hands, feet (clapping, stomping)	microphone + sound card (+ speech recognizer)	(speech synthesizer) + sound card + speakers
touch – *tactile*	skin	hands and feet	touch screen, pen & tablet, tactile glove	tactile glove with force feedback
data / information / knowledge	reasoning		data ports / network	

in computers is even less likely to occur.

This leaves us with three senses. To provide data to the senses we can generate sound, vision, and touch. Table 2.1 provides an overview of the three senses and how humans and computer use them and generate output for them. An extra, abstract sense, 'data/information/knowledge', has been added as well. This represents external knowledge used to solve ambiguity. For a human this can be a sense of the situation they're in, knowledge on what 'makes sense'. For a computer, contextual information could be data from a GPS to provide information on where the computer is, or the history of a dialog to determine what a user means when a word is omitted (ellipsis). This is information humans derive through reasoning.

Unimodal interfaces accept just a single input modality. For example, telephone-based voice response systems are unimodal, as they use only speech.

A multimodal interface is created when a machine receives input from multiple senses at once. For this reason, another name for this type of interface is *multisensory*. This excludes interfaces using one or more modalities for output yet using just a single modality for input from being called multimodal, such as systems that take speech input and generate speech and visual (i.e. on-screen) output. A true multimodal interface must accept multiple input modes, such as speech, gaze, pen, etc. and combine them to derive user intention. A system that generates output in multiple modalities, such as speech and visual, as well as having multimodal input is called 'doubly multimodal' by [12].

The purpose of using a multimodal human-computer interface is to gain a better insight in what the user's intention is when interacting with the machine. Traditional interfaces ignore much of this information, leading to frustration as the user expects the computer to "know" what he/she wants.

2.10.1 Advantages of Multimodal

The goal of using multiple modalities is twofold. The first is mutual disambiguation, the second is naturalness.

Mutual disambiguation is the act of using information from one modality to fill in or correct missing or ambiguous information in another modality: the weaknesses of speech are compensated by the use of gesture, and vice versa [9]. For example, in an application for travel planning, the computer might not be

sure as to whether the user said "I want to travel to Boston" or "I want to travel to Austin". By using information from another source as well, such as images of the speaker's lip movements while speaking or gesture information on where the user was pointing on the map, the system can be more certain of what the speaker intended.

The reason multimodal interfaces are more natural than either traditional WIMP interfaces or even unimodal speech interfaces, is that humans communicate multimodally. Our brain is designed to process multiple streams of information to assess the state of the world. This is why we use our hands when we speak, reflect the semantics of what we are saying in our facial expressions, etc. This is also why we have a harder time understanding people on the phone, and tend to pay more attention to someone's face when talking in noisy situations, such as in clubs. Similarly, research has shown that people instinctively use the most appropriate modality or combination of modalities for a task and switch to another set of modalities when a command is not understood the first time around [59]. This self-correcting behavior results in better performance and less frustration compared to a situation in which users are constrained to using a single modality that may not be optimal for the task at hand.

Additionally, providing multiple input modalities makes computers accessible to users with disabilities. For instance, users unable to use their hands for mouse or keyboard input can use speech and/or gaze to control the computer.

Research shows that gaze and gesture are particularly suited for applications that involve spatial information. Speech, on the other hand, has a great advantage for non-visible objects, such as off-screen ones [9]; these can be referred to by speech, but not by a single mouse or gesture movement (the object can be scrolled onto the screen and then accessed, but this is a relatively time-consuming process). Combining speech with gaze or gesture is, therefore, a logical step that combines the strengths of both modalities to provide a more pleasing user interface.

Oviatt et al. [60] state that multimodal pen-voice interaction can result in a 10% increase compared to just speech. (An even greater increase can be expected when using gaze and speech, due to the great speed of eye movements.) Also, 36% less errors are made, and speech is more fluent due to simpler constructions. Users have an overwhelming preference to multimodal interaction.

2.10.2 Multimodal Devices

Multimodal interfaces draw their strength from using devices that convert human input into data the computer can understand and use. These devices are often initially designed for unimodal use, but combining them with speech yields a more powerful and natural interface. We will quickly review some relevant devices.

Gaze Tracker

The goal of using other modalities besides speech is understanding what the user's intent is. A lot can be told about the user's focus of attention, and therefore his/her intent, by looking at eye movements. This is what a gaze tracker tries to capture. Different gaze tracking techniques exist [31]. A popular one, because it is minimally intrusive and fairly accurate, uses an infrared light

source to send a ray of light onto the eye, and an infrared-sensitive camera to capture the eye. In the image of the eye, the pupil is located (a dark area surrounded by a light area). The pupil is approximated by a circle, and the center of this circle is calculated. Then the 'glint' — the bright corneal reflection of the infrared light source — is located. From the location of these two points, the angle between the user's gaze direction and the screen's perpendicular can be calculated. Given the distance between the user and the screen, the fixation point on the screen can be determined.

Tactile Glove

Tactile gloves are commonly used in 3D or virtual reality environments. A tracker attached to the glove determines the direction the user is pointing, as well as the orientation of the hand. This enables the glove to be used as a pointing device. Additionally, feedback can be given to the user. A simple mechanism is vibration. The Rutgers Master II-ND glove [54] uses pneumatic pistons to provide a feedback force against the fingertips.

Touch Screen

Touch screens have been around for a while and provide access to information in kiosk-type situations, such as ATMs, ticket machines at train stations, and catalog information systems in stores. They avoid the need for a keyboard or mouse, and are more intuitive as the user can directly touch objects on the screen, instead of through an intermediary device such as a mouse.

PDAs effectively also come with touch screens, although these are usually operated with a pen or 'stylus'.

Different types of touch screen technology exist. *Analog resistive touch* [71] is relatively inexpensive and accurate. A glass plate with a conductive, transparent layer is placed over the touch screen. On top of that a plastic plate is placed, with its conductive side facing the glass plate. In between the plates are thousands of dots keeping the layers separated. When the screen is touched, the layers make contact and the position of the touch is calculated.

Microphone Array

Although not a new modality, microphone arrays can be of great aid in creating a more natural multimodal user interface. Traditional microphones require the user to stay close to the microphone at all times, which can be constraining. Head-worn microphones are unnatural as they require taking the microphone and putting it on before using the system. Microphone arrays create a large area in which speech can be captured, while keeping out noise and reverberation [41].

2.10.3 Current Multimodal Systems

The first known multimodal interface was built in 1980 by Richard A. Bolt [11]. It provided an interface in which shapes could be created, moved, copied, removed, and named using a combination of speech and pointing, for example "put that to the left of the green triangle", "copy the green triangle there", "copy that there", "call that the calendar". Fusion was done at the parse level. Every time an pronoun or deictic reference was recognized, the system

would immediately see where the user was pointing and resolve the reference. The system also had an ability to learn new words. When the user said "call that...", the system would tell the speech recognizer to switch from recognition mode to training mode so that the name that the user gave the object would be learned.

Quickset [18] is a multimodal system developed by the Oregon Graduate Institute. It features a map-based application that can be controlled using either or both of speech and pen gestures. Pen gestures can be used to indicate the target of a deictic reference, such as "show me the hotels in *this area*". Pen can also be used independently. A single command gesture can be given, for example drawing an X through an object to have it removed. Also, handwriting can be used in combination with selection gestures, such as drawing a circle around an area on a city map and writing "restaurants" to find out which restaurants are in that area.

MIT is developing a multimodal system based on the Galaxy architecture, which they are extending to support dialog applications with multimodal input. Their system allows one to build a solar system by creating, modifying, and naming planets that are placed around a sun [80]. The system currently accepts speech and mouse input, in addition to having a 'traditional' WIMP GUI. A unique property of this system is that the objects in it are dynamic. The planets continuously rotate around the sun, and choosing one with pen or gesture involves tracking it for a short while. During multimodal fusion, the system must look back at the objects that were at the location the user was pointing at, at the time he was pointing.

The Future Combat System developed at CAIP features a map on which military missions can be planned by creating and positioning units, such as tanks and infantry. The system is hands-free, using gaze tracking for object and location selection and speech for command input.

Advanced multimodal systems currently only exist as research prototypes [80]. Speech recognition is making its way to the desktop [53, 5], but the only true multimodal systems that have been deployed commercially at this point are kiosk-type systems such those described in [38]. These systems have specific hardware requirements that a typical home or business user will not be able to fulfill. Large scale consumer multimodal applications will require the use of affordable and small hardware. PDAs are a logical choice, since they already offer a pointing modality (the pen, or stylus) and can easily be equipped with a microphone for speech recognition. Also, they are already in wide use. Processing power is increasing continually, and is at a point where speech recognition is feasible, at least for small vocabularies. Microsoft's MiPad prototype [47] uses speech and pen to fill in text fields on handhelds. However, it cannot truly be considered multimodal, since modalities are not used at the same time: pen is used to select a field, after which speech is used to dictate the contents of the field. Little natural language processing is present, only to parse items such as numbers and dates.

2.10.4 From Unimodal to Multimodal

Multimodal interfaces can greatly enhance interaction between a user and a computer in some types of applications. Therefore it seems like a logical next step to transform existing unimodal systems into multimodal systems.

Speech-only systems have reached a level of maturity that allow them to be used commercially. Telephone-based dialog systems are replacing traditional menu based and touch tone controlled customer support systems [33]. Another popular application for dialog systems is in the travel domain: both Carnegie Mellon, Colorado University, and SRI have developed systems to plan airplane travels [14, 65, 15, 21, 87]; Delft University of Technology has developed a speech application for public transport information in The Netherlands [85]. The travel domain has favorable properties for dialog systems: dialog flow is fairly straightforward, and vocabulary and grammar are limited, known a-priori, and generally nonambiguous. These systems in their current form, however, would not benefit from other modalities, as the entire system is focused on optimizing performance using just the speech modality.

Both Carnegie Mellon and Colorado Unversity have produced systems within the context of DARPA's Communicator project. MIT's Galaxy [79, 67] distributed architecture has been chosen as the basis for work in this task (Galaxy is now maintained by the MITRE corporation). MITRE defines Galaxy as a 'distributed, message-based, hub-and-spoke infrastructure optimized for constructing spoken dialogue systems'. Until recently, no attempt had been made to produce a multimodal system using Galaxy. Various changes are needed to the dialog components, including timestamping incoming modalities to allow synchronization. MIT has begun an effort to transform its Galaxy servers to allow multimodal interaction [80].

2.10.5 Reuse of Programming Code in Multimodal Dialog Systems

In a perfect world, we could take a multi-agent architecture, download dialog components for it, configure them, and have a working multimodal dialog system. In practice, this is not possible. Few universities or project groups are able or willing to release their dialog components. Sometimes a demo system is available, but then the components are often very specific to the application, which means the source code needs to be modified to use the component in another application. This takes away from the advantage of using existing code. Also, source code is often poorly documented, which makes it hard to make the needed adjustments.

OGI's Center for Spoken Language Understanding does make available a toolkit for rapid development of dialog systems [13]. Although this works well for quickly prototyping a dialog system, the system features very simple dialog management and no parsing, just pattern matching to a set of expected responses.

MITRE is developing an Open Source Toolkit for dialog systems based on the Communicator infrastructure but this is still in its infancy, and does not support multimodal interfaces (yet). The inputs and outputs of the various components are fairly well documented, which makes it possible to use these components without having to dig into the source code itself.

CMU's Communicator system is freely available, including source code, but contains large domain-specific parts, specifically dialog management and natural language generation. Additionally, it is a unimodal system, so the components are not made for use in a multimodal system. Documentation is minimal: a very high level overview of the components is available, with no implementation

notes, and in-code documentation is virtually non-existent. The speech recognizer, Sphinx, and Parser, Phoenix, used in the system *are* available as separate components, though, and are domain independent.

Knowledge representation used in a dialog system is a key issue in reusing components. Even though components might communicate on a data-transport level, if the way the knowledge is encoded in the data differs, the two components still won't be able to work together without some sort of conversion component in between. CU and CMU work with the knowledge representation used in Phoenix's output, which is a text representation of semantic frames. MIT's Galaxy systems also use semantic frames, but encoded differently — using Communicator's internal frame representation — and with more complexity. Until a common meaning representation is decided, re-use of components, even using a common middleware such as Communicator, will be hard to achieve. MITRE poses the Communicator hub scripting language as the solution for component incompatibilities, but the differences in representation between Phoenix-based systems and MIT systems are far beyond what can be resolved in a hub script.

2.10.6 The Future of Multimodal Interfaces

Conversational interfaces have matured in the past ten years. This is especially true for speech-only interfaces, which are now being used in phone-based customer support systems. Piggy-backing on these developments, multimodal systems have the opportunity to achieve the same level of maturity. This requires moving away from computer-imposed restrictions on interfaces and allowing truly natural input.

The following advances have brought this goal closer to reality:

- **Robust parsing**: The possible set of commands that Bolt's system recognized were limited by the grammar used. Some synonyms were provided, e.g. "move" for "put", but the grammar had to be strictly adhered to. Current robust or partial parsers can extract information from spoken text even when the user does not strictly adhere to the grammar, or when errors occur in speech recognition [37].

- **Multi-turn and mixed initiative dialogs**: Bolt's system provided a single command-action sequence. If information was missing (e.g. "move that . . ."), the system would wait until the missing information was spoken (". . . there"). Current systems provide mixed initiative in which the computer can request missing information ("where do you want to move it?").

- **Support for more complex gestures**: Bolt's system used the "point-and-speak" paradigm where the only gestures used are pointing gestures. Current systems support a variety of other gestures such as drawing paths, indicating multiple objects or an area on the screen by drawing a elliptical area, handwriting recognition [18], as well as pointing at moving objects (as opposed to static ones) [80].

- **More advanced applications**: Although this doesn't signify a change on the side of human computer interface technology, creating more advanced multimodal applications brings multimodal user interfaces from

the research and "toy" stage into an area where it is actually used to do something useful.

Multimodal system development seems to take place in disjoint 'bubbles', with each university or company building their own set of tools, developing their own meaning representation, etc. This is causing the wheel to be invented over and over. The following issues can be identified as obstacles to easily creating new multimodal systems and advancing multimodal technology, despite the existence of some fine systems.

- **Extensibility**: Most multimodal system accept a fixed set of modalities, for example speech and pen. An extensible architecture where modalities can be added and removed is more desirable. For an efficient and reusable implementation this requires a separation between the modalities themselves and the data they provide, since different modalities can provide the same type of data. Such a separation would improve code reuse.
- **Modularity**: Tight integration between the multimodal fusion and the details of an application make it easier to quickly implement functionality for a particular domain. However, it makes it harder to reuse the fusion code in another system. Strict separation of fusion and domain-specific information makes the system more general-purpose, so that it can be applied to other domains without having to rewrite parts of the fusion code.
- **Documentation**: Although algorithms and ideas are usually published in papers and magazines, code-level documentation is scarce, if the code is made public at all. This makes it cumbersome to use existing code and improve on it.

In short, while the state-of-the-art of multimodal systems is becoming quite mature, it has not yet advanced to the level of, for example, GUI interfaces, for which countless toolkits or frameworks are available, such as MFC [46] or QT [81]. Such toolkits actually make it possible to develop new GUI applications, without having to implement the low-level tasks like drawing widgets, converting mouse events into window events, etc. However, the absence of such a toolkit for multimodal interfaces makes that we have to do exactly that for multimodal interfaces. We need to implement device handling, parsing of device events, fusion, etc., while these things are common to all multimodal systems.

A framework for multimodal interfaces could, analogously to GUI frameworks, provide the tools to quickly implement multimodal systems in all kinds of domains while abstracting away from low-level details in the framework. A well-designed framework should have the aforementioned properties of extensibility, modularity, and documentation.

Chapter 3

Design of the Multimodal Framework

This chapter describes the design of a multimodal framework — an application-neutral collection of components that are to facilitate rapid development of the robot control application as described later on in this thesis. The rationale for developing a framework is discussed, followed by a brief description of object-oriented frameworks. Finally, the framework architecture design is presented.

3.1 Developing a New Multimodal Interface

To support the requirements of reusability and modularity in the multimodal system I was to build, the initial research was towards creating a generic framework for multimodal applications. The goal of creating a framework was to support rapid development of a multimodal application by providing an environment tailored towards the use of multiple modalities and deriving meaning from them, without restricting the type of application that can be developed, in other words, without making assumptions about how to derive meaning from a modality, and how to use it to perform a certain function in an application. This requires a very generic approach, which translates into time savings once this is implemented, since adding features can then take advantage of the reusable code that will already be present.

3.1.1 Rationale

The main purpose for creating a generic multimodal framework was to facilitate building the multimodal interface for the robot control application. Starting with generic code and building the application-specific interface on top of that results in a much better design in terms of modularity and abstraction than if an interface was made just for this application in which application specific demands were integrated with the fusion and dialog management algorithms. Modularity and abstraction should also make the resulting code easier to understand for a new developer, who can look at it one layer at a time, instead of being overwhelmed by all the details of the multimodal interface at once.

Additionally, if the application-independent framework is designed properly, it will provide a basis to build future multimodal interfaces on. Also, if new features are added to the framework itself, they can be used in previous applications that were built on the framework as well, as a direct result of the principle of modularity.

3.1.2 Design Goals

Design goals for the framework were to maintain a clear separation between dialog and fusion functionality on the one hand, and application functionality on the other. Multisensory input can be parsed or filtered by application-neutral code to form information from data, but actual semantics — what this information means within the context of an application — can only be given by the application itself.

For example, a hand gesture such as a circle or a line can be detected by the framework from a series of points emitted by the hand tracker, but giving meaning to this gesture — e.g. that a circle indicates selection of a group of objects, that a line indicates that the user wants to move an object from the starting point of the line to its ending point — is a task of the application. Having said that, the framework should provide as much support for this as possible without becoming application specific. Since multimodal interfaces generally deal with spatial information, support for gesture detection related to object selection and movement is a basic service that a multimodal framework should provide.

3.1.3 Interaction Style

Since the interface that was to be eventually created is for a command-and-control application, the framework will inevitably be biased towards this type of interaction. Characteristics are:

- Commands are generally short and self-contained.
- The language used has a low level of complexity, e.g. anaphora can generally be resolved by looking within a two-sentence window, there are no nested dialogs: commands are issued sequentially.
- Dialog is mostly user initiative, with the system taking initiative only when user input is ambiguous — in which case the system will ask for clarification or confirmation — or when something changes in the application that the user needs to be alerted of.

3.1.4 Development Procedure

A simple test application was made and used during development of the framework to gauge progress and provide some actual measure of the quality of the framework.

Then, using the framework, a multimodal application used to command and control the robot could then be rapidly developed using the framework. This is a real-life application, which would also test and hopefully prove the usefulness of the framework in creating multimodal applications and possibly bring to

light some weaknesses of the design that can then be improved upon. This is described in Chapter 5.

Finally, vocabulary and grammar for the robot domain needed to be formulated. For the initial implementation, a grammar was be made up. For the future, "Wizard of Oz" experiments using actual people that are to be working with the system can be used to obtain a more representative vocabulary and grammar. The design and implementation of the robot control application interface using the framework are discussed in Chapter 5. This chapter will focus on the framework proper.

3.2 Object Oriented Frameworks

Object oriented frameworks play in important role in modern software development [40]. Frameworks support software development in a particular domain[1] by providing a core of functionality that applies to the entire domain. In several points in this core's programming code there are calls to code that is not implemented by the framework itself. The points where this happens are called "hot spots". It is the task of the developer using the framework to implement these hot spots to create a specific working application within the domain of the framework.

Framework can be characterized by the way in which a new application is created. In white box frameworks, the implementor must subclass or implement classes and interfaces in the framework, making calls to the framework's libraries. Such an approach is also called architecture-driven. Black box, or data-driven frameworks, on the other hand, are customized by a configuration script or wizard which creates the necessary classes. Therefore the developer doesn't need to know the framework's internals. Combinations are also possible, so-called gray-box frameworks.

The power of frameworks lies in inversion of control, also called "old code calls new code". The framework's core calls functions that are not implemented by the framework, but instead need to be provided by the developer using the framework. These are also called "callback functions", because they are a known location the framework calls back to when needed. The developer only needs to comply with the interface the framework dictates for its unimplemented parts; he/she doesn't need to know the framework's details, such as when this code is called and how often. This is different from use of "traditional" toolkits or libraries. When using those, a developer must take the effort of "gluing together" calls to the library or toolkit components, for which he/she must know which functions are available, the condition in which to call them, etc. In general, a framework provides a more convenient way to write applications because it abstracts away from details and allows the developer to focus on just the things specific to the application.

An important tradeoff in developing a framework is that between flexibility and genericity on the one hand, and functionality on the other. A very generic framework can produce a wide variety of applications but requires a lot of effort from the developer. An extreme is the universal framework given

[1] Domain is used here in a different sense than in the rest of this thesis. In the context of frameworks, by domain we mean an application type, such as a GUI application, or a database application

in Listing 3.1. This framework can be extended to implement any deterministic solution provider; the developer need only implement the three functions **problemSolved()**, **solveProblem()**, and **solution()**. Of course, this framework is useless because it provides hardly any functionality.

```
while (!problemSolved()) {
    solveProblem();
}
return solution();
```

Listing 3.1: A universal framework

Also, frameworks with many hot spots that need to be implemented place a burden on the developer, even though such frameworks are very versatile. On the other hand, a framework in which most of the implementation is already present limits the possibilities in creating new applications. A common solution that aims to provide the 'best of both worlds' is to have a good amount of hot spots, but provide default implementations for them. If these default implementations are used, the framework functions as a specialized framework, enabling a new application to be created in very little time. However, the default implementations can be replaced by custom ones, giving the framework the flexibility of a very generic framework.

This is the approach I have tried to take in building the multimodal framework. A multimodal application can be created very quickly by using readily available hotspot implementations and implementing the few hotspots that have no default implementations.

3.3 A Multimodal Framework

Development of a multimodal framework was started due to the absence of freely available dialog systems that were both multimodal and could easily be applied to other domains. The resulting framework was to be a valuable tool that could be used to create a plethora of multimodal applications, providing the support needed by developers of multimodal software, while remaining sufficiently generic to be applicable to a wide range of solutions.

The main focus was towards command and control applications, since that is the type of application that would eventually need to be created for this project.

3.3.1 Design Goals

The goal was a framework with all the characteristics of an object oriented software framework, as well as having some domain specific requirements (for the domain of multimodal applications). The following list enumerates all these requirements:

- **Reusable**: all domain-dependent information is located in external configuration files that can be easily modified. Domain dependent code, where needed, is separated from the core code by using well-defined interfaces. Configuration files serve as the link between the framework and domain-dependent code.

- **Multimodal**: the framework is to fuse information from different modalities to obtain a reliable assessment of the user's intention. Since the use of multimodal interfaces implies the presence of spatial information, the framework provides extensive support for referencing of objects using spatial and other attributes.

- **Minimal impact on existing applications**: developers wishing to use the framework to make their applications multimodal should be able to easily do so, keeping the original application and the framework as separate as as possible. Communication between the framework and the application is to be done through a well-defined Application Programmers Interface (API).

- **Multilingual / Language-independent**: the framework should have no language-dependent code. Given the availability of a speech recognizer for a target language, the framework should easily enable developing multimodal applications in that language.

- **Efficient**: direct manipulation using multimodal techniques requires a response time in the order of hundreds of milliseconds. The framework should be speedy, to avoid user irritation caused by slow response times.

As so often, design goals conflict:

- Reusability and efficiency tend to be enemies. Reusability requires modularity and levels of indirection, while for efficiency, tight integration of application components can be advantageous.

- To effectively use multimodal human computer interaction, any existing WIMP application will need to be redesigned to take advantage of this style of interaction. Using speech and other modalities to control menus, windows, and buttons, which would require very little modification to the application, is not using multimodal interaction to its full potential. Making an application multimodal will always have some level of impact on that application.

In developing the framework, a balance will need to be found between conflicting goals so that performance criteria are met while maintaining a flexible and elegant design. We can keep Moore's Law in mind though, which promises a doubling of computer processing power every eighteen months [51]. A suboptimal system in terms of efficiency will perform better over time thanks to Moore's Law; however, an inflexible design will cause problems for a long time. Therefore I will prefer flexibility over efficiency in most cases.

3.3.2 Approach

I took iterative approach, in which an initial design was created, followed by an initial prototype implementation, so that a working system could be created a soon as possible, even when some components have very trivial initial implementations. Later on these components could then be refined.

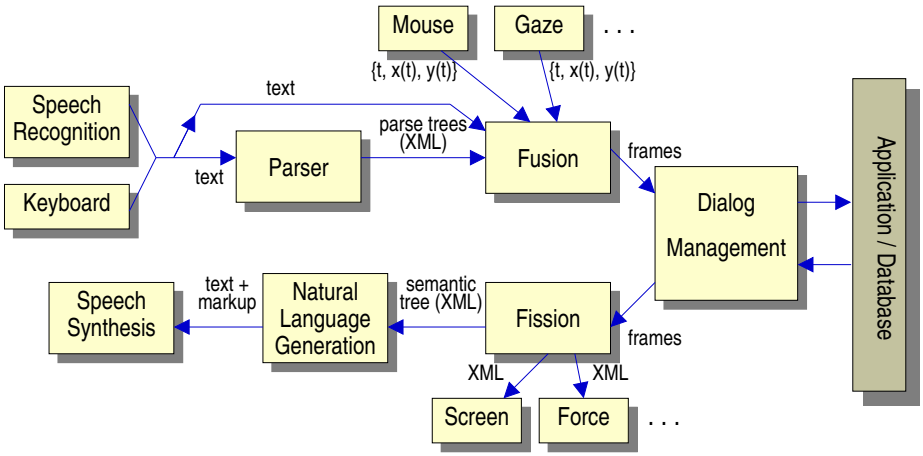

Figure 3.1: Data flow in a multimodal system

3.4 Architecture

The architecture used for the framework is similar to that shown in Figure 2.3. However, since we are dealing with *multimodal* dialog systems, two extra components need to be added: one to fuse multiple modality streams, before doing dialog management, and one to split dialog manager output into multiple output streams for the various output modalities. This is also the approach that was taken in [12]. A graphical representation is given in Figure 3.1.

As can be seen from the figure, the approach is speech-centric. Perhaps "language-centric" would be a better term, as text can come from either a speech recognizer or be typed in with a keyboard. Language is the "main" modality, and other modalities are used to resolve deictic references, pronouns, and other anaphora as well as ellipsis in the text. For the type of interface the framework will be used for — an interface to a command and control application — most dialog actions will in fact be accompanied by speech, so this is not a problem. Additionally, a user can always revert to using the application's GUI with the mouse, if for some reason this is better for a particular command or situation. Some tasks are done more easily with the mouse, and one of the advantages of a multimodal interface is that the user has a choice, especially when, as in our case, the multimodal interface is combined with a GUI interface. The limitations of this approach are discussed in Section 3.8.2.

A few things need to be said about this figure. First, the speech recognizer output is text, and not a word graph. The reason for this is that our current speech recognizer does not provide word graphs, but provides an N-best list. However, no confidence scores accompany the alternatives, and time stamps are only provided for the best alternative. Therefore, only the best alternative is sent to the parser, as text. The time stamps are encoded in the text, as described in Section 3.7.1.

The figure shows that spoken or typed text is not only sent to the parser, but also directly to the fusion component. This has two reasons. One is general in nature: some applications may need to have access to the complete unparsed utterance, for whatever reason. For example, there might be words in the orig-

inal utterance that the parser does not know — possibly because they are not known a priori — but that the application needs. This is the case if by using the system new words are introduced into the system, or if arbitrary names are used for objects or locations in the interface. The other reason is that the parser we use does not output timestamps. Therefore, we need to link the timestamps from the speech recognizer output back to the words in the parse tree, for which we need both the parse tree and the original utterance with timestamps.

Mouse and gaze are shown as two possible modalities. Many other modalities could be filled in for the ellipsis in the figure. The framework does not make any assumptions about these modalities. Currently no support is provided for 'parsing' them, although this would be a useful addition in the future. For example, a module for detecting gesture patterns such as lines and circles in (x,y)-input could be written. This could be used for mouse, touch, and glove inputs.

The components function individually and are configured separately. Some components are off-the-shelf, others were written specifically for the framework. The next section describes which components were chosen and why.

3.5 Components

In designing the framework, an implementation needed to be found for each of the components a multimodal dialog system is comprised of. Additionally, these components need to work together in a single system, so they need to be compatible in some way. Since several dialog systems use the Galaxy Communicator architecture, there are some good Communicator-compliant (or "hub-compliant") components available. Therefore I chose Communicator as the common infrastructure for the framework.

Communicator has the additional advantage that it is distributed. This means the different components can run on different machines if needed. It also means that in a running system, one component can be restarted while the remaining components keep running. Restarting a component can be useful when a configuration file is changed that affects just that component, or when a component displays erratic behavior.

3.5.1 A Common Infrastructure: Communicator

Communicator was described in Section 2.9. In short, it is a TCP/IP based infrastructure in which all components communicate with each other through a central *hub*, which directs traffic between the components, or *servers*.

Speed can be a concern using Communicator, since all traffic goes over TCP/IP, which, even in the local case, does incur some overhead. Also, as all traffic must pass through the hub, possibilities for parallelizing are limited. However, using Communicator, I have not witnessed any significant delays. The number of messages sent between components and the time spent in the hub by each message is too small to have a big influence on the system as a whole.

The framework is spread out over five communicator components, or *servers*. Each of these has a separate connection to the hub. The layout is shown in Figure 3.2.

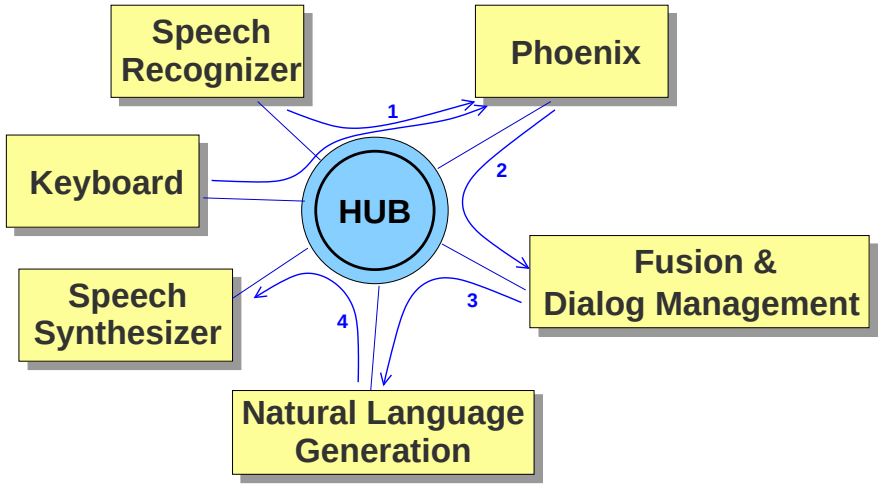

Figure 3.2: The layout of the framework's servers in the Communicator infrastructure

Communicator and system robustness

Some people are concerned with the single point of failure that the hub presents. While this seems like a reasonable concern at first, one should realize that splitting a system into multiple components can only reduce the risk of failure. And if one component *does* fail, at least the others will continue running. Also, most Communicator components do not maintain state, so if a process monitor is present to bring processes back up if they go down, component failure can go almost unnoticed. Components that do keep state (such as the hub) will have more effect on the system when they fail, but even then they will only lose their own data, not that of other components. In a system consisting of a single process, failure of that process causes the entire system to go down, losing all of its data.

3.6 New vs. Off-the-Shelf Components

Implementations for components on the left side of Figure 3.1, which are more low level and application-independent, are much easier to find than the (more application-dependent) ones on the right. In most conversational systems, fusion, dialog management, and natural language generation are implemented in a very application-dependent and hardwired way. This means the system itself works very well, but it demands a lot of effort to adapt the code of these components for use in another system. Because of this, the only off-the-shelf components I was able to use were speech recognizers, parsers, and speech synthesizers. I took it upon myself to develop fusion, dialog management, and natural language generation components, and in such a way that they would be flexible and not just usable in the application I was currently developing, but instead configurable to meet application-specific demands for a range of applications with the same component.

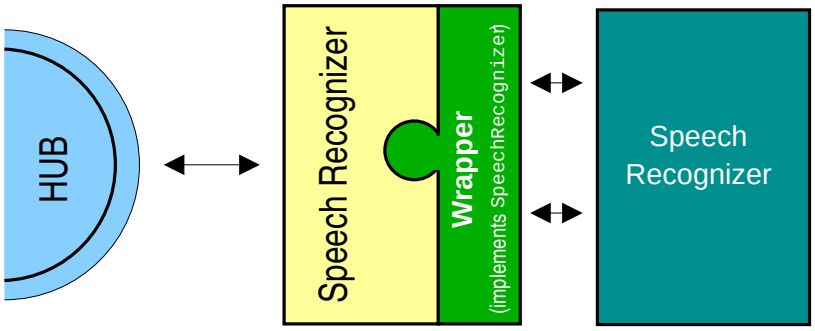

Figure 3.3: Abstracting away from the speech recognizer implementation

3.7 Off-the-Shelf Components

3.7.1 Speech Recognizer

Currently, IBM ViaVoice is used for speech recognition. ViaVoice is a commercial speech recognizer, but targets the consumer market, and is therefore reasonably priced. It supports the JavaSpeech API (JSAPI), and it is very easy to train: the user just reads a story from the screen for about fifteen to thirty minutes; ViaVoice then builds an acoustic model for that user. Disadvantages of using ViaVoice through Java were already named: no confidence scores are provided and time stamps are only given for the best alternative. However, many other speech recognizers do not provide time stamps at all, so until other viable options are built (the upcoming Sphinx-4 looks promising) ViaVoice will be the best choice.

To avoid locking the framework in to a specific speech recognizer, however, I created an interface abstracting away from the actual speech recognizer used. Following the wrapper design paradigm, whenever a new speech recognizer needs to be used in the application, an implementation of the `SpeechRecognizer` interface is implemented that communicates with the actual speech recognizer. This can be done through function calls, TCP/IP, or whatever method of communication the speech recognizer supports. This is illustrated in figure 3.3.

A logical question to ask is why the speech interface used is not an existing one, such as JSAPI, which would be a logical choice since the framework is largely written in Java. The reason is that we only need a subset of the features provided by these interfaces. Using JSAPI would require implementing all of its methods, which requires a lot of effort; effort that would go unused since the framework only requires a few basic features. A `SpeechRecognizer` implementation for JSAPI *is* provided, so that most JSAPI-compliant speech recognizers can be used easily.

Output from the speech recognizer is sent to the hub as text, with timing and confidence information encoded in the text, for example:

```
MOVE{startTime=66041061|endTime=66041481|confidence=0.5}
THIS{startTime=66041638|endTime=66042367|confidence=0.5}
ARMY{startTime=66042419|endTime=66043212|confidence=0.5}
THERE{startTime=66043497|endTime=66044072|confidence=0.5}
```

Since JSAPI doesn't provide confidence scores, the JSAPI implementation currently uses a fixed value of 0.5.

3.7.2 Parser

Colorado University's Phoenix parser is a popular parser that has been proven through its use in several Communicator systems. Phoenix is robust, completely domain-independent, and hub-compliant. In addition, it can be downloaded at no charge and is licensed under an open source license, so that the source can be modified to change the parser.

Because Phoenix is open source I had the opportunity to change it or add features. Initially I wanted to add functionality for including metadata such as timestamps and confidence scores in the parse output, but this proved to be difficult to combine with the way Phoenix was designed: each word in the grammar is given an index, and parse input is converted into a list of index numbers, so the identity of words is lost. I did add code to strip incoming text of the metadata (the text between braces), since otherwise words would not be recognized.

Linking metadata back to the parse output is now done by Java code in the fusion component. The method used there could also be used in the parser, but implementing it in Java in the Fusion component was easier than doing it in plain C, which is what Phoenix was written in.

3.7.3 Speech Synthesizer

The speech synthesizer is implemented analogous to the speech recognizer. That is, a uniform interface is provided that is used in the framework, and any speech synthesizer can be used as long as a wrapper is written for it that conforms to the interface dictated by the framework. An implementation was made for JavaSpeech, and IBM's ViaVoice speech synthesizer was used.

3.8 New Components

Fusion, dialog management, and natural language generation components needed to be written from scratch, as off-the-shelf implementations were not available.

3.8.1 Reusability in Components

Making components reusable does not only mean designing them to be technically configurable and usable in different applications, but also limiting their complexity — e.g. the number of classes that need to be configured or extended for a new application—, designing comprehensive interfaces, and providing sufficient documentation so that users will find it easy and intuitive to work with the components.

XML was used extensively in the three new components described in the following sections — fusion manager, dialog manager, and natural language generation. XML has some positive aspects that we take advantage of here, namely structure and self-documentation:

- XML's **structure** — i.e. as a tree — makes it very suitable for representing natural language parses. XPath provides a way for testing for the presence of certain words or concepts in the parse tree, or retrieving them. XSLT can be used to flatten structure into text by a natural language generation module.

- By **self-documentation** I mean that because XML tags are human-readable, an XML file usually needs little extra documentation to be understood. Configuration files for the fusion and dialog management components are in XML format, which makes it easy and intuitive for a developer to modify them, without the learning curve imposed by some other configuration file formats, and without the need for extensive documentation.

3.8.2 Fusion

The fusion component is the place where modality data comes together to determine a single unambiguous meaning, or user intention, which is then passed on to the dialog manager. A modality and application independent fusion manager was necessary for the multimodal framework. This required a novel approach to fusion, which is described in this section.

The main goal of the fusion component is not the fusion of sensory information itself, since that is only a means. The end is *context resolution*: taking a user's parsed spoken utterance and using context — in whatever form available — to resolve inclarities and ambiguities in it and convert it into a canonical form. In unimodal conversational systems, context resolution uses dialog history, common knowledge, and possibly some form of reasoning. In multimodal systems, this is enhanced by having access to sensory inputs as an extra source of information. The fusion manager described in this section takes the task of traditional discourse managers and generalizes it by allowing all sorts of information to be used as context, regardless of where it came from. Several methods for using this information to resolve ambiguity and create a canonical representation are available, and the framework can easily be extended with more.

The Fusion Process

The input to the fusion process is a semantic parse tree with time stamps as generated by the natural language parser component of the speech interface. This parse tree needs to be transformed into frames that the dialog manager can use to make calls to the application. To create these frames, the natural language concepts in the parse tree need to be mapped to application concepts. In addition, ambiguity needs to be resolved. Ambiguity exists, for example, when the user uses pronouns or other anaphora, or deictic references, for example "remove *that*', or "do reconnaissance *here*". Another case of ambiguity is ellipsis, a linguistic construct in which words that are implied by context are omitted, such as "rotate this clockwise ... *and this too*" — the last phrase can be expanded to "and rotate this clockwise, too".

Needless to say, this is far from trivial. The fusion algorithm explained below addresses some forms of ambiguity, but does not (yet) resolve all forms.

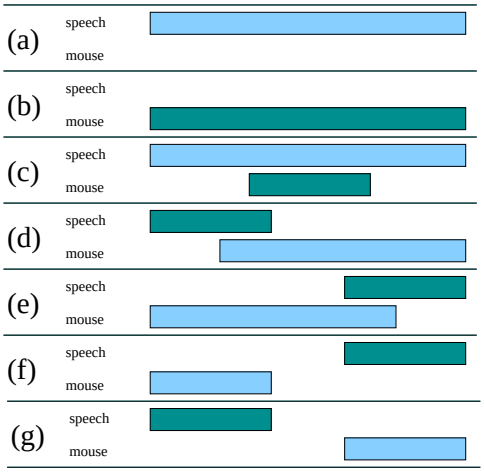

Figure 3.4: Different alignment cases for speech and mouse

In multimodal interaction, input from different modalities can be simultaneous, overlap partially, or be completely disjunct. Ideally, a multimodal system should be able to handle all of these cases. Fusion in the framework is currently speech-centric. This means that all multimodal actions must contain speech. While this seems restrictive, it is intuitive for most users to at least use speech when interacting with a computer through a multimodal interface. From a design viewpoint, using speech to decide when an interaction starts and stops is most obvious, as speech signifies a conscious act on the side of the user. Determining from other modalities, such as gaze, whether the user is consciously doing something is much more difficult. Additionally, because modalities can overlap, non-speech-centric multimodal interaction requires the use of time thresholds to determine how long to wait after input from a modality has stopped before deciding the user has completed an action. This is the method used in [80], but it will necessarily impose delays on the system, which can be annoying for the user. On the other hand, these delays are present as well when using speech: speech recognizers will wait for a certain duration of silence before assuming the user is done speaking.

Figure 3.4 shows the different ways in which speech and mouse can be aligned (mouse can be substituted by other modalities). The frameworks handles these cases as follows:

(a), (c), (e), (f) **Speech-only, speech encompasses/follows mouse** — Speech-only interaction is just a specialization of interaction using speech and other modalities simultaneously. The framework works very well for this type of alignment. Mouse movements are stored in a buffer until the user finishes speaking, at which point they are used for fusion. So mouse movements can start before speech, but mouse movements done after speech ends are not used for that speech act. (There is actually a delay between when the user stops speaking and when the buffered mouse data is retrieved, so mouse movements made shortly after the user stops speaking *will* be registered).

(b) **Mouse only** — For the framework to detect a multimodal action, it must currently be accompanied by speech. Multimodal processing is done from speech, and speech determines which type of frame is instantiated — non-speech modalities can only influence what the contents of that frame will be. To change this, a voting mechanism similar to that used for filling the slots could be used to choose a frame, i.e. based on each modality a choice could be made, and the majority's vote (with speech having a greater weight, for instance) would have the final decision. Also, time thresholds for each modality would dictate after how much time of inactivity the user can be considered done with it, and therefore ready for the system to process the multimodal command. For speech this is already done by the speech recognizer, which waits for a certain duration of silence before assuming the user is done speaking. Inter-modality time thresholds would also need to be set, to allow a user to use two modalities in succession. This time threshold determines after what duration of time the user can be expected not to be using another modality to supplement the command just given with one modality or combination of modalities. More advanced methods that time thresholds for determining this intent of the user would be desirable, as they might avoid delays and help create better system response times.
Since the framework augments GUI interfaces, a user can also perform traditional direct manipulation using the mouse or a device emulating a mouse (such as a touch screen). In this case the framework is not involved.

(d), (g) **Mouse follows speech** — As described for the first case, mouse data is stored until speech is received by the fusion component. Therefore, mouse movements done after speech are not used for that speech act (they accumulate in the buffer for the next speech act). Therefore the following form of interaction does not work with the framework as it is currently implemented:
"move along this line. ⟨*draws line*⟩"
If the modification in the framework described in the previous point is done (to allow mouse clicks and drags to drive fusion), this type of action *would* be possible (given that the mouse action follows the speech act before the set time threshold). An alternative using the current framework is to have a the speech act ("move along this line" in this example) set a toolbar button or change state in the application which would cause the following mouse movement to generate the desired result, outside of the framework. Again, this would only work with a mouse or a device emulating a mouse (as traditional GUI methods are used).

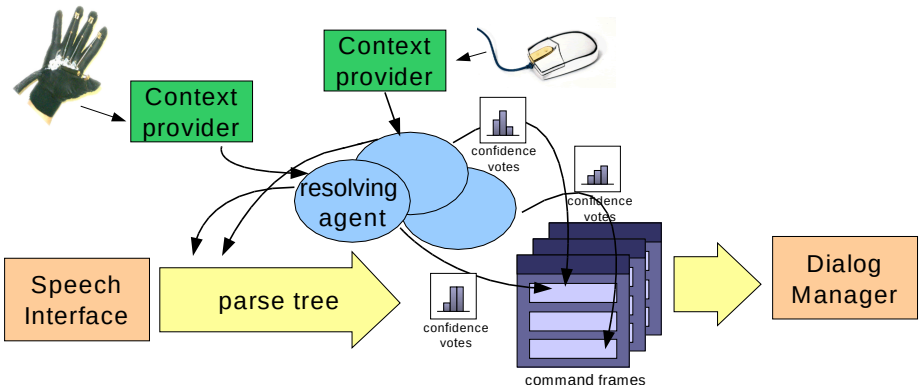

Figure 3.5: The design of the fusion process

The Fusion Design

Figure 3.5 shows the conceptual design of the fusion process as it is now implemented. This will be explained in the following section. The fusion manager, which directs the fusion process, is not shown in this figure, in order to make the flow of data more apparent. However, it is actually the fusion manager that takes fragments from the parse tree and passes them to the resolving agents. It also collects the confidence votes from the agents and combines them to fill the command frames which it sends to the dialog manager. So, although not displayed, the fusion manager is involved in every step of the fusion process.

The fusion design bears some resemblance to that used in OGI's QuickSet system [18], which employs a Members-Teams-Committee technique, using parallel agents to estimate *a posteriori* probabilities for various possible recognition results, and weighting them to come to a decision. However, our approach is more reusable as it separates the data – or feature – acquisition from the recognition. Also, it supports a variety of simultaneous modalities whereas QuickSet seems to be built solely for pen and speech-based interaction.

The Fusion Manager

The fusion manager directs the fusion process. It has three responsibilities which it either carries out itself or delegates to subcomponents:

1. **Collecting data from modalities connected to the system** — The fusion component is the central locus in which modality data come together. The data are stored with timestamps, so that they can later be synchronized. Modality data can be of different types, such as spatial data (coordinates), semantic data (frames), or symbolic data. The fusion manager doesn't need to know the specific type of data, which makes the system very extensible, i.e. other data types can be added without modifying the fusion manager. The data is used by resolving agents, which use it along with the speech input to find out what the user's intent was. This process is described in detail in the remainder of this section.

2. **Combining this data with speech to come to an unambiguous meaning** — As mentioned, resolving agents are used to process modality

data. This results in assessments of meaning that are sent back to the fusion manager. The manager then makes a final decision on what the user most likely intended, based on the various possibilities proposed by the resolving agents, and the *a posteriori* probabilities they provided.

3. **Sending the meaning to the dialog manager** — Once an unambiguous meaning is derived, a frame can be sent to the dialog manager, which will update the dialog context, perform some task in the application by making function calls, and provide user feedback. A frame is also sent to the dialog manager if an unambiguous meaning can *not* be determined, in which case it can prompt the user for more specific information.

Collecting data from modalities A continuously active process is that of collecting data from the various modalities connected to the system. Classes that collect this data and provide it to the framework implement the `ContextProvider` interface. They are spawned by the fusion manager and generally run in their own threads, independent of the fusion manager. Through the **`getData()`** method, the fusion manager can obtain data from the `ContextProvider`.

Not all `ContextProvider`s get data directly from a modality. A `ContextProvider` can get input from another `ContextProvider` and use its data. This can be used to implement modality parsers or filters. For example, the `Clustering` class takes coordinate data from an eye tracker or pointing-modality and finds fixation clusters in the coordinate stream, which it then outputs.

Modality data is usually buffered so that it can be used when a speech utterance is completed. The `BufferedContextProvider` continuously polls another context provider and stores its data. The frequency with which the underlying context provider is polled and the size of the buffer are determined by the developer in the fusion configuration file. Usually a few seconds of buffered data are sufficient. For example, if a modality is polled every 60 milliseconds, a buffer of size 100 would be enough to store 6 seconds of data. Since a speech utterance only lasts a few seconds, this is sufficient. Smaller sizes are better since they use less memory, and result in less waste of time when the buffered data is searched by a resolving agent.

The `ContextProvider` class and related classes will be described in detail in Chapter 4.

Combining Data with Speech When a parse tree is received by the fusion manager, it needs to create a semantic frame from it. Through the fusion configuration file, a developer using the framework can specify exactly how slots in this frame are filled using the data in the parse tree and the modalities connected to the system. Because multimodal systems deal with uncertainty, there is usually no clear answer to what value should be filled in a slot. Instead, there are several possibilities, generated by what the user has said (which is reflected in the parse tree), multimodal inputs, and the context, or history, of the dialog. Some possibilities may be more likely than others.

The fusion manager spawns resolving agents which provide for *one* fragment of the parse tree and *one* modal input, a list of possible values for a slot, along with confidence scores. Resolving agents implement the `Resolver` interface,

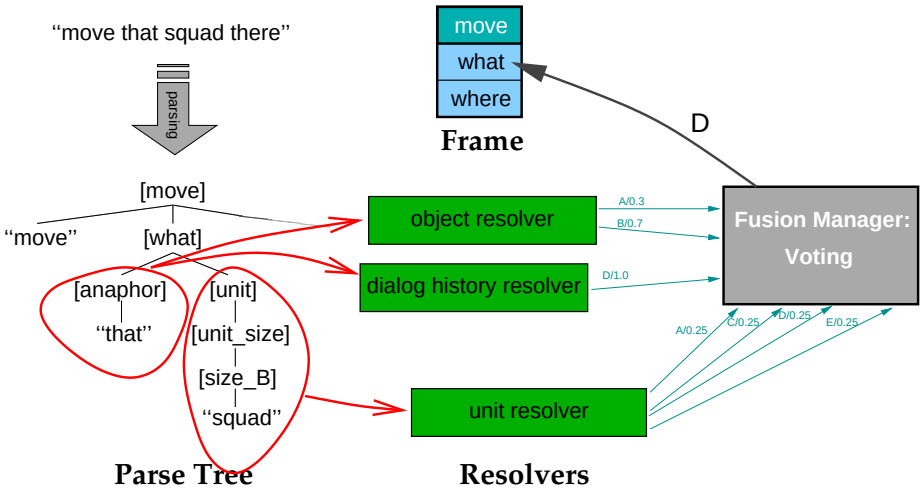

Figure 3.6: An example of how a slot is filled by resolving agents

but, as with the `ContextProvider`s in the previous section, the fusion manager needs to have no knowledge of *how* they do their work. All the fusion manager does is send a fragment of the parse tree, as dictated by the fusion configuration file, to a resolving agent, and retrieves a list of possible slot values and corresponding confidence scores.

Because the resolving agents use just a part of the parse tree, and access just one modality, their design can be kept simple. Since they themselves don't fill the slots of the frame that the fusion manager creates, but instead send possible values to the fusion manager, they don't need to be concerned with combining their possible values with those of other agents. Instead, they can operate locally on their data. This makes the agents parallelizable, which means they can run at the same time and take advantage of multiprocessor hardware, if it is available.

A visual example of all this is given in Figure 3.6.

Thus, for each slot the fusion manager obtains a list of possible values for that slot. Each value is accompanied by a confidence score that the resolving agent provided. The fusion manager must now use these lists to decide what to put into the slot. There are many ways to do it, and the currently used method is very simple and generic. The confidence scores for each value are multiplied with a weight specific to the resolver the value came from. This weight is provided by the developer in the configuration file and provides a way to 'prefer' certain agents over others. For example, we might find mouse input more reliable than eye input, so we allocate a higher weight to the `Resolver` using mouse input than to the one using eye input. The resulting scores are summed per possible value (as in a histogram). Finally, the highest scoring value is used to fill the slot.

Sending Meaning to the Dialog Manager To avoid lock-in to the Java language, the fusion and manager communicate using a language-neutral representation, encoded in XML. The `DomainCodec` interface provides methods for encoding and decoding data between the language-neutral representation

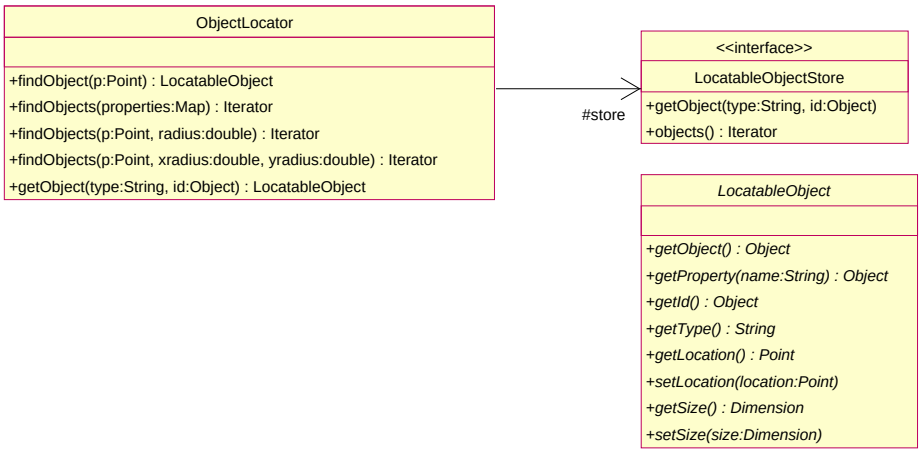

Figure 3.7: The `ObjectLocator` and related classes

and Java. The default implementation provided by the framework — `DomainCodecImpl` — can encode and decode primitive types, `String`s, `Point`s, and on-screen objects. Note that objects are not encoded by (Java-) object type, but by a higher level indication of type. According to this classification, the `DomainCodecImpl` distinguishes `object`s, `location`s, `literal`s, and `value`s.

A `location` (`Point`), for example, is encoded as:

```
<location x="473" y="216" />
```

Services Provided by the Fusion Manager

The Fusion Manager provides services that resolving agents can access when determining values for a slot in the output frame.

Querying objects Resolvers may query the application for on-screen objects through an `ObjectLocator` class that the fusion manager provides. This provides an application-independent way for finding objects using criteria such as location, name, or other properties. The `GeometricAnaphorResolver` that comes with the framework uses this to find objects in the vicinity of where the user was pointing or looking. `ObjectResolver` queries an `ObjectLocator` to find objects matching what the user was looking for, e.g. "move the *enemy squad* there" would cause `ObjectResolver` to query the `ObjectLocator` for objects whose **affiliation** property is "hostile" and whose **size** property is "squad". The resolver uses the parse tree and configuration information from the fusion configuration file to determine what to query on.

As shown in Figure 3.7, `ObjectLocator` uses an instance of `LocatableObjectStore` that provides the actual access to the objects in an application. The objects returned by `LocatableObjectStore` are instances of `LocatableObject`, which is a wrapper class that wraps application objects in an application-independent interface that allows the framework to obtain its location, size, and attributes in a uniform way.

`Application` and `Session` objects The framework uses two objects, `Application` and `Session`, to provide access to global settings and resources. An application can register a *resource* with the framework, which will make it available through the `Application` object. In this way, `ContextProvider`s and `Resolver`s can have access to parts of the application. For example, the `Mouse ContextProvider` needs a `Component` resource that it captures mouse events from. Also, `Session` provides access to the framework's logging facilities, allowing `ContextProvider`s and `Resolver`s to generate informational and debugging output that can be used for troubleshooting, or just to gain insight in what a `ContextProvider` or `Resolver` is doing.

The `Application` and `Session` objects have other features, but these are only used by the dialog manager, and will be described in the next section.

3.8.3 Dialog Manager

As described in the preceding section, the dialog manager receives semantic frames from the fusion component. These frames are fully resolved and unambiguous. However, they are not necessarily complete. That is, some slots may be empty.

The dialog manager is configured through an XML file that declares the possible frames, and *action script* code to be executed for each. Currently the script code has both the responsibility of determining whether the frame is complete and carrying out the actions for that frame. This is an example action script:

```
1 if (frame.glyph) {
2     application.api.deleteGlyph(frame.glyph.glyph);
3 } else {
4     session.speak('<$missing><frame>delete</frame>' +
5             '<slot>glyph</slot></missing>');
6 }
```

The code is JavaScript. The first line checks if the frame is complete by testing if the `glyph` slot exists. If so, a call to the application is made. If not, feedback is given through the natural language generation component. This will generate speech asking the user to specify what he or she wants to delete.

Dialog Manager Tasks

The dialog manager becomes active when a frame is received from the fusion component. It has three responsibilities:

1. Maintaining dialog context
2. Maintaining dialog history
3. Making application calls

Maintaining dialog context The dialog manager keeps a context frame that represents the current dialog context. Incoming frames are merged with this frame until a complete frame is filled. Whether a frame is complete is determined

but the **isComplete()**. Currently the default dialog manager implementation just returns **true** and to make real use of the context frame, a developer must subclass the DialogManager class and override the **isComplete()** method. However, as described before, currently the action scripts are used to check if a frame is complete. In a later version of the framework, a more generic method for checking for frame completeness might be developed, for example based on rules, which could be implemented in the base DialogManager, alleviating the need to subclass it for this purpose.

Maintaining dialog history Dialog history is implemented as a separate class, DialogHistory, that is owned and kept current by the dialog manager. When a frame is completed and executed, its slots are added to the dialog history. There is currently room for 7 slots. Since the dialog history is used to find antecedents that a user refers to in the dialog, there is no need to store more entities than the user will refer to. As the human mind can only maintain approximately 7 entities at a time, we do not need to store more than that number in the dialog history. That said, the number is easily changed by modifying a constant in the DialogHistory class.

DialogHistory implements the ContextProvider interface and can therefore be used by resolving agents to resolve anaphors or fill in blanks in frames based on the dialog history.

Making application calls As described, when a frame is completed, the action script for that frame is retrieved and executed. Although this script can do anything the developer wants, in general it will use the application, session, and frame objects made available to it to read the values of the frame's slots and make calls to the application or send feedback to the user through the natural language generation component.

3.8.4 Dialog Manager Limitations

Dialog context management is a very complex subject, and the dialog manager presented here is a very simple implementation. This is in part because many traditional dialog manager tasks are handled by the fusion manager: choosing a frame type based on the possible parse trees, "inheriting" slot data from the dialog history, and resolving various types of partial data such as anaphora and relative dates.

Another reason for a simple dialog manager is that I've made some assumptions as to the type of dialog that can be expected: command and control systems will generally have straightforward and accurate dialog, so advanced dialog management techniques should not be necessary.

The current dialog has the following limitations:

- **Nested dialogs** are not recognized. For example, a dialog of the following style would not be handled correctly with the proposed design:

User:	Move the infantry army here ⟨*points*⟩.
Computer:	There are two infantry armies, which one do you want to move?
U:	‖ Show me both armies.
C:	‖ *Scrolls to show both armies and hilights them.*
U:	OK. Uhm, move that one ⟨*points*⟩ then.
C:	Moves the army to the requested location

The user starts a dialog, asking the computer to move an army. Because there is uncertainty as to which army to move, the user starts a subdialog where he asks the system to show both armies, which the system then does. At that point the original dialog is resumed and the army is moved.

To handle this simple case would be trivial. The system could detect a topic change (from "move" to "show"), push the current context frame on a stack and start a new dialog. Then when the old topic is resumed, it could pop the frame and continue working on it. This would even work with multiple nested dialogs.

However, it is very difficult to determine whether a topic change indicates the start of a subdialog, or whether the user justs wants to change the topic and has no intent of resuming the original conversation. This could be solved by implementing a "time-out", for instance if the user doesn't return to the original topic within n sentences, the context frame is dropped from the stack. However, it is difficult to determine what this n should be. Further research could show if this approach would work, and what would be a good value for n. For now, I stayed with a simple dialog manager, to at least have a working prototype. This means the user's speech from the previous dialog would result in the following user-computer dialog with the current dialog manager:

User:	Move the infantry army here ⟨*points*⟩.
Computer:	There are two infantry armies, which one do you want to move?
U:	Show me both armies.
C:	*Scrolls to show both armies and hilights them.*
U:	OK. Move ⟨*points*⟩ that one then.
C:	Where do you want to move it?

Since the context frame from the first part of the dialog is dropped when the user changes topics, a new "move " frame is created in the last part, but the slot that should contain the destination is now empty, causing the computer to ask for it.

- The only **state information** that is maintained is the context frame. This means that the dialog manager can only make decisions based on the current frame and the context frame. If more complex behavior is desired, a developer should extend the dialog manager, or implement this behavior in the API class that connects the dialog manager with the application.

3.8.5 Abstractions and Assumptions

In order to be applicable for many different applications, the framework makes several abstractions. Each abstraction results in a "hot-spot", or interface, as

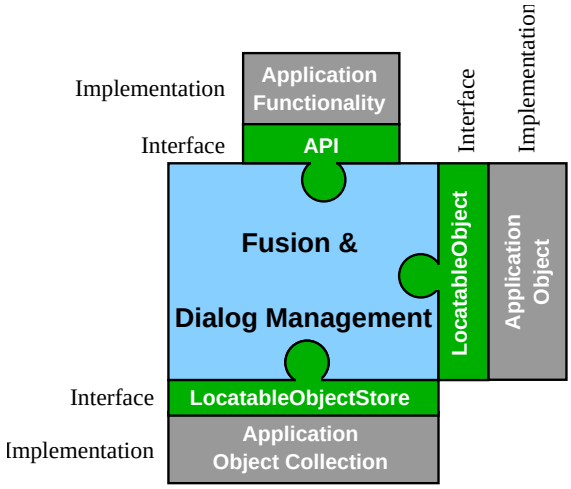

Figure 3.8: Hot spots in the framework for application implementation details

described in Section 3.2. These hot spots need concrete implementations that are often specific to the application, the modalities used, or the way in which modalities are used to resolve ambiguities in speech. Along with abstractions, assumptions are made. By defining the hot spots, the framework's core is also defined, and this embodies assumptions we've made about the type of interface that will be built with the framework.

Application Abstractions and Assumptions

I've assumed two things about applications in implementing fusion and dialog management. This is shown schematically in Figure 3.8. The assumptions, and their corresponding abstractions, are:

- Since the application is to be made multimodal, it has objects or spots on the screen that can be referenced in a multimodal fashion, i.e. by pointing to them or by describing them. This implies that the objects on screen have a location, represented by an (x,y) coordinate pair, and have attributes that can be used to describe them, for instance size, color, or type of object. The `LocatableObjectStore` interface and the `LocatableObject` abstract class described previously are the results of this abstraction.

- An application has a certain functionality that it wishes to expose to the framework so that it can be accessed by multimodal commands. I'm assuming all this functionality is provided in a single class, shown as "API" in Figure 3.8. If an application has multiple classes it wishes to expose, the API class can be built as a "mediator" class that just provides **`get`** methods for these other classes.

Fusion Abstractions and Assumptions

Abstractions in the fusion manager were made to avoid locking in the fusion component of the framework to specific modalities or resolution methods. The

following abstractions and assumptions were made:

- Speech is the "driving force" for fusion. That is, fusion is started when a speech utterance is received.
- Late fusion is used. Fusion is done at the slot level on frames that are instantiated from a parse tree, that is, a slot-filling approach is used.
- Modalities provide data, and it is possible for different modalities to provide the same type of data. These two abstractions led to the `ContextProvider` interface.
- An operation using as input modality data and a fragment from the parse tree can provide possible values to use in a frame slot as well as a likelihood of each possibility. This is independent of both the values and probabilities generated from other combinations of modality data and parse tree fragment as from the values in other slots. This is represented by resolving agents and the corresponding `Resolver` interface, as well as the voting mechanism used in the fusion manager to combine the resolving agents value-probability lists.

Dialog Management Abstractions and Assumptions

As described in Section 3.8.4, a simple dialog manager design was used, which implies a significant number of assumptions or abstractions were made. Two of these were already mentioned in Section 3.8.4.

- There are no subdialogs.
- The dialog manager's response to a new frame is only dependent on that frame and the current context frame.
- The action taken for a frame can be expressed as a piece of script code. This script can call methods and access variables of one object that belongs to or interfaces with the application.
- Whether a frame is complete, that is whether a sufficient number of slots has data so that the frame has meaning, is determined by the action script that the dialog manager runs for that frame.

3.8.6 Fission

There was no time to implement fission. Currently, dialog manager output is sent directly to the natural language generation module, so that, besides the screen output provided by the application itself, only speech output is given to the user. A fission manager would be responsible for distributing output over multiple modalities, so that a user could be alerted through different sensory channels.

In symmetry with the way fusion sends frames to the dialog manager, the dialog manager would send frames to the fission component. A fission manager can then call agents for each output modality which take information from the parse tree and transform it into a form that can be sent to the respective

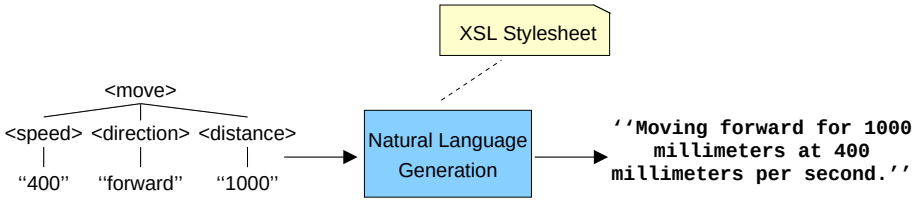

Figure 3.9: Natural Language Generation

modalities. The fusion manager would need to control where and how redundancy it used. It would also need to orhestrate the modalities so that outputs are properly synchronized, for example, a unit on the screen is hilighted at the same time the speech synthesizer says "this unit".

Designing and implementing this properly would probably take a year by itself, which is why I have mentioned fission but have not made any specific design or implementation for it.

3.8.7 Natural Language Generation

I designed and implemented a very simple, but nevertheless powerful natural language generation component. The component takes advantage of XML and code that is readily available, notable from the Apache Xerces [4] and Xalan [3] projects, to perform XML parsing and transformations.

Output from the dialog manager comes in a language-neutral form, in an XML representation. The canonical way to transform XML into something else, in this case into text, is through an XSL Transformation (XSLT) [86]. In the XML stylesheet (XSL), *templates* are declared that dictate how a particular type of XML node is converted. The XSLT then traverses the XML tree and applies the appropriate template for each node.

The natural language generator uses an XSL to convert this into actual text to be spoken by the speech synthesizer, as depicted in Figure 3.9.

Chapter 4

Implementation of the Multimodal Framework

This chapter describes implementation of the multimodal framework.

4.1 Choice of Language

As shown in the previous chapter, several new components needed to be written to have working framework, notably the fusion and dialog manager components. I chose to write these new components in Java. This language has some favorable properties that are particularly useful when writing a framework. Through its "Reflection" API, Java allows objects to be instantiated, methods to be called, and fields to be accessed of classes that were not known at compile time. This makes it easy to extend the framework with new modalities and resolution agents, without having to modify or recompile the core framework code. It is true that there are other programming languages that provide a similar type of functionality, but these are most often script languages that need to be interpreted. Interpreted languages are usually slower than their compiled counterparts. Also, the distinction between fixed code (the framework core) and configurable code (hotspot implementations) can become blurred when all code is human readable.

In addition, Java features strong typing, which avoids programming errors by pointing them out at compile time. Errors due to using a wrong type of pointer than the code expects can cost significant amounts of time to find and correct in a language like C, where this is not checked at compile time.

An extensive and well-documented class library are a great time savings by providing virtually bug-free data structures, utility methods, and much more.

Finally, Java applications can run on different platforms without recompiling, which means the framework can be deployed on any Java 1.4 capable platform.

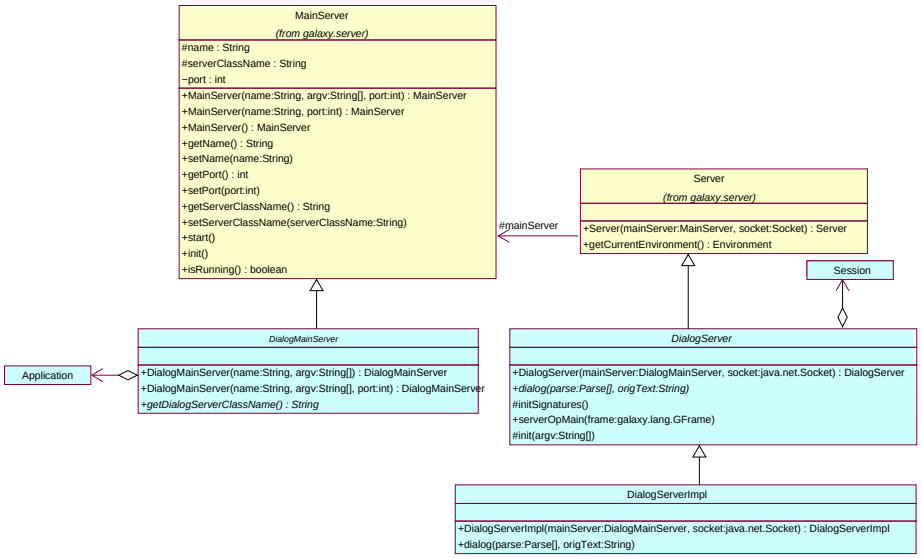

Figure 4.1: A class diagram of the `galaxy.server-MainServer` and `galaxy.server.Server` classes, and their subclasses in the framework

4.2 Programming with Communicator

Chapter 3 described that the system is divided into components that communicate using the Communicator middleware. Although Communicator is written in C, MITRE makes available a Java programming interface as well. To enable Java code to work as a server in the Communicator infrastructure, two classes must be used. An instance of `galaxy.server.MainServer` listens for (TCP) connections from the hub. When the hub connects, the `MainServer` instantiates a subclass of `galaxy.server.Server`. Which exact subclass is instantiated is determined by a previous call to **`setServerClassName()`**. The `Server` instance then receives frames from the hub and can also send frames back.

Incoming frames are handled by a *dispatch* method in the `Server` subclass. A dispatch method will generally read the frame's slots, perform some action, and possibly return a frame.

4.2.1 Configuring the Hub

The Communicator hub is configured through a *hub script*. This is a file that contains the name, address, and port number of each server in the system that the hub is to contact. It can also include routing instructions, if the default routing is not adequate.

To simplify configuring the hub, I've placed server definitions inside an XML file. When the framework is compiled, this file is transformed into a hub script using an XSL Transformation. The XML file also contains information on which program to run to bring up the server. This is for the process manager. A process manager configuration file is created from the same XML file, but using a different stylesheet. Because most programs have different names under Win-

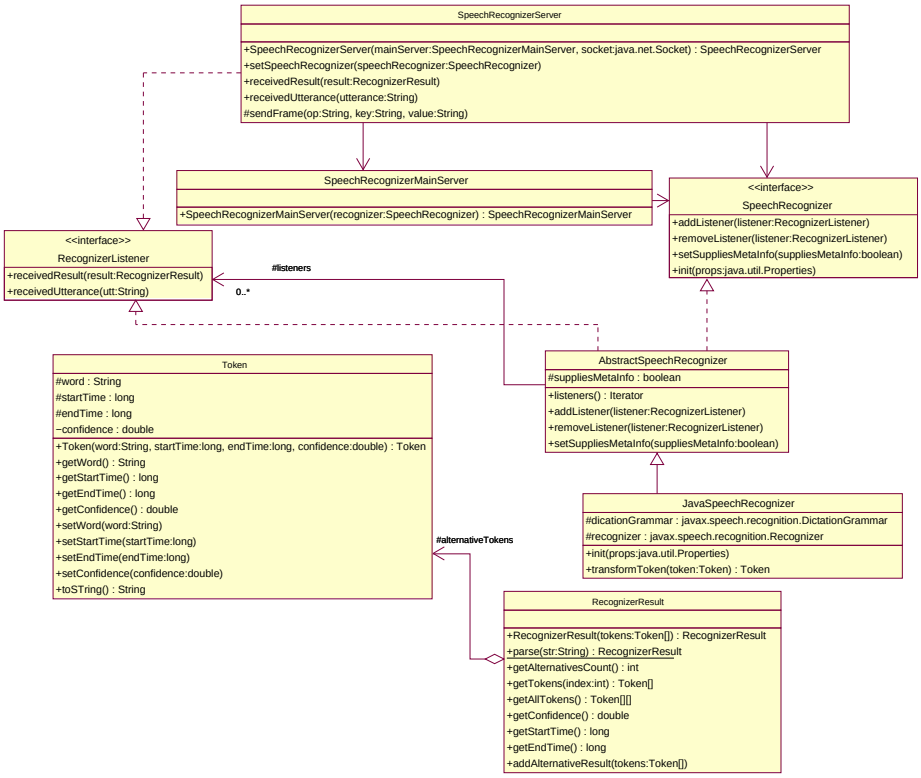

Figure 4.2: A UML class diagram of the speech recognizer classes

dows and Unix (Windows executables end in `.exe`, whereas Unix ones don't), most servers have two definitions, one for Windows and one for Unix.

A fragment of the XML file is shown in Listing 4.1. The resulting hub script is shown in Listing 4.3. The resulting process monitor file is shown in Listing 4.2.

4.3 Speech Recognizer

A UML class diagram of the speech recognizer server is shown in Figure 4.2. A speech recognizer server sends a `send_to_parse` frame to the hub when a **`receivedUtterance`** event is received. The frame contains the utterance text and meta-information as follows:

send_to_parse	
`:parse_input`	The text that was spoken and is to be parsed

```
<server os="nt">
  <name>HUB</name>
  <dir><basedir />\bin\hub</dir>
  <path><basedir />\bin\hub\hub.exe -verbosity 3 -pgm_file ↩
project.pgm</path>
</server>

<server os="posix">
  <name>HUB</name>
  <dir><basedir /></dir>
  <path><basedir />/bin/hub/hub -verbosity 3 -pgm_file ↩
bin/hub/project.pgm</path>
</server>

<server os="nt">
  <name>Phoenix Parser</name>
  <dir><basedir />\bin\phoenix\phoenix.config</dir>
  <path><basedir />\bin\phoenix\phoenix_server.exe -config ↩
default.config</path>
  <galaxy name="phoenix" port="12348">
    <operation>reinitialize</operation>
    <operation>send_to_parse</operation>
  </galaxy>
</server>

<server os="posix">
  <name>Phoenix Parser</name>
  <dir><basedir />/bin/phoenix/phoenix.config</dir>
  <path><basedir />/bin/phoenix/run_phoenix</path>
  <galaxy name="phoenix" port="12348">
    <operation>reinitialize</operation>
    <operation>send_to_parse</operation>
  </galaxy>
</server>

<server os="nt">
  <name>Natural Language Generator</name>
  <dir><basedir /></dir>
  <path><jredir />\bin\java -cp class;res;lib/galaxy.jar; ↩
lib/log4j.jar;lib/xml-apis.jar;lib/xercesImpl.jar;lib/xalan.jar ↩
edu.rutgers.caip.communicator.nlg.NaturalLanguageGenerationMainServer↩
</path>
  <galaxy name="nlg" port="65427">
    <operation>speak</operation>
  </galaxy>
</server>

<server os="posix">
  <name>Natural Language Generator</name>
  <dir><basedir /></dir>
  <path><jredir />/bin/java -Djava.util.prefs.systemRoot=/tmp -cp ↩
class:res:lib:lib/log4j.jar:lib/galaxy.jar:lib/xml-apis.jar:↩
lib/xercesImpl.jar:lib/xalan.jar ↩
edu.rutgers.caip.communicator.nlg.NaturalLanguageGenerationMainServer ↩
</path>
  <galaxy name="nlg" port="65427">
    <operation>speak</operation>
  </galaxy>
</server>
```

Listing 4.1: A fragment of the server definitions XML file

```
EXPAND: $BASEDIR /home/fflippo/flatscapex.robot
EXPAND: $JAVADIR /opt/jdk
EXPAND: $TELNET /usr/bin/telnet
EXPAND: $ROBOTIP 192.168.100.32
EXPAND: $COMMAND /bin/sh

PROCESS: $BASEDIR/bin/hub/hub -verbosity 3 -pgm_file ↩
bin/hub/project.pgm
PROCESS_TITLE: HUB
PROCESS_MONITOR_ARGS: --open --start --input_line

PROCESS: $BASEDIR/bin/phoenix/run_phoenix
PROCESS_TITLE: Phoenix Parser
PROCESS_MONITOR_ARGS: --open --start --input_line

PROCESS: $JAVADIR/bin/java -Djava.util.prefs.systemRoot=/tmp -cp ↩
class:res:lib:lib/log4j.jar:lib/galaxy.jar:lib/xml-apis.jar: ↩
lib/xercesImpl.jar:lib/xalan.jar ↩
edu.rutgers.caip.communicator.nlg.NaturalLanguageGenerationMainServer
PROCESS_TITLE: Natural Language Generator
PROCESS_MONITOR_ARGS: --open --start --input_line
```

Listing 4.2: The configuration file that is created for the process monitor on Unix when the code in Listing 4.1 is transformed

```
PGM_SYNTAX: extended

SERVER: phoenix@localhost:12348
OPERATIONS: reinitialize send_to_parse

SERVER: nlg@localhost:65427
OPERATIONS: speak

```

Listing 4.3: The hub script that is generated when 4.1 is transformed

If the recognizer hears "go forward two yards", the resulting frame might look like this:

```
{c send_to_parse
   :parse_input "go{startTime=66041061|endTime=66041481|confidence=0.5}
forward{startTime=66041492|endTime=66042104|confidence=0.5} two{startTi
me=66042213|endTime=66042287|confidence=0.5} yards{startTime=66042287|e
ndTime=66042563|confidence=0.5}"
}
```

4.4 Parser

The input to the Phoenix Communicator server is a `send_to_parse` frame, with the following format:

main	
`:parse_input`	The spoken (or typed) text that is to be parsed

For typed text the frame will look like this:

```
{c send_to_parse
   :parse_input "go forward two yards"
}
```

In a frame from the speech recognizer, the `parse_input` will also contain meta-information like time stamps and confidence scores. However, these are lost in the parsing process, so they are not present in the parser output.

The output frame contains all the slots of the input frame, plus a slot `:parse_output` containing the possible parses of the input text, for example:

```
{c main
   :parse_input "GO{STARTTIME=66041061|ENDTIME=66041481|CONFIDENCE=0.5}
FORWARD{STARTTIME=66041492|ENDTIME=66042104|CONFIDENCE=0.5} TWO{STARTTI
ME=66042213|ENDTIME=66042287|CONFIDENCE=0.5} YARDS{STARTTIME=66042287|E
NDTIME=66042563|CONFIDENCE=0.5}"
   :parse_output "PARSE_0:
move:[move] ( GO )
move:[direction] ( [forward] ( FORWARD ) )
move:[distance] ( [Number] ( TWO ) [d_unit] ( [du_yd] ( YARDS ) ) )
END_PARSE

" }
```

The parse is not transmitted in Communicator's internal frame representation (which was actually designed to represent parses and natural language constructs). Instead, a structured string representation is made and put into a single slot. This representation is discussed in the next section.

4.4.1 Phoenix frame representation

Phoenix outputs its frames as a string with a strictly defined, line-based structure. Each parse begins with a line `PARSE_`n`:`, where n is the parse number,

starting at 0. Each line of the parse contains a frame name, followed by a colon, followed by the contents of one slot. The slot name is enclosed in square brackets, followed by its contents enclosed in parens. Slots can be nested, so slots can contain other slots. In this way a *parse tree* is formed. The leaves of this tree are the words from the input text. The words are capitalized: because parsing is not case sensitive, Phoenix converts the input text to upper case before parsing it. The end of a parse is marked by the token `END_PARSE` on a line by itself.

Since parsing is robust, not all words from the input text need to be present in a parse. However, only parses with the maximum number of words from the input text are preferred. That is, if a parse covering n words is possible, no parses covering less than n words will be returned. Also multiple frame types can be present in the parse, but Phoenix prefers parses with the least amount of frames. These two rules limit the number of parses that are be returned, since only 'optimal' solutions are allowed.

4.5 Fusion and Dialog Management

Fusion and dialog management function as a single Communicator server. It handles frames from the parser, which have the following format:

main	
`:parse_input`	The original spoken (or typed) text
`:parse_output`	The possible parses of `:parse_input`

Figure 4.1 shows the classes in the framework that inherit from `galaxy.server.MainServer` and `galaxy.server.Server`. `DialogMainServer`, in its constructor, creates an `Application` object. `Application` stores global application state information for the framework, provides logging facilities, stores a reference to the API object of the application that will be controlled, and generally maintains state information that is global and not specific to a dialog session. Application state is valid regardless of whether a user is currently using the application.

`DialogServer` extends `galaxy.server.Server` by declaring a handler method for the `main` frame. It also provides a constructor that creates a `Session` object. `Session` creates and configures the fusion and dialog manager components, provides access to the scripting engine used for dialog manager action scripts, provides access to the natural language generation module through the **`speak()`** method, and generally maintains state information that is specific to a dialog session. A session is destroyed when a user disconnects, while the application keeps running.

`DialogServer` is an abstract class, with an unimplemented **`dialog()`** method. When a `main` frame is received, `DialogServer` converts the parses to XML DOM trees. The leafs of this tree are the actual words that were spoken. Using this tree and the original utterance, which has time stamps for every word, the words from the parse tree are matched to the words in the utterance, and their timestamps are attached to the parse tree, creating an *annotated* parse tree. These timestamps are propagated up the tree, so that not only the leafs (words) have timestamps, but every node in the parse tree. For a node, the start time stamp is set to the smallest start time stamp of all its child nodes

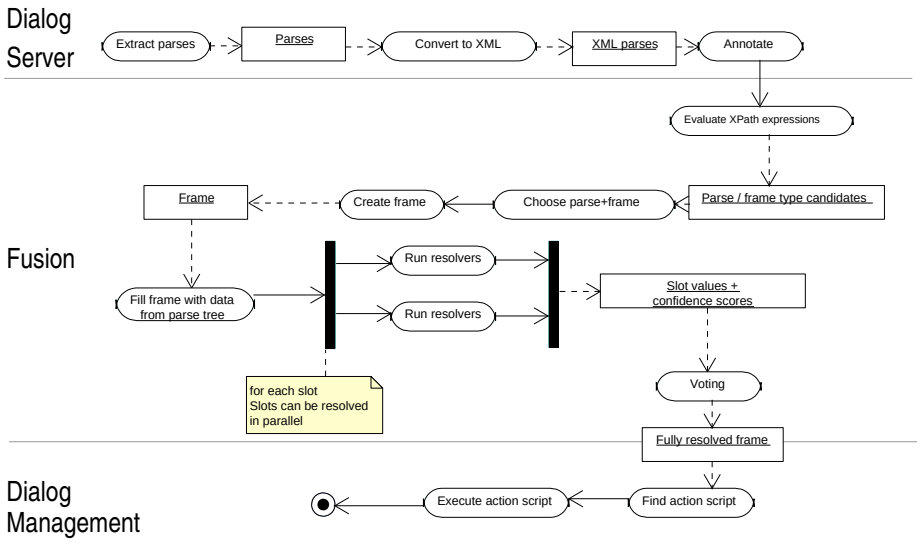

Figure 4.3: A UML activity diagram showing the steps from parser output to dialog manager

and leafs. The end time stamps is set to the largest end time stamps of all its child nodes and leafs.

`DialogServer` then calls its abstract method **`dialog()`** with all the annotated parse trees — each parse tree is an alternate parse of the same utterance. A default implementation `DialogServerImpl` is available that implements the **`dialog()`** by sending the parses that are received to the fusion component, and sending the resulting frames to the dialog manager. The whole process is depicted in the UML activity diagram in Figure 4.3.

4.5.1 Fusion Resources

One task of the previously mentioned `Application` object is to maintain a resource pool. This is simply a hash table of objects that are registered with the `Application` under a unique String identifier. The resource pool is used to reference application objects from the fusion and dialog manager configuration file, or from action scripts. This works as follows:

1. A `DialogMainServer` subclass creates the `Application` object as well as the actual application that the multimodal interface is to control.

2. For each object that needs to be registered, the `DialogMainServer` obtains a resource handle from the `Application` object by calling `Application.`**`getResource`**`(identifier)`, passing the identifier the new resource is to have. This returns a `Resource` object, in which the application object can be set with a call to **`setValue`**.

3. In the fusion configuration file, the resource identifier is used as the value of a `Resolver` or `ContextProvider` parameter of type `resource`.

These steps are shown schematically in Figure 4.4.

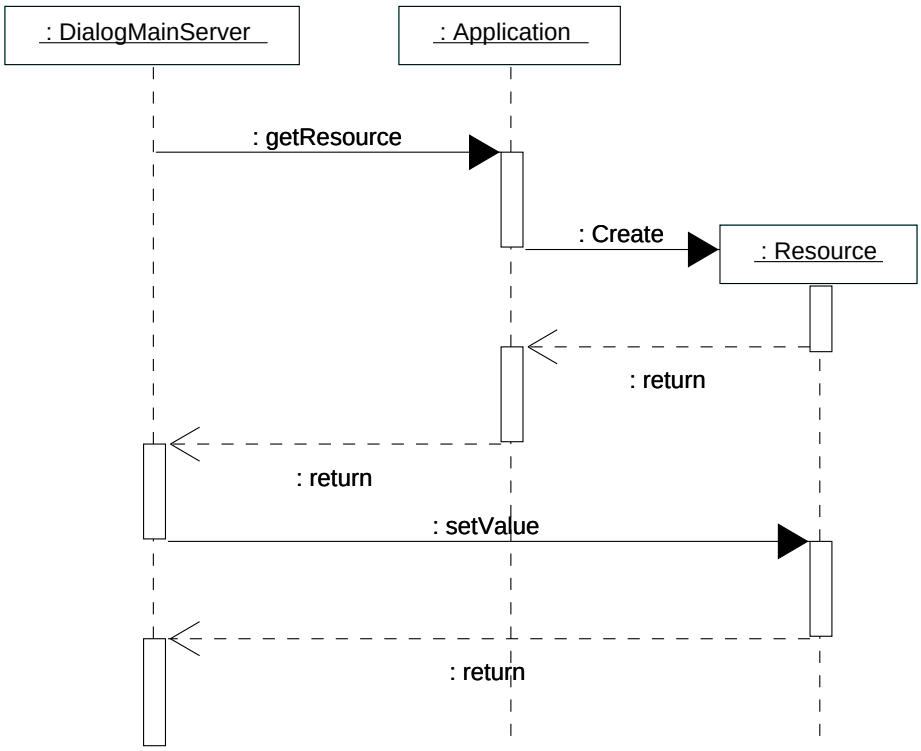

Figure 4.4: A UML Sequence diagram showing how resources are created

The `Resource` class extends `Observable`. This means that context providers and resolvers using resources can respond to changes in their values. For example, the `Mouse` context provider has a parameter of type `resource` that references the `Component` that `Mouse` listens to for mouse events. If a new value is set for this resource (through a call to **`setValue`**, `Mouse` is notified and removes itself as a listener from the old `Component` and attaches itself to the new one.

4.6 Fusion Manager

It is the fusion manager's task to take the possible values for a slot from the resolving agents and choose the one that is optimal, based on the resolving agents' probabilities, and a confidence value or weight for each agent. Currently a very simple voting algorithm is employed, in which the scores for each value are summed and the one with the highest total sum is selected. However, the framework can accommodate any algorithm desired, and we are looking into using fuzzy reasoning and/or Bayesian networks for this purpose, using models trained on empirical data, to obtain better predictions of the user's intent.

```
<frame name="delete" test="delete/delete">
 <slot name="glyph">
  <source select="delete/what/anaph">
   <resolve resolver="mouseObjectResolver" weight="0.5" />
   <resolve resolver="eyeObjectResolver" weight="0.4" />
  </source>
  <source select="delete/what/unit">
   <resolve resolver="unitResolver" weight="0.3" />
  </source>
  <source select="delete">
   <resolve resolver="dialogObjectResolver" weight="0.1" />
  </source>
 </slot>
 <action language="javascript">
  if (frame.glyph) {
   application.api.deleteGlyph(frame.glyph.glyph);
  }
 </action>
</frame>
```

Listing 4.4: A sample frame declaration

The fusion manager, resolving agents, and context providers are configured through an XML file containing declarations for frames, resolvers and context providers. An example frame declaration is shown in Listing 4.4. The declaration specifies the name of the frame, an XPath test on the parse tree that must succeed for the frame to be used, and a set of slots with XPath expressions for each slot specifying their data source and a list of one or more resolvers (resolving agents) for each source.

The fusion manager uses the XPath test to determine which frame to instantiate. If multiple XPath expressions evaluate to 'true', the fusion manager prefers the frame that was used in the previous utterance, if possible. This ensures that multiple-step dialogs continue as intended. If no frame is of the same

type as the previous one and there are multiple frames to choose from, feedback can be generated asking the user to be more specific. The actual implementation of this is left up to the developer, so any message can be generated, but output on the screen is also possible, for instance.

In the future we may defer choosing of the actual frame until after they are filled, so as to use the same type of reasoning to choose a frame as is presently done on the slot level. However, this may lead to a combinatorial explosion, and since it has not yet been investigated whether doing this would be advantageous, we use the current algorithm for now, which, for situations we have encountered, seems to work adequately. Applications with complex dialogs, however, may require a more sophisticated approach.

4.6.1 Resolving Contradictory Inputs

Contradiction between modalities can arise. For example, a user may say "move that infantry squad" while pointing to or looking at an infantry army. On the slot level, speech is treated like any modality, so the results from resolving "infantry squad" — that is, a list of all infantry squads — will participate in the voting process along with the result of resolving "that" using a pointing modality and possibly the dialog history, which will also result in a list of objects — those in the vicinity of the location the user was pointing. Ultimately, the developer decides which modality 'wins' in this case by the weights that he or she allocates to each modality. By adjusting the weights appropriately, the developer can achieve that the pointing modality wins if the object being pointed at is under the mouse cursor, but if the pointer is a certain threshold away from any object, the speech modality will win instead. Since the scores from each resolving agent are summed, the fusion manager may choose an object that is somewhat close to the pointer and is (in our example) an infantry squad over an object directly under the pointer.

A better approach in case of unresolvable ambiguity could be, again, to ask the user for clarification, e.g. "That is an infantry army, not an infantry squad. What do you want to move?".

4.6.2 Fusion Interfaces

In the spirit of an object-oriented framework, the framework core is separated from implementation-specific details by well-defined interfaces. These interfaces, their purpose, definition, and standard implementations provided, if any, are discussed in the following sections.

- Resolver
- ContextProvider
- ContextData
- LocatableObjectStore
- CoordinateTransform

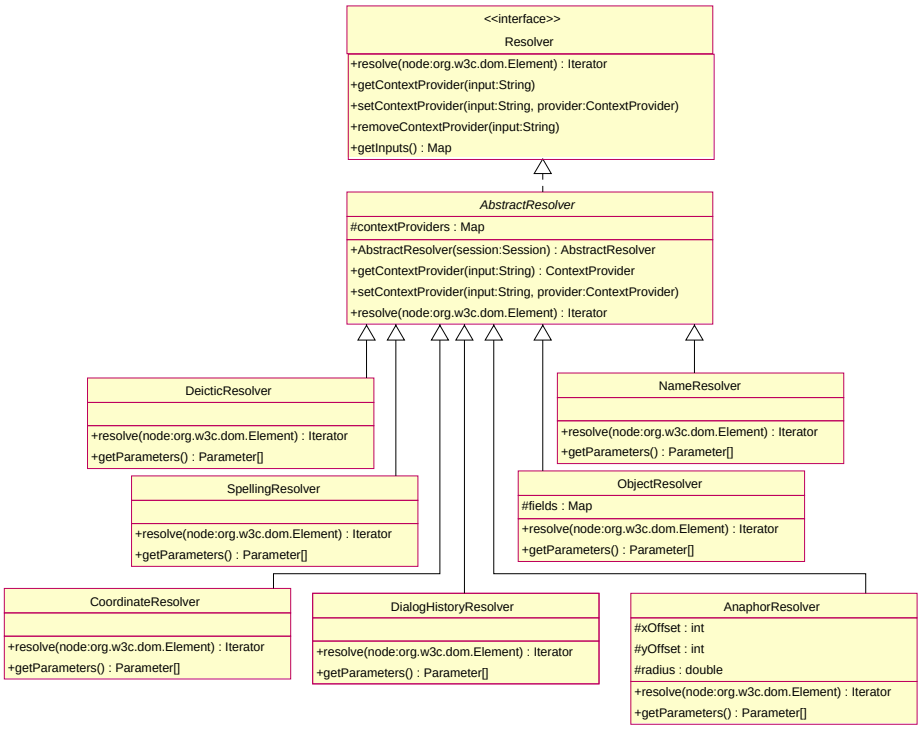

Figure 4.5: The `Resolver` class hierarchy and related classes

Resolver

All resolving agents implement the `Resolver` interface. A resolving agent is initialized by the fusion manager, which passes the initialization parameters as read from its configuration file to the `init()` method.

`Resolver`s will usually be created by subclassing the `AbstractResolver` class, which implements the `Resolver` interface and inherits from `AbstractSessionObject`. The latter class provides common functionality for objects that operate within a multimodal dialog session and can be dynamically instantiated. The key method of the `Resolver` interface is the `resolve()` method, which takes a DOM `Element` representing a parse tree fragment and returns an `Iterator` containing the agents' resolutions along with their assessed probabilities.

The standard resolvers provided by the framework are briefly described in the following sections. For a detailed description of all their parameters and implementation details, the reader is referred to the framework's JavaDoc documentation, which can be found at the address listed in Appendix E.

`GeometricAnaphorResolver` Resolves a demonstrative pronoun to the on-screen object it refers to as determined from the input modality. The probabilities that accompany the solutions are proportional to the object's distance from the (x, y)-data point that is selected. This resolver does not actually look at the data in the parse tree fragment, it just uses the time stamps.

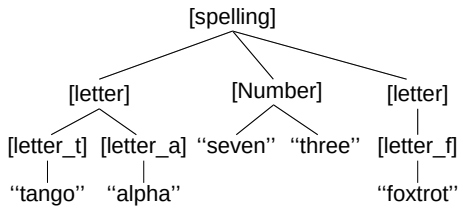

Figure 4.6: An example of a spelling parse tree node

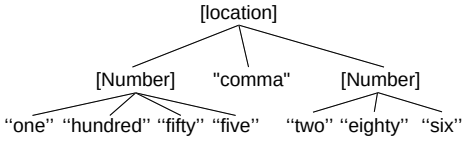

Figure 4.7: An example of a coordinate parse tree node

`DeicticResolver` Resolves a deictic reference to its location on the screen as determined from the input modality. The probabilities that accompany the solutions are determined by the frequency of the point in the (x, y) data list. This resolver does not actually look at the data in the parse tree fragment, it just uses the time stamps.

`ObjectResolver` Resolves a set of object attributes to the objects that match those attributes. For example, when the user is to be able to select an object using speech only with an expression such as “remove the hostile infantry army”, this resolver is used to return all objects matching the attributes named: “hostile”, “infantry”, and “army” in the example. This can also be used to resolve a small set of anaphora and elliptical expressions, for example “move the *friendly one*”, or ”remove the hostile army ... and the *neutral*”.

`SpellingResolver` Takes a parse tree fragment containing nodes that represent spelled letters and returns the word they spell out. Spelling can be very useful when a parser or speech recognizer is used that requires an a priori vocabulary or grammar and cannot learn new words while running. In this way, out of vocabulary terms, such as names, can be spelled out. Spelling using a specialized alphabet, such as the NATO alphabet, is also very accurate. This resolver returns a single `String`, with a probability of 1.

`DialogHistoryResolver` Returns the first item in the dialog history for the slot. This is the most recently used value for the slot. Slots are identified by name, so this also looks for slots with the same name used in other frames.

`CoordinateResolver` Resolves a parse node representing a (2D) coordinate, such as shown in Figure 4.7, to a Java `Point` object. One `Point` is returned, with a probability of 1.

`NameResolver` A trivial resolver that simply returns the name of the parse tree fragment's top level node. Some simple transformations can be done on

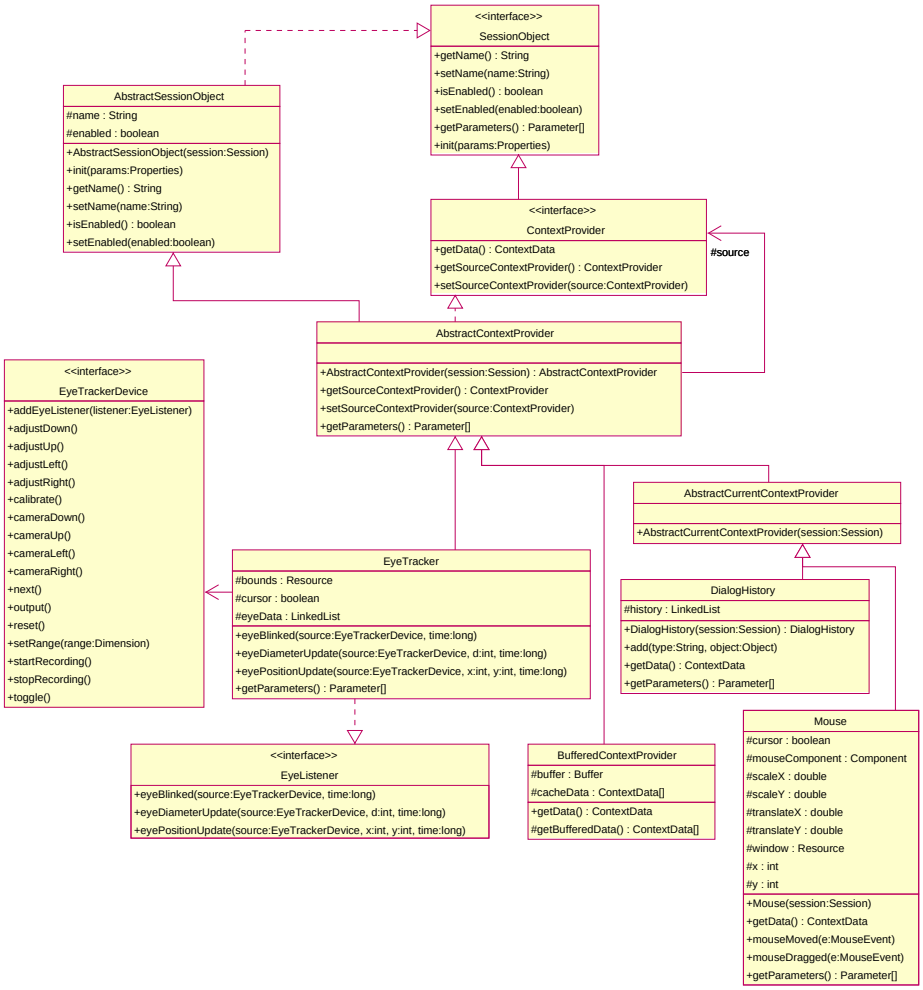

Figure 4.8: The `ContextProvider` class hierarchy and related classes

the returned name, including stripping fixed leading and trailing strings, and applying a translation.

ContextProvider

Classes that provide access to modalities implement the `ContextProvider` interface. The key method is `getData()`, which returns a `ContextData` instance that provides the modality's data along with a timestamp for which the data is valid.

ContextProviders can be *stacked*. By doing this, data from a sensor or modality can be transformed in several atomic steps into something more useful. To promote reusability, each ContextProvider should only perform a simple operation. Figure 4.9 shows a stack of ContextProviders in which a clustering algorithm is applied eye tracker data data; the resulting data are buffered to create a `ContextDataList` of points that can be used by a resolving agent to resolve anaphora or deictic references, by searching this list of points to find

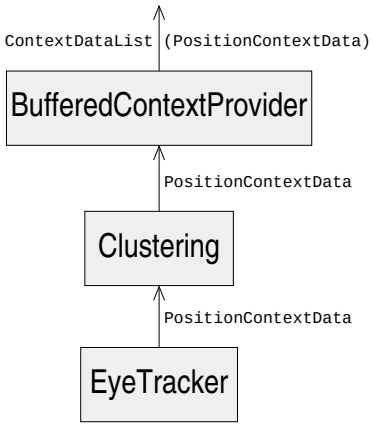

Figure 4.9: A stack of three context providers: eye tracker coordinates are clustered and buffered in a `ContextDataList`

out where the user was looking when he/she uttered the anaphor or deictic reference.

The base context providers (context providers that don't stack on other context providers, but generate their own data) supplied by the framework are currently `EyeTracker`, `Mouse`, and `DialogHistory`. Secondary context providers are currently `Clustering`, `MouseFilter`, and `BufferedContextProvider`.

ContextData

`ContextData` implementations represent data from a modality. Three implementations are supplied with the framework: `PositionContextData`, `Entity`, and `ContextDataList`. `ContextDataList` is a container for other `ContextData` objects, and is returned by the `BufferedContextProvider`, which polls another context provider and caches its data for the duration of an utterance, so that multiple resolvers can use the same data through a single `BufferedContextProvider` instance. Most resolvers expect a `ContextDataList` as input.

LocatableObjectStore

As shown in Figure 4.5, an `LocatableObjectStore` implementation plugs unto the framework's `ObjectLocator` class to enable it to query the application for on-screen objects. This is used in the `AnaphorResolver` to find objects in the neighborhood of the point on the screen the user indicated when speaking a pronoun. It is used by the `ObjectResolver` to find objects matching a set of attributes. The elements of the `LocatableObjectStore` are instances of a concrete subclass of `LocatableObject`. `LocatableObject` uses the "wrapper" design pattern to provide a uniform interface to access an object's location, size, and attributes.

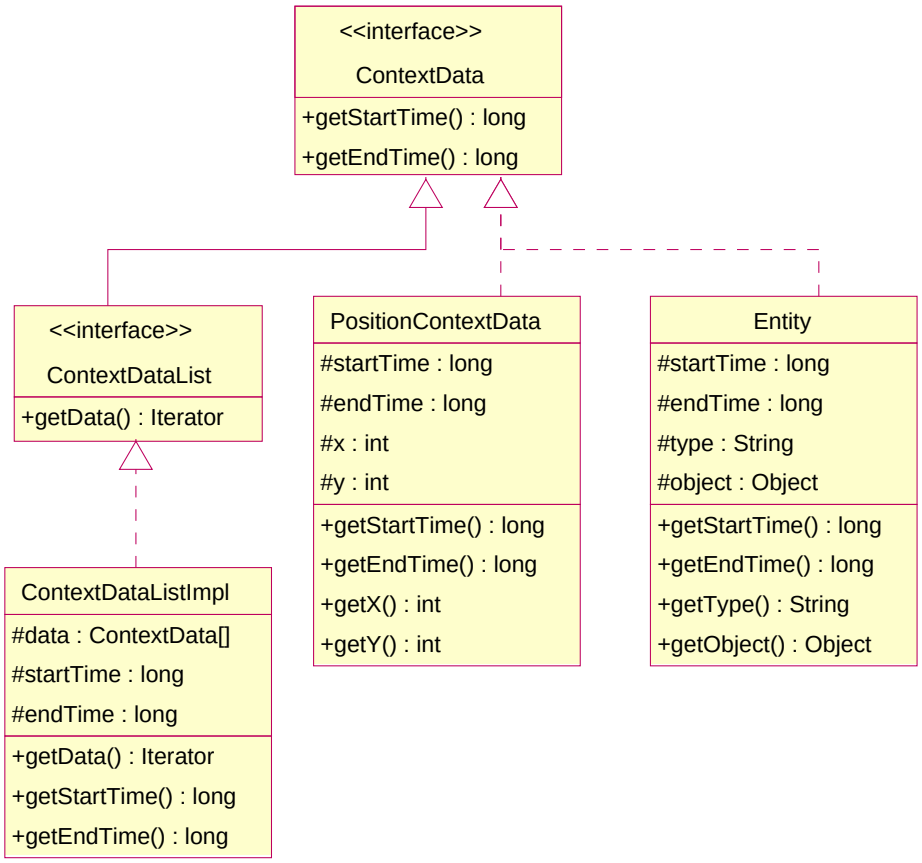

Figure 4.10: The `ContextData` class hierarchy

CoordinateTransform

Rarely are window or screen coordinates used directly in the application. Rather, these coordinates are transformed into logical coordinates in the application workspace. This transformation can be dynamic, depending on scroll or pan position, zoom factor, and other factors. Since resolvers need logical coordinates to perform their resolution activities, the window or screen coordinates captured by context providers such as `Mouse` and `EyeTracker` need to be transformed before they can be sent to a resolver. Since this transformation is very much application- dependent, this is captured in an interface that applications must provide an implementation for, to be able to use context providers that generate window or screen coordinates. This interface is `CoordinateTransform`.

Applications register a `CoordinateTransform` with the framework through the resource manager, giving it an identifier. This same identifier is used in the context provider configuration to determine which `CoordinateTransform` to use. Since resources are dynamic, the `CoordinateTransform` can be changed at runtime, changing the transformation algorithm used, although this will rarely be necessary.

4.7 Dialog Manager

Dialog management consists of maintaining dialog context and making application calls when frames are completed. The first task is implemented by having the dialog manager maintain a `DialogHistory` object, which implements the `ContextProvider` interface, so that it can provide data to resolving agents, notably the `DialogHistoryResolver`.

Application calls are made by the dialog manager through JavaScript. When the dialog manager receives a frame, it retrieves the associated script and executes it through the Bean Scripting Framework [2], a framework that provides scripting language support to Java applications. JavaScript provides a flexible way to interface between the dialog manager and the application. Since JavaScript is interpreted code, all that is needed to change the behavior of the multimodal interface is to modify the script and re-run the application; no code needs to be compiled.

4.7.1 Dialog History

where	(500, 950)
what	⟨object-015⟩
what	⟨object-012⟩
affiliation	“friendly”
unittype	“infantry”
unitsize	“squad”
where	(300, 950)

Figure 4.11: An example dialog history

The Dialog Manager also updates the dialog history. Each slot in the frame is put in the dialog history. Figure 4.11 shows an example dialog history as it might look after a few commands. Note that the history is capped to seven entries, as it is very unlikely the user will want to reference anything that was mentioned longer ago than that.

The dialog history allows for simple references to things previously mentioned in the dialog, and enables the fusion manager to inherit slots for elliptical expressions. For example, “move the infantry army there ... delete *it*” (anaphor referring to “the infantry

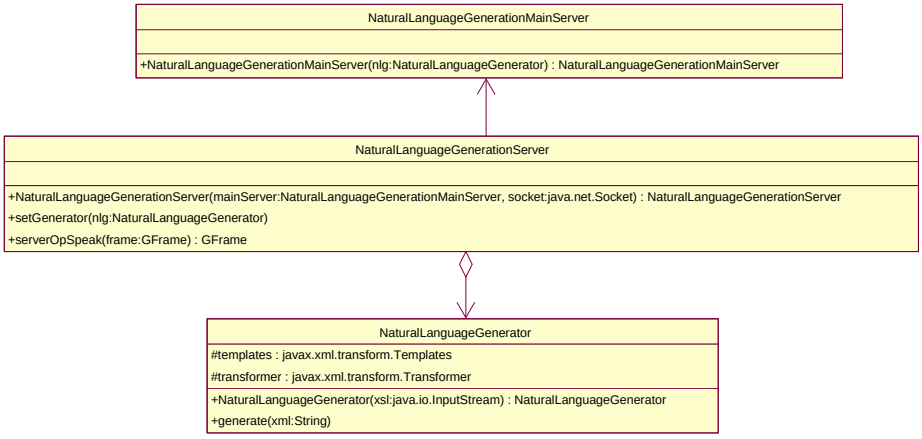

Figure 4.12: A UML class diagram of the natural language generation component

army" in the previous sentence). Or "drive forward for five yards ... three more yards" (elliptical expression which expands to "drive forward for three more yards").

A limitation of the dialog manager is that fully resolved slots are stored. So if the last example in the previous sentence were said as "drive forward for five yards ... three more", it would not have the intended result. The fact that the user said "yards" in the previous sentence is not used, as only the canonical distance value (in millimeters) is stored, and not what the user originally said ("five yards"). The result would be that the robot would drive forward for three millimeters, obviously not what was intended.

4.8 Natural Language Generation

The Natural Language generation classes are shown in Figure 4.12. The `NaturalLanguageGenerator` has a single method **`generate()`** that is called by the `NaturalLanguageGenerationServer` in response to an incoming `speak` frame. The frame is expected to have the following form:

speak	
`:xml`	The XML to transform to text
`:lang`	The language to generate text in (an ISO language code)

After generation, the following frame is output:

speak	
`:xml`	The XML that was transformed to natural language
`:lang`	The language that text was generated in
`:text`	The text that was generated

This frame is sent to the hub which will generally route it to a speech synthesizer to speak the generated text.

Chapter 5

Building the Robot Control Application

The goal set in Chapter 1 was to create a multimodal interface to control a robot. The previous two chapters have described a generic multimodal framework. This chapter will show how this framework was used to quickly create an interface that enables the robot to be control using voice, mouse, gaze, and possibly other modalities.

5.1 The Robot

The robot used was a Pioneer 2-AT robot manufactured by ActivMedia Robotics [1]. The model owned by CAIP is powered by three batteries, has a mounted pan–tilt–zoom camera, an onboard PC with a 400 MHz processor, a wireless 802.11b card for wireless networking, and runs GNU/Linux.

Figure 5.1: The Pioneer 2-AT all-terrain robot (pictured here with the gripper accessory, which CAIP's model does not have)

5.2 Design

Software for controlling the robot using a traditional GUI interface was already in place when I started this project. CAIP's Flatscape software was used for this. Flatscape is a collaborative mission planning tool developed by CAIP for the U.S. military. It features a map overlay on the screen on which military units can be placed; units are represented by icons with different color, size, and decorations depending on affiliation (hostile, friendly, neutral, or unknown), size (anywhere from unit to army group), and other properties (whether the unit is tracked, airborne, etc.) — all this is described in [57]. Different situ-

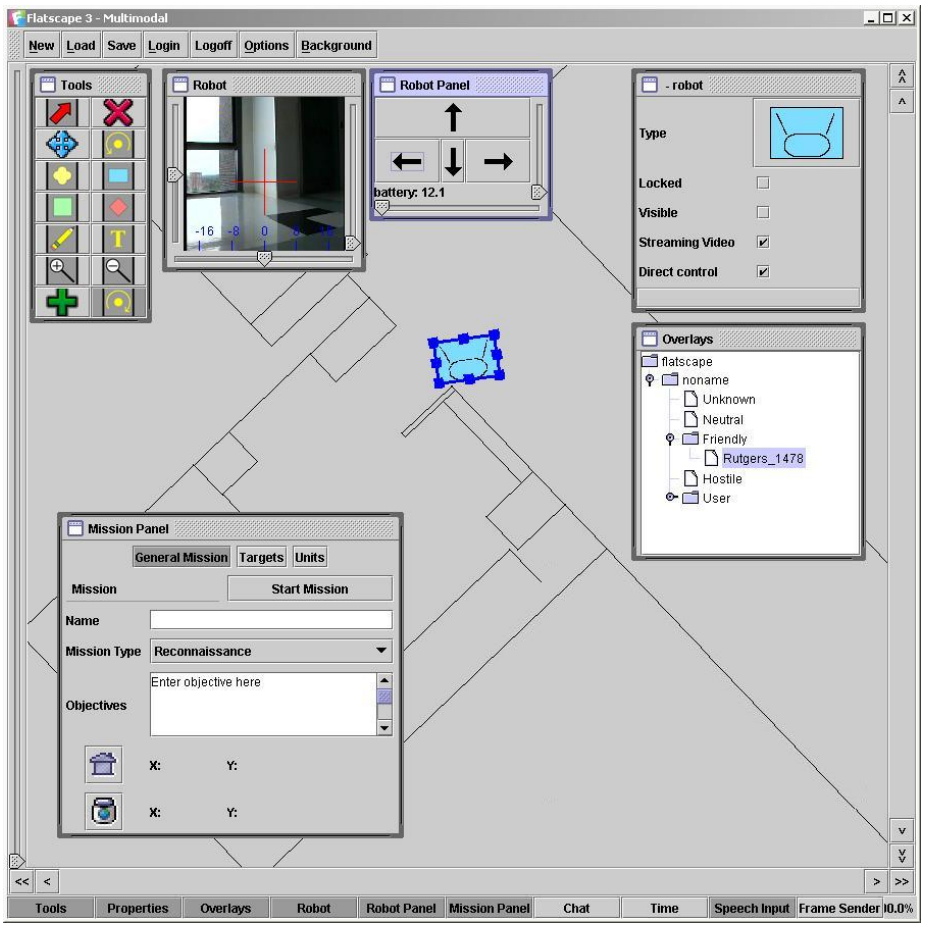

Figure 5.2: Flatscape with panels for monitoring and controlling the robot

ations can be set up on a time line and a mission can thus be completely planned ahead. Flatscape's GUI is very configurable, and new panels and toolbars can easily be added by creating a new subclass of the panel or toolbar superclass and including it in Flatscape's configuration file.

Flatscape runs on DISCIPLE [39], CAIP's collaborative middleware, which means that multiple people can participate in the mission planning, using different PC's or PDA's. The changes one person makes are reflected on the screen of the other participants. This is very useful as mission planning is seldom done alone.

Robot control is provided as two new panels on the Flatscape screen; one panel consists of four arrow buttons for moving the robot forward and back and rotating it left and right, as well as two sliders for setting travelling and rotation speed. The other panel shows camera images as recorded by the pan–tilt–zoom camera mounted on the robot, and has three sliders to control pan (horizontal camera movement), tilt (vertical camera movement), and zoom. Additionally, there is a new icon type for robots on the map, and a properties panel to edit a robot's properties (like what mode it is in, and whether images from the camera

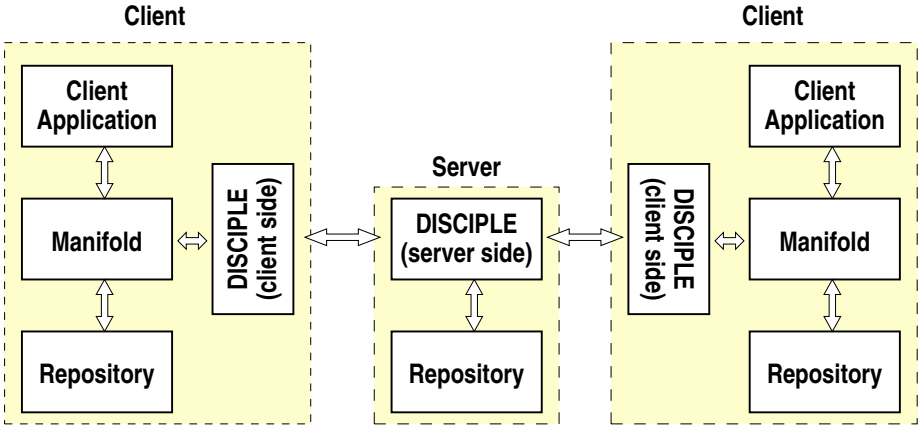

Figure 5.3: Overview of the DISCIPLE system architecture (from [39])

should be shown on the screen). All this can be seen in Figure 5.2.

5.2.1 The Robot Model

The robot model which contains every aspect of the robot's state is stored on the DISCIPLE server in a repository. The model is a DOM (Document Object Model) tree. Every connected client has a copy of the model in its own repository, which is updated by node and property change events sent by the DISCIPLE server in response to changes made by the clients. This is depicted in Figure 5.3.

The robot model's structure is shown in Table 5.1. Fields that I added are marked with an asterisk (*). **targetLocation** and **locationReached** were added to be able to move the robot to any arbitrary location on the map, by pointing, for instance. With the new **distance** and **angle** fields, the robot can be told to move or rotate a certain distance, where previously seperate "move" and a "stop" commands were needed to achieve the same effect. This was born partly out of necessity. As the speech recognizer can exhibit delays of up to a second, a "stop" command would sometimes not reach the robot until a second after it was spoken, long after the robot has passed the point at which it was to stop, causing to even run into walls at times. By telling the robot "Move back 30 centimeters", for example, this problem is alleviated since no corresponding "stop" command is needed; the robot will simply stop after having moved the requested distance.

Fields that Flatscape uses are marked in Table 5.1 with a dagger (†). The **name** property is used by Flatscape as a label in the "Overlays" panel, which shows a tree view of all objects on the map. Each icon, or "glyph", in Flatscape terminology, has a **transform** property which contains position, orientation, and scaling information for three dimensions. Flatscape uses this to draw the glyph in the correct location on the screen, and with the correct rotation and scaling applied. **unittype**, **affiliation**, and **modifiers** are the three properties of all "unit" glyphs on the map, which are military units. These attributes are used to determine which icon to use (there is a seperate icon for each legal combination of these three properties).

Table 5.1: The robot data model; the last two columns indicate who "controls" a property. A check in the F column means the Flatscape client (or another client) sets the property to request a change on the robot. A check in the R column indicates that the robot will update that property to synchronize the model with the robot's state

property name	data type	description	F	R
name †	`String`	The robot's name (initially set on the robot side)		√
forward	`boolean`	Whether the robot is to move forward	√	
back	`boolean`	Whether the robot is to move back	√	
right	`boolean`	Whether the robot is to turn right	√	
left	`boolean`	Whether the robot is to turn left	√	
transspeed	`int`	Desired travelling speed, in in millimeters per second (mm/s)	√	
rotspeed	`int`	Desired rotation speed	√	
compassdata	`String`	Bearings in degrees as indicated by the compass (or `"0"` if there is no compass)		√
poseInterval	`int`	Frequency with which updated location and bearings are sent	√	
poseDistance	`int`	Minimum distance to move before updated location and bearings are sent	√	
stream	`boolean`	Whether to accept and display streaming video (only used by the Flatscape client; the robot's video broadcasting is done outside of DISCIPLE)	√	
controlstate	`boolean`	Whether the robot is in direct control mode (**`true`**) or autonomous mode (**`false`**)	√	
unittype †	`String`	The robot's unit type. Initially set to `"XXX"` on the robot side	√	
affiliation †	`String`	The robot's affiliation. Initially set to `"F"` (friendly) on the robot side	√	
modifiers †	`String`	The robot's modifiers. Initially set to `""`	√	
transform †	`double[]`	The robot's position and bearings encoded in a nine-dimensional double array		√
battery	`double`	Battery voltage.		√
batteryInterval	`int`	Frequency with which battery voltage updates are sent		√
pan	`int`	Camera pan in degrees; 0 is straight ahead (in the direction the robot is facing), negative values pan left, positive pan right	√	
tilt	`int`	Camera tilt in degrees; 0 is level, negative values tilt down, positive pan up	√	
zoom	`int`	Camera zoom.	√	
distance *	`int`	The distance the robot is to move, in millimeters. Positive values will move the robot forward, negative values will move it backward. Is set to zero when the desired distance has been travelled	√	
angle *	`int`	The angle the robot is to rotate, in degrees. Positive values will rotate the robot left, negative values right. Is set to 0 when the desired angle has been reached	√	
maxtransspeed *	`int`	The maximum travelling speed, in in millimeters per second. This is used when the robot moves autonomously It will never go faster than this	√	
maxrotspeed *	`int`	The maximum rotation speed. This is used when the robot moves autonomously. It will never rotate faster than this	√	
targetLocation *	`int[]`	A two-dimensional array (x,y) with a point the robot is to move to	√	
locationReached *	`boolean`	Set to **`false`** when targetLocation is set, and **`true`** when the robot has reached the desired location, or determined it is unreachable	√	√

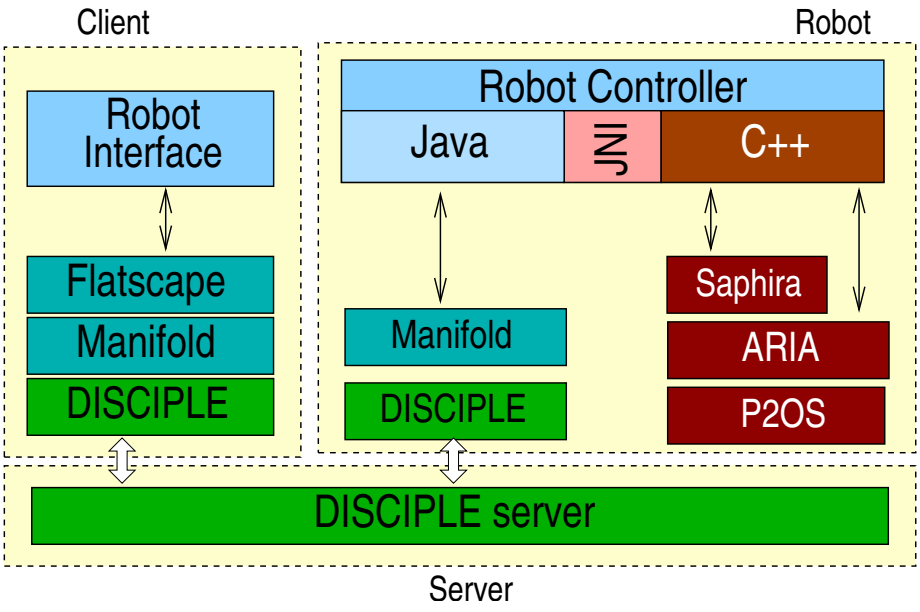

Figure 5.4: Model of the various modules used in the robot system and how they interact

A model of the system is shown in Figure 5.4. The system clearly consists of two distinct parts: the Flatscape client that is the user's interface to the robot's functions, as described in the previous paragraph, and the robot-side client that actually communicates with the robot. Both parts communicate over DISCIPLE's collaboration bus, emobodied in the DISCIPLE server.

On the robot side, a robot controller, written partly in Java, receives property changes made in the Flatscape client over DISCIPLE's collaboration bus (the collaboration bus is an abstraction of the network connection between a DISCIPLE client and the DISCIPLE server). Based on the property change, it will have to make function calls in the robot's control software to achieve that the property change is reflected in the robot's state.

The robot hardware is directly controlled by its onboard operating system, P2OS. However, this operating system can be controlled from the onboard PC over a serial link using a high-level C++ interface called Aria. Aria provides commands for moving and rotating the robot, both directly as well as through more abstract "behaviors". On top of Aria, Saphira provides additional intelligence for gradient path planning and localization, given an *a priori* map.

Aria and Saphira have C++ interfaces. However, DISCIPLE and the part of the robot controller that communicates with DISCIPLE were written in Java. Therefore, JNI (Java Native Interface [74]) is used to be able to call Aria and Saphira functions from the Java part of the Robot Controller. Part of the robot controller is written in C++ and provides access to Aria and Saphira's functionality. The JNI layer makes these functions available on the Java side. Most of this was already implemented previously [27], however I did have to make some additions and improvements to accomodate the multimodal interface. This will be described later on.

5.2.2 Design of the Multimodal Interface

Adding a multimodal interface to the application described in the previous section required doing the following things:

1. Analyzing which commands the interface should recognize and perform.
2. Writing a parser grammar for those commands.
3. Developing new resolving agents specific to this application, to be used by the fusion manager.
4. Writing an API class that provides an interface to Flatscape and DISCIPLE, with methods that can be called from the action scripts in the fusion configuration file.
5. Configuring the fusion manager by defining which frames are instantiated as a result of parser output, and which modalities are used to resolve each frame's slots.
6. Making changes to the robot-side Java and C++ code to add new features.

The new design of the Flatscape client with the multimodal interface is shown in Figure 5.5.

5.2.3 Robot Commands

The initial assessment of multimodal commands was based on what what possible using the GUI interface. This resulted in the following command categories:

- **Movement:** Moving forward and backward, rotating left and right, stopping.
- **Camera:** Panning, tilting, and zooming the camera. Toggling reception and display of streaming video.
- **Missions:** Setting mission name, home base, resupply point, and type. Adding points to visit during missions. Starting, pausing and aborting missions.

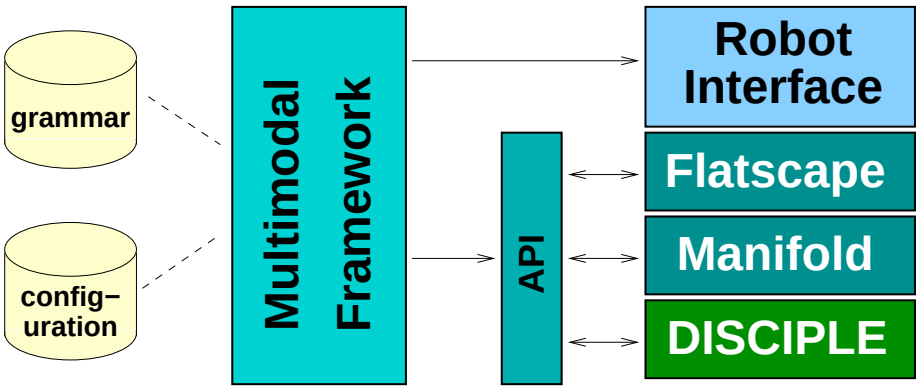

Figure 5.5: Design of the multimodal robot client

- **Flatscape commands:** Logging in, loading a map, positioning and moving units.

Additional commands were added that were either useful or very natural in a multimodal interface:

- Moving a predefined distance
- Rotating a predefined distance
- Moving to a specific point on the map
- Returning to the home base

5.2.4 Writing a Grammar

Writing a grammar requires determining in which manner commands can be articulated by a user. Since we're using a robust parser, including just the salient phrases — that is, the words or phrases that actually contain relevant information — in the grammar should suffice. For instance, there's a myriad of ways to tell the robot to move forward, e.g.

"robot, please move forward"

"drive forward"

"move in the forward direction"

Obviously the most important part of these sentences is the word "forward". Also, words like "move", "drive", or "go" can indicate the user wishes the robot to move, as opposed to looking forward, or shooting forward. Additional information can be the speed the robot is to move at, the distance it is to move for, or a specific point on the map it is to move towards. The frame definition of **move** contains these — and just these — properties, or slots, and is shown in Listing 5.1.

```
FRAME: move
NETS:
        [move]
        [direction]
        [distance]
        [where]
        [speed]
;
```

Listing 5.1: The frame definition of the move frame

Other frames were defined in a similar fashion for the following frames:

- **rotate** — For rotating the robot
- **stop** — For stopping the robot
- **control** — For switching between direct control to autonomous movement

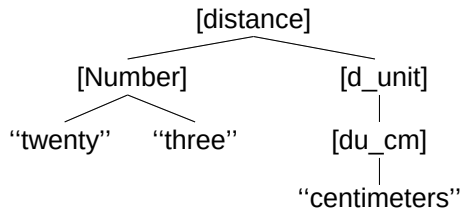

Figure 5.6: An example of a `distance` parse tree node

- **camera_pan** — For panning the mounted camera
- **camera_tilt** — For tilting the mounted camera
- **camera_zoom** — For zooming in and out with the mounted camera
- **camera_visual** — For setting whether camera images are to be displayed
- **mission** — For setting mission properties (name, type, home base, resupply point)
- **mission_task** — For adding a task to a mission
- **mission_control** — For starting, pausing, continuing and aborting a mission
- **login** — For logging in and out of the DISCIPLE server
- **connect** — For opening a map and establishing a connection with the robot
- **create** — For creating a new military unit on the screen
- **delete** — For deleting an object on the screen
- **identify** — For identifying an object on the screen
- **name** — For labelling a location on the screen, so that it can later be referred to, e.g. “call this tango–five ... move to tango–five”

A complete list of all frames and their grammars is given in Appendix C.

5.2.5 Developing New Resolving Agents

The Robot domain called for some extra resolving agents. I added some of these to the framework because they seemed sufficiently general to be applicable to other domains as well. Others are specific to the robot domain and are kept seperate from the framework.

The new resolvers are discussed in this section.

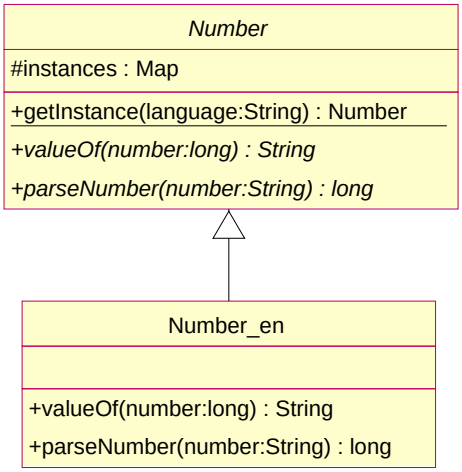

Figure 5.7: The `Number` abstract class

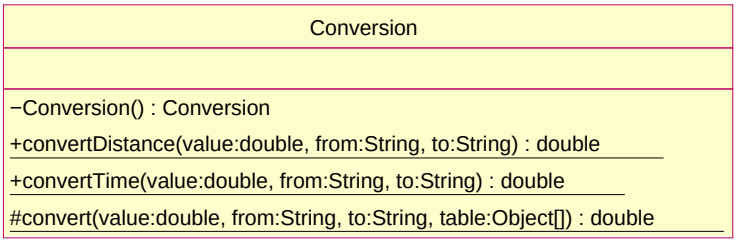

Figure 5.8: The `Conversion` class

`DistanceResolver`

There are many ways to indicate a distance in natural language. Humans often use fuzzy terms, such as “a little bit”, ”a lot”, etc. For more accurate distance measurements, we can use a range of distance units, such as meters, yards, inches, etc. The `DistanceResolver` was built to derive a canonical distance representation from a parse tree node. An example `distance` parse tree node is shown in Figure 5.6.

To interpret the `Number` part of the parse tree, I designed an abstract class `Number` which uses the “Multiton” pattern. It has a static **`getInstance()`** method that will return a concrete, language-specific implementation, and two abstract methods to convert a `String` into a number and vice-versa. Each concrete implementation implements these two methods. A UML diagram is shown in Figure 5.7. By putting this in a seperate, abstract class, the framework maintains its design goal of language-independence, as stated in Section 3.3.1.

Given the distance unit in the parse tree, the `DistanceResolver` uses a utility class `Conversion` (shown in Figure 5.8) to convert the number from that unit to the canonical unit (the default is millimeters).

`SpeedResolver`

`SpeedResolver` works very similarly to the `DistanceResolver` described

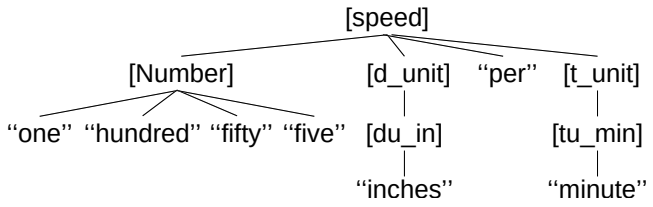

Figure 5.9: An example of a `speed` parse tree node

in the previous ection, but parses speed values instead of distance values. This involves two conversions, since speed is expressed in a distance unit per a time unit, which must be converted to a canonical unit (millimeters per second is the default). The `Conversion` class shown in Figure 5.8 is used for this as well.

A sample `speed` node is shown in Figure 5.9.

Besides converting exact natural language speed indications to millimeter per second values, `SpeedResolver` can also be configured to recognize some constant terms and assign values to them. In the robot system, the `SpeedResolver` recognizes “slow”, “moderately fast”, ”fast”, and ”as fast as you can” and assigns corresponding values.

5.2.6 Writing an API Class

The action scripts in the fusion/dialog configuration file have access to the application through a single object, of which a reference is stored in the `application.api` variable that all action scripts have access to. As shown in Figure 5.10, the API class has references to application objects and provides getter and setter methods that operate on these objects. This greatly simplifies the action script since in most cases all they need to do is make a single method call on the API object.

5.2.7 Writing the Fusion Manager Configuration File

For all the frame types described in Section 5.2.4, declarations were written for the fusion and dialog managers. Some frames, such as `login` and `stop` were trivial as they are always speech-only and unambiguous. Other frames, such as `move` are more complicated since they can contain anaphora, references to other modalities, etc.

Context providers (modalities) were defined first. The only non-speech modalities are mouse and gaze, so the configuration file contains two declarations, shown in Listing 5.2.

A total of twenty-four resolvers are defined. These are summarized in Table 5.2. The actual declarations are in Appendix D, so I just summarize them here briefly

Some resolvers have very trivial functions. **homeResolver**, for instance is used to resolve the word “home”, or “home base” to the actual home base coordinate as set in the application. Others are more complicated, and use input from another modality to do their work. The resolvers can be categorized based on their task and complexity:

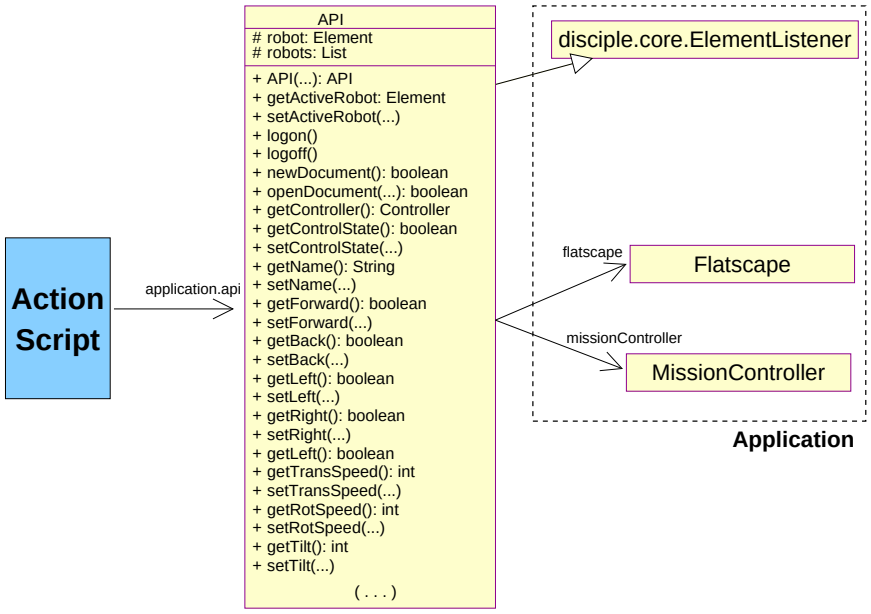

Figure 5.10: Diagram of the robot API class and related classes

name	class	context input
eyeObjectResolver	AnaphorResolver	eyeTracker
mouseObjectResolver	AnaphorResolver	mouse
eyeLocationResolver	DeicticResolver	eyeTracker
mouseLocationResolver	DeicticResolver	mouse
coordinateResolver	CoordinateResolver	
dialogObjectResolver	DialogContextResolver	_DIALOG
distanceResolver	DistanceResolver	
panAmountResolver	DistanceResolver	
tiltAmountResolver	DistanceResolver	
zoomAmountResolver	DistanceResolver	
speedResolver	SpeedResolver	
rotationSpeedResolver	SpeedResolver	
spellingResolver	SpellingResolver	
unittypeResolver	NameResolver	
unitsizeResolver	NameResolver	
affiliationResolver	NameResolver	
unitResolver	ObjectResolver	
unitsizeHistoryResolver	DialogContextResolver	_DIALOG
unittypeHistoryResolver	DialogContextResolver	_DIALOG
affiliationHistoryResolver	DialogContextResolver	_DIALOG
missionTypeResolver	NameResolver	
taskTypeResolver	NameResolver	
homeResolver	HomeResolver	
namedLocationResolver	LocationResolver	

Table 5.2: The resolver instances used in the robot interface

```
<contextproviders>
  <contextprovider id="eyeTracker"
  class="edu.rutgers.caip.communicator.fusion.BufferedContextProvider">
    <param name="poll-interval" value="0" />
    <param name="buffer-size" value="500" />
    <contextprovider id="eyeClustering"
    class="edu.rutgers.caip.communicator.modalities.Clustering">
      <contextprovider id="eyeTrackerSource"
          class="edu.rutgers.caip.communicator.modalities.EyeTracker">
        <param name="in-port" value="COM1" />
        <param name="out-port" value="COM2" />
        <param name="cursor" value="on" />
        <param name="emulate" value="off" />
        <param name="bounds" value="bounds" />
      </contextprovider>
    </contextprovider>
  </contextprovider>

  <contextprovider id="mouse"
  class="edu.rutgers.caip.communicator.fusion.BufferedContextProvider">
    <param name="poll-interval" value="100" />
    <param name="buffer-size" value="50" />
    <contextprovider id="mouseFilter"
    class="edu.rutgers.caip.communicator.modalities.MouseFilter">
      <param name="inactive-timeout" value="1000" />
      <param name="poll-interval" value="100" />
      <contextprovider id="mouseSource"
          class="edu.rutgers.caip.communicator.modalities.Mouse">
        <param name="window" value="canvas" />
        <param name="cursor" value="on" />
        <param name="transform" value="flatscape-transform" />
      </contextprovider>
    </contextprovider>
  </contextprovider>
</contextproviders>
```

Listing 5.2: The contextproviders section from the fusion configuration file for the robot control application

- **Simple evaluation** — `homeResolver`, `coordinateResolver`, `missionTypeResolver`, `taskTypeResolver`, `unittypeResolver`, `unitsizeResolver`, `affiliationResolver`

  These resolvers all return a single result, as they perform a trivial evaluation or mapping. `homeResolver` returns a `Point` that it obtains from the application, `coordinateResolver` reads the x and y coordinates from a `Point`, the other resolvers listed merely return the name of the parse node that the fusion manager passes to it (as directed by the frame declaration that uses the resolver).

- **Evaluation/conversion** — `distanceResolver`, `panAmountResolver`, `tiltAmountResolver`, `zoomAmountResolver`, `speedResolver`, `rotationSpeedResolver`, `spellingResolver`

  These are all instances of `DistanceResolver` or `SpeedResolver`, both of which were described in Section 5.2.5. They convert a parse tree node containing a distance or speed value into an integer value representing the speed or distance in the desired unit (which, for the robot, is millimeters for distance and millimeters per second for speed). The reason that there are separate instances of this resolver for pan, tilt, and zoom values, is that the values of the symbolic terms are different. For panning, "a little" means 30 degrees; for tilting, it means 5 degrees (because the total tilt range is only 30 degrees).

- **Ellipsis resolution** — `unitResolver`

  Resolves phrases like "the hostile one", or "the squad", using a partial description of a unit to find the matching units.

- **Dialog context** — `unittypeHistoryResolver`, `unitsizeHistoryResolver`, `affiliationHistoryResolver`, `dialogObjectResolver`

  These take data from the dialog history to have the current frame "inherit them". For example, if the user says "create a hostile infantry army here", followed by "and a friendly one there", these resolvers achieve that "infantry" and "army" are inherited as unit type and size, respectively.

- **Multimodal resolution** — `eyeObjectResolver`, `mouseObjectResolver`, `eyeLocationResolver`, `mouseLocationResolver`

  Used in multimodal commands. The object resolvers resolve pronouns to the geometric objects on the screen that the user was looking at while uttering the pronoun. The location resolvers do the same, but for points onthe screen instead of objects.

5.2.8 Modifying Robot-Side Code

New Commands

Since some new commands and one new mission type were added to the client interface, the code on the robot needed to be extended as well. Several classes need to be changed to implement a new command. A UML Sequence diagram for a typical command is shown in Figure 5.11. In `RobotCentral`, a jump is

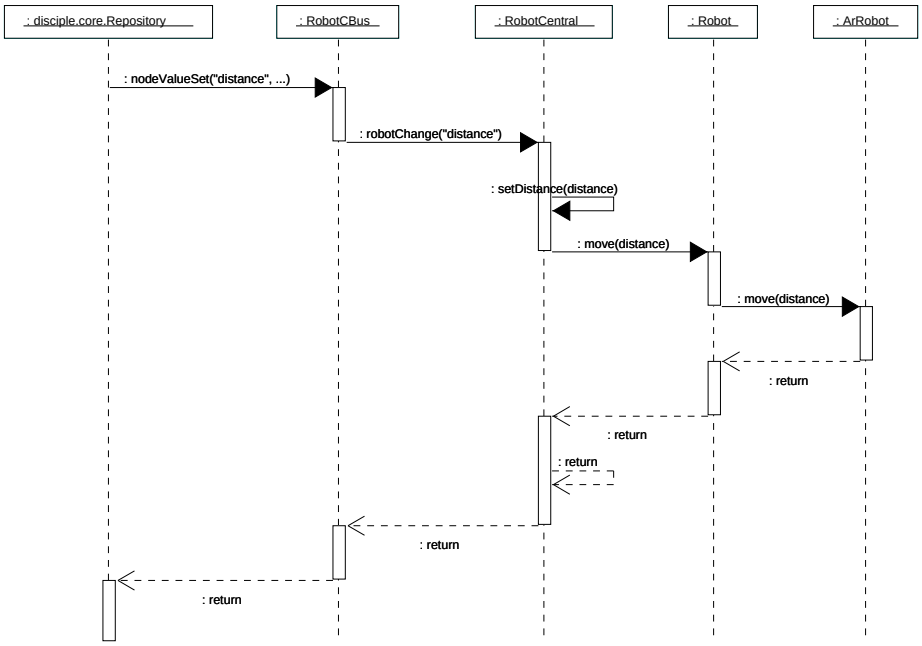

Figure 5.11: A UML Sequence Diagram for a `setDistance` command

made from Java code to C code when the **`setDistance()`** method is called. On the C side, the native implementation uses the Java object pointer to find the corresponding C object pointer for the `Robot` class, and then the **`move()`** method is called on that instance.

The three new commands that were added are **`setDistance`**, **`setAngle`**, and **`setTargetLocation`**. Besides mapping these commands to method calls on Aria's `ArRobot` class, I also needed to add three so-called "user tasks". These are wrappers to functions that are called periodically by the robot software. I use them to monitor whether the robot has reached its desired distance, angle, or target location. A callback is then made to the appropriate Java method — **`moveDone()`**, **`rotationDone()`**, or **`locationReached()`** or **`locationFailed()`**, depending on whether the robot could find a path to the requested target location. The robot model is then updated to reflect this.

New Mission Type

A new mission type — `PatrolMission` — was also added. The main reason was to support experiments that others at CAIP were doing with wireless networking, and for which the robot could be used (since it has a wireless link). But it also provided a nice proof of concept: that a new mission type can be added fairly easily. The mission was based on the existing `ReconMission`, but instead of returning home after having visited the desired reconnaissance points, `PatrolMission` keeps circling between all the points, like a patrol guard. The multimodal interface was also changed to recognize this new mission type — a simple modification.

JNI Troubles

Programming JNI is not simple, as I found out. Because part of the code is in C, using JNI means leaving the safe environment of Java where every pointer reference and every array bound is checked. A bug in the C code could have very strange results; sometimes the robot client would stop listening, or show confusing and undeterministic behavior. This usually turned out to be a logic error in the C code, such as forgetting to dereference a pointer, or using a pointer that was no longer valid at that point. Because the code is called by Java it is even harder to debug than a normal C program. However, I eventually got everything running without problems.

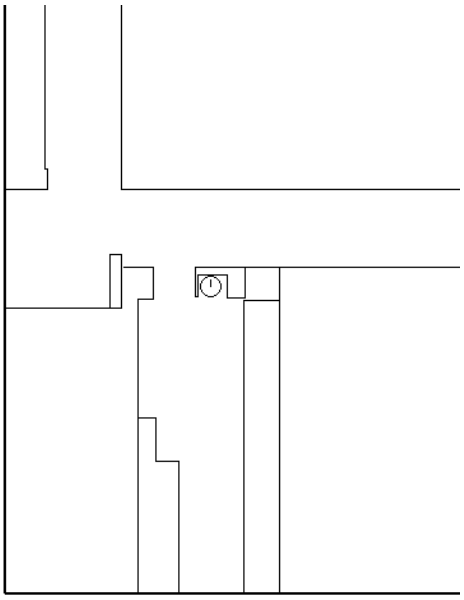

Figure 5.12: The map used by the robot for localization and path planning

5.2.9 Miscellaneous Tasks

Some tasks needed to be done that did not directly attribute to the project goals, but were nevertheless necessary. These are described in this section.

Mapping

The person who worked on this project before I did, set up a map of the floor we work on containing the hallways and his office. The robot uses this map (or "world") for two purposes. Using "Markov Localization", the robot software can more accurately estimate the robot's location. Without a map, and without the help of a GPS or similar device, the robot can only determine the actual distance it travels and rotates through dead reckoning: deriving it from the rotation and diameter of the wheels. Due to slippage of the, wheels, the cumulative error can quickly become very large. The localization algorithm approximates where the robot is, based on its last position and the current sonar and/or laser range finder readings.

The map is also used for path planning. Using the world map, the path planning algorithm sets up a gradient map that tells the robot in which direction to travel from any point on the map to get closer to the desired destination.

Since my office was not on the map, I had to add it. Luckily, all offices at CAIP have more or less the same size, so I could use the measurements of the office already on the map. The localization algorithm is fairly robust, so it works even if the map is not 100% accurate. Obstacle avoidance behavior ensures that the robot will not run into walls or other obstacles when the robot is following a planned path, even if these obstacles are not on the map, or the walls are not in exactly the same place in reality as they are on the map.

Flatscape uses a map for display purposes, but since it is in a different

format, it is maintained separately from the robot's world map, so I needed to modify it as well. For some reason unknown to me, the scale of the map was also off by a factor 10. For this reason, the map needed to be zoomed out by a factor of 10, and the robot's icon then appeared very small. I wrote a small program to scale the map down, and changed the code so that the reported position of the robot was also divided by 10.

DISCIPLE V3

The original robot control software was written for DISCIPLE version 2. When I started on this project, version 3 had been developed. The major change is that XML is now used as the native data representation, instead of the "UForms" that were previously used. UForms were flat structures with a unique ID and a key-value list. XML makes hierarchial data representation much easier since XML was designed to represent structured data. Formerly, UForms would have to been linked based on UForm ID, much like in relational databases, to create hierarchy.

Where 1:n or n:m relations are still needed — for example, a single mission can involve several robots— XML nodes are referenced by XPath, such as `/flatscape/time/overlay[2]/robot`.

Video Transmission

Video is transmitted from the robot to the Flatscape client using RTP (Real-time Transport Protocol [66]). Vic [43] is used to compress and send the RTP stream from the robot. The Java Media Framework [48] is used to receive, decode, and display the video in a Flatscape panel on the client side. The video stream is multicasted, so multiple clients can receive the video stream at once.

Sending and receiving of video was implemented by two previous students at CAIP. However, I found all this in a non-working state. I had to install a driver for the robot's camera frame grabber, so that video could be transmitted. However, this proved to be unstable; the robot's onboard PC would sometimes freeze when Vic was started up. It was hard to find the cause of this since it did not happen consistently, i.e. sometimes it just worked. I finally found out that a new version of the frame grabber driver `bttv` was available. When I installed that, it worked flawlessly.

Chapter 6

Evaluation

In this chapter I will look back at the project goals stated in Chapter 1 and evaluate how they were realized. Reusability and performance measurements are also given.

6.1 Evaluation of Project Approach

The goal was to create a multimodal interface to control a wheeled robotic vehicle, extending the existing GUI interface that was already in place. One goal was to investigate existing dialog systems and infrastructures to find out if existing technology could be used. For this I did an extensive literature study, which provided me with a lot of information on (multimodal) conversational interfaces in general, as well as a view of what systems are available and what the state of the art is. The Communicator architecture appeared to be a good basis to work on, as several dialog systems used it and some components that worked with in Communicator infrastructure were freely available, especially a robust natural language parser, Phoenix.

The literature showed a great lack of generic multimodal dialog systems. Some components, such as Phoenix, were implemented in a reusable manner, but the majority of the systems contain large amounts of code in which application details are hard-coded, so that this code cannot be easily used in other systems. Since I did not want to make this same mistake, I decided to begin by developing a multimodal *framework*, and build the interface for the robot control application on that. This way the design would be modular and extensible, as well as possibly providing a starting point for implementation of future multimodal interfaces.

Based on existing conversational interface models, I set up a model for a multimodal interface, designing a fusion method that works with various modalities and types of data, so that it would be easy to add new commands and modalities to the system.

I then built the conversational interface for the robot control application on CAIP's Flatscape software using this framework. An initial interface was made in only a few days, which shows that once the framework was in place, rapid development of this interface was possible. The initial interface was speech-only. I later extended this so that mouse and gaze could be used for multimodal

interaction, so that instead of having to say "go to (330, 900)" or "go forward for twenty feet", a user could just say "go *there*", and point to the desired location on the screen. Pointing was enabled by use of a touch screen which emulates a mouse. So having implemented the mouse modality, I got pointing for free.

The result is a prototype multimodal interface that can be used to directly control the robot using speech, keyboard, mouse, pointing, and gaze (although the latter has not been thoroughly tested). The robot can be rotated and moved, the onboard camera can be panned, tilted, and told to zoom in and out. Video images can be displayed on screen with a speech command. Also, autonomous movement is possible by creating missions ahead of time for which points on the map are set (again, using any combination of modalities) that the robot will then go to without user intervention.

6.2 Reusability

Table 6.1 shows the implementation effort for the multimodal interface for the robot control application expressed in kilobytes of code. As can be seen in the table, 92% of the code is framework code. This indicates that the framework is a good generalization for this type of application and that much effort is saved by using the framework.

Table 6.1: Implementation effort for the situation map tool

Application-specific code	**Size**
Grammar	28K
New Java code	46K
Fusion and Dialog Manager configuration (XML + Javascript)	33K
Natural language generation configuration	6K
Framework code	**Size**
Framework code	1152K
Parser code	148K
Code re-use	**92%**

6.3 Response Times

Figure 6.1 shows response times for five different types of speech acts, both speech-only and multimodal. Times were measured on a Pentium 4, 1.7 GHz with 512MB of RAM. As can be seen in the figure, speech recognizer delays can be significant; the reason for this is that present-day speech recognizers wait for a certain period of silence before assuming the user has finished speaking. These delays, due to their nature, can not be reduced by acquiring faster hardware.

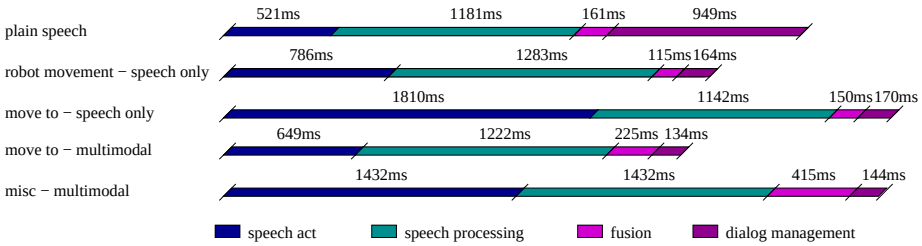

Figure 6.1: Response times for five types of speech acts

6.4 Conclusions

Looking back at the initial design goals from Chapter 1, we can state that most of these have been reached:

- **Use of existing systems:** Communicator is used as a common infrastructure. Phoenix is used as a parser within this infrastructure. IBM ViaVoice is used for speech recognition and synthesis. Usable implementations for other components were not found.
- **Reusability:** The multimodal framework can be used to build other, similar systems. For this particular interface, approximately 92% of the code was framework code. A new interface with similar interaction paradigm could therefore be very quickly implemented.
- **Efficiency:** As the measurements in the previous section show, response times vary from approximately 1500 milliseconds to a little over two seconds. However, most of this is from the speech recognizer waiting for a certain amount of silence after the user has finished speaking, to set the speech boundaries. This delay is inevitable with current speech recognition technology. The other system components have fast response times, as can be witnessed when keyboard input is used instead of speech.
- **Modularity:** The system is very modular. It is component-based with components communicating through the Hub using a well-defined interface. The new components were also written with modularity in mind: it is easy to add new modalities, resolution methods, command types, etc.
- **Interface with previously developed robot control software:** The existing robot code was used as much as possible. Not all the code was usable because a lot of it was not documented, and it wasn't clear where the latest version of the code was located. The current system runs on Flatscape and DISCIPLE, and uses the part of the robot control system I was able to get working, which provides streaming video, direct control, and simple mission planning.

6.5 Future Work

The interface prototype works well, but improvements can be made. Several assumptions were made about fusion and dialog management. One was that

the system was language-centric. The framework could be changed to support the use of other modalities without speech or keyboard input to give a command. A use for this would be to do handwriting or gesture recognition for commands, as in [18].

Fusion is currently done on the slot level. While this works for the type of interaction in this prototype, supporting fusion on a higher, frame level could be better, since there is often not only ambiguity concerning the values of the slots, but also on the frame to instantiate. Currently this is done in a "crisp" manner, but in analogy to the resolving agents, which return probability scores for the possibly slot values, we could provide probabilities for which frame to instantiate, based on the previous frame, the application state, what the user said, modality inputs, etc.

Sentence structure is currently not taken into account for fusion, even though it could provide valuable hints to better assess a user's intent. This would require a syntactic parse as well as a semantic parse. A more accurate anaphor and ellipsis resolution could be done than is currently possible. However, implementing this in an application-independent manner could prove to be difficult.

The voting mechanism used in the fusion manager could be replaced by a more sophisticated approach, if that would prove to give better results. Ideas on using Bayesian networks have been discussed within CAIP, but designs have only been presented for low-level fusion. It is unclear whether and how belief networks could be used for higher level semantic fusion as well.

Dialog management is currently very straightforward. Support for nested dialogs is currently absent. Also, as explained in Section 4.7.1, a very simple dialog history of fully resolved slots is stored, which will not be sufficient for some dialogs. Straightforward dialogs in which the intent is clear from the start work best. More advanced dialog management could be implemented for a more natural interface.

Speech recognition performance quickly deteriorates in the presence of noise. Fusion is currently not really used to solve this, partly due to the fact the the interface is speech-centric. If a speech act is misrecognized, fusion is useless, as there is nothing to resolve in that case. Speech recognition could be improved by combining it with lip-reading. Data from other modalities could be used more actively to pick a suitable alternative from the N-best list (if all alternatives had time stamps, which they do not at this time).

The current interface does not have multimodal output, because there is not fission component yet. Implementing this would make a doubly multimodal system possible where both system input and output are multimodal. This would create an even more natural interaction between user and computer, in which feedback is sent to the user as synchronized streams of spoken text, other audio, visual feedback, force feedback, and possibly other modalities, such as 3D output over an augmented reality display. Fission is slowly starting to become a topic of interest in the multimodal research community, so we can expect systems using multimodal output to appear in the future, that our system could possibly borrow from.

Improvements made in the field of speech recognition, fusion, dialog management, fission, and conversational systems in general may be used in the future to replace parts of the system with sophisticated implementations. Of course, this implies that these implementations are not just internal prototypes, but that they will be available publicly.

Appendix A

Publication for the Fifth International Conference on Multimodal Interfaces (ICMI-PUI'03)

The following paper was submitted to and accepted for oral presentation at ICMI-PUI'03.

A Framework for Rapid Development of Multimodal Interfaces

Frans Flippo * †
fflippo@caip.rutgers.edu

Allen Krebs *
krebs@caip.rutgers.edu

Ivan Marsic *
marsic@caip.rutgers.edu

* Rutgers University
CAIP Center
Piscataway, NJ 08854-8088
+1 732 445 0542

† Delft University of Technology
Dept. Information Technology and Systems
2628 CD Delft, The Netherlands
+31 15 278 7504

ABSTRACT

Despite the availability of multimodal devices, there are very few commercial multimodal applications available. One reason for this may be the lack of a framework to support development of multimodal applications in reasonable time and with limited resources. This paper describes a multimodal framework enabling rapid development of applications using a variety of modalities and methods for ambiguity resolution, featuring a novel approach to multimodal fusion. An example application is studied that was created using the framework.

Categories and Subject Descriptors

H.5.2 [**Information Interfaces and Presentation**]: User InterfacesInteraction styles, Natural language, Voice I/O; H.5.1 [**Information Interfaces and Presentation**]: Multimedia Information Systems

General Terms

Algorithms, Design, Human Factors

Keywords

Multimodal interfaces, multimodal fusion, application frameworks, command and control, direct manipulation

1. INTRODUCTION

Multimodal interfaces provide a very natural way for humans to perform tasks on a computer, using direct manipulation and speech: interaction methods that are used daily in human-to-human communication. However, despite the availability of high-accuracy speech recognizers and the maturing of multimodal devices such as gaze trackers, touch screens, and gesture trackers, very little applications take advantage of these technologies. One reason for this may be that the cost in time of implementing a multimodal interface is prohibitive. One desiring to equip an application with such an interface must start from scratch, implementing access to external sensors, developing ambiguity resolution algorithms, and making calls to the application's API based on the determined user intention.

However, when properly implemented, a large part of the code in a multimodal system can be reused. This aspect was identified and used to implement a multimodal application framework. The framework uses a novel and parallelisable application-independent fusion technique that can be easily augmented to support application-specific demands as well as new modalities. The fusion algorithm separates the three parts of fusion: obtaining data from modalities, fusing that data to come to an unambiguous meaning, and calling application code to take an action based on that meaning. Separation of these three tasks makes the framework applicable to a wide range of applications and modalities.

1.1 Requirements and Constraints

The framework enables existing applications to be equipped with a multimodal interface. Therefore the design is to be minimally intrusive on existing application code, but rather function alongside it, calling application code when data is needed from or actions need to be performed in the application; additionally, the interface accepts callbacks from the application when changes occur that change the discourse context or that need to be reported to the user.

To gain user acceptance for a multimodal system, response times must be reasonably small. If the system takes too long to process a user's spoken command, the user will think the command was not understood and repeat it, resulting in confusion when the command then gets carried out twice. All this results in annoyance and should be avoided at all cost. Ideally, response times should be below a second.

Many multimodal systems are geocentric in nature [6]. Combining speech with gesture, gaze, and mouse serves to link spoken references to spatial data (i.e. objects and locations on-screen) with their antecedents as derived from the aforementioned modalities. Therefore our framework is optimized for this type of fusion.

Direct manipulation is currently the most popular mode of interaction. It appeared in the late 1970's [9]. Its software design was described in [12]. It is a proven form of interaction that should not be replaced, but rather be com-

ICMI'03, November 5–7, 2003, Vancouver, British Columbia, Canada.

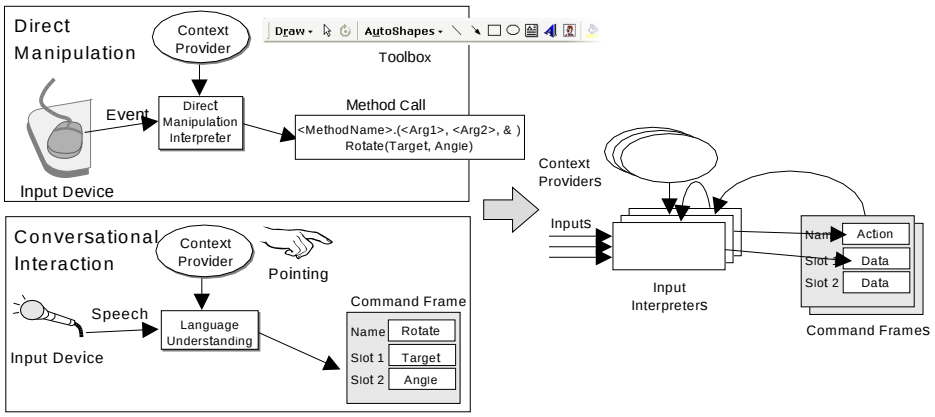

Figure 1: A high-level view of software architectures for direct manipulation and conversational interaction. In direct manipulation, the toolbox (in this figure we show a part of a PowerPoint toolbox) determines the context of the manipulation and the end result is a method call on the application. Similarly, in conversational interaction, pointing helps to resolve ambiguities in speech. The architecture on the right hand side shows a generalisation of these two paradigms.

plemented by conversational interfaces. The key components of the software architecture for direct manipulation are summarized in Figure 1. Primary inputs are mouse clicking and dragging and command selection through menus or toolboxes. On the other hand, recent multimodal interfaces focus on conversational interaction (also summarized in Figure 1). Unfortunately, almost none of these interface include direct manipulation in conversational interaction. Pointing is used only to resolve deictic references in speech. Our objective is to provide an architecture which will support both. This architecture is summarized in Figure 1 on the right hand side. Creating an architecture which supports direct manipulation and conversational interaction *in parallel* combines the strengths of both interaction styles while compensating for the weaknesses. By maintaining direct manipulation as a choice of interaction, we are not limiting user actions to those that are speech-centric. The user can choose between traditional direct manipulation, speech, or a combination of both.

Our focus is on single-user multimodal interfaces. Multi-user interfaces have many more problems, such as having to recognize multiple voices, determine the source of gestures, etc. Our system *does* allow collaboration where users are on different machines, and the application described at the end of the paper is an example of this.

We will first discuss some related work. In Section 2 we describe the conceptual design of the framework, followed by its implementation in Section 3. We then look at the multimodal interface that was made for Flatscape, our collaborative situation map tool using the framework, followed by conclusions and ideas for future work.

1.2 Related Work

The first known multimodal interface was built in 1980 by R. Bolt [1]. It provided an interface in which shapes could be created, moved, copied, removed, and named using a combination of speech and pointing, for example "put that to the left of the green triangle," "copy that there," "call that the calendar," "move the calendar here". Fusion was done at the parse level. Every time an anaphor or deictic reference was recognized, the system would immediately see where the user was pointing and resolve the reference. The system also had an ability to learn new words. When the user said "call that ⟨*name*⟩," the system would tell the speech recognizer to switch from recognition mode to training mode so that the name that the user gave the object would be learned. The components of the system necessarily had to be tightly integrated because of the way the system was designed. While performing fusion during speech yields a straightforward implementation of fusion, gestures and speech are in general not synchronized, that is, gesture precedes or follows a spoken reference, and assuming that they are demands the user to change his normal behavior to use the system: the system trains the user. This is not desirable.

Krahnstoever [6] describes a multimodal framework targeted specifically at fusing speech and gesture with output being done on large screen displays. Several applications are described that have been implemented using this framework. The fusion process is not described in great detail, but appears to be optimized for and limited to integration of speech and gesture, using inputs from cameras that track a user's head and hands.

The W3C has set up a multimodal framework specifically for the web [7]. This does not appear to be something that has actually been implemented by the W3C. Rather, it proposes a set of properties and standards — specifically the Extensible Multimodal Annotation Markup Language (EMMA) — that a multimodal architecture should adhere to.

The QuickSet system [4], built by the Oregon Graduate Institue, integrates pen with speech to create a multimodal system. QuickSet goes beyond simple point-and-speak commands and recognizes more complex pen gestures, like arrows and lines to better support direct manipulation.

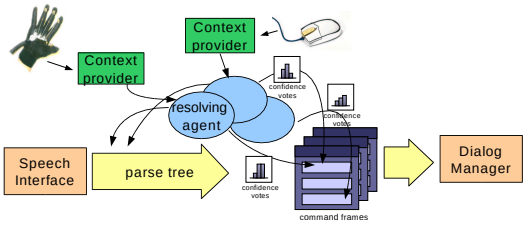

Figure 2: The fusion process – from parse tree to frames

This allows for richer and more natural multimodal interaction. The system employs a Members-Teams-Committee technique very similar to the fusion technique described in this paper, using parallel agents to estimate a posteriori probabilities for various possible recognition results, and weighing them to come to a decision. However, our approach is more reusable as it separates the data — or feature — acquisition from the recognition. Also, it supports a variety of simultaneous modalities whereas QuickSet seems to be built solely for pen and speech-based interaction.

2. FRAMEWORK DESIGN

2.1 An Object-Oriented Framework

The fusion system described in this paper has been implemented as a framework. A framework is a reusable, "semi-complete" application that can be specialized to produce custom applications [5]. Developing a framework involves determining what functionality is common to the applications in the target domain and abstracting away from application-specific functionality. This common functionality is the framework's immutable *core*, while *"hot spots"* in the framework are places in which a developer must "plug in" code to come to a specific, working application [8]. An important aspect of an application framework is the inversion of control, or "old code calls new code". In an application created with a framework, the framework core (old code) makes calls to the code that is plugged in to the hot spots (new code). It is this inversion that makes a framework an attractive paradigm for application development: the developer does not need to have knowledge of the framework's internals, but only needs to implement the interfaces that define the hot spots.

Configuration of the implemented framework is largely declarative: the user specifies (declares) structure, not procedure. The framework uses the user's specification to perform fusion, but the user needs to have no knowledge of *how* the framework does this. The user provides the "what" knowledge, while the framework contains the "how" knowledge. This makes the framework an ideal tool for non-experts. It also allows the framework to be changed without affecting existing applications, since the same declarations will still apply for another framework implementation.

2.2 Fusion

Our framework features a new approach to fusion that is reusable across applications and modalities. The process is depicted in Figure 2. The input to the fusion process is a semantic parse tree with time stamps as generated by the natural language parser component of the speech interface. This parse tree needs to be transformed into frames that the dialog manager can use to make calls to the application. To accomplish this, the natural language concepts in the parse tree need to be mapped to application concepts. In addition, ambiguity needs to be resolved. Ambiguity exists when the user uses pronouns or deictic references, for example "remove *that*", or "do reconnaissance *here*". Another case of ambiguity is ellipsis, a linguistic construct in which words that are implied by context are omitted, such as "rotate this clockwise ... *and this too*". The last phrase can be expanded to "and rotate this clockwise, too".

Resolving agents operate on the parse tree to realize the aforementioned mapping of concepts and resolution of ambiguity. The implementation details of resolving agents are not specified by the framework. All that is expected is that the agents take a fragment from the parse tree, perform some transformation on it, and use it to fill a slot in the semantic frame that is sent to the dialog manager. The agents can use data from a modality (through an access object we call "context provider") to give them a context in which to perform their task. Context providers can provide data from an external sensor, such as a gaze tracker, but also from more abstract data sources such as dialog history or application state (e.g. which toolbox button is selected). An agent performing pronoun resolution might have access to gaze or gesture input to resolve a pronoun to an object on the screen that the user pointed or looked at. Any agent will typically have access just one such input. This keeps the design of the agents simple, as they do not need to be concerned with combining data from multiple sources. This combining is done by the fusion manager. It *is* possible for resolving agents to share the same modality, and the framework is designed so that this is possible, even when a device has an exclusive use policy.

The agents themselves do not actually perform fusion. Their task is to perform an assessment of what they think the contents of a slot in the frame should be. Each agent will provide zero or more possible solutions with corresponding probability scores. The whole of the solutions provided by all agents will finally determine what the slot will contain. This is shown in Figure 3.

To make resolving agents reusable, the resolution process is separated from the acquisition of data from modalities. The resolution process is implemented in the resolving agents, while the acquisition of data is the responsibility of the context providers. Resolving agents merely specify the type of data they expect to receive from their context provider. In this way, an agent that requires (x, y)-data points to do its work can accept data from any context provider that provides (x, y)-data, such as a mouse, a gaze tracker, or a haptic glove. In a system with a mouse and a gaze tracker, for instance, two copies of the same pronoun resolution agent might be active, one using data from the mouse, and another using data from the gaze tracker. Each will give its resolutions along with corresponding probability scores, based on the data they have access to.

Thus, resolving agents operate *locally* with only the information they have access to, namely the fragment of the parse tree they use and the data they receive from their modality, if any. However, all agents together create a *global* result that takes into account all of the parse tree and all of the available modalities. Because each resolving agent works

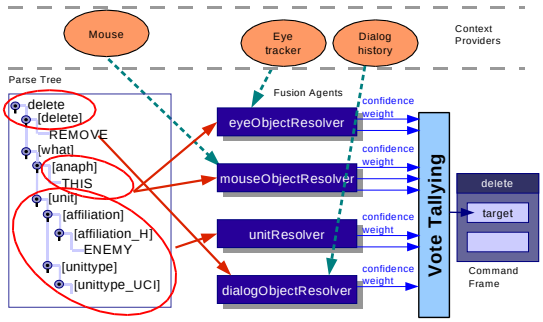

Figure 3: Combining results from different resolving agents (resolvers) to fill a slot

independently of the others, the agents can work in parallel, taking advantage of multiprocessor hardware to increase performance.

Context providers provide timestamps along with their data. These can be used by the resolvers so select data that are applicable to the parse tree fragment they are handling, using the timestamps that the natural language parser provides. For instance, the pronoun resolver agent mentioned before will look at data points that were generated around the time that the pronoun was spoken. Timestamps for speech data and context data ensure that the modality streams are properly synchronized.

2.2.1 Fusion Manager

The fusion manager controls the fusion process. It is responsible for:

1. choosing a frame for the parse tree and creating it
2. spawning resolving agents and passing them parse tree fragments to work with
3. taking the possible values for each slot from the resolving agents and choosing one, based on the probability scores provided and the weight assigned to each resolving agent.
4. merging frames from the conversational interface with method calls from the application's GUI, resolving ambiguities to create a frame with unambiguous meaning
5. handing a completed frame to the dialog manager

Direct manipulation and conversational interaction run in parallel and may influence each other. Commands uttered by the user can change the interpretation of mouse events. Conversely, the state of an application's GUI elements, such as a toolbox, can influence the meaning of spoken commands or gestures. The fusion manager implements the merging of direct management and conversational input.

If direct manipulation is done by itself, it must be done with the mouse or a device emulating a mouse. Currently, multimodal actions and actions using other devices than the mouse need speech to drive them. Having speech as the 'primary modality' in this way avoids delays imposed by systems that allow other modalities to be used by themselves and need time thresholds to define the end of a dialog action, as in [11].

The fusion manager is configured through an XML file and will be described in more detail in Section 3.

2.3 Dialog Management

The dialog manager receives frames from the fusion manager. First it updates the dialog history with the contents of the frame's slots. Then it calls an action script that is defined for the frame. This script will ultimately make a call to the application to perform a certain task, but first it will typically check whether the frame is 'complete', that is, whether all the slots that are required to be filled are indeed filled. If not, it can send feedback to the user to request him or her to provide the missing data. In the future we intend to implement this as part of the dialog manager, so that specifying which slots are required is all that is needed to have the dialog manager check for missing slots and report missing data to the user. However, slot requirements can be conditional, so implementation is not straightforward. The current solution is flexible, at the cost of some extra implementation effort.

As the rest of the framework, the dialog manager contains no application-specific code. All application-specific data is in the configuration file, which the dialog manager shares with the fusion manager.

2.4 Fission

Fission [2] is currently unimplemented in the framework. However, conceptually it is the inverse of fusion and it will be implemented as such. Given a semantic frame, the fission manager will distribute its slots to different fission agents, which create output to send to a modality as well as placing corresponding text in the semantic tree that is sent to the natural language generator. Where resolving agents *resolve* ambiguity by using multimodal input, fission agents *create* it for a single multimodal output. For instance, an anaphor generator adds a pronoun to the parse tree while creating a command for its modality to point to or hilight the object being referred to on the screen. The various fission agents will be responsible for sending output at the correct time, with the fission manager driving the fission agents.

```
<frame name="delete" test="delete/delete" uses="delete">
 <slot name="glyph">
  <source select="delete/what/anaph">
   <resolve resolver="mouseObjectResolver" weight="0.5" />
   <resolve resolver="eyeObjectResolver" weight="0.4" />
  </source>
  <source select="delete/what/unit">
   <resolve resolver="unitResolver" weight="0.3" />
  </source>
  <source select="delete">
   <resolve resolver="dialogObjectResolver" weight="0.1" />
  </source>
 </slot>
 <action language="javascript">
  if (frame.glyph) {
   application.api.deleteGlyph(frame.glyph.glyph);
  }
 </action>
</frame>
```

Figure 4: A sample frame declaration

3. IMPLEMENTATION

The framework is implemented in Java. Java was chosen due to its strong typing, extensive class library, dynamic object instantiation and object reflection capabilities, and the fact that Java applications can run on different platforms without recompiling. The first two aspects result in quicker development time and less errors, the third gives the framework much of its power, while the last allows applications created with the framework to be deployed on any Java 1.4 capable platform.

3.1 Fusion Manager

It is the fusion manager's task to take the possible values for a slot from the resolving agents and choose the one that is optimal, based on the resolving agents' probabilities, and a confidence value or weight for each agent. Currently a very simple voting algorithm is employed, in which the scores for each value are summed and the one with the highest total sum is selected. However, the framework can accomodate any algorithm desired, and we are looking into using fuzzy reasoning and/or Bayesian networks for this purpose, using models trained on empirical data, to obtain better predictions of the user's intent.

The fusion manager, resolving agents, and context providers are configured through an XML file containing declarations for frames, resolvers and context providers. An example frame declaration is shown in Figure 4. The declaration specifies the name of the frame, an XPath test on the parse tree that must succeed for the frame to be used, and a set of slots with XPath expressions for each slot specifying their data source and a list of one or more resolvers (resolving agents) for each source.

The fusion manager uses the XPath test to determine which frame to instantiate. If multiple XPath expressions evaluate to 'true', the fusion manager prefers the frame that was used in the previous utterance, if possible. This ensures that multiple-step dialogs continue as intended. If no frame is of the same type as the previous one and there are multiple frames to choose from, feedback can be generated asking the user to be more specific. The actual implementation of this is left up to the developer, so any message can be generated, but output on the screen is also possible, for instance.

3.1.1 Resolving Contradictory Inputs

Contradiction between modalities can arise. For example, a user may say "move that infantry squad" while pointing to or looking at an infantry army. On the slot level, speech is treated like any modality, so the results from resolving "infantry squad" — that is, a list of all infantry squads — will participate in the voting process along with the result of resolving "that" using a pointing modality and possibly the dialog history, which will also result in a list of objects — those in the vincinity of the location the user was pointing. Ultimately, the developer decides which modality 'wins' in this case by the weights that he or she allocates to each modality. By adjusting the weights appropriately, the developer can achieve that the pointing modality wins if the object being pointed at is under the mouse cursor, but if the pointer is a certain threshold away from any object, the speech modality will win instead. Since the scores from each resolving agent are summed, the fusion manager may choose an object that is somewhat close to the pointer and is (in our example) an infantry squad over an object directly under the pointer.

A better approach in case of unresolvable ambiguity could be, again, to ask the user for clarification, e.g. "That is an infantry army, not an infantry squad. What do you want to move?".

3.2 Fusion Interfaces

Four interfaces are crucial to the fusion process:

- **Resolver** – All resolving agents implement the `Resolver` interface. A resolving agent is initialized by the fusion manager, which passes the initalization parameters as read from its configuration file to the `init()` method.

  `Resolver`s will usually be created by subclassing the `AbstractResolver` class, which implements the `Resolver` interface and inherits from `AbstractSessionObject`. The latter class provides common functionality for objects that operate within a multimodal dialog session and can be dynamically instantiated. The key method of the `Resolver` interface is the `resolve()` method, which takes a DOM `Element` representing a parse tree fragment and returns an `Iterator` containing the agents' resolutions along with their assessed probabilities.

  The following standard resolvers are provided:

  - `AnaphorResolver` — Resolves a pronoun to the on-screen object it refers to as determined from the input modality. The probabilities that accompany the solutions are proportional to the object's distance from the (x, y)-data point that is selected. This resolver does not actually look at the data in the parse tree fragment, it just uses the time stamps.
  - `DeicticResolver` — Resolves a deictic reference to its location on the screen as determined from the input modality. The probabilites that accompany the solutions are determined by the frequency of the point in the (x, y) data list. This resolver does not actually look at the data in the parse tree fragment, it just uses the time stamps.

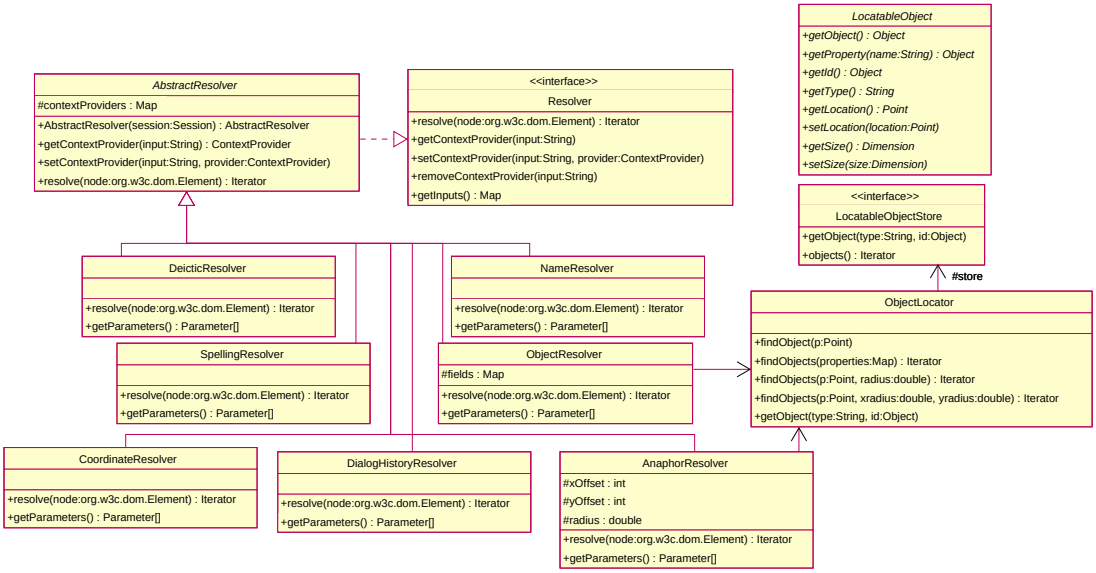

Figure 5: The `Resolver` class hierarchy and related classes

– `ObjectResolver` — Resolves a set of object attributes to the objects that match those attributes. For example, when the user is to be able to select an object using speech only with an expression such as "remove the hostile infantry army", this resolver is used to return all objects matching the attributes named: "hostile", "infantry", and "army" in the example.

– `SpellingResolver` — Takes a parse tree fragment containing nodes that represent spelled letters and returns the word they spell out. Spelling can be very useful when a parser or speech recognizer is used that requires an a priori vocabulary or grammar and cannot learn new words while running. In this way, out of vocabulary terms, such as names, can be spelled out. Spelling using a specialized alphabet, such as the NATO alphabet, is also very accurate. This resolver returns a single `String`, with a probability of 1.

– `DialogHistoryResolver` – Returns the first item in the dialog history for the slot. This is the most recently used value for the slot. Slots are identified by name, so this also looks for slots with the same name used in other frames.

– `CoordinateResolver` – Resolves a coordinate specification, such as "five hundred comma two fifty" to a Java `Point` object. One `Point` is returned, with a probability of 1.

– `NameResolver` – A trivial resolver that simply returns the name of the parse tree fragment's top level node. Some simple transformations can be done on the returned name, including stripping fixed leading and trailing strings, and applying a translation.

- **ContextProvider** – Classes that provide access to modalities implement the `ContextProvider` interface. The key method is `getData()`, which returns a `ContextData` instance that provides the modality's data along with a timestamp for which the data is valid. The context providers supplied by the framework are currently `EyeTracker`, `Mouse`, and `DialogHistory`.

- **ContextData** – `ContextData` implementations represent data from a modality. Three implementations are supplied with the framework: `PositionContextData`, `Entity`, and `ContextDataList`. `ContextDataList` is a container for other `ContextData` objects, and is returned by the `BufferedContextProvider`, which polls another context provider and caches its data for the duration of an utterance, so that multiple resolvers can use the same data through a single `BufferedContextProvider` instance. Most resolvers expect a `ContextDataList` as input.

- **LocatableObjectStore** – As shown in Figure 5, an `LocatableObjectStore` implementation plugs unto the framework's `ObjectLocator` class to enable it to query the application for on-screen objects. This is used in the `AnaphorResolver` to find objects in the neighborhood of the point on the screen the user indicated when speaking a pronoun. It is used by the `ObjectResolver` to find objects matching a set of attributes. The elements of the `LocatableObjectStore` are instances of a concrete subclass of `LocatableObject`. `LocatableObject` uses the "wrapper" design pattern to provide a uniform interface to access an object's location, size, and attributes.

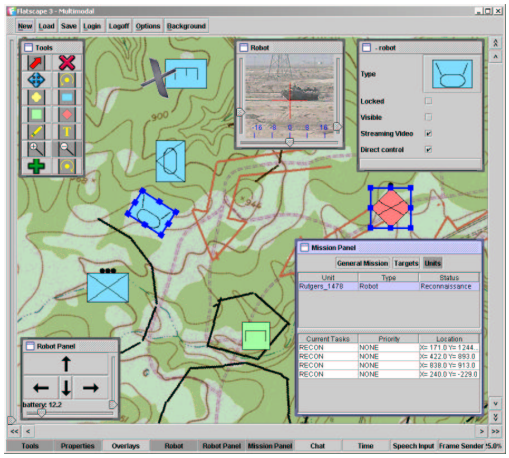

Figure 6: Flatscape

3.3 Dialog Management

Dialog management consists of maintaining dialog context and making application calls when frames are completed. The first task is implemented by having the dialog manager maintain a `DialogHistory` object, which implements the `ContextProvider` interface, so that it can provide data to resolving agents, notably the `DialogHistoryResolver`.

Application calls are made by the dialog manager through JavaScript. When the dialog manager receives a frame, it retrieves the associated script and executes it through the Bean Scripting Framework [10], a framework that provides scripting language support to Java applications. JavaScript provides a flexible way to interface between the dialog manager and the application. Since JavaScript is interpreted code, all that is needed to change the behavior of the multimodal interface is to modify the script and re-run the application; no code needs to be compiled.

4. CASE STUDY

4.1 Flatscape

Using the framework, a multimodal interface was created for Flatscape, our collaborative situation map application. With this application we can plan military missions by placing and moving icons representing military units on a map overlay. Additionally, we can track moving robotic vehicles on the same map and give them direct control commands or assign them higher level "missions". Camera feedback from the robots is available and can be viewed on screen. The camera can be rotated horizontally (panned) and vertically (tilted) and has a zoom function. Camera images are also used by the robot for target recognition.

The system runs on top of our DISCIPLE [3] collaborative middleware. Both Flatscape and the robots function as clients in this infrastructure, communicating over the collaboration bus (cBus) to exchange and synchronize state information. A change made by the user in Flatscape that is relevant to a robot will cause this robot to be notified of it, possibly making it perform some action, as the user indicated, such as moving to a different location.

The new multimodal interface allows the above things to be accomplished with multimodal commands:

- **Unit creation and manipulation**
  "create a friendly infantry squad *here*"
  "move *this* anti armor unit over *there*"
  "delete *that*"
  "move this to five fifty comma two hundred"

- **Robot control**
  "go forward fifty feet"
  "turn left"
  "back up slowly ... stop"
  "camera on ... look left ... zoom in"

- **Mission planning**
  "new recon mission"
  "home base is *here*"
  "new reconnaissance task *there*"
  "start mission"

Additionally, we are able to use the direct manipulation commands already present in the GUI alongside the multimodal commands. For each task, users can select whichever interaction paradigm is most suitable and use both interaction methods interchangably.

4.2 Evaluation

4.2.1 *Reusability*

Table 1 shows the implementation effort for this new multimodal interface expressed in kilobytes of code. As can be seen in the table, 92% of the code is framework code. This indicates that the framework is a good generalization for this type of application and that much effort is saved by using the framework.

Table 1: Implementation effort for the situation map tool

Application-specific code	**Size**
Grammar	28K
New Java code	46K
Fusion and Dialog Manager configuration (XML + Javascript)	33K
Natural language generation configuration	6K
Framework code	**Size**
Framework code	1152K
Parser code	148K
Code re-use	**92%**

4.2.2 *Response Times*

Figure 7 shows response times for five different types of speech acts, both speech-only and multimodal. Times were measured on a Pentium 4, 1.7 GHz with 512MB of RAM. As can be seen in the figure, speech recognizer delays can be significant; the reason for this is that present-day speech recognizers wait for a certain period of silence before assuming the user has finished speaking. These delays, due to their nature, can not be reduced by acquiring faster hardware.

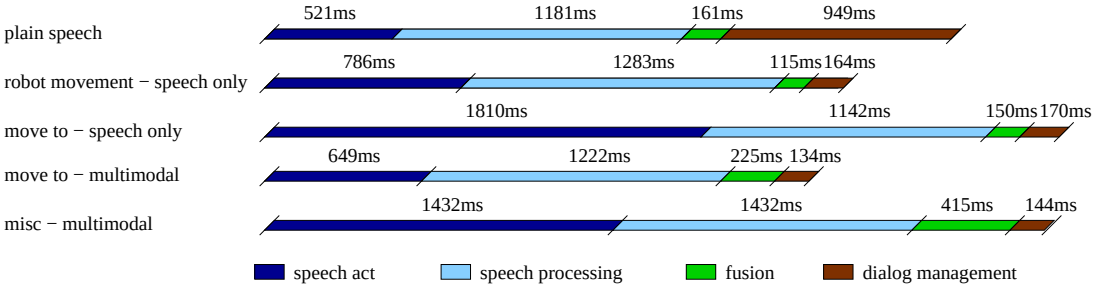

Figure 7: Response times for five types of speech acts

5. CONCLUSIONS AND FUTURE WORK

We have showed that the framework can be used to implement a multimodal interface with relatively little effort. The resulting interface has a reasonable response time, which can be improved by advances in speech recognition technology. The system currently uses mouse and gaze input, but other modalities can be added easily by creating new implementations of the `ContextProvider` interface. Direct manipulation and speech are used together to form a user interface that gives users the freedom to choose and combine interaction methods to create a more efficient and pleasant way of working.

One of the main thrusts for future work is implementing more complex direct manipulation examples and integrating them into the framework. The architecture presented in [12] gives a generic framework but there are also parts that are application-specific and will need to be implemented by the application developer.

We are investigating the use of fuzzy reasoning and Bayesian networks in fusion. Fuzzy values can be a better representation of uncertainty in multimodal systems than discrete probabilities and may enable a more natural way of configuring the fusion manager and resolving agents.

Bayesian belief networks trained with data captured from multimodal dialogs can be used to improve fusion. Statistical data on the way users use various modalities can be used to estimate a user's intent with a multimodal action.

Finally, adding a fission agent as proposed in the text would make a true multimodal system in which both user and system can communicate in a multimodal fashion, creating a more natural experience for the user.

6. ACKNOWLEDGMENTS

The research is supported by US Army CECOM Contract No. DAAB07-02-C-P301, a grant from New Jersey Commission on Science and Technology, and by the Center for Advanced Information Processing (CAIP) and its corporate affiliates.

7. REFERENCES

[1] R. A. Bolt. "Put-that-there": Voice and gesture at the graphics interface. *Computer Graphics (SIGGRAPH '80 Proceedings)*, 14(3):262–270, July 1980.

[2] L. Boves and E. den Os. Multimodal multilingual information services for small mobile terminals (MUST). Technical report, Eurescom, 2002. `http://www.eurescom.de/~pub/deliverables/documents/P1100-series/P1104/p1104-d1.pdf`.

[3] CAIP. DISCIPLE - mobile computing and collaboration. `http://www.caip.rutgers.edu/disciple`.

[4] P. Cohen, M. Johnston, D. McGee, S. Oviatt, J. Pittman, I. Smith, L. Chen, and J. Clow. Quickset: Multimodal interaction for distributed applications. *ACM International Multimedia Conference, New York: ACM*, pages 31–40, 1997.

[5] M. Fayad and D. C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

[6] N. Krahnstoever, S. Kettebekov, M. Yeasin, and R. Sharma. A real-time framework for natural multimodal interaction with large screen displays. In *Proc. of Fourth Intl. Conference on Multimodal Interfaces (ICMI 2002), Pittsburgh, PA, USA*, October 2002.

[7] J. A. Larson and T. V. Raman. W3C multimodal interaction framework. `http://www.w3.org/TR/mmi-framework`, 2 December 2002. W3C Note.

[8] M. E. Markiewicz and C. J. Lucena. Object oriented framework development. *ACM Crossroads*, 2001.

[9] B. Shneiderman. The future of interactive systems and the emergence of direct manipulation. *Behaviour and Information Technology*, 1(3):237–256, 1982.

[10] The Apache Jakarta Project. Jakarta BSF – bean scripting framework. `http://jakarta.apache.org/bsf`.

[11] D. Toledano, S. Wang, S. Cyphers, and J. Glass. Extending the galaxy communicator architecture for multimodal interaction research. submitted to ACM Trans. on Human-Computer Interaction, Aug 2002.

[12] J. Vlissides and M. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268, 1990.

Appendix B

Glossary

Various terms and acronyms used in this document are briefly explained here.

anaphor A word, such as a pronoun, used to avoid repetition. The referent of an anaphor is determined by its antecedent [35]. Anaphora include pronouns, such as he, she, it, they; the word "one", as in "the blue one"; and descriptive phrases, for example, "I bought a red Cadillac yesterday, and showed *the car* to my friends.

BNF Backus Naur Form / Backus Normal Form: a formal metasyntax used to express context-free grammars. Developed in the early 1960's to describe the ALGOL syntax.

context-free grammar A context-free grammar is a formal grammar in which every production rule is of the form $V \rightarrow w$ where V is a nonterminal symbol and w is a string consisting of terminals and/or nonterminals. The term "context-free" comes from the feature that the variable V can always be replaced by w, no matter in what context it occurs. [62].

ellipsis Omission or suppression of parts of words or sentences. For example, "I like olives, but my friend doesn't *[like olives]*. The elided (omitted) text here is placed in square brackets.

framework A software architecture that is to support the rapid development of applications. Although in itself a framework is not a runnable application, it enables rapid application developent by including commonly used functions and tasks into libraries that developers of applications can use. Frameworks are typically targeted towards a certain type of application. An examples of a framework is Apache's Cocoon[4] for server application development.

Java VM Java Virtual Machine. This is the simulated (hence "virtual") machine on which Java programs run. The byte code that the Java compiler generates is the machine language of this machine. A VM running on a personal computer translates the byte code commands into commands that can be executed on the local platform. No physical Java machine exists, so Java programs will always run in an emulated environment. This is the reason for portability of Java programs and relative security, but it also causes Java programs to run slower than their native equivalents.

multimodal Indicates the use of different modalities, modes, or channels of communication. Human communication is inherently multimodal, since humans (often unconciously) use speech, looking, pointing, and face expressions together when communicating. To create a natural interface to a computer, the machine must be programmed to interpret these modalities.

VM Virtual Machine. See Java VM for a description of the Java VM.

WIMP Acronym for Windows, Icons, Mouse, Pointer. Used to refer to the standard graphical user interface developed over twenty years ago by Xerox and Apple and found today on most personal computers.

Appendix C

Grammar

The complete Phoenix grammar is shown below. Note that this is still a prototype, therefore some parts of the grammar are not used, but rather included for future purposes. Other parts could be improved by performing Wizard of Oz experiments on the target users to determine the way in which they normally articulate their commands.

C.1 Frames

```
# Eye tracker calibration and adjustment
FRAME: eyetracker
NETS:
        [eye_reset]
        [eye_calibrate]
        [eye_next_point]
        [eye_output]
        [eye_adjust]
        [eye_camera]
;

# Connect to the robot by creating
# a new overlay that the Robot code
# will attach its uForm to
FRAME: connect
NETS:
        [new_document]
        [connect]
        [map_name]
;

# Log in to or out of the DISCIPLE server
FRAME: login
NETS:
        [login]
        [logout]
;

FRAME: control
```

```
NETS:
        [direct_control]
        [autonomous]
;

FRAME: rotate
NETS:
    # "rotate", "turn"
    [rotate]

    # "left", "right", "clockwise", "counterclockwise"
    [rotate_direction]

    # <number> degrees
    [degrees]

    # rotation speed
    [speed]

        # currently unused
        [when]
;

FRAME: stop
NETS:
        [stop]
;;

FRAME: move
NETS:
        # possible 'move' or 'put' word
        # this is to avoid having a possible 'move' parsed as an action
        # in the task frame
        [move]

        # These three slots are actually mutually exclusive. The dialog
        # manager is responsible for removing ambiguities should more
        # than one slot be filled.

        # e.g. move there
        [what]

        # e.g. move there
        [where]

        # e.g. north, south, east, west
        [direction]

        # e.g. move north 100 miles
        [distance]

        # e.g. move forward at 20 millimeters per second
        [speed]

```

```
84          # An excellent place for language acquisition would be the naming of certain
85          # areas on the map. We would need a catch-all here that would trigger
86          # a question for the user: "Where is <new phrase>" or the like
87          # Requirements: (*) large vocabulary (check: ViaVoice has that)
88          #               (*) Adaptive parser  (use Sorin's work, or adapt Phoenix)
89
90          # Go when?
91          [when]
92  ;
93
94  FRAME: task
95  NETS:
96          # destroy, locate, etc.
97          [action]
98
99      # Usually we should have either [unit] or [anaph], not both, e.g.
100         # "kill it the enemy" is not legal english
101         # However, "kill this enemy" is allowed. In that case the postparser
102         # has information it needs to fuse from another modality, such as
103         # gaze or pointing. Enemy can than be used to clear up any ambiguity,
104         # e.g. if there are both enemy and neutral units where the user is
105         # pointing, we know which one to pick
106         [what]
107
108         [when]
109 ;
110
111 FRAME: camera_pan
112 NETS:
113         # "pan", "move" etc.
114         [camera_pan]
115
116         # left, right, middle
117         [pan_direction]
118
119         # pan: how much?
120         [amount]
121
122         # relative? (e.g. (more) to the left)
123         [relative]
124
125 ;
126
127 FRAME: camera_tilt
128 NETS:
129         # "tilt", "look", etc.
130         [camera_tilt]
131
132         # up or down
133         [tilt_direction]
134
135         # tilt: how much?
136         [amount]
137 ;
```

```
138
139 FRAME: camera_zoom
140 NETS:
141         # "zoom" and co.
142         [camera_zoom]
143
144         # for zoom: in, out
145         [zoom_inout]
146
147         # zoom: how much?
148         [amount]
149 ;
150
151 FRAME: camera_visual
152 NETS:
153         [camera_on]
154         [camera_off]
155 ;
156
157 FRAME: quit
158 NETS:
159         [quit]
160 ;
161
162 FRAME: identify
163 NETS:
164         [identify]
165         [anaph]
166 ;
167
168 FRAME: create
169 NETS:
170         [create]
171         [unit]
172         [where]
173 ;
174
175 FRAME: delete
176 NETS:
177         [delete]
178         [what]
179 ;
180
181 FRAME: mission
182 NETS:
183         [mission_name]
184         [mission_type]
185         [mission_homebase]
186         [mission_resupply]
187 ;
188
189 FRAME: mission_task
190 NETS:
191         [mission_task]
```

```
192         [what]
193         [task_type]
194         [where]
195 ;
196
197 FRAME: mission_control
198 NETS:
199         [mission_start]
200         [mission_stop]
201         [mission_pause]
202         [mission_resume]
203         [mission]
204 ;
205
206 FRAME: name
207 NETS:
208         [call]
209         [what]
210         [where]
211         [name]
212 ;
213
214 FRAME: ellipsis
215 NETS:
216         [unit]
217         [what]
218         [where]
219 ;
220
```

C.2 BNF Grammars

```
1  [letter]
2          ([letter_a])
3          ([letter_b])
4          ([letter_c])
5          ([letter_d])
6          ([letter_e])
7          ([letter_f])
8          ([letter_g])
9          ([letter_h])
10         ([letter_i])
11         ([letter_j])
12         ([letter_k])
13         ([letter_l])
14         ([letter_m])
15         ([letter_n])
16         ([letter_o])
17         ([letter_p])
18         ([letter_q])
19         ([letter_r])
20         ([letter_s])
21         ([letter_t])
```

```
        ([letter_u])
        ([letter_v])
        ([letter_w])
        ([letter_x])
        ([letter_y])
        ([letter_z])
        ([letter_-])
        ([letter__])
;

[letter_a]
        (alpha)
;

[letter_b]
        (bravo)
;

[letter_c]
        (charlie)
;

[letter_d]
        (delta)
;

[letter_e]
        (echo)
;

[letter_f]
        (foxtrot)
;

[letter_g]
        (gulf)
;

[letter_h]
        (hotel)
;

[letter_i]
        (india)
;

[letter_j]
        (juliet)
;

[letter_k]
        (kilo)
;

```

```
[letter_l]
        (lima)
;

[letter_m]
        (mike)
;

[letter_n]
        (november)
;

[letter_o]
        (oscar)
;

[letter_p]
        (pappa)
;

[letter_q]
        (quebec)
;

[letter_r]
        (romeo)
;

[letter_s]
        (sierra)
;

[letter_t]
        (tango)
;

[letter_u]
        (uniform)
;

[letter_v]
        (victor)
;

[letter_w]
        (whiskey)
;

[letter_x]
        (x-ray)
        (x ray)
;

[letter_y]
```

```
        (yankee)
;

[letter_z]
        (zulu)
;

[letter_-]
        (dash)
        (hyphen)
        (mark)
;

[letter__]
        (capital)
        (upper case)
;

# Anaphoric expressions
# Seperate file, since these are not domain dependent

[deictic]
        (*right *about *over here)
        (*right *about *over there)
        (*on this spot)
        (*on this location)
        (*on this site)
        (*at this spot)
        (*at this location)
        (*at this site)
        (*to this spot)
        (*to this location)
        (*to this site)
;

# The exact nature of these anaphora and anaphoric expressions
# is determined by the postparser, which has language specific
# information on anaphora (e.g. gender, number, etc.)
[anaph]
        (this)
        (this one)
        (that)
        (him)
        (her)
        (it)
        (they)
        (them)
        (these)
        (those)
;
[camera]
        ([camera_tilt])
        ([camera_zoom])
        ([camera_pan])
```

```
184 ;
185
186 [camera_tilt]
187         (*move camera)
188         (tilt *camera)
189         (camera tilt)
190         (look)
191 ;
192
193 [camera_zoom]
194         (zoom *camera)
195         (camera zoom)
196 ;
197
198 [camera_pan]
199         (pan)
200         (move camera)
201         (pan *camera)
202         (move camera)
203         (camera pan)
204         (camera move)
205         (look)
206         (show *me)
207 ;
208
209 [zoom]
210         (zoom)
211 ;
212
213 [zoom_inout]
214         ([zoom_in])
215         ([zoom_out])
216 ;
217
218 [zoom_in]
219         (in)
220         (closer)
221 ;
222
223 [zoom_out]
224         (out)
225         (far)
226         (wide)
227         (wider)
228         (farther)
229         (further)
230 ;
231
232 [amount]
233         ([small_amount])
234         ([medium_amount])
235         ([large_amount])
236         ([Number])
237 ;
```

```
238
239 [small_amount]
240         (*a little)
241 ;
242
243 [medium_amount]
244         (some)
245         (somewhat)
246         (more)
247         (medium)
248         (moderate)
249         (moderately)
250 ;
251
252 [large_amount]
253         (*a lot)
254         (all the way)
255 ;
256
257 [tilt_direction]
258         ([up])
259         ([down])
260 ;
261
262 [pan_direction]
263         ([left])
264         ([mid])
265         ([right])
266 ;
267
268 [up]
269         (up)
270         (above)
271 ;
272
273 [down]
274         (down)
275         (below)
276 ;
277
278 [mid]
279         (middle)
280         (mid)
281         (center)
282         (straight)
283         (ahead)
284         (forward)
285         (front)
286 ;
287
288 [relative]
289         (more)
290         (further)
291         (little)
```

```
;

[camera_on]
        (visual)
        (camera on)
        (view camera)
        (*show video *on)
        (on screen)
;

[camera_off]
        (hide visual)
        (visual off)
        (hide camera)
        (camera off)
        (remove visual)
        (video off)
        (hide video)
;
[new_document]
        (new document)
        (new map)
        (create document)
;

[connect]
        (connect *to *the robot)
        (connecting *to *the robot)
        (*establish link *to *the robot)
        (open *the *map)
        (load *the *map)
        (bring up *the *map)
;

[map_name]
        (+LETTER_OR_NUMBER)
LETTER_OR_NUMBER
        ([letter])
        ([Number])
;
[direct_control]
        (enable direct control)
        (switch to direct control)
        (direct control *on)
        (teleoperate)
        (stop autonomous)
        (stop wandering)
        (listen)
;

[autonomous]
        (disable direct control)
        (direct control off)
        (autonomous)
```

```
        (go ahead)
        (you're on your own)
        (wander)
        (wandering)
        (mission)
;
[eye_reset]
        (reset eye tracker)
        (reset calibration)
        (restart calibration)
;

[eye_calibrate]
        (start calibration)
        (calibrate *eye *tracker)
;

[eye_next_point]
        (next *calibration *point)
        (next one)
;

[eye_output]
        (start running)
        (end calibration)
        (stop calibration)
        (calibration complete)
        (*eye *tracker output)
;

[eye_adjust]
        (adjust *calibration [direction])
;

[eye_camera]
        (move camera [direction])
        (adjust camera [direction])
;
[where]
        (*TO [location])
        ([deictic])
        (*TO [home])
        (*TO [named_location])
TO
        (at)
        (to)
;

[location]
        (+[Number] comma +[Number])
        (+[Number] x +[Number] y)
        (x *is *equals *equal *to +[Number] y *is *equals *equal *to +[Number])
;

```

```
[home]
        (*YOUR home)
        (*YOUR home base)
YOUR
        (the)
        (your)
        (our)
;

[named_location]
        (+LETTER_OR_NUMBER)
LETTER_OR_NUMBER
        ([letter])
        ([Number])
;
[login]
        (log in *to *disicple)
        (login *to *disicple)
        (*connect *to disciple)
;

[logout]
        (log off *from *disciple)
        (log out *of *disciple)
        (logout *of *disciple)
        (disconnect *from disciple)
;
[repair]
        (no)
        (sorry)
        (excuse me)
;

[frame_boundary]
        (then)
        (finally)
        (next)
        (after that)
        (done)
;
[mission_type]
        (mission type *is [mission_type_spec])
        (this *mission *is *a [mission_type_spec])
        (set mission type *TO [mission_type_spec])
        (new [mission_type_spec])
TO
        (to)
        (as)
;

[mission_name]
        (mission name *is +LETTER_OR_NUMBER)
        (mission is called +LETTER_OR_NUMBER)
LETTER_OR_NUMBER
```

```
454         ([letter])
455         ([Number])
456 ;
457
458 [mission_type_spec]
459         ([mission_Reconnaissance])
460         ([mission_Mine_Detection])
461         ([mission_Map_Region])
462         ([mission_Patrol])
463 ;
464
465 [mission_Reconnaissance]
466         (reconnaissance *mission)
467         (recon *mission)
468 ;
469
470 [mission_Patrol]
471         (patrol *mission)
472         (patrolling *mission)
473 ;
474
475 [mission_Map_Region]
476         (*region mapping *mission)
477         (map *region *mission)
478 ;
479
480 [mission_Mine_Detection]
481         (mine *detection *mission)
482 ;
483
484 [mission_homebase]
485         (home *base *is [where])
486         ([where] *is *the home base)
487         ([anaph] *is *the home base)
488 ;
489
490 [mission_resupply]
491         (resupply *point *is [where])
492         ([anaph] *is *the resupply point)
493         ([where] *is *the resupply point)
494 ;
495
496 [mission_task]
497         (mission task)
498 ;
499
500 [task_type]
501         ([task_recon])
502         ([task_map])
503         ([task_destroy])
504 ;
505
506 [task_destroy]
507         (destroy)
```

```
;

[task_map]
        (map)
;

[task_recon]
        (*DO *a reconnaissance *TASK)
        (*DO *a recon *TASK)
        (new recon *TASK)
        (new reconnaissance *TASK)
DO
        (do)
        (perform)
        (put)
        (add)
        (new)
TASK
        (task)
        (target)
        (point)
;

[mission_start]
        (start mission)
        (commence mission)
;

[mission_stop]
        (stop mission)
        (cancel *mission)
        (abort *mission)
        (terminate *mission)
        (halt mission)
;

[mission_pause]
        (pause *mission)
        (suspend *mission)
;

[mission_resume]
        (resume)
        (continue)
;

[mission]
        (*the mission)
        (*a mission)
;
[move]
        (move)
        (go)
        (come)
```

```
562         (drive)
563         (get)
564         (ride)
565 ;
566
567 [direction]
568         ([north])
569         ([south])
570         ([east])
571         ([west])
572         ([northwest])
573         ([northeast])
574         ([southwest])
575         ([southeast])
576         ([forward])
577         ([backward])
578 ;
579
580 [north]
581         (north)
582         (up)
583 ;
584
585 [south]
586         (south)
587         (down)
588 ;
589
590 [east]
591         (east)
592         (right)
593 ;
594
595 [west]
596         (west)
597         (left)
598 ;
599
600 [southwest]
601         (southwest)
602         (south west)
603         (down *and left)
604         (left *and down)
605 ;
606
607 [southeast]
608         (southeast)
609         (south east)
610         (down *and right)
611         (right *and down)
612 ;
613
614 [northwest]
615         (northwest)
```

```
616         (north west)
617         (up *and left)
618         (left *and up)
619 ;
620
621 [northeast]
622         (northeast)
623         (north east)
624         (up *and right)
625         (right *and up)
626 ;
627
628 [forward]
629         (forward)
630         (forwards)
631         (front)
632 ;
633
634 [backward]
635         (backward)
636         (backwards)
637         (back)
638         (rear)
639         (backup)
640 ;
641
642 [distance]
643         ([Number] *[d_unit])
644         ([Number] [d_unit] *and [distance])
645         (A *[d_unit])
646 A
647         (a)
648         (an)
649 ;
650
651 [speed]
652         ([Number] [d_unit] *per [t_unit])
653         ([Number] [d_unit] an [t_unit])
654         ([slow_speed])
655         ([medium_speed])
656         ([fast_speed])
657         ([max_speed])
658 ;
659
660 [slow_speed]
661         (*do *take *make *it slow)
662         (*do *take *make *it slowly)
663         (*do *take *make *it carefully)
664 ;
665
666 [medium_speed]
667         (*do *take *make *it medium)
668         (*do *take *make *it moderate)
669         (*do *take *make *it moderately fast)
```

```
        (*don't *not *do *it *too fast)
;

[fast_speed]
        (*do *take *make *it fast)
;

[max_speed]
        (*do *take *make *it *as fast *as you can *go)
        (*do *take *make *it *at *your maximum speed)
        (*do *take *make *it *at *your highest speed)
;

[d_unit]
        ([du_in])
        ([du_cm])
        ([du_ft])
        ([du_yd])
        ([du_m])
        ([du_mi])
        ([du_km])
        ([du_px])
;

[du_in]
        (inch)
        (inches)
;
[du_cm]
        (centimeter)
        (centimeters)
        (cm)
;
[du_ft]
        (foot)
        (feet)
        (ft)
;
[du_m]
        (meter)
        (meters)
;
[du_mi]
        (miles)
        (mile)
;
[du_yd]
        (yard)
        (yards)
        (yd)
;
[du_km]
        (kilometer)
        (kilometers)
```

```
724         (km)
725 ;
726
727 [du_px]
728         (pixel)
729         (pixels)
730         (point)
731         (points)
732         (dot)
733         (dots)
734 ;
735
736 [t_unit]
737         ([tu_h])
738         ([tu_min])
739         ([tu_s])
740 ;
741
742 [tu_h]
743         (hour)
744         (hours)
745 ;
746
747 [tu_min]
748         (minute)
749         (minutes)
750 ;
751
752 [tu_s]
753         (second)
754         (seconds)
755 ;
756 [call]
757         (call)
758         (name)
759 ;
760
761 [name]
762         (+LETTER_OR_NUMBER)
763 LETTER_OR_NUMBER
764         ([letter])
765         ([Number])
766 ;
767
768 # Two problems:
769 # 1) Language independence. Clearly this is very specific to English and changing
770 #    it to work with another language is far from trivial
771 # 2) ViaVoice outputs numbers formatted, i.e. not as text, so these rules are
772 #    not of much use there unless we convert the formatted number back to a
773 #    textual representation first
774
775 [Number]
776         (MIL *THOU *HUN *and *BASE)
777         (THOU *HUN *and *BASE)
```

```
778         (HUN *and *BASE)
779         (BASE)
780         (+DIGIT)
781 A
782         (a)
783         (an)
784 MIL
785         (BASE million)
786 THOU
787         (BASE thousand)
788         (DIGIT hundred *BASE thousand)
789 HUN
790         (*a hundred)
791         (DIGIT hundred)
792         (TEEN hundred)
793 BASE
794         (DIGIT)
795         (TEEN)
796         (DECADE *DIGIT)
797 DIGIT
798         (zero)
799         (one)
800         (two)
801         (three)
802         (four)
803         (five)
804         (six)
805         (seven)
806         (eight)
807         (nine)
808 TEEN
809         (ten)
810         (eleven)
811         (twelve)
812         (thirteen)
813         (fourteen)
814         (fifteen)
815         (sixteen)
816         (seventeen)
817         (eighteen)
818         (nineteen)
819 DECADE
820         (twenty)
821         (thirty)
822         (forty)
823         (fifty)
824         (sixty)
825         (seventy)
826         (eighty)
827         (ninety)
828 ;
829
830 [Ordinal]
831         (ORDINAL_DIGIT)
```

```
832         (ORDINAL_TEEN)
833         (ORDINAL_DECADE)
834 ORDINAL_DIGIT
835         (first)
836         (second)
837         (third)
838         (fourth)
839         (fifth)
840         (sixth)
841         (seventh)
842         (eighth)
843         (ninth)
844 ORDINAL_TEEN
845         (tenth)
846         (eleventh)
847         (twelfth)
848         (thirteenth)
849         (fourteenth)
850         (fifteenth)
851         (sixteenth)
852         (seventeenth)
853         (eighteenth)
854         (nineteenth)
855 ORDINAL_DECADE
856         (twentieth)
857         (thirtieth)
858         (fortieth)
859         (fiftieth)
860         (sixtieth)
861         (seventieth)
862         (eightieth)
863         (ninetieth)
864         (*one hundredth)
865 DECADE
866         (twenty)
867         (thirty)
868         (forty)
869         (fifty)
870         (sixty)
871         (seventy)
872         (eighty)
873         (ninety)
874 ;
875
876 [what]
877         (*a *an [unit] *one)
878 # the optional "one" is for ellipses, i.e. "create a hostile one here"
879         ([anaph])
880 ;
881
882 [unit]
883         ([affiliation] *ONE)
884         ([size] *ONE)
885         ([unittype] *ONE)
```

```
# Beyond type it gets a little too complicated, so
# either just do that through the GUI, or add commands
# later (i.e. "set unit modifier: tracked" or "change to tracked"
# or something
ONE
        (one)
        (unit)
;

[affiliation]
        ([affiliation_F])
        ([affiliation_N])
        ([affiliation_H])
        ([affiliation_U])
;

[affiliation_F]
        (friendly)
;

[affiliation_N]
        (neutral)
;

[affiliation_H]
        (hostile)
        (enemy)
        (enemies)
        (bad guy)
        (bad guys)
;

[affiliation_U]
        (unknown)
;

[size]
        ([size_A])
        ([size_B])
        ([size_C])
        ([size_D])
        ([size_E])
        ([size_F])
        ([size_G])
        ([size_H])
        ([size_I])
        ([size_J])
        ([size_K])
        ([size_L])
        ([size_M])
;

[size_A]
        (team)
```

```
940         (teams)
941 ;
942
943 [size_B]
944         (squad)
945         (squads)
946 ;
947
948 [size_C]
949         (section)
950         (sections)
951 ;
952
953 [size_D]
954         (platoon)
955         (platoons)
956 ;
957
958 [size_E]
959         (company)
960         (companies)
961 ;
962
963 [size_F]
964         (battalion)
965         (battalions)
966 ;
967
968 [size_G]
969         (regiment)
970         (regiments)
971 ;
972
973 [size_H]
974         (brigade)
975         (brigades)
976 ;
977
978 [size_I]
979         (division)
980         (divisions)
981 ;
982
983 [size_J]
984         (corps)
985         (corpses)
986 ;
987
988 [size_K]
989         (army)
990         (armies)
991 ;
992
993 [size_L]
```

```
994         (army group)
995         (army groups)
996 ;
997
998 [size_M]
999         (region)
1000        (regions)
1001 ;
1002
1003 [unittype]
1004        ([unittype_UCA])
1005        ([unittype_UCAA])
1006        ([unittype_UCV])
1007        ([unittype_UCI])
1008        ([unittype_UCE])
1009        ([unittype_UCF])
1010        ([unittype_UCR])
1011        ([unittype_XXX])
1012 ;
1013
1014 [unittype_UCA]
1015        (armor)
1016        (armour)
1017        (armors)
1018        (armours)
1019 ;
1020
1021 [unittype_UCAA]
1022        (anti armor)
1023        (anti-armor)
1024        (anti armour)
1025        (anti-armour)
1026        (anti armors)
1027        (anti-armors)
1028        (anti armours)
1029        (anti-armours)
1030 ;
1031
1032 [unittype_UCV]
1033        (aviation)
1034        (aviations)
1035 ;
1036
1037 [unittype_UCI]
1038        (infantry)
1039        (infantries)
1040 ;
1041
1042 [unittype_UCE]
1043        (engineer)
1044        (engineers)
1045 ;
1046
1047 [unittype_UCF]
```

```
1048         (field artillery)
1049         (field artilleries)
1050 ;
1051
1052 [unittype_UCR]
1053         (reconnaissance)
1054         (recon)
1055         (recons)
1056 ;
1057
1058 [unittype_XXX]
1059         (robot)
1060         (robotic)
1061         (robots)
1062 ;
1063 [quit]
1064         (quit)
1065         (exit)
1066         (*good bye)
1067         (close application)
1068 ;
1069 [rotate]
1070         (rotate)
1071         (turn)
1072 ;
1073
1074 [rotate_direction]
1075         ([left])
1076         ([right])
1077 ;
1078
1079 [degrees]
1080         (+[Number] *degrees)
1081 ;
1082
1083 [left]
1084         (counterclockwise)
1085         (left)
1086 ;
1087
1088 [right]
1089         (clockwise)
1090         (right)
1091 ;
1092 [stop]
1093         (stop)
1094         (halt)
1095         (hold)
1096         (wait)
1097 ;
1098 [action]
1099         ([destroy])
1100         ([find])
1101 ;
```

```
[put]
        (put)
        (move)
        (relocate)
        (transport)
;
[destroy]
        (destroy)
        (kill)
        (nuke)
;
[find]
        (find)
        (locate)
        (look for)
;
[scan]
        (scan)
        (investigate)
        (map)
;

[when]
        ([now])
        ([snap])
        ([queue])
;

[now]
        (*right now)
        (immediately)
        (straight away)
;

[snap]
        (when i snap my fingers)
;

[queue]
        (when i say now)
;

[test]
        (test)
;
[identify]
        (identify)
        (identified)
        (id)
        (i d)
        (information)
        (what is)
        (what's)
;
```

```
1156
1157 [create]
1158         (create *A *new)
1159         (make *A *new)
1160         (*place *A new)
1161         (*put *A new)
1162         (put *A)
1163         (place *A)
1164 A
1165         (a)
1166         (an)
1167 ;
1168
1169 [delete]
1170         (delete)
1171         (remove)
1172         (scratch)
1173         (erase)
1174 ;
```

Appendix D

Fusion and Dialog Manager Configuration

```
<fusion>

<!-- ************************************************** -->
        <frame name="rotate" test="rotate" uses="rotate">
                <slot name="direction">
                        <source select="rotate/rotate_direction/*" />
                </slot>
                <slot name="angle">
                        <source select="rotate/degrees">
                                <resolve resolver="distanceResolver" />
                        </source>
                </slot>

                <slot name="speed">
                        <source select="rotate/speed">
                                <resolve resolver="rotationSpeedResolver" />
                        </source>
                </slot>

                <action language="javascript">
                var angle = 90;
                var speed = 400;
                // If just "turn" is said, assume turn all the way around (180 degre
                if (!frame.direction &amp;&amp; !frame.angle) angle = 180;
                // cancel any movements
                application.api.transSpeed = 0;
                if (frame.angle) angle = frame.angle;
                if (frame.speed) speed = frame.speed;
                if (frame.direction &amp;&amp; frame.direction.toLowerCase().equals(

                var xml = '<rotate>';
                if (frame.speed) xml += '<speed>' + speed + '</speed>';
                xml += '<direction>' + (angle > 0 ? "left" : "right") + '</direction
                xml += '<angle>' + java.lang.Math.abs(angle) + '</angle>';
                xml += '</rotate>';
```

```
36                 session.speak(xml);
37
38                 application.api.rotate(angle, speed);
39                 </action>
40         </frame>
41
42 <!-- ************************************************* -->
43 <!-- A frame to move a glyph. -->
44         <frame name="move" test="move and ((move/where) or (move/direction) or (move/speed)
45                 <slot name="destination" required="true">
46                         <source select="move/where/deictic">
47                                 <!-- The resolver gets as input the tree fragment with
48                                                 top-level node deictic. Its job is to tran
49                                                 it and return the possible tree fragments
50                                                 the "deictic" node, along with confidence
51                                 -->
52                                 <resolve resolver="eyeLocationResolver" />
53                                 <resolve resolver="mouseLocationResolver" />
54                         </source>
55                         <source select="move/where/location">
56                                 <resolve resolver="coordinateResolver" />
57                         </source>
58                         <source select="move/where/named_location">
59                                 <resolve resolver="namedLocationResolver">
60                                         <element name="name"><values resolver="spellingReso
61                                 </resolve>
62                         </source>
63                         <source select="move/where/home">
64                                 <resolve resolver="homeResolver" />
65                         </source>
66                 </slot>
67                 <slot name="glyph">
68                         <source select="move/what/anaph">
69                                 <resolve resolver="mouseObjectResolver" />
70                                 <resolve resolver="eyeObjectResolver" />
71                                 <resolve resolver="dialogObjectResolver" weight="0.1"/>
72                         </source>
73                         <source select="move/what/unit">
74                                 <resolve resolver="unitResolver" />
75                         </source>
76                 </slot>
77                 <slot name="direction">
78                         <source select="move/direction/*" />
79                 </slot>
80                 <slot name="distance">
81                         <source select="move/distance">
82                                 <resolve resolver="distanceResolver" />
83                         </source>
84                 </slot>
85                 <slot name="speed">
86                         <source select="move/speed">
87                                 <resolve resolver="speedResolver" />
88                         </source>
89                 </slot>
```

```
90                  <slot name="time" required="true">
91                          <source select="task/when" />
92                  </slot>
93
94                  <action language="javascript">
95                          var robot;
96                          if (frame.glyph) {        // move the glyph
97                                  // What's the glyph's type?
98                                  if (frame.glyph.glyph.model.nodeName.equals('robot')
99                                          // special action
100                                         robot = glyph.glyph.model;
101                                 } else {
102                                         if (frame.destination) {
103                                                 java.lang.System.out.println("Moving
104                                                 frame.glyph.location = frame.destina
105                                         } else {
106                                                 session.speak('<missing><frame>move<
107                                                 // return "missing";
108                                         }
109                                 }
110                         } else {
111                                 robot = application.api.activeRobot;
112                         }
113                         if (robot) {
114                                 java.lang.System.out.println("Moving robot " + robot
115                                 application.api.activeRobot = robot;
116                                 // move the active robot
117                                 // cancel any rotations
118                                 application.api.rotSpeed = 0;
119                                 var speed = 200;
120                                 if (frame.speed) speed = frame.speed;
121                                 if (frame.distance) {
122                                         var distance = frame.distance;
123                                         var direction = frame.direction.toLowerCase(
124                                         if (frame.direction &amp;&amp; direction.equ
125
126                                         var xml = '<move>';
127                                         xml += '<speed>' + speed + '</speed>';
128                                         xml += '<direction>' + (distance > 0 ? "forw
129                                         xml += '<distance>' + java.lang.Math.abs(dis
130                                         xml += '</move>';
131                                         session.speak(xml);
132
133                                         application.api.travel(distance, speed);
134                                 } else {
135                                         application.api.transSpeed = speed;
136                                         if (frame.destination) {
137                                                 application.api.travelTo(frame.desti
138                                         } else if (frame.direction) {
139                                                 var direction = frame.direction.toLo
140
141                                                 var xml = '<move>';
142                                                 xml += '<speed>' + speed + '</speed>
143                                                 xml += '<direction>' + (direction) +
```

```
144                                                 xml += '</move>';
145                                                 session.speak(xml);
146 
147                                                 if (direction.equals("forward")) {
148                                                         application.api.back = false;
149                                                         application.api.forward  = true;
150                                                 } else
151                                                 if (direction.equals("backward") || directi
152                                                         application.api.forward  = false;
153                                                         application.api.back = true;
154                                                 }
155                                         } else {
156                                                 session.speak('<missing><frame>move</frame>
157                                                 // return "missing";
158                                         }
159                                 }
160                         }
161                 </action>
162         </frame>
163 
164 <!-- ************************************************* -->
165     <!-- A frame to stop the robot. This resets the forward, backward,
166        - left, and right fields to 0
167          -->
168         <frame name="stop" test="stop" uses="stop">
169                 <action language="javascript">
170                         session.speak('<stop />');
171                         application.api.forward  = false;
172                         application.api.back     = false;
173                         application.api.left     = false;
174                         application.api.right    = false;
175                 </action>
176         </frame>
177 
178 <!-- ************************************************* -->
179     <!-- A frame to identify a unit, or the active robot
180          -->
181         <frame name="identify" test="identify" uses="identify">
182                 <slot name="glyph">
183                         <source select="identify/anaph" weight="1.0">
184                                 <resolve resolver="mouseObjectResolver" />
185                                 <resolve resolver="eyeObjectResolver" />
186                         </source>
187                         <source select="identify" weight="0.5">        <!-- last resort typ
188                                 <resolve resolver="mouseObjectResolver" />
189                                 <resolve resolver="eyeObjectResolver" />
190                                 <resolve resolver="dialogObjectResolver" />
191                         </source>
192                 </slot>
193                 <action language="javascript">
194                         java.lang.System.out.println("Unit is " + frame.glyph);
195                         var UFormUtils = Packages.flatscape.util.UFormUtils;
196                         if (frame.glyph) {
197                                 var locatable = frame.glyph;
```

```
198                                 var unit = frame.glyph.glyph;
199
200                                 var model = unit.model;
201                                 var loc = locatable.location;
202                                 // identify a particular unit
203                                 session.speak('<identify><unittype>' + UFormUtils.ge
204                         } else {
205                                 // identify the currently selected robot
206                                 if (application.api.name != null)
207                                         session.speak('<robot><name>' + application.
208                         }
209                 </action>
210         </frame>
211
212 <!-- ************************************************ -->
213           -->
214         <frame name="create" test="create/unit or create/where" uses="create">
215                 <slot name="affiliation">
216                         <source select="create/unit/affiliation/*">
217                                 <resolve resolver="affiliationResolver" />
218                         </source>
219                         <source select="create" priority="2">
220                                 <resolve resolver="affiliationHistoryResolver" weigh
221                         </source>
222                 </slot>
223                 <slot name="unittype">
224                         <source select="create/unit/unittype/*">
225                                 <resolve resolver="unittypeResolver" />
226                         </source>
227                         <source select="create" priority="2">
228                                 <resolve resolver="unittypeHistoryResolver" weight="
229                         </source>
230                 </slot>
231                 <slot name="size">
232                         <source select="create/unit/size/*">
233                                 <resolve resolver="unitsizeResolver" />
234                         </source>
235                         <source select="create" priority="2">
236                                 <resolve resolver="unitsizeHistoryResolver" weight="
237                         </source>
238                 </slot>
239                 <slot name="location">
240                         <source select="create/where/deictic" priority="1">
241                                 <resolve resolver="mouseLocationResolver" />
242                                 <resolve resolver="eyeLocationResolver" />
243                         </source>
244                         <source select="create/where/location" priority="1">
245                                 <resolve resolver="coordinateResolver" />
246                         </source>
247                         <source select="create" priority="2">
248                                 <resolve resolver="mouseLocationResolver"  weight="0
249                                 <resolve resolver="eyeLocationResolver"  weight="0.3
250                                 <resolve resolver="locationHistoryResolver" weight="
251                         </source>
```

```
252                 </slot>
253                 <action language="javascript">
254                         var text = '';
255                         var unittype = '';
256                         var size = '';
257                         var affiliation = '';
258 
259                         if (frame.unittype) {
260                                 text += '<unittype>' + frame.unittype.substring(2) + '</uni
261                                 unittype = frame.unittype;
262                         }
263                         if (frame.size) {
264                                 text += '<unitsize>' + frame.size + '</unitsize>';
265                                 size = frame.size;
266                         }
267                         if (frame.affiliation) {
268                                 text += '<affiliation>' + frame.affiliation + '</affiliatio
269                                 affiliation = frame.affiliation;
270                         }
271                         if (!frame.unittype) {
272                                 session.speak("<missing><frame>create</frame><slot>unittype
273                                 // return "missing";
274                         } else {
275                                 if (!frame.location) {
276                                         session.speak("<missing><frame>location</frame><slo
277                                         // return "missing";
278                                 }
279                                 session.speak('<create>' + text + '</create>');
280                                 application.api.createUnit(affiliation, unittype, size, fra
281                         }
282                 </action>
283         </frame>
284 
285 <!-- ************************************************* -->
286     <!-- A frame to delete a glyph
287           -->
288         <frame name="delete" test="delete/delete" uses="delete">
289                 <slot name="glyph">
290                         <source select="delete/what/anaph">
291                                 <resolve resolver="mouseObjectResolver" />
292                                 <resolve resolver="eyeObjectResolver" />
293                         </source>
294                         <source select="delete/what/unit">
295                                 <resolve resolver="unitResolver" />
296                         </source>
297                         <source select="delete" priority="2">
298                                 <resolve resolver="dialogObjectResolver" weight="0.1" />
299                         </source>
300                 </slot>
301                 <action language="javascript">
302                         if (frame.glyph) {
303                                 application.api.deleteGlyph(frame.glyph.glyph);
304                         } else {
305                                 session.speak('<missing><frame>delete</frame><slot>glyph</s
```

```
306                                 // return "missing";
307                         }
308                 </action>
309         </frame>
310 <!-- ************************************************* -->
311     <!-- A frame to report battery power
312           -->
313         <frame name="battery" test="battery" uses="battery">
314                 <slot name="glyph">
315                         <source select="identify/anaph">
316                                 <resolve resolver="mouseObjectResolver" />
317                                 <resolve resolver="eyeObjectResolver" />
318                         </source>
319                 </slot>
320                 <action language="javascript">
321                   if (frame.unit) {
322                                 // report for a particular unit
323                         } else {
324                                 // report battery of the currently selected robot
325                                 session.speak('<battery><name>' + application.api.na
326                         }
327                 </action>
328         </frame>
329
330 <!-- ************************************************* -->
331         <frame name="eyetracker" test="eyetracker" uses="eyetracker">
332                         <slot name="reset">
333                                 <source select="eyetracker/eye_reset" />
334                         </slot>
335                         <slot name="calibrate">
336                                 <source select="eyetracker/eye_calibrate" />
337                         </slot>
338                         <slot name="next">
339                                 <source select="eyetracker/eye_next_point" />
340                         </slot>
341                         <slot name="output">
342                                 <source select="eyetracker/eye_output" />
343                         </slot>
344                         <slot name="camera">
345                                 <source select="eyetracker/camera" />
346                         </slot>
347                         <slot name="adjust">
348                                 <source select="eyetracker/adjust" />
349                         </slot>
350
351                         <action language="javascript">
352                         java.lang.System.out.println("frame == " + frame);
353                         java.lang.System.out.println("frame.calibrate == " + frame.c
354                         // This could throw a null pointer exception in a lot of pla
355                         // we define everything below it should all be good
356                         var eyeTracker = session.getContextProviderPool().get("eyeTr
357                         if (frame.reset) {
358                                 java.lang.System.out.println("Resetting eye tracker"
359                                 eyeTracker.reset();
```

```
360                         }
361                         if (frame.calibrate) {
362                                 java.lang.System.out.println("Calibrating eye tracker");
363                                 eyeTracker.calibrate();
364                         }
365                         if (frame.next) {
366                                 java.lang.System.out.println("Jumping to next calibration p
367                                 eyeTracker.next();
368                         }
369                         if (frame.output) {
370                                 java.lang.System.out.println("Starting eye tracking");
371                                 eyeTracker.output();
372                         }
373                         if (frame.toggle) {
374                                         eyeTracker.toggle();
375                         }
376                         if (frame.camera) {
377                                 if (frame.camera.left)
378                                         eyeTracker.cameraLeft();
379                                 if (frame.camera.right)
380                                         eyeTracker.cameraRight();
381                                 if (frame.camera.up)
382                                         eyeTracker.cameraUp();
383                                 if (frame.camera.down)
384                                         eyeTracker.cameraDown();
385                                 if (frame.camera.left)
386                                         eyeTracker.cameraLeft();
387                         }
388                         if (frame.adjust) {
389                                 if (frame.adjust.left)
390                                         eyeTracker.adjustLeft();
391                                 if (frame.adjust.right)
392                                         eyeTracker.adjustRight();
393                                 if (frame.adjust.up)
394                                         eyeTracker.adjustUp();
395                                 if (frame.adjust.down)
396                                         eyeTracker.adjustDown();
397                                 if (frame.adjust.left)
398                                         eyeTracker.adjustLeft();
399                         }
400                         </action>
401         </frame>
402
403 <!-- ************************************************* -->
404         <frame name="destroy" test="task/action/destroy" uses="destroy">
405                         <slot name="object" required="true">
406                                         <source select="task/what/anaph" mode="support">
407                                                 <resolve resolver="mouseObjectResol
408                                                 <resolve resolver="eyeObjectResolve
409                                         </source>
410                         </slot>
411                         <slot name="time">
412                                         <source select="task/when" />
413                         </slot>
```

```
414
415         </frame>
416
417
418 <!-- ************************************************ -->
419         <frame name="connect" test="connect/connect" uses="connect">
420                 <slot name="map">
421                         <source select="connect/map_name">
422                                 <resolve resolver="spellingResolver" />
423                         </source>
424                 </slot>
425                 <action language="javascript">
426                         var map = "core7";
427                         if (frame.map) map = frame.map;
428                         session.speak("<open>" + map + ".xml</open>");
429                         application.api.openMap(map + ".xml");
430                 </action>
431         </frame>
432
433 <!-- ************************************************ -->
434         <frame name="new" test="connect/new_document" uses="connect">
435                 <action language="javascript">
436                         session.speak("<new></new>");
437                         application.api.newMap();
438                 </action>
439         </frame>
440
441 <!-- ************************************************ -->
442         <frame name="login" test="login" uses="login">
443                 <slot name="login">
444                         <source select="login/login" />
445                 </slot>
446                 <slot name="logout">
447                         <source select="login/logout" />
448                 </slot>
449                 <action language="javascript">
450                         <!--
451                            - Log in or out of Disciple
452                            -->
453                         if (frame.login) {
454                                 session.speak('<login />');
455                           application.api.logon();
456                         }
457                         if (frame.logout) {
458                                 session.speak('<logout />');
459                                 application.api.logoff();
460                         }
461                 </action>
462         </frame>
463
464 <!-- ************************************************ -->
465         <frame name="control" test="control" uses="control">
466                 <slot name="direct">
467                         <source select="control/direct_control" />
```

```
468                 </slot>
469                 <slot name="autonomous">
470                         <source select="control/autonomous" />
471                 </slot>
472                 <action language="javascript">
473                         if (frame.direct) {
474                                 if (application.api.controlState) {
475                                         session.speak('<direct-control repeat="true" />');
476                                 } else {
477                                         session.speak('<direct-control />');
478                                 }
479                                 application.api.controlState = true;
480                         }
481                         if (frame.autonomous) {
482                                 if (!application.api.controlState) {
483                                         session.speak('<autonomous repeat="true" />');
484                                 } else {
485                                         session.speak('<autonomous />');
486                                 }
487                                 application.api.controlState = false;
488                         }
489                 </action>
490         </frame>
491
492 <!-- ************************************************ -->
493         <!-- it seems that natural language for panning is, unlike
494          that for tilting and zooming, absolute in its "amount"
495          specifications, that is, "look left" means look left absolutely,
496          not left relative to where you are looking now. Testing
497          with real users will show whether this is correct, but for
498          now we assume it
499         -->
500         <frame name="pan" test="camera_pan" uses="camera_pan">
501                 <slot name="relative">
502                         <source select="camera_pan/relative" />
503                 </slot>
504                 <slot name="direction">
505                         <source select="camera_pan/pan_direction/*" />
506                 </slot>
507                 <slot name="amount">
508                         <source select="camera_pan/amount">
509                                 <resolve resolver="panAmountResolver" />
510                         </source>
511                 </slot>
512                 <action language="javascript">
513                         amount = 90;
514                         if (frame.amount) amount = frame.amount;
515                         if (frame.relative) {
516                                 // average between what we have now and
517                                 // the extreme. Amount determines the
518                                 // weighing factor
519                                 if (frame.direction == "left")
520                                         application.api.pan = (amount * -90 + (90-amount) *
521                                 if (frame.direction == "right")
```

```
522                                     application.api.pan = (amount * 90 + (90-amo
523                             if (frame.direction == "mid")
524                                     application.api.pan = ((90 - amount) * appli
525                     } else {
526                             if (frame.direction == "left")
527                                     application.api.pan = -amount;
528                             if (frame.direction == "right")
529                                     application.api.pan = amount;
530                             if (frame.direction == "mid")
531                                     application.api.pan = 0;
532                     }
533             </action>
534     </frame>
535
536 <!-- ************************************************ -->
537     <frame name="tilt" test="camera_tilt" uses="camera_tilt">
538             <slot name="direction">
539                     <source select="camera_tilt/tilt_direction/*" />
540             </slot>
541             <slot name="amount">
542                     <source select="camera_tilt/amount">
543                             <resolve resolver="tiltAmountResolver" />
544                     </source>
545             </slot>
546             <action language="javascript">
547                     amount = 10;
548                     if (frame.amount) amount = frame.amount;
549                     if (frame.direction == "up")
550                             application.api.tilt = Number(application.api.tilt)
551                     if (frame.direction == "down")
552                             application.api.tilt = Number(application.api.tilt)
553             </action>
554     </frame>
555
556 <!-- ************************************************ -->
557     <frame name="zoom" test="camera_zoom/camera_zoom" uses="camera_zoom">
558             <slot name="zoom">
559                     <source select="camera_zoom/zoom_inout/*" />
560             </slot>
561             <slot name="amount">
562                     <source select="camera_zoom/amount">
563                             <resolve resolver="zoomAmountResolver" />
564                     </source>
565             </slot>
566             <action language="javascript">
567                     amount = 300;
568                     if (frame.amount) amount = frame.amount;
569                     if (frame.zoom == "zoom_out")
570                             application.api.zoom = Number(application.api.zoom)
571                     if (frame.zoom == "zoom_in")
572                             application.api.zoom = Number(application.api.zoom)
573             </action>
574     </frame>
575
```

```
<!-- ************************************************* -->
        <frame name="visual" test="camera_visual" uses="camera_visual">
                <slot name="on">
                        <source select="camera_visual/camera_on" />
                </slot>
                <slot name="off">
                        <source select="camera_visual/camera_off" />
                </slot>
                <action language="javascript">
                        if (frame.on)  {
                                session.speak('<video mode="on" />');
                                application.api.video = true;
                        } else {
                                session.speak('<video mode="off" />');
                                application.api.video = false;
                        }
                </action>
        </frame>

<!-- ************************************************* -->
        <frame name="quit" test="quit" uses="quit">
                <action language="javascript">
                session.speak('<quit />');
                application.api.quit();
                </action>
        </frame>

<!-- ************************************************* -->
        <frame name="mission" test="mission" uses="mission">
                <slot name="name">
                        <source select="mission/mission_name">
                                <resolve resolver="spellingResolver" />
                        </source>
                </slot>
                <slot name="type">
                        <source select="mission/mission_type/mission_type_spec/*">
                                <resolve resolver="missionTypeResolver" />
                        </source>
                </slot>
                <slot name="homebase">
                        <source select="mission/mission_homebase/where/deictic">
                                <resolve resolver="mouseLocationResolver" />
                                <resolve resolver="eyeLocationResolver" />
                        </source>
                        <source select="mission/mission_homebase/anaph">
                                <resolve resolver="mouseLocationResolver" />
                                <resolve resolver="eyeLocationResolver" />
                        </source>
                        <source select="mission/mission_homebase/where/location">
                                <resolve resolver="coordinateResolver" />
                        </source>
                </slot>
                <slot name="resupplyPoint">
                        <source select="mission/mission_resupply/where/deictic">
```

```
630                             <resolve resolver="mouseLocationResolver" />
631                             <resolve resolver="eyeLocationResolver" />
632                     </source>
633                     <source select="mission/mission_resupply/anaph">
634                             <resolve resolver="mouseLocationResolver" />
635                             <resolve resolver="eyeLocationResolver" />
636                     </source>
637                     <source select="mission/mission_resupply/where/location">
638                             <resolve resolver="coordinateResolver" />
639                     </source>
640             </slot>
641             <action language="javascript">
642                     var missionController = application.api.missionController;
643                     var nameSet, typeSet, homebaseSet, resupplyPointSet;
644                     if (frame.name) {
645                             missionController.missionName = frame.name;
646                             nameSet = true;
647                     }
648                     if (frame.type) {
649                             missionController.missionType = frame.type;
650                             typeSet = true;
651                     }
652                     if (frame.homebase) {
653                             missionController.homeBase = frame.homebase;
654                             homebaseSet = true;
655                     }
656                     if (frame.resupplyPoint) {
657                             missionController.resupplyPoint = frame.resupplyPoin
658                             resupplyPointSet = true;
659                     }
660                     var text = '<mission-type set="' + typeSet + '">' + missionC
661                     text    += '<name set="' + nameSet + '">' + missionControlle
662                     if (homebaseSet) text    += '<homebase><x>' + missionControl
663                     if (resupplyPointSet) text    += '<resupply><x>' + missionCo
664                     session.speak('<mission>' + text + '</mission>');
665             </action>
666     </frame>
667
668 <!-- ************************************************ -->
669     <frame name="mission_control" test="mission_control/mission_start or mission
670             <slot name="start">
671                     <source select="mission_control/mission_start" />
672             </slot>
673             <slot name="stop">
674                     <source select="mission_control/mission_stop" />
675             </slot>
676             <slot name="pause">
677                     <source select="mission_control/mission_pause" />
678             </slot>
679             <slot name="resume">
680                     <source select="mission_control/mission_resume" />
681             </slot>
682             <action language="javascript">
683                     var missionController = application.api.missionController;
```

```
684                         if (frame.start) {
685                                 missionController.sendMission();
686                                 session.speak('<mission-start><name>' + missionController.m
687                         }
688                         if (frame.stop) {
689                                 missionController.cancelMission();
690                                 session.speak('<mission-stop><name>' + missionController.mi
691                                 application.api.controlState = true;
692                         }
693                         if (frame.pause) {
694                                 application.api.controlState = true;
695                                 session.speak('<mission-pause><name>' + missionController.m
696                         }
697                         if (frame.resume) {
698                                 application.api.controlState = false;
699                                 session.speak('<mission-resume><name>' + missionController.
700                         }
701                 </action>
702         </frame>
703
704 <!-- ************************************************ -->
705         <frame name="name" test="(name/call and name/name) or (name/what and name/name) or
706                 <slot name="glyph">
707                         <source select="name/what/anaph">
708                                 <resolve resolver="mouseObjectResolver" />
709                                 <resolve resolver="eyeObjectResolver" />
710                                 <resolve resolver="dialogObjectResolver" />
711                         </source>
712                         <source select="name/what/unit">
713                                 <resolve resolver="unitResolver" />
714                         </source>
715                 </slot>
716                 <slot name="location">
717                         <source select="name/where/deictic">
718                                 <resolve resolver="mouseLocationResolver" />
719                                 <resolve resolver="eyeLocationResolver" />
720                         </source>
721                         <source select="name/what/anaph">
722                                 <resolve resolver="mouseLocationResolver" />
723                                 <resolve resolver="eyeLocationResolver" />
724                         </source>
725                         <source select="name/where/location">
726                                 <resolve resolver="coordinateResolver" />
727                         </source>
728                 </slot>
729                 <slot name="name">
730                         <source select="name/name">
731                                 <resolve resolver="spellingResolver" />
732                         </source>
733                 </slot>
734                 <action language="javascript">
735                         if (frame.glyph)
736                                 application.api.nameGlyph(frame.glyph, frame.name);
737                         else if (frame.location)
```

```
738                                 application.api.nameLocation(frame.location, frame.n
739                         </action>
740                 </frame>
741 <!-- ************************************************* -->
742                 <frame name="mission_task" test="mission_task/task_type" uses="mission_task"
743                         <slot name="task">
744                                 <source select="mission_task/task_type/*">
745                                         <resolve resolver="taskTypeResolver" />
746                                 </source>
747                         </slot>
748                         <slot name="glyph">
749                                 <source select="mission_task/what/anaph" weight="1.0">
750                                         <resolve resolver="mouseObjectResolver" />
751                                         <resolve resolver="eyeObjectResolver" />
752                                 </source>
753                         </slot>
754                         <slot name="location">
755                                 <source select="mission_task/where/deictic">
756                                         <resolve resolver="mouseLocationResolver" />
757                                         <resolve resolver="eyeLocationResolver" />
758                                 </source>
759                                 <source select="mission_task/where/location">
760                                         <resolve resolver="coordinateResolver" />
761                                 </source>
762                         </slot>
763                         <slot name="priority">
764                                 <source select="mission_task/priority">
765                                 </source>
766                         </slot>
767                         <action language="javascript">
768                                 var missionController = application.api.missionController;
769                                 var unit;
770                                 if (frame.unit) unit = frame.unit.glyph.model;
771                                 else unit = application.api.activeRobot;
772                                 var task = new Packages.flatscape.robot.Task();
773                                 if (frame.task.equals("recon")) task.taskType = task.RECON;
774                                 if (frame.task.equals("map")) task.taskType = task.MAP;
775                                 if (frame.task.equals("destroy")) task.taskType = task.DESTR
776                                 java.lang.System.out.println("location = " + frame.location)
777                                 task.locationX = frame.location.x;
778                                 task.locationY = frame.location.y;
779                                 if (frame.priority) task.priority = frame.priority;
780                                 missionController.addUnitTask(unit, task);
781                         </action>
782                 </frame>
783
784
785                 <!-- ***************************************** -->
786                 <!-- The anonymous frame is a generic catch-all frame used in
787                      case of ambiguity. If more than one frame applies,
788                                  and no frame can be continued based on the previous
789                                  frame, this frame is instantiated. Basically,
790                                  this frame type has preference over the others.
791                                  Usually the only thing to be done is to send
```

```
792                          output to the natural language generator, possibly
793                          based on the contents of the slots.
794                          The main thing is that the dialog history is
795                          updated with the slots here; otherwise they would
796                          be lost.
797            -->
798         <frame name="" test="ellipsis">
799                 <slot name="location">
800                         <source select="ellipsis/where/deictic">
801                                 <resolve resolver="mouseLocationResolver" />
802                                 <resolve resolver="eyeLocationResolver" />
803                         </source>
804                         <source select="ellipsis/where/location">
805                                 <resolve resolver="coordinateResolver" />
806                         </source>
807                         <source select="move/where/named_location">
808                                 <resolve resolver="namedLocationResolver">
809                                         <element name="name"><values resolver="spellingReso
810                                 </resolve>
811                         </source>
812                 </slot>
813                 <slot name="glyph">
814                         <source select="ellipsis/what/anaph">
815                                 <resolve resolver="mouseObjectResolver" />
816                                 <resolve resolver="eyeObjectResolver" />
817                                 <resolve resolver="dialogObjectResolver" />
818                         </source>
819                         <source select="ellipsis/what/unit">
820                                 <resolve resolver="unitResolver" />
821                         </source>
822                 </slot>
823                 <action language="JavaScript">
824                         var text = '';
825                         if (frame.glyph) text += '<glyph />';
826                         if (frame.location) text += '<location />';
827                         session.speak('<ambiguous>' + text + '</ambiguous>');
828                 </action>
829         </frame>
830
831 <!-- ************************************************* -->
832 <!-- Resolvers are the access to other modalities, dialog context, etc. -->
833 <!-- These are used to resolve certain slots to other slots -->
834         <resolver id="eyeObjectResolver" class="edu.rutgers.caip.communicator.fusion.Geomet
835                 <!-- The time-offset paramater is user-dependent, not just application
836                         dependent. Therefore, it should be defined in some user profile,
837                         which isn't known until runtime. We need some form of run-time
838                         variables. Let's define:
839                          ${expression}                    - Expression substitution
840                         expression will be a JavaScript expression that we evaluate
841                         through the Bean Scripting Framework. We might prepend some code
842                         to make global variables such as session and application available.
843                 -->
844                 <param name="interval" value="-400;400" />
845                 <param name="x-offset" value="0" />
```

```
846                 <param name="y-offset" value="0" />
847                 <!--param name="radius" value="1.5cm" /-->
848                 <param name="radius" value="20" />
849                 <input name="points" contextprovider="eyeTracker" />
850                 <param name="locator" value="glyphs" />
851         </resolver>
852
853         <resolver id="mouseObjectResolver" class="edu.rutgers.caip.communicator.fusi
854                 <param name="interval" value="-400;400" />
855                 <param name="x-offset" value="0" />
856                 <param name="y-offset" value="0" />
857                 <param name="radius" value="1" />
858                 <input name="points" contextprovider="mouse" />
859                 <param name="locator" value="glyphs" />
860         </resolver>
861
862         <resolver id="eyeLocationResolver" class="edu.rutgers.caip.communicator.fusi
863                 <param name="interval" value="-400;400" />
864                 <param name="x-offset" value="0" />
865                 <param name="y-offset" value="0" />
866                 <param name="ignore" value="(0,0)" />
867                 <input name="points" contextprovider="eyeTracker" />
868         </resolver>
869
870         <resolver id="mouseLocationResolver" class="edu.rutgers.caip.communicator.fu
871                 <param name="interval" value="-400;400" />
872                 <param name="x-offset" value="0" />
873                 <param name="y-offset" value="0" />
874                 <param name="ignore" value="(-1,-1)" />
875                 <input name="points" contextprovider="mouse" />
876         </resolver>
877
878         <resolver id="coordinateResolver" class="edu.rutgers.caip.communicator.fusio
879         </resolver>
880
881         <resolver id="dialogObjectResolver" class="edu.rutgers.caip.communicator.dia
882                 <input name="dialog" contextprovider="_DIALOG" />
883                 <param name="slot" value="glyph" />
884                 <param name="encoding" value="object" />
885         </resolver>
886
887         <resolver id="distanceResolver" class="edu.rutgers.caip.communicator.robot.D
888         </resolver>
889
890         <resolver id="panAmountResolver" class="edu.rutgers.caip.communicator.robot.
891                 <param name="constants" value="small_amount => 30; medium_amount =>
892         </resolver>
893
894         <resolver id="tiltAmountResolver" class="edu.rutgers.caip.communicator.robot
895                 <param name="constants" value="small_amount => 5; medium_amount => 1
896         </resolver>
897
898         <resolver id="zoomAmountResolver" class="edu.rutgers.caip.communicator.robot
899                 <param name="constants" value="small_amount => 300; medium_amount =>
```

```
900         </resolver>
901
902         <resolver id="speedResolver" class="edu.rutgers.caip.communicator.robot.SpeedResolv
903                 <param name="distance" value="mm" />
904                 <param name="time" value="s" />
905                 <param name="constants" value="slow_speed => 100; medium_speed => 500; fast
906         </resolver>
907
908         <resolver id="rotationSpeedResolver" class="edu.rutgers.caip.communicator.robot.Spe
909                 <param name="distance" value="deg" />
910                 <param name="time" value="s" />
911                 <param name="constants" value="slow_speed => 100; medium_speed => 500; fast
912         </resolver>
913
914         <resolver id="colorResolver" class="edu.rutgers.caip.communicator.fusion.NameResolv
915                         <param name="encoding" value="color" />
916         </resolver>
917
918         <resolver id="spellingResolver" class="edu.rutgers.caip.communicator.language.Spell
919
920         <resolver id="affiliationResolver" class="edu.rutgers.caip.communicator.fusion.Name
921                 <param name="prefix" value="affiliation_" />
922         </resolver>
923
924         <resolver id="unittypeResolver" class="edu.rutgers.caip.communicator.fusion.NameRes
925                 <param name="prefix" value="unittype_" />
926         </resolver>
927
928         <resolver id="missionTypeResolver" class="edu.rutgers.caip.communicator.fusion.Name
929                 <param name="prefix" value="mission_" />
930                 <param name="translation-table" value="_; " />
931         </resolver>
932
933         <resolver id="taskTypeResolver" class="edu.rutgers.caip.communicator.fusion.NameRes
934                 <param name="prefix" value="task_" />
935                 <param name="translation-table" value="_; " />
936         </resolver>
937
938         <resolver id="unitsizeResolver" class="edu.rutgers.caip.communicator.fusion.NameRes
939                 <param name="prefix" value="size_" />
940         </resolver>
941
942         <resolver id="unitResolver" class="edu.rutgers.caip.communicator.fusion.ObjectResol
943                 <param name="locator" value="glyphs" />
944                 <param name="fields" value="affiliation/* => affiliation; unittype/* => uni
945                 <param name="resolvers" value="affiliation/* => affiliationResolver; unitty
946         </resolver>
947
948         <resolver id="unitsizeHistoryResolver" class="edu.rutgers.caip.communicator.dialog.
949                 <param name="slot" value="size" />
950                 <input name="dialog" contextprovider="_DIALOG" />
951         </resolver>
952
953         <resolver id="unittypeHistoryResolver" class="edu.rutgers.caip.communicator.dialog.
```

```
954                 <param name="slot" value="unittype" />
955                 <input name="dialog" contextprovider="_DIALOG" />
956         </resolver>
957
958         <resolver id="affiliationHistoryResolver" class="edu.rutgers.caip.communicat
959                 <param name="slot" value="affiliation" />
960                 <input name="dialog" contextprovider="_DIALOG" />
961         </resolver>
962
963         <resolver id="homeResolver" class="edu.rutgers.caip.communicator.robot.HomeR
964         </resolver>
965
966         <resolver id="namedLocationResolver" class="edu.rutgers.caip.communicator.ro
967                 <param name="locator" value="glyphs" />
968                 <param name="fields" value="name/* => name; 'text => " />
969         </resolver>
970
971
972         <contextproviders>
973
974                 <contextprovider id="eyeTracker" class="edu.rutgers.caip.communicato
975                         <param name="poll-interval" value="0" />
976                         <param name="buffer-size" value="500" />
977                         <contextprovider id="eyeClustering" class="edu.rutgers.caip.
978                                 <contextprovider id="eyeTrackerSource" class="edu.ru
979                                         <param name="in-port" value="COM1" />
980                                         <param name="out-port" value="COM2" />
981                                         <!--param name="in-baud" value="9600" /-->
982                                         <!--param name="out-baud" value="9600" /-->
983                                         <param name="cursor" value="on" />
984                                         <param name="emulate" value="off" />    <!--
985                                         <param name="bounds" value="bounds" />
986                                 </contextprovider>
987                         </contextprovider>
988                 </contextprovider>
989
990                 <contextprovider id="mouse" class="edu.rutgers.caip.communicator.fus
991                         <param name="poll-interval" value="100" />
992                         <param name="buffer-size" value="50" />
993                         <contextprovider id="mouseFilter" class="edu.rutgers.caip.co
994                                 <param name="inactive-timeout" value="1000" />
995                                 <param name="poll-interval" value="100" />
996                                 <contextprovider id="mouseSource" class="edu.rutgers
997                                         <param name="window" value="canvas" />
998                                         <param name="cursor" value="on" />
999                                         <param name="transform" value="flatscape-tra
1000                                </contextprovider>
1001                        </contextprovider>
1002                </contextprovider>
1003
1004        </contextproviders>
1005 </fusion>
```

Appendix E

Online Resources

I've made the following items available online, since that is a more convenient way of accessing them than on paper:

- JavaDoc documentation — `http://www.caip.rutgers.edu/~fflippo/apidoc`

- Online Literature — `http://www.caip.rutgers.edu/~fflippo/docs`

Bibliography

[1] Activmedia. http://www.activmedia.com, http://robots.activmedia.com.

[2] The Apache Jakarta Project. Bean Scripting Framework. http://jakarta.apache.org/bsf/.

[3] Apache Project. Xalan-java 2. http://xml.apache.org/xalan-j.

[4] Apache Project. Xerces-java. http://xml.apache.org/xerces-j.

[5] Apple. Mac OS - Speech technologies. http://www.apple.com/macos/speech.

[6] Srinivas Bangalore and Owen Rambow. Exploiting a probabilistic hierarchical model for generation. In *Proceedings of the 18th Conference on Computational Linguistics (COLING'2000)*, Saarbrucken, Germany, 2000.

[7] Lauren Baptist and Stephanie Seneff. GENESIS-II: A versatile system for language generation in conversational system applications. In *Proc. 6th International Conference on Spoken Language Processing*, Beijing, China, Oct 2000.

[8] Steve Bett. On the number of phonemes in the English language. http://www.unifon.org/number-of-phonemes.html, 2001.

[9] Mark Billinghurst. Put that where? Voice and gesture at the graphics interface. *ACM SIGGRAPH Computer Graphics*, 32(4), Nov 1998. http://www.siggraph.org/publications/newsletter/v32n4/contributions/billinghurst.html.

[10] Alan W. Black, Paul Taylor, and Richard Caley. *The Festival Speech Synthesis System — System documentation*, 1.4, for festival version 1.4.0 edition, Jun 1999.

[11] R. A. Bolt. "Put-That-There": Voice and gesture at the graphics interface. *Computer Graphics (SIGGRAPH '80 Proceedings)*, 14(3):262–270, Jul 1980.

[12] Lou Boves and Els den Os. Multimodal multilingual information services for small mobile terminals (MUST). Technical report, EURESCOM, 2001. EURESCOM published project result.

[13] Oregon Graduate Institute Center for Spoken Language Research. Cslu toolkit. http://cslu.cse.ogi.edu/toolkit/.

[14] CHCC current and past research. http://www.cse.ogi.edu/CHCC/Research/main.html.

[15] CMU Communicator. http://www.speech.cs.cmu.edu/Communicator.

[16] M. Cohen, H. Franco, N. Morgan, D. Rumelhart, V. Abrash, and Y. Konig. Combining neural networks and hidden Markov models. In *Proceedings of the DARPA Speech and Natural Language Workshop*, Harriman, NY, 1992.

[17] M.M. Cohen, D.W. Massaro, and Clark R. Training a talking head. In *Proceedings of IEEE Fourth International Conference on Multimodal Interfaces (ICMI'02)*, Pittsburgh, Pennsylvania, October 2002.

[18] P.R. Cohen, M. Johnston, D. McGee, S. Oviatt, J. Pittman, I. Smith, L. Chen, and J. Clow. Quickset: Multimodal interaction for distributed applications. *ACM International Multimedia Conference, New York: ACM*, pages 31–40, 1997.

[19] Colorado University. *Phoenix Parser User Manual*. http://communicator.colorado.edu/phoenix/Phoenix_Manual.pdf.

[20] Alistair Conkie. A robust unit selection system for speech synthesis. *Joint Meeting of ASA/EAA/DAGA*, Mar 1999.

[21] The CU Communicator. http://communicator.colorado.edu.

[22] Hercules Dalianis. ASTROGEN - Aggregated deep and Surface naTuRal language GENerator. http://www.dsv.su.se/~hercules/ASTROGEN/ASTROGEN.html, 2002.

[23] Department of Speech, Music and Hearing, Royal Institute of Technology, Stockholm, Sweden. Multimodal speech. http://www.speech.kth.se/multimodal.

[24] J. Dowding, J. Gawron, D. Appelt, J. Bear, L. Cherny, R. Moore, , and D. Moran. Gemini: A natural language system for spoken-language understanding. *Communications of the ACM*, 39(1):51 – 87, 1993.

[25] John Dowding, Robert Moore, Francois Andry, and Doug las Moran. Interleaving syntax and semantics in an efficient bottom-up parser. In *Proc. of the 32nd Annual Meeting of the Association for Computational Linguistics*, pages 110 – 116, New Mexico State University, Las Cruces, New Mexico, 1994.

[26] Giovanni Rimassa Fabio Bellifemine, Agostino Poggi. JADE — a FIPA-compliant agent framework. Technical report, CSELT, 1998.

[27] Paul Fernandes. Autonomous robots and their software architecture. Master's thesis, Rutgers University, 2002.

[28] Edward A. Filisko. A context resolution server for the GALAXY conversational systems. Master's thesis, Massachusetts Institute of Technology, 2002.

[29] The Foundation for Intelligent Physical Agents. `http://www.fipa.org`.

[30] FIPA abstract architecture specification. Technical report, FIPA, December 2002.

[31] Arne John Glenstrup and Theo Engell-Nielsen. Eye controlled media: Present and future state, Jun 1995. Bachelor's Thesis.

[32] D. Goddeau, H. Meng, J. Polifroni, S. Seneff, and S. Busayapongchai. A form-based dialogue manager for spoken language applications. In *Proc. ICSLP '96*, volume 2, pages 701–704, Philadelphia, PA, 1996.

[33] Allen L. Gorin, Alicia Abella, Tirso Alonso, Giuseppe Riccardi, and Jeremy H. Wright. Automated natural spoken dialog. *Computer Magazine*, Apr 2002. `http://www.research.att.com/~algor/hmihy/papers/computer_magazine.pdf`.

[34] Andrew J. Hunt and Alan W. Black. Unit selection in a concatenative speech synthesis system using a large speech database. In *Proc. ICASSP-96*, Atlanta, GA, May 1996.

[35] Hyperdictionary.

[36] Koji Iwano, Satoshi Tamura, and Sadaoki Furui. Bimodal speech recognition using lip movement measured by optical-flow analysis. In *Proceedings International Workshop on Hands-Free Speech Communication*, pages 187 – 190, Kyoto, Japan, 2001.

[37] E. Kaiser. Robust, finite-state parsing for spoken language understanding. In *Student Session Proceedings of ACL '99*, College Park, Maryland, Jun 1999.

[38] N. Krahnstoever, S. Kettebekov, M. Yeasin, and R. Sharma. A real-time framework for natural multimodal interaction with large screen displays. In *Proc. of Fourth Intl. Conference on Multimodal Interfaces (ICMI 2002), Pittsburgh, PA, USA*, Oct 2002.

[39] A. Krebs, M. Ionescu, B. Dorohonceanu, and I. Marsic. The DISCIPLE system for collaboration over the heterogeneous web. In *Proceedings of the 36th Hawaiian International Conference on System Sciences (HICSS-36)*, Jan 2003. 10 pages/CD-ROM.

[40] Marcus Eduardo Markiewicz and Carlos J.P. Lucena. Object oriented framework development. *ACM Crossroads*, 2001.

[41] I. Marsic, A. Medl, and J.L. Flanagan. Natural communication with information systems (invited paper). *Proceedings of the IEEE*, 88(8):1354 – 1366, Aug 2000.

[42] D. L. Martin, A. J. Cheyer, and D. B. Moran. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1-2):91–128, Jan – Mar 1999. http://www.ai.sri.com/~cheyer/papers/oaa.pdf.

[43] Steven McCanne and Van Jacobson. vic: A flexible framework for packet video. In *Proceedings of the ACM Multimedia Conference*, pages 511–522, 1995.

[44] Michael McTear. Modelling spoken dialogues with state transition diagrams: experiences with the cslu toolkit. In *Proceedings of the 5th International Conference on Spoken Language Processing (ICSLP-98)*, pages 1223 – 1226, Sydney, Australia, 1998.

[45] Michael F. McTear. Spoken dialogue technology: enabling the conversational interface. *ACM Computing Surveys*, 34(1):90 – 169, Mar 2002.

[46] Microsoft. Microsoft foundation class library. http://msdn.microsoft.com/library/en-us/vcmfc98/html/mfchm.asp.

[47] Microsoft. Mipad: Speech powered prototype to simplify communication between users and handheld devices, May 2000. Press release.

[48] Sun Microsystems. Java media framework. http://java.sun.com/products/java-media/jmf/.

[49] MIT Spoken Language Systems Laboratory. The MIT TINA system. http://www.sls.lcs.mit.edu/TINA.html.

[50] MITRE. Galaxy communicator open source toolkit. http://communicator.sourceforge.net/sites/MITRE/distributions/OSTK-20021004, Oct 2002.

[51] Gordon E. Moore. Cramming more components into integrated circuits. *Electronics*, 38(8), Apr 1965.

[52] Nelson Morgan. What's new in government-sponsored speech recognition research. *Speech Technology Magazine*, 2002.

[53] Jeremy Moskowitz. Speech recognition with Windows XP. http://www.microsoft.com/windowsxp/expertzone/columns/moskowitz/02september23.asp, Sep 2002.

[54] Mourad Mouzit, George Popescu, Grigore Burdea, and Rares Boian. The rutgers master ii-nd force feedback glove. In *Proceedings of IEEE VR 2002 Haptics Symposium*, Orlando, Florida, Mar 2002.

[55] NetByTel.com, Inc. History of speech recognition. http://www.netbytel.com/literature/e-gram/technical3.htm.

[56] NTNU. Spoken dialog systems for telephone services. http://www.tele.ntnu.no/projects/spodis/.

[57] Department of Defense. Common warfighting symbology. Department of Defense Interface Standard MIL-STD-2525B, Department of Defense, Jan 1999.

[58] OGI Center for Human-Computer Communication. The adaptive agent architecture. `http://chef.cse.ogi.edu/AAA`.

[59] Oviatt. Designing robust multimodal systems for diverse users and environments. In *Workshop on universal accessibility of ubiquitous computing: providing for the elderly*, 2001.

[60] S. Oviatt, P. Cohen, L. Wu, J. Vergo, L. Duncan, B. Suhm, J. Bers, T. Holzman, T. Winograd, J. Landay, J. Larson, and D. Ferro. Designing the user interface for multimodal speech and pen-based gesture applications: state-of-the-art systems and future research directions. *HCI2000*, 15:263–322, 2000.

[61] Shimei Pan and Kathleen R. McKeown. Integrating language generation with speech synthesis in a concept to speech system. `http://www.ldc.upenn.edu/acl/W/W97/W97-1204.pdf`, 1997.

[62] B. Pellom, W. Ward, J. Hansen, K. Hacioglu, J. Zhang, X. Yu, and S. Pradhan. University of colorado dialog systems for travel and navigation. In *Proceedings of the Human Language Technology Conference (HLT-2001)*, San Diego, Mar 2001.

[63] James Poniewozik. Will smell-o-vision replace television? `http://www.time.com/time/reports/v21/tech/mag_smell.html`.

[64] Owen Rambow, Srinivas Bangalore, and Marilyn Walker. Natural language generation in dialog systems. In *Proceedings of the First International Conference on Human Language Technology Research (HLT2001)*, San Diego, USA, 2001.

[65] A. Rudnicky, E. Thayer, P. Constantinides, C. Tchou, R. Shern, K. Lenzo, W. Xu, and A.Oh. Creating natural dialogs in the carnegie mellon communicator system. *Proceedings of Eurospeech*, 4:1531–1534, 1999.

[66] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. Rtp: A transport protocol for real-time applications. Request for Comments — Standards Track RFC3550, The Internet Engineering Task Force, July 2003.

[67] Stephanie Seneff, Ed Hurley, Raymond Lau, Christine Pao, Philipp Schmid, and Victor Zue. Galaxy-II: A reference architecture for conversational system development. In *Proceedings of ICSLP '98*, pages 931–934, Nov 1998. `http://www.sls.lcs.mit.edu/sls/publications/1998/icslp98-galaxy.pdf`.

[68] Stephanie Seneff and Joseph Polifroni. Dialogue management in the Mercury flight reservation system. Presented at Satellite Dialogue Workshop, ANLP-NAACL, Apr 2000.

[69] A. Shaikh, S. Juth, A. Medl, I. Marsic, C. Kulikowski, and J. Flanagan. An architecture for multimodal information fusion. In *Proceedings of the Workshop on Perceptual User Interfaces*, pages 91 – 93, Banff, Alberta, Canada, Oct 1997.

[70] Ronnie W. Smith. Practical issues in mixed-initiative natural language dialog: An experimental perspective. In *Proceedings of the 1997 AAAI Spring Symposium on Computational Models for Mixed Initiative Interaction*, pages 158–162, March 1997.

[71] Spire Controls. Touch screen technology. http://www.spirecontrols.com/touch-screen-technology.htm.

[72] SRI Artifical Intelligence Center. OAA® news. http://www.ai.sri.com/~oaa/news.html.

[73] Sun Microsystems. Java speech API. http://java.sun.com/products/java-media/speech/.

[74] Sun Microsystems. JNI — Java Native Interface. http://java.sun.com/products/jdk/1.2/docs/guide/jni/.

[75] Sun Microsystems. Java speech grammar format specification. http://java.sun.com/products/java-media/speech/forDevelopers/JSGF/, Oct 1998.

[76] Sun Microsystems. http://java.sun.com/products/java-media/speech/forDevelopers/JSML, 1999.

[77] Sun Microsystems. http://www.w3.org/TR/jsml, 2000.

[78] Paul Taylor and Amy Isard. Ssml: A speech synthesis markup language. *Speech Communication*, 21(1–2):123–133, 1997.

[79] The Darpa Communicator Team. Galaxy communicator 4.0 documentation. http://fofoca.mitre.org/sites/MITRE/distributions/GC4point0.pdf, 2002.

[80] D. Toledano, S.B. Wang, S. Cyphers, and J. Glass. Extending the galaxy communicator architecture for multimodal interaction research. submitted to ACM Trans. on Human-Computer Interaction, Aug 2002.

[81] Trolltech. Qt 3.1 whitepaper. http://www.trolltech.com/products/qt/whitepaper/qt-whitepaper.html.

[82] University of california at santa cruz - perceptual science lab. http://mambo.ucsc.edu.

[83] University of Bremen. KPML. http://www.fb10.uni-bremen.de/anglistik/langpro/kpml/README.html, 2003.

[84] University of Edinburgh Centre for Speech Technology Research. Festival. http://www.cstr.ed.ac.uk/projects/festival/.

[85] R.J. van Vark, J.P.M. de Vreught, and L.J.M. Rothkrantz. An automated speech processing system for public transport information services. *3rd International International Congress on Information Engineering*, pages 212–221, 1997. ftp://ftp.kbs.twi.tudelft.nl/pub/alparon/publications/1997/R.J.vanVark-ICIE-97.ps.gz.

[86] W3C. XSL transformations. `http://www.w3c.org/TR/xslt`.

[87] M. Walker, J. Aberdeen, J. Boland, E. Bratt, J. Garofolo, L. Hirschman, A. Le, S. Lee, S. Narayanan, K. Papineni, B. Pellom, J. Polifroni, A. Potamianos, P. Prabhu, A. Rudnicky, G. Sanders, S. Seneff, D. Stallard, and S. Whittaker. Darpa communicator travel planning systems: the June 2000 data collection. In *Proceedings from EUROSPEECH*, pages 1371–1374, 2001. `http://www.ai.sri.com/~communic/euro-eval7.doc`.

[88] Tom Weston. Voice recognition technology. `http://florin.stanford.edu/~t361/Fall2000/TWeston/home.html`, Nov 2000.

[89] Jacek C. Wojdel Pascal Wiggers and Leon J.M. Rothkrantz. An audio-visual corpus for multimodal speech recognition in Dutch language. In *Proceedings of the International Conference on Spoken Language Processing (ICSLP2002)*, pages 1917 – 1920, Denver CO, USA, 2002.

[90] Pascal Wiggers and Leon J.M. Rothkrantz. Integration of speech recognition and automatic lip-reading. In *Proceedings of the Fifth International Conference on Text, Speech and Dialogs (TSD2002)*, pages 205 – 212, Brno, Czech Republic, Sep 2002.

[91] Pascal Wiggers, Jacek C. Wojdel, and Leon J.M. Rothkrantz. Medium vocabulary continuous audio-visual speech recognition. In *Proceedings of the International Conference on Spoken Language Processing (ICSLP2002)*, pages 1921 – 1924, Denver CO, USA, Sep 2002.

[92] Lizhong Wu, Sharon L. Oviatt, and Philip R. Cohen. Multimodal integration — a statistical view. *IEEE Transactions on Multimedia*, 1(4):334 –3 41, 1999.

[93] Dongsuk Yuk. *Robust Speech Recognition using Neural Networks and Hidden Markov Models*. PhD thesis, Rutgers, The State University of New Jersey, 1999.