Abstract

Today, user interfaces normally consist of a screen, and a pointing device and a keyboard for input. However, as more advanced technology and methods appears, there should be good chances to utilize these for more natural and effective human-computer interfaces. The main motivation is to get a more natural and easy to use interface, and the computer should understand the user without too much effort from the user. Intelligent interfaces could be a solution to achieve this goal.

The main focus in this thesis, is multimodal input which combines different input modalities to achieve the user's goal. A framework has been designed where the user has the possibility to change between input modalities. The system should integrate the information given in different input modalities to one joint meaning. In this architecture, input could either be location or command input, and different modalities could be used for each input type. The example described later on in this thesis combines either speech or written text as command input, with either map input or physical position for location input.

An agent-based blackboard architecture are used for collecting input. Agents collect information directly from the user. Each agent represent their own input modality, and is responsible to analyse input. As this is done, the agent send the information to a common blackboard which hold the latest information from each agent. An own agent which is responsible for fusing this information to one common meaning, collects the information from the blackboard and integrate it to one joint meaning. This joint interpretation decides what should be done to which object.

Since the modalities are independent of each other, other modalities could easily be added with just small changes to other parts of the system as long as it is an command or location input which agrees to the currently representation structure.

Contents

List of Figures

Definition of Terms

Agent	An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors.[25]
Blackboard	a central global workspace with a collection of agents which act upon it
DARPA	Defense Advanced Research Projects Agency
Frame	A data structure for representing situations in Artificial Intelligence
FIPA	Foundation for Intelligent Physical Agents, standard for agent-based systems
GUI	Graphical User Interface
HMM	Hidden Markov Model: Modern speech recognition systems are generelly based on this model
IBM ViaVoice	Speech recognition software offered by IBM
ISO	International Organization for Standardization
JavaSpace	A part of the JINI architecture with the possibilities of exchange Java objects
JADE	Java Agent DEvelopment Framework for developing MAS. Compatible with the FIPA standard
JINI	Jini Is Not Initials: a network architecture for the construction of distributed systems
JSAPI	JavaSpeech API: API provided by Sun to easily develop speech-based application
MAS	Multiagent software
Multimodal integration	Integration of inputs from various senses to form a multimodal representation
RMI	Remote Method Invocation

Chapter 1

Introduction

This work is about multimodal interfaces, and the main focus is on multimodal input. Multimodal interfaces combines two or more input modes from the user. There has been a growing interest in this field because of the advantages it provides. In this thesis I will present some thoughts and theories about this topic and I will present a prototype of an agent-based multimodal system.

1.1 Motivation

An important aspect in interaction design, is usability. ISO 9241 defines usability as: "The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use." Focus on the user and good usability is very important as the benefit of researching this area are many. Good interaction design reduce the users frustration rate, and the tasks can be more focused on. An important factor of motivation behind the work of this thesis, is to improve usability by using multimodal input over unimodal or traditional interfaces. Multimodal interfaces could include important advantages as flexibility, availability, adaptability, efficiency, lower error rate, and more intuitive and natural interaction.

The most common types of input when discussing multimodal interfaces include the traditional devices as keyboard, mouse, pen and touch screen, and more advanced input as speech recognition, 2D and 3D gesture recognition, lip movement and gaze tracking. During the 1980's and early 90's, the most common method of making multimodal systems was by adding speech to the more traditional direct-manipulation interfaces. During the last decade a new class of multimodal systems has occurred with major improvements in new input technologies and algorithms, hardware speed, distributed computing, and spoken language technology. In this thesis I will show some examples of systems designed during this period, and how the different input modes are combined.

1.2 Objectives and requirements

The main goal with this work is to develop an architecture of an agent-based system for multimodal input, and to prototype a part of it. The system is based on a blackboard model where each modality works on their own, and sends analysed information to the blackboard as soon as accepted input is detected and finalized. A fusion agent is used to integrate the multimodal information to one specific meaning.

The following requirements are defined:

R1: The different input modalities should work independently of each other.

R2: Sharing of input information.

R3: A common representation structure of the input from the different modalities.

R4: Multimodal integration of at least two different input modalities.

R5: Mobility

1.3 Structure

The structure of this thesis is divided into 7 chapters which are as follows:

- Chapter 2 introduces general theory of important research issues in this framework, primary software agents and intelligent interfaces.
- Chapter 3 describes previous research and systems that influenced this work.
- Chapter 4 describes this architecture.
- Chapter 5 describes the current implementation and which modalities that are chosen for the prototype.

- Chapter 6 shows a test run of the current implementation situated in the implemented example.
- Chapter 7 gives a conclusion of this work, and suggestions for future work.

Chapter 2

Relevant study

2.1 Agents

The term "agent" is outside the computer world often used when describing people helping us with something, like for example a travel agent. Agents like this has a job where they should act autonomously to fulfill our goals, and then reduce our workload. Software agents have the similar goal, to help the user to be more productive. When using this terminology in user-interfaces, the system can act for example as a butler or a secretary.

2.1.1 Intelligent agents

There are several definition of the term agent, but no universally accepted definition. Russel&Norvig has a general definition which says "an agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors."[25] Wooldridge defines an agent as "a computer system that is situated in some environment, and is capable of autonomous action in this environment in order to meet its design objectives." [36] An agent do not have to be intelligent or to cooperate with others, but for an agent to be intelligent it should have skills as being reactive, proactive and social. A reactive agent are able to react to changes in its environment. A proactive agent has a goal-directed behaviour and are able to take the initiative when appropriate. Social agents are able to interact with other agents to accomplish their tasks and to achieve the complete goal.

Nwana points out some reasons to the problem on agreeing on a consensus definition for the term "agent"[19]. "Agents researchers do not own this term" and "agent is an umbrella term for a heterogeneous body of research and development". They also identify 7 types of agents which are defined by what they do and what technology underpins them. Using these agent types defines an agent as collaborative, interface, mobile, information/internet, reactive, hybrid and smart.

Lieberman [15] see it from an user-interface perspective and has a definition of an intelligent interface agent. He breaks up this definition in three parts. The short version of the term computer intelligence is that the computer exhibit behaviour that is like what we call intelligence in people. By this he do not mean that the interface should have human-level intelligence, but have some characteristic of human intelligence. This could be either reasoning, domain-knowledge or procedures which do useful tasks. He defines the term interface agent as an agent which communicates directly with the user. An interface agent observes the users actions and behaviour and can take actions on behalf of the user. When he describes the term agent, he points out the different definitions of an agent, but he like to focus on the function of the agent. An agent should have an assistive role to the user, and its job is to help the user in succeeding it's goals.

Context plays an important role for interface agents. Lieberman defines context as everything the user do not explicitly tell the system, and a context-aware application takes its decision based on information other than what it gets explicitly from the user. People are sensitive to context as we often take action which suits the context. Context-aware agents may sense their additional input from the environment using sensory input.

Agents and multiagent systems seems useful when building systems where data, control, expertise and/or resources are distributed. Agents will give a natural metaphor for these kinds of systems, where you have several participants which knows what it can do and act in cooperation with others.

2.1.2 Collaborating software

Collaborating software systems has by AI researchers been a way to solve large and difficult problems for a long time. This is an effective divide-and-conquer approach for complex software applications, where a number of smaller software modules are applied to form a complete system. Blackboard systems were the first attempt made at integrating cooperating software modules with the goal of achieving the flexible, brainstorming style of problem solving used by a group of humans working together on a problem. The multiagent approach is another approach to get the same effect[7].

According to Corkill[7], there are some key challenges to think about when creating effective collaboration software. These include:

- Representation - how to get software modules to understand each other.

- Awareness - make modules aware when something which involves them happens.
- Investigation - help modules to quickly find relevant information.
- Interaction - create modules that are able to use work done by others while working on a shared task.
- Integration - combine result produced by other modules.
- Coordination - get the modules to focus on right thing in right moment.

Multiagent cooperation

Coordination of the agents in a system is important to get the agents to reach the overall goal. Because of the distributed expertise, there is a need to coordinate everyone to prevent chaos and to make the system more efficient. Usually the different agents work toward a common goal, and therefore there is no conflict between them. The individual agents objectives does not matter, only the overall system. This is what Wooldridge [36] mean about "benevolence assumption". In contrast some agents are self-interested. These type of agents have goals that will be in conflict with other agents. However, they still need to cooperate and it is important to find the best way to cooperate.

Coherence and coordination are two issues that need to be considered to decide how successful a multiagent system are. Coherence is the ability of a system to behave as a unit. Coherence is "measured in terms of solution quality, efficiency of resource usage, conceptual clarity of operation, or how well system performance degrades in the presence of uncertainty or failure" [36]. Coordination is "a process in which agents engage in order to ensure their community acts in a coherent manner"[20]. In a perfectly coordinated system agent do not need to bother about others sub-goal while achieving a common goal.

There are several ways different agents can work together to solve problems. Contracting is one solution to coordinate agents to work together. By using the contract net protocol standardized by FIPA[29], the agents can cooperate by sharing tasks. A manager announces the problem to the other agents. As the agents listen to announcements and evaluate them with respect to their own resources, they place a bid if they find a suitable task. Several agents can bid for the same task, then the manager has to decide from the information of the bid which agent should win the bidding round and then will be awarding the contract.

Another way to let agents cooperate to solve a problem, is result sharing. This will typically be that each agent solve small problems which later on will be become larger solution. Result sharing is when agents may share information relevant to their sub-problems. Durfee, referred to by [36], has suggested 4 ways to improve group performance:

- **Confidence:** When independently derived solutions can be cross-checked, the confidence in the overall solution is increased.
- **Completeness:** Agents that share their local views to achieve a better overall global view.
- **Precision:** The precision of the overall solution is increased when agents share results.
- **timeliness:** As several agents work on the solution, the result could be derived more quickly.

Blackboard systems

Blackboard systems is another method to coordinate a problem solving process. Blackboard systems were first developed in the early 1970's. A blackboard system consist of three main components: Knowledge sources, blackboard, and a control component. The knowledge sources are the individual modules that contain the expertise, the blackboard are a shared working memory and a communication medium, and the control component makes runtime decisions about the course of the problem solving. When using this method, the solution evolves step by step from the information available on the blackboard.

In this prototype, a space implementation is used, and this will be discussed in detail in later chapters. By using a space implementation of a blackboard, the participants do not have to know anything about each others, they do not need to share the same process or machine, and they are temporally independent of each other.

The space model first appeared in the early eighties when Gelernter and his co-workers made the Linda programming system and their tuple-space model [8]. I will give an example of an implementation based on the tuple-space model called JavaSpaces[32].

JavaSpaces is based on Java RMI and JINI, and is a framework to exchange distributed objects. JavaSpaces is used to implement a blackboard model where the JavaSpace is the blackboard itself, and the different agents are the knowledge sources.

Jini is a distributed computing environment, and can be used in any network where there is some change. Several clients and services communicates by using the Jini-protocol. A Jini system consists of services, clients and lookup services. A lookup service act as locators between services and clients.

A JavaSpace is a shared, network-accessible repository for objects. The agents uses this space for object storage and to exchange these objects. The objects that are stored on the space is referred to as entries. An entry is a Java object with the advantages that follows. There are four primary operations in JavaSpaces:

- **Write** an entry to the JavaSpace.
- **Read** an entry from a JavaSpace.
- **Take** an entry from a JavaSpace.
- **Notify** an object when specific entries are written to the JavaSpace.

When reading or taking entries from the space, simple value-matching lookup is used to find the relevant entries. These entries are just passive data, and it is not possible to modify them. In the case of modifying, the agent has to remove the old one, update it and write it to the space once more. Notify uses a listener that can be specified to listen for specific entries. When one such entry is written to the space, the listener notifies, and a specific action takes place.

An advantage with using JavaSpaces, is that the knowledge sources are loosely coupled. The interaction is through the space, and not directly between each of them. The senders and receivers are then not required to know each others identity, or not even have to be active at the same time.

2.1.3 Multi-Agent architectures

When several agents working together to solve a problem, they need to communicate. Agents do this by sending messages via a central hub, where messages are structured according to some high-level representation and are transported using TCP/IP. Some architectures are made, and they provide a transport layer or infrastructure on which multiagent applications can be built. I will now present some common multiagent architectures, and since Jade is the one which is used in this project, this will be presented more detailed than the others.

Galaxy Communicator

The Galaxy Communicator[30] was developed by MIT and was funded by the Defense Advanced Research Projects Agency(DARPA). Galaxy is an open source distributed, message-based infrastructure optimized for dialogue system design, and was designed to support the creation of speech-enabled interfaces that could scale across modalities.

Open Agent Architecture(OAA)

Open Agent Architecture(OAA)[16] is SRI International's distributed software architecture. OAA could be used for a several types of applications, but are generally for distributed problem solving. Each entity in OAA is called agent, and these agents work together with the user to reach a goal. The communication is controlled by a central facilitator, and is done using the specialized language ICL(Inter-agent Communication language).

Adaptive Agent Architecture(AAA)

AAA[28] is a bit different from the previous two in that they have a central hub or facilitator, this could have a team of facilitators or brokers. This could be an advantage in greater robustness in facilitator failure. AAA is also backwards compatible with OAA, which means that any OAA system could run using AAA facilitator.

Jade(Java Agent DEvelopment Framework)

Jade [31] is a software framework designed for developing Multi Agent Software. It is built on Java, and is designed to simplify agent development. The Jade-Board was founded in march 2003 and is a non-profit organization and has the intention to promote and work to make it a de-facto standard. The source code is Open Source and distributed under LGPL(Lesser GNU Public License).

A definition of JADE is: "Jade is an enabling Technology, a middle ware for the development and run-time execution of peer-to-peer applications which are based on the agents paradigm and which can seamless work and interoperate both in wired and wireless environment"[1]. From this we can conclude that Jade has a conceptual model consisting of two major parts, a distributed system with peer-to-peer networking, and software component architecture with agent paradigm. The first part is about how the components are linked together, and is seen as a peer-to-peer model. The different peers is called agents in Jade. The second part is what each component is expecting from the others. As it is based on the agent paradigm, it has the agent properties described in the previous subsection. JADE is in fully compliance with the FIPA-specification [29], and make it therefore easy to implement agents using the FIPA standard, which also means that Jade agents can interoperate with agents built on other agent frameworks as long as they are built on the same standard. FIPA was formed in 1996 to produce software standards for heterogeneous and interacting agents and agent based systems. It is a pure interface specification which specifying both the communication language and the semantics of the messages used in the communication. The messages are written in an agent communication language(ACL), the content is in a content language. Since it is a interface specification, the implementation details are left up to the implementor.

Jade has several advantages and is a middleware that simplifies developing applications. When developing distributed applications with autonomous entities, Jade is a framework that hides the complexity so the developer can focus on the logic of the application. Each agent in Jade control their own thread of execution, and can be programmed to initiate execution on the basis of goal and state changes, called pro-activity.

2.2 Intelligent user interfaces

User interfaces today mostly offers input from mouse and keyboard. This are often known as direct manipulation interfaces, and makes use of graphical user interfaces and menus, and the users are presented some objects and a set of discrete actions to perform on them. Electronical pen devices has recently become popular, and is also a tool for direct manipulation interfaces.

As mentioned earlier, designing human-computer interfaces with focus on usability should make them more efficient to use, easier to learn, and more satisfying to use. Next generation interfaces will in addition be "intelligent", and will provide benefits to the users as adaptivity, context sensitivity, and task assistance. Intelligent user interfaces are [17] "human-machine interfaces that aim to improve the efficiency, effectiveness, and naturalness of human-machine interaction by representing, reasoning, and acting on models of the user, domain, task, discourse, and media". As traditional interfaces, these should be learnable, usable, and transparent, and in addition provide advantages so the user could enhance interaction.

As traditional interfaces support sequential and unambiguous input from input devices as mentioned(keyboard and pointing devices), these constraint are not that important in intelligent user interfaces. A broader range of input devices are usual, for example recognition based input as speech, eye tracking and gestures. These type of interfaces support asynchronous, ambiguous and inexact input by using a more sophisticated input analysis.

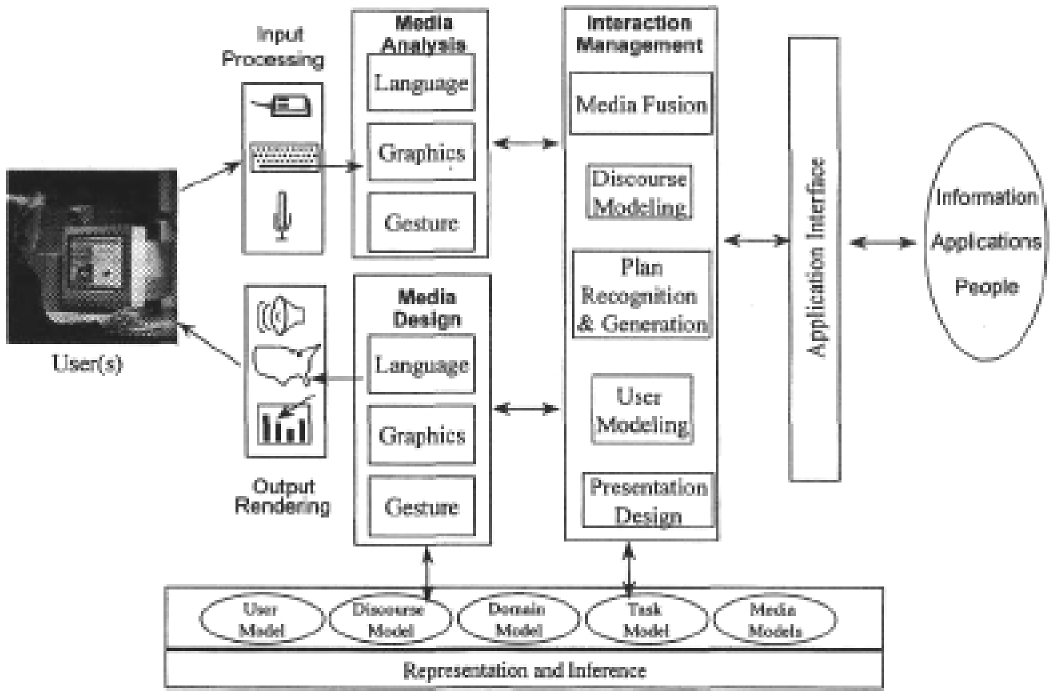


Figure 2.1: Architecture of Intelligent User Interfaces[17]

Intelligent interfaces research consist of a few main areas such as input analysis, generation of coordinated output, and modeling. Figure 2.1 illustrates an architecture of intelligent user interfaces. In this thesis, the main focus is in input, and I will discuss some methods for analysing and fusing multimodal input in the following section about multimodal interfaces. In the generation phase, the planning and realization of the output will be done by using analysed input and different models. These models could for example be of user, discourse, task, and situation and interaction management.

2.3 Multimodal interfaces

I will start this section by defining the term multimodal. This word is built from two words, multi and mode. Mode is here a physical sense that is used in communication. Humans have five senses in sight, sound, touch, taste, and smell, were the last two are not so important in communication.

For human-computer interfaces, the focus therefore are on sound, vision and touch. In addition an extra more abstract sense knowledge could also be used as a mode. This knowledge sense is external knowledge which could be useful, and for computers this could for example be context information as location [4]. The word ”multi” means that the machine receives input from multiple modes. The word multimodal should refer to modalities regardless of nature, but many researcher use the word when referring to modalities commonly used in communication between peoples, and multimodal usually refers to user input.

As the importance of computers being more intelligent increases, we like to communicate with the computer rather than operating it. When humans communicates, we usually get the information from several modes which could be a combination of spatial and semantic knowledge. A multimodal system coordinate a combination of two or more user input modes. Speech is the most important form of communication between humans, but other modes as hand gestures and facial gestures is important to interpret the meaning in human-human communications. Identical spoken words could have different meanings when combined with different gestures. These sort of interfaces are seen as more natural, and allows the user to concentrate on the tasks instead of how to control the interface.

The interest in multimodal interface design has been growing recently, and is inspired by the goal of more flexible, efficient, and powerfully expressive means of human-computer interaction. It is expected that multimodal user interfaces will be easier to use and learn than traditional interfaces. A system with human like sensors and sensors on the surrounding physical environment, will have the capability to adapt to the user, task and environment in an intelligent manner.

Input to GUIs is atomic and certain while input as speech is uncertain. That means that any system is probabilistic, which again means that easy events now requires interpretation and can be misinterpreted. Another issue is that traditional interfaces is a sequence of events as keyboard and mouse clicks. Multimodal interfaces require simultaneously continuing input stream processing. Important challenges to build successful multimodal systems is to manage the problem with uncertainty of recognition and process parallel input from several modalities.

2.3.1 Input modalities

As defined, multimodal interfaces are user interfaces where two or more input modalities are combined. Several types of input modalities can be chosen, and I will discuss some possibilities here. Since speech recognition based systems is seen as a very important input source in multimodal systems, I will give a more detailed presentation of it in the next subsection.

Traditionally direct manipulation interfaces has been common for more than a decade. These usually combines a keyboard with a pointing device, normally a mouse, but pen/stylus input is more and more used lately. Direct manipulation interfaces has several positive qualities [5]. Communication is generally fast and concise and the input techniques are relatively easy to learn and remember. Another important strength with these interfaces, is that the user usually knows what can be done since the visual presentation of what is possible to do is easy to access. However, direct manipulation system also have some limitations, especially when the user try to access or describe entities that are not visualized. Natural language could be a way of reducing this problem.

Natural language interfaces have an advantage in describing entities not displayed on the screen, in specifying temporal relations between entities or actions, and in identifying members of sets. Common input modalities for natural language interfaces includes speech, typing and handwriting.

Natural language interfaces and direct manipulation interfaces has opposite strengths and weaknesses, and therefore seems to be very complementary modalities. That is why several multimodal systems combines these. I will present some of these systems in chapter 3.

The modalities needed to fully implement the example described in this thesis, is a speech recognizer, keyboard, pointing device, and a location aware modality. The latter modality would probably not hold the definition of a multimodal system, as it is not a direct user input, but more like a context-aware input.

Research on more advanced input modalities have been done, but these are not that much commonly used, at least not yet. Examples of these sort of input modalities include gestures, eye tracking. Both eye tracking and gestures could be used for deictic purposes, and could be seen as pointing devices. Gestures by for example using a 3D glove, could also be interpreted to action commands, were different movements have a specific meaning.

Vision based modalities are also widely researched. Lip movements are an example of a input modality which can be used together with speech. Visuals could be very useful to accomplish several tasks, and could be used to watch the users to identify by face recognition, expression analysis, eye tracking, body tracking, head and face tracking, hand tracking.

Recently systems have also started to combine them with more physiological input modalities like fingerprints. This is often referred to as multibiometric multimodal interfaces. These have the advantage of identify users by using physiological or behavioural features associated with that person. Currently identification methods include fingerprints, hand geometry, iris, retina, face, facial thermograms, signature, gait, palm print, and voice print[12].

2.3.2 Speech recognition

Since speech is the most important way to communicate by humans, there is extensive research in speech recognition systems in computer science. The motivations behind speech recognition systems could be to build hands-free system where users can give commands to the computer while using their hands to other things. It also opens new possibilities when designing systems for small units.

As the demand for smaller and portable computers increase, there will be smaller keyboards and screens on them, and this will decrease our ability to manipulate them. Speech will be a useful input medium for these systems, since speech do not need physical space, and this is one important motivation for speech interfaces

Speech theory

Research on speech recognition dates back to Bell in the 1870's as he wanted to create a machine that could visualize speech, the phonautograph. Even if he failed in this project, this lead to the inventing of the telephone[14]. The development went slow the next hundred years, but in the 1970's serious speech recognition research started. This research was mainly driven by the US government under its Defense Advanced Research Projects Agency(DARPA). Over a 5 years period starting in 1971, they provided $3 million to research every year in its Speech Understanding project. Speech recognition has become increasingly better from these years until now, and a number of products have become available. The goal is of course to get 100% accuracy, but this has not been reached yet, however it can be reached in very constrained cases, but usually the accuracy lies somewhere below.

Speech is a complex process in the human brain, and humans are the only species that have a natural language. Animals like parrots are only capable to mimic sound. When humans produce speech, there are some logical steps to do, and the language has a linguistic structure which consists of a few levels known as[10]:

- Phonetics: the sounds of speech.
- Phonology: the sound system of any particular language.
- Morphology: Word formation.

- Syntax: The combination of words into phrases and sentences.
- Semantics: the meaning of words, phrases and sentences
- Discourse: activities using language which extend beyond individual sentences, such as stories.

The first step is to form the idea we want to communicate, and then use the grammatical rules in the language to form the idea to phrases. Each phrase consist of words that will give meaning to the idea. Each word consists of morphemes, which again are created from phonemes. Phonemes are the smallest units in speech, while morphemes are the smallest units which carry meaning. Waves are produced from this speech, and these waves are received by the hearing mechanism of the receiver. This mechanism is so advanced that it can not be copied by todays technology. Humans have a very large number of brain neurons working in parallel which makes the human perceptual and cognitive systems very complex. It is thought that if the machine should be possible to get human performance in processing normal conversation, it need to have extensive linguistic knowledge and a high ability to simulate human intelligence. One solution is to simplify the task by constraining what can be said so speech can be used in many situations.

In speech technology there are two areas that correspond to speaking and hearing, speech synthesis and speech recognition. Speech synthesis is the process where speech is produced from digitized text. Speech recognition is the process where computers listen to spoken language and determine what has been said by processing audio input and converting it to text. In this thesis my focus is on input, which means I only work on recognition and not on synthesis.

The Hidden Markov Model is a popular technique to use for speech recognition. This approach was invented by Lenny Baum of Princeton University in the early 1970's and was shared with several ARPA (Advanced Research Projects Agency) contractors including IBM. It is s a complex mathematical pattern-matching strategy which became adopted by all leading speech recognition companies. It uses the probabilities of the user speaking a certain phoneme given the frequency spectrum at a certain segment in speech and the previous phoneme[24].

When to use speech

Since speech is so important in human-human communication, we tend to have extreme high expectations when using speech in human-computer interaction. This makes it a lot easier to get frustrated and disappointed if things do not work as expected. Because of this, speech should not be used at any time, and there should be a good reasons for using it. Some good reasons for using speech interfaces have been identified, and includes:

- No keyboard available. This could for example be when using small units.
- Users hands are occupied.
- Commands are embedded in deep structures.
- The users are unable to type, or they could be uncomfortable with it.
- Physical disability.

There exists times when it is not reasonable to use speech recognition systems. This includes when the task requires the user to speak with other people, the environment is very noisy, and it will be more easy to use a keyboard and/or mouse.

Important challenges

When designing speech recognition interfaces, there are some important challenges which is needed to be faced before the application could be robust. This lies in the features of speech.

- Speech is transient: Once you hear it, it is gone. This could be a problem with respect to the humans limited ability to remember. When working with graphical interfaces, the graphic typically stays on the screen until the user perform some action.
- Speech is invisible: In graphically user interfaces, the elements and the functionality of an application is visible to the user. This is a problem with speech recognition interfaces in which it is more difficult to indicate to the user what they should say to perform actions.
- Speech is asymmetric: Humans can produce speech easily, but can not listen that easily. Therefore humans speaks faster than they write, but listen more slowly than they read.
- Speech recognizers do mistakes: Recognizers are not perfect listeners. To build as robust system as possible, it is important that the designers know which errors that can occur, and what causes them. It is difficult to guide the users to fix some type of recognition errors, and it is important that the users are cooperative.

- Flexible vs accurate: Which combination is the best in flexible versus accurate systems? As more flexible a system is, the more combinations of words are possible to perform a command. Meanwhile you often will trade away some accuracy in the system by allowing several ways to say the same thing since the grammars will be more complex with several similar words.

Accuracy

The problem with speech recognition today, is that it has two major limitations: it does not fully transcribe free-form speech, and it makes mistakes. The first is in their constrain to grammars, and this will be discussed in a later section.

Recognition accuracy is often used as a measurement of speech recognizers reliability, and this is usually not good enough for us to trust it in many types of systems. Some major factors that influence the accuracy [33]:

- Accuracy usually higher in quiet environment.
- Quality of microphone and audio hardware.
- Users that speaks clearly get higher accuracy.
- Accents get lower accuracy.
- Higher accuracy with simpler grammars.
- Higher accuracy with less confusable grammars.

These factors should be considered when designing and using speech recognition systems, but probably there are still recognition errors. These falls into three categories:

- Rejection: The recognizer does not understand what the user say.
- Misrecognition: The recognizer return a word which is different to what the user said
- Misfire: The recognizer return a word when the user did not speak.

Recognition errors is a problem in all recognition-based interfaces, but there are ways of "repairing" errors. Repeating input is probably the preferred correction method in human-human dialogue, but in recognition-based interfaces repeating of input in the same modality does not eliminate the cause of recognition error. However, there are an alternative approach which seems promising by using repetition. This could be by switching modality for repetition, and correlating the correction input with repair context. If the primary input is speech, the user could switch to handwriting if an recognition error occur. Another approach to reduce recognition errors, is by using context information, and to eliminate alternatives from the recognition vocabulary that are known to be incorrect.

2.3.3 Speech recognizer

A typical speech recognizer has some major steps [33]:

- grammar design: The grammar defines the words which may be spoken by the users. It must be activated for the recognizer to know what to listen for.
- Signal Processing: Analyze the spectrum characteristics of incoming audio.
- Phoneme recognition: Compare the spectrum patterns to the phonemes in the specified language.
- Word recognition: Compare the likely phonemes against the words specified in the activated grammars
- Result generation: The information about recognized word is given to the application. It indicates the best guesses, but may also indicate alternative guesses.

These steps are mostly controlled automatic by the speech recognizer, and are beyond the application developers control. The applications control of the recognizer is through the grammar, and this is the way a developer can design how the system shall work.

When deciding the quality of a speech recognizer, there are some properties which should be investigated: The size of the vocabulary, whether training is needed, whether continuous or discrete speech, word error rate, whether speech processing is done real time or offline, and whether the recognizer performance is independent of the speakers gender and age. The ideal speech recognizer would then be a speaker independent, continuous recognizer with large vocabulary and low error rate. With todays speech recognizers, tradeoffs has to be made. If high accuracy is wanted, smaller vocabulary would be better. However, if both high accuracy and large vocabulary is needed, training is unavoidable.

JavaSpeech API

Sun Microsystems has designed Java Speech API(JSAPI) in cooperation with leading speech technology companies like Apple Computer, Dragon Systems, IBM Corporation, Novell, Philips Electronics, BV and Texas Instruments. The goal with this API was to make an easy way to design application based on speech technology. Sun only provide the API, while the implementations was created in cooperation with the speech technology companies.

Java Speech API provide a standard, easy to use cross platform interface to state-of-the-art speech technology, and support both speech recognition and synthesis. This API makes it easier to implement speech based interfaces in Java, and the developers will have access to state-of-the-art speech technology from leading companies. Since most existing speech technology applications is written in C and C++, speech vendors have implemented Java Speech API on top of their existing speech software by using Java Native Interface. Speech synthesizers and Speech recognizers can then easily be written in Java software, and therefore take advantage from the portability Java provide.

JSAPI use grammars to decide what to listen for in the input. Two basic grammar types are supported: rule grammars and dictation grammars. Rule grammars are most common today, and are defined by a set of rules. These rules use the specification of Java Speech Grammar Format(JSGF) [34]. In rule based systems the developer of the application design the rules to decide what the recognizer should listen for. These rules constrain the recognition process, which means that the work with designing the rules are very important to build good systems. The key is to design the rules that allow the users freedom of expression while still limiting what might be said to ensure quick and accurate recognition process. This type of system will make the recognition process more accurate and faster.

Dictation grammar makes the user free to speech almost what he wants, and this form of grammar is close to the ideal free-form speech, but it has several major disadvantages today. The most important is that it makes more errors. As from the application developer point of view, it's easier to implement a dictation grammar as much of the complexity is in the speech recognizer and not in the application, but as it is a more complex type of grammar it requires more computing resources.

JSAPI has a ResultListener which is the main listener. It is based on the same model as graphical user interfaces. The ResultListener defines what to should happen, after listening to events. The events are generated from the grammars defined in the application.

IBM ViaVoice is used for speech recognition in this implementation. The implementation of JSAPI made by Sun and IBM is called Speech for Java and is designed to work with ViaVoice.

ViaVoice is mainly used to create text from the users speech. It has a vocabulary offering over 300.000 words, which means there are possibilities of creation of almost any text [35]. ViaVoice creates a model of the user to increase accuracy. After some training the performance should be over 90 percent which could be quite satisfactory in several systems. As mentioned before, performance depends on other factors as hardware and the user, and that could be the reason why this performance sometimes is never reached.

2.3.4 Cognitive aspects of multimodal interfaces

The development of human-computer interfaces has historically been driven by the technology, and technologist have assumed that users can easily adapt to what is built. For the user to communicate with the system, he have to get instructions, training, and practice to fully take advantage of the capabilities. Interfaces based on natural behaviour as human speech, gaze, touch, and movements are recognition-based and are not under full conscious control. In these kind of systems it is not possible for even the most cooperative user to adapt the behaviour to the systems limitations. Both speech and complex gaze are vulnerable for recognition errors, and no matter how cooperative the user is, recognition errors could occur. This is a reason why it is important that multimodal interface design should considerate cognitive science research. Some of the most important cognitive science themes that are relevant to design multimodal interfaces will be presented in this subsection [23]:

User preferences: Users have a strong preference to interact multimodally before unimodally. Several observations across a variety of application-domains seems to document this, and as much as 95-100% preferred multimodally interaction when they were free to use either speech or pen input in a map-based spatial domain. Since each mode could have its strengths and weaknesses in different situations, and by giving the user a choice of which modality to use, will be an important advantage by multimodal interfaces.

When users interact multimodally: Users prefer multimodal interaction over unimodal, but that does not mean that they always will communicate multimodally. Several studies of users has been done, and it seems like users are most likely to express commands multimodally when describing spatial information about location, number, size, orientation, or shape of an object. When performing general actions without spatial components, users will seldom express themselves multimodally. This is why it is important to consider when users are and are not communicating multimodally.

Integrations and synchronization patterns: Early multimodal systems used spoken deictic terms as "that" together with a pointing reference. Studies has shown that people normally will not speak these deictic terms together with the pointing. Multimodal system designers should therefore not count on overlapped signals to achieve successful processing. Research should focus on fine-grained integration patterns between different input modes, and this could be found in the cognitive science literature. I will discuss some integration techniques later in this section.

Individual differences: A multimodal interface allows the user to have control on how to interact with the computer, and has the potential to accommodate a broader range of peoples based on age, skill level, native language, cognitive style, sensory impairments, and temporary or permanent handicap or illness.

Complementary versus redundancy: It is claimed that the content from different modalities contains high redundancies during multimodal interaction, but the dominant view is that it is complementary. Studies in speech/pen based interfaces shows they provide complementary semantic information, where the subject, verb and object of a sentence is spoken, and locative information comes from pen. One goal is to integrate complementary modalities such that each mode can be used to overcome weaknesses in the other mode. This has promoted the philosophy of using modes and components technology where they fits best, and then combine them to permit mutual disambiguation of the partial information associated with that mode.

Performance and Linguistic Efficiency: Several studies shows that multimodal interfaces gives enhanced performance and linguistic efficiency. In pen/voice based interaction, linguistic indirection that typically is spoken language is replaced with more direct commands. Brief and direct multimodal language also contains fewer referring expressions, and is instead deictic multimodal expressions.

2.3.5 Multimodal interface characteristics

As mentioned earlier in this chapter, speech has several characteristics that is different compared to traditional modalities. The most notable is that it is temporary; as once uttered, it is no longer available. Speech can also be used from a distance, which makes it ideal for hands-busy and eyes-busy situations. When using speech, users often tends to overestimate the capabilities, and they often treat the system as another person. This makes it a more human and a more natural interacting method.

By combining speech interfaces with direct-manipulation in a multimodal interface, several advantages could be identified. Research shows that speech and direct manipulation have complementary strengths and weaknesses that could be combined in a multimodal interface. One's strengths is usually the others weakness. As speech suits for hands/eyes free operation, direct manipulation use direct engagement. Other strengths with direct manipulation interfaces include, simple, intuitive actions, consistent look and feel, and no reference ambiguity, while speech recognition interfaces makes it possible for complex actions, reference is not dependent of location, and there are multiple ways to refer to entities[9].

Multimodal interfaces has some advantages over traditional keyboard and mouse interfaces or unimodal recognition-based interfaces. Some identified advantages include flexibility, availability, adaptability, efficiency, and lower error rate[21].

Multimodal interfaces allows flexible use of input modes. This includes that the users can choose how to interact with the system, and select input mode which suits type of input. Different modalities are well suited in some situations, and less ideal in others. For mobile use where the conditions are continually changing, it is important that the interface is flexible.

The availability characteristic of multimodal interfaces means that they have the potential to accommodate a broader range of users than traditional interfaces. This includes users with different ages, skill levels, native language status, cognitive styles, sensory impairments, and other illnesses or handicaps. For example users with visual impairments and stress injuries would probably prefer speech, while users with hearing impairments would prefer visual modes.

Multimodal interfaces provide the adaptability that is needed in continuous changing conditions of mobile use. Modes as speech, pen and touch are particularly suitable for mobile tasks where users can shift among these as environmental conditions change. The users could temporally be unable to use a particular input mode, for example using a navigation system when driving, the user might not be able to remove the gaze from the road, and a purely map-based interface would not be able to guide him. However, use of verbal directions would be a solution to this problem.

For certain tasks multimodal interfaces offers greater efficiency. This includes task which normally would require a series of sequential input events in a unimodal interface. As multimodal interfaces has the ability to process input in parallel, it would gain some efficiency in this area.

One important advantageous feature of multimodal interfaces, is its method of error handling, both in terms of error avoidance and recovery. This avoidance and recovery could both be user based and system based. A user would for example often choose the input mode which is less error prone, and tends to error avoidance. A speech-pen based system have several advantages in error handling compared to a speech-only system. The user could change to pen-input when communicate a word that is error-prone, as a foreign surname. Users language also tends to be simplified when interacting multimodal, and would therefore reduce the complexity. After a recognition error, the user likes to change mode, which facilitate error recovery. System based error recovery are also improved in multimodal systems, and is rooted in its ability to process parallel input from complementary input modes. The best joint interpretation is not always the same as the best from the different single modes. A multimodal system can therefor be more robust than a unimodal system as the complementary input modes can overcome the weakness of the others.

Several strategies have emerged for error suppression, and basically more information means greater likelihood of resolving missing or conflicting information, and could lead to successful disambiguation of input[22]. By increasing the number of input modes interpreted within the multimodal system, would lead to more effective supplementation and disambiguation of partial and conflicting information in individual input modes. These modes should also represent semantically rich information sources. Increasing the heterogeneity of the combined information sources by using completely different input modalities would lead to greater increase in robustness than fusing different data sources within an modality. As more complementary they are, as more their strengths can overcome the others weaknesses.

2.3.6 Multimodal integration

As multimodal interfaces combines data from multiple modalities, it is a need for merging the data at some point to one single data representation. This process is called multimodal fusion or integration.

Many early multimodal interfaces were based on a multimodal integration which occurred during the process of parsing spoken language. As the user spoke a deictic term, the system searched for a gestural act that suits that term. This point-and-speak multimodal integration are not usable for many tasks, and multimodal systems should be able to process other pen-based input than just pointing. That is why it is important with a general processing architecture which handle both different speech-and-gestures integrations, and interpretations of unimodal inputs and combined multimodal input.

There are two main types of multimodal architectures to handle joint interpretation of input. Early fusion integrates signals at feature level, and late fusion integrate information at semantic level. Feature-level fusing is a method for fusing of feature information of parallel input modes, where the recognition process in one mode influence the recognition process in others. This could be used to process closely synchronized input such as speech and lip movements where the "phonemes" created by speech and the "visemes" observed from lip movements are highly correlated. This architecture is based on machine learning and uses statistical methods such as hidden markov models or temporal neural networks for viseme-phoneme correlations, and therefore needs to be trained with real data. These systems are generally considered appropriate for modes that have similar time scales, but would however have more problem when the modes have substantially different information content.

Systems that use the late semantic fusion method have been applied to process input modes which are temporally less coupled. These input modes often provides different but complementary information. This method usually includes individual recognizers and a sequential integration process. Individual recognizers as speech are already publicly available, and could be trained using speech data. This means that unimodal data can be train using unimodal data and could be easier attached to the whole system, and the whole system could easy be scaled up both in number of input modes and each mode could be extended. The use of complementary input modes is one major design goal of multimodal systems. A well-designed system should be able to integrate these modes so the strengths of each mode could overcome the weaknesses of others.

Semantic fusion requires a meaning representation network that is common for each mode. In recent years, two data structures has become accepts as de-facto to represent meaning, frames and feature structures. Both these structures represent objects and relations as consisting of nested sets of attribute/value pairs. Feature-structures are logic-based and the primary operation is unification.

Typed-feature-structure unification determines the consistency of two representational structures and combines them to a single result if they are consistent. In typed-feature-structure, the feature structure consist of a type which indicates the type of entity it represent. This is associated with a collection of feature-value or attribute-value pairs, where the value could either be nil, a variable, an atom, or another feature structure. When two feature-structures are unified with respect to a type hierarchy, their values of identical attributes also are matched. If they are atoms, they have to be identical, if its a value, it became bounded to the value of the corresponding feature in the other feature structure. Feature structure unification is seen as well-suited for multimodal integration since unification can combine complementary input from different modes while it rules out contradictory input.

Chapter 3

Previous work

During the last decade it has emerged some new multimodal systems which allows users to communicate more naturally with computers, including modalities as voice, hand and/or pen gestures, gaze and body movement. A lot of experimentation has been done to discover how different modalities best cooperates. In this chapter I will present some systems which tries to accomplish this and which has been influential in this work.

3.1 Early systems

Early systems as "Put-That-There" [3], CUBRICON[18] and XTRA[26] was based on adding speech to traditional graphical user interfaces to give the user greater expressive capability. The more recent systems moves toward using speech in parallel with more expressive input methods and technologies. I will present some recent systems in the sections following this.

3.1.1 Put-That-There

Put-That-There was one of the first systems which integrated speech and gestures, and was designed at Massachusetts Institute of Technology.

This system use speech recognition in parallel with gesture recognition, and are tested in the MIT "Media Room", a physical room where the user's terminal is a room instead of a desk-top screen. The room also includes a wall-sized screen, and works together with the user's real-space in the "Media Room" as one continuous interactive space.

When the user sits in it's chair infront of the wall-sized screen, a space-sensing cube attached to the wrist is used. These deictic gestures are used to identify objects by specifying their locations. Meanwhile the system's microphone is listening for speech commands from the systems repertoire of commands the user can perform. This includes basic commands as create items, move items, delete items and naming items. By speaking the commands, and point on the screen, the system get a common meaning from speech and gesture. The user has to speak both the command and a object, either the name of the object or "that" while pointing.

3.1.2 CUBRICON

CUBRICON is an interface for combining spoken and typed natural language with deictic gestures, and is designed as a military situation assessment tool. Input are provided via a mouse pointing device, and is selected from the screen. The spoken input specify an action that refers back to the selected objects. CUBRICON also has a knowledge base with models of the user and the ongoing interaction. These models are dynamic and will influence the generated results.

As input comes from speech, keyboard or mouse, an input coordinator process the input streams and combines them to a single stream which is passed on to the multimedia parser and interpreter. By using this information together with the system's knowledge sources, a result is generated and passed on to the executor. The knowledge sources used both for understanding input and composing output includes an lexicon, a grammar defining the multi-modal language, a discourse model, a user model, and a knowledge base of task domain and interface information.

XTRA(eXpert TRAnslation)

XTRA is designed at the German Research Center for Artificial Intelligence, University des Saarlandes, and is an intelligent multimodal interface to expert systems which combine natural language, graphics, and pointing for input and output. XTRA acts as an intelligent agent, a translator that is an intermediary between the user and the expert system, and which provide natural language access to the expert system.

Figure 3.2 illustrates the use of this interface, Wahlster shows how to fill in a tax form. By using a mouse or similar pointing device, the user can specify locations and areas on the tax form. The user can choose between pointing modes as exact pointing with pencil, standard pointing with index finger, vague pointing by entire hand, and encircling regions with @-sign. In addition to these are three types of movement gestures as point, underline and encircle. The interface shown on the screen consist of three parts, one which

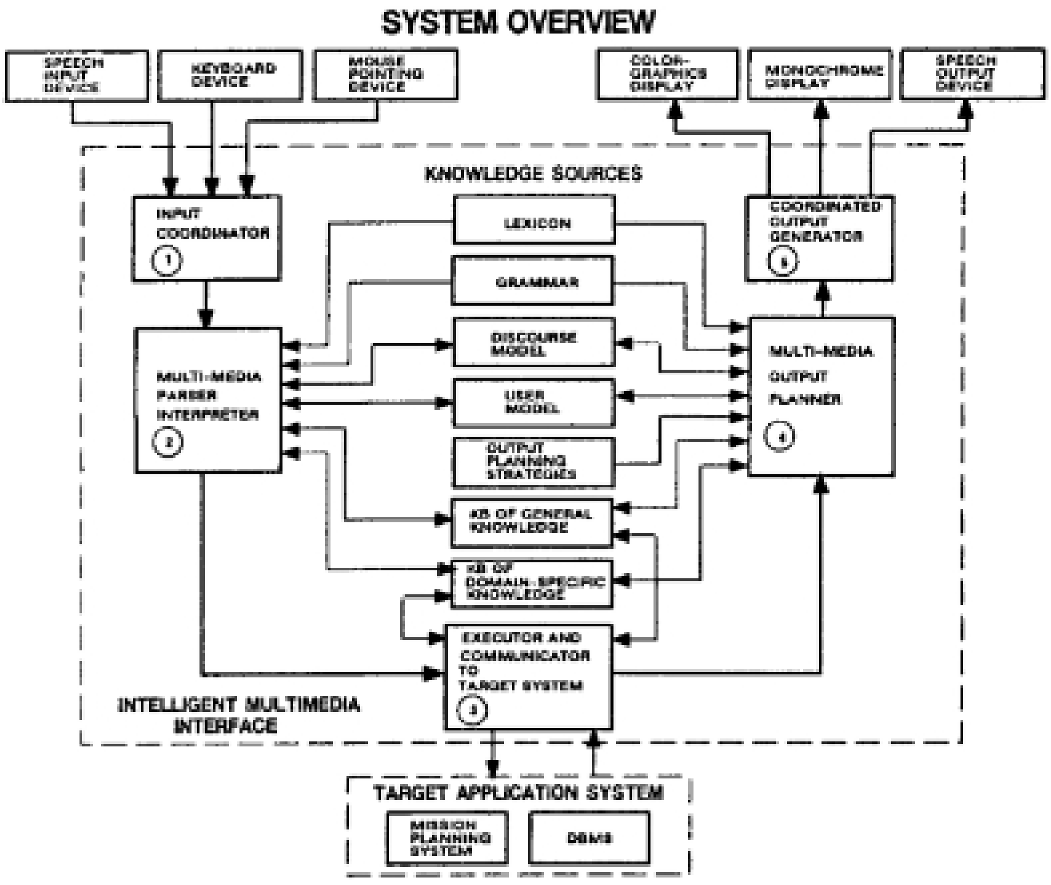
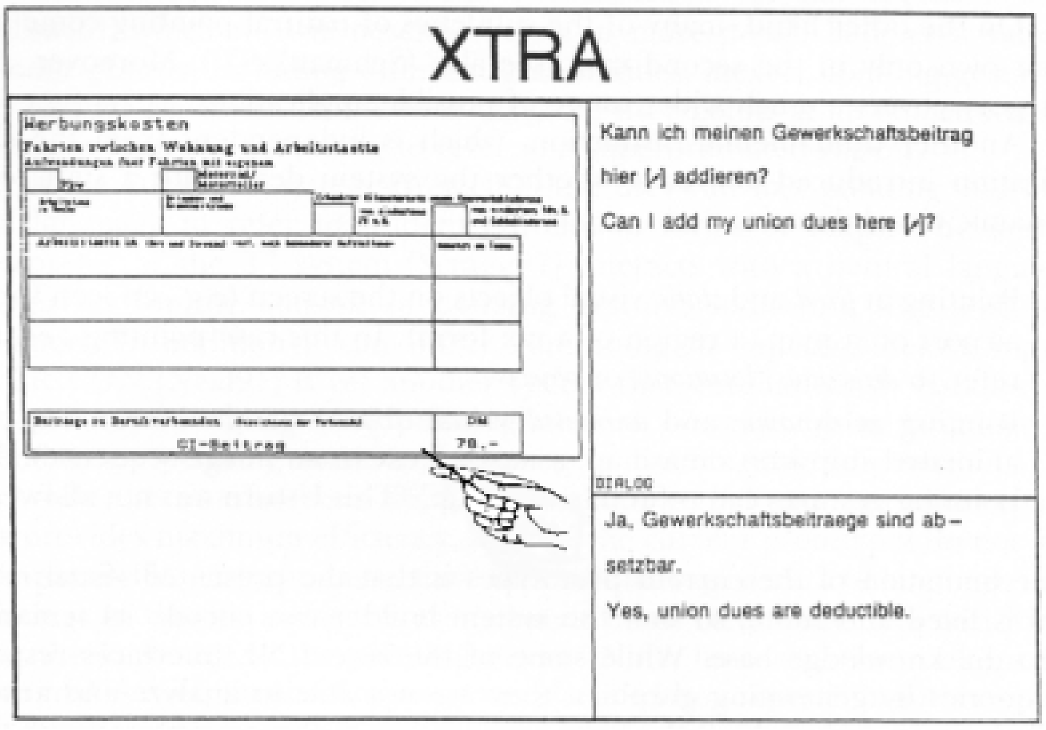


Figure 3.1: System overview of CUBRICON[18]



XTRA

Werbungskosten

Fahrten zwischen Wohnung und Arbeitsstaette

Aufwendungen fuer Fahrten mit eigenem

Pkw

Motorrad/
Motorroller

Beitraege zu Berufsverbaenden

DM

GI-Beitrag

78,-

Kann ich meinen Gewerkschaftsbeitrag hier [↗] addieren?

Can I add my union dues here [↗]?

DIALOG

Ja, Gewerkschaftsbeitraege sind ab-setzbar.

Yes, union dues are deductible.

Figure 3.2: Example of the User Interface in XTRA[26]

shows the tax form where the user can refer to points by tactile gestures, one for natural language input, and one for the systems response.

3.2 Recent multimodal interfaces

Recent systems combines speech in parallel with more expressive input methods. These systems has an advantage in their ability to utilize two recognition based input modalities. I will give a briefly overview of the Portable Voice Assistant and the Field Medic Information System in this section, and then I will give a more detailed overview of Quickset and MATCH in the following sections.

3.2.1 Portable Voice Assistant

The Portable Voice Assistant [2] is developed at BBN Technologies, and is a pen/voice based multimodal interface for web applications. It runs on a mobile pen-based computer with microphone and wireless connection, and the user could browse the World Wide Web using voice and a stylus.

The Portable Voice Assistant can interpret either individual or simultaneous pen/voice input, and integrate them using a late semantic frame-based integration. To demonstrate the portable voice assistant interface, a prototype is implemented, VoiceLog. This prototype is developed in Java, and both speech recognizer and the pen recognizer works as separate threads, and use time-stamps to which is used by the integration thread.

VoiceLog allows the user to order parts from a catalog by using speech and pen input. The user could select images from the catalog, which contains "hot" regions. These regions corresponds to parts individual parts that can be selected via speech or pen input. The interface also have an order form which get its input from either speech or written pen input.

3.2.2 Field Medic Information System

The Field Medic Information System [11] is developed by NCR Corporation, and is a speech/pen based multimodal interface for medical use. This system allows medical personal to document patient care and status in the field as it occurs. This information is then sent electronically to the hospital to prepare the arrival of the patient.

The system consist of two major hardware components, the Field Medic Associate(FMA) and the Field Medic Coordinator(FMC). FMA is a small wearable computer which uses a headset with microphone and earphones, and

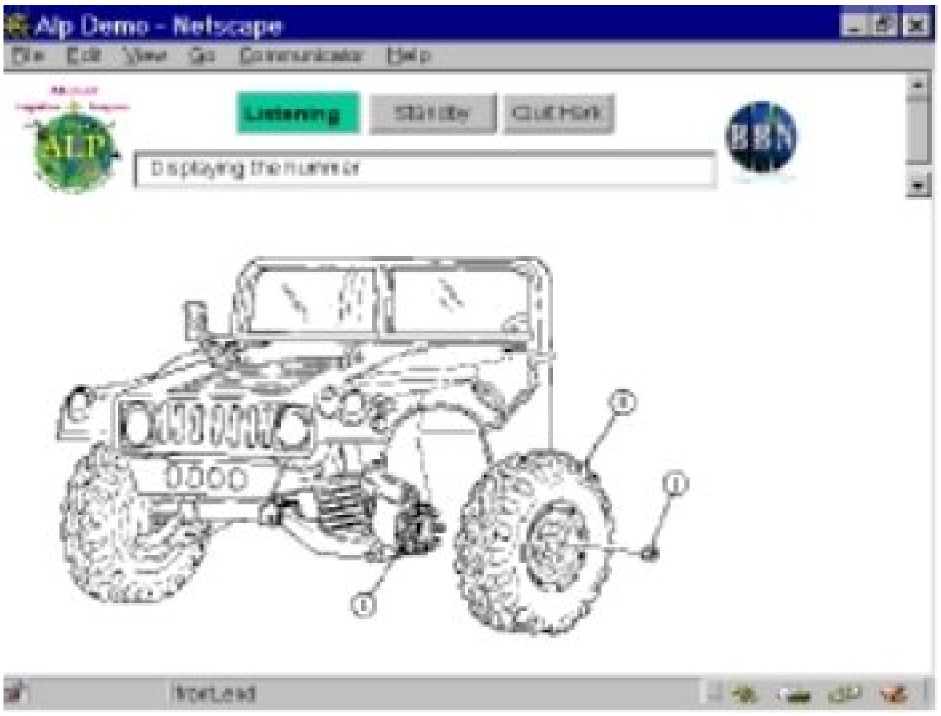


Figure 3.3: voiceLog[2]

FMC is a handhold tablet computer. Unlike the other presented speech/pen based interfaces, this is not designed to use input modalities simultaneously. The goal with this system is to have a mobile, hands-free speech based system with the ability to use pen-based input using the tablet interface.

3.3 QuickSet

QuickSet is developed at the Oregon Graduate Institute of Science and Technology, and is a prototype of a multimodal interface which uses pen and voice as input on a handhold PC. It uses a multi-agent architecture to communicate through wireless LAN. QuickSet work as a military training system where it is used to control a simulator and a 3-D virtual terrain visualization system. The first prototype of this system was finished in 1994, and is one of the earliest speech/pen based interfaces.

QuickSet is a handhold system and includes technologies like speech recognition, gesture recognition, natural language processing, multimodal integration, distributed agent technologies and reasoning. The most important parts of the system will be described in the following sections. QuickSet has been deployed for the US Navy, US Marine Corps. and the US Army.

3.3.1 System architecture

QuickSet [6] is based on the distributed agent technology Open Agent Architecture which means it has the ability to run different places and supports user mobility. The QuickSet architecture has a central facilitator with a blackboard, and then several agents which have their different jobs to do. This section will describe some agents briefly.

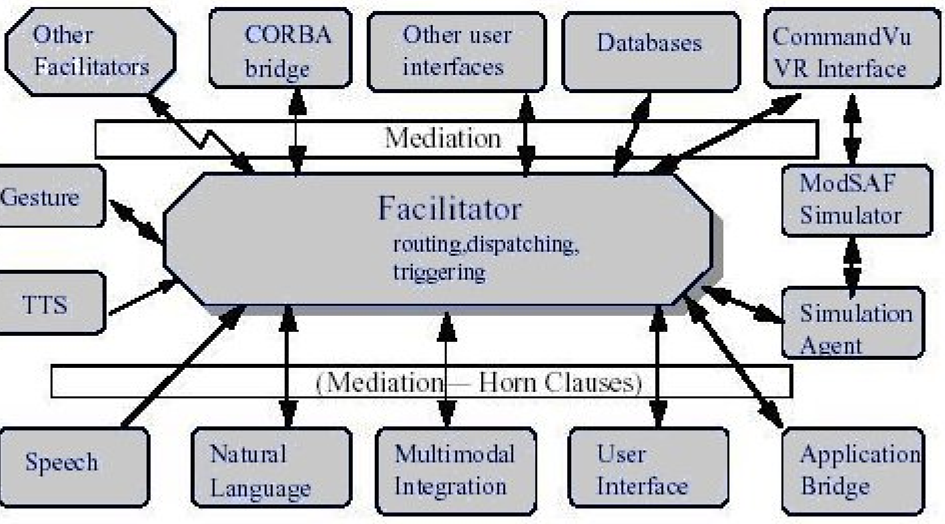


Figure 3.4: Architecture overview of QUICKSET[6]

Figure 3.4 shows an overview of the QuickSet architecture with a central facilitator in the middle, and all capable agents around.

Quickset interface: This shows the map of the region with entities placed in their actual terrain. While using pen and speech the user has the possibility to create points, lines and areas, create entities and give them behaviour, and watch the simulation from the handhold.

Speech recognition: This agent is built on IBM's VoiceType Application Factory and VoiceType 3.0 and Microsoft Whisper speech recognizer.

Gesture recognition agent: QuickSet uses a pen-based gesture recognizer. This consist of both a neural network and a hidden Markov model.

The system can recognize several pen-gestures, including military map symbols, editing gestures, route indications and area indications.

Natural language agent: Typed feature structures are produced as a representation of the utterances meaning.

Text-to-Speech: This agent uses Microsoft's text-to-speech system.

Multimodal integration agent: This agent is responsible to get the individual interpretations of each input agent(speech and gesture), and then identify the best potential unified interpretation. This agent give as output the preferred interpretation and could be either unimodal or multimodal.

Simulation agent: This serve as the communication channel between the OAA agents and the ModSAF simulation system.

Web display agent: This agent can be used to create entities, points, lines and areas, and the entities can be viewed over a WWW connection.

CommandVu agent: CommandVu is a virtual reality system, and can be used to create entities and to fly the user through the 3-D terrain.

Application bridge agent: This generalizes the underlying applications API to typed feature structures, and therefore provide an interface to the various applications. This allows a domain-independent architecture.

CORBA bridge agent: Converts OAA messages to CORBA IDL for the Exercise Initialization(ExInit) project. ExInit allows user to create large-scale(divisions and brigades) exercises.

3.3.2 Multimodal integration

QuickSet uses a distributed agent architecture, and use the advantages from this architecture in designing the multimodal integration. The different input agents as speech and gesture works in parallel. To get a clearly defined and well understood common meaning from the different input modes, they uses typed feature structures. To accomplish multimodal integration, unification is used.

In QuickSet input needs to be both temporally and semantically compatible before they will be fused to one integrated meaning. Temporal compatibility means speech follow gestures within a short time interval. That means that if speech does not follow within that interval, the gesture will be interpreted unimodally. This architecture requires time stamps for both the beginning and the end of each input stream.

To accomplish the goal of semantic compatibility, QuickSet uses unification of typed feature structures. Unification requires consistency of two representational structures to get one single result. A feature-structure is a collection of feature-value pairs. The value could be either an atom, a variable or another feature structure. Typed feature structure is an extension where feature structures or pairs of atoms are being unified to be compatible in type.

They identify some significant advantages by using typed feature structure unification. The first one is that it allows specification of partial meanings. That means that both speech and gesture input could specify an feature structure where certain features are not specified, and by using both input modes there are possibilities to interpret the input to one single meaning. This will also lead to multimodal compensation, where the input modes compensate for the other input modes recognition errors.

3.4 MATCH

MATCH(Multimodal Access To City Help) [13] is developed at AT&T Labs - Research, and provides a mobile multimodal speech-pen interface to restaurant and subway information in New York City. This system should work on mobile devices with limited screen and no keyboard or mouse as PDAs, tablet computers and mobile phones.

Users can find restaurants based on cooking, price, and location, and can get information such as reviews, phone numbers, and addresses. The system could also guide the users to the restaurant by using the subway guidance which can help users come from one location to another. MATCH runs on a handhold PC with a browser-based graphical user interface and integrates AT&T's WATSON speech recognition technology, Natural Voices text-to-speech, handwriting recognition, and gesture recognition.

3.4.1 System architecture

The purpose by the architecture is that it is designed for highly mobile applications, it enables flexible multimodal input, and provides multimodal output.

The architecture consists of a series of agents. I will present some of the key components in this section.

MCUBE: This is a Java-based facilitator which enables the agents to pass messages to each other. MCUBE messages are encoded in XML, and provide a general mechanism for parsing messages.

Multimodal User Interface: The Multimodal UI is browser-based and what the users are interacting through with the system. It's communication with MCUBE are based on TCP/IP. The user interface provide a map which enables both pen-based interaction and traditional GUI interaction. The speak input are possible to turn on and off, depending on which mode is natural and for noisy environment.

Speech Recognition: The system uses AT&T's Watson speech engine.

Gesture and handwriting recognition: The recognitions could be performed both on individual strokes and combinations of strokes. The handwriting recognizer supports a vocabulary of 285 words, and the gesture recognizer recognizes a set of 10 basic gestures.

Multimodal Integrator: This receives gesture and speech input, and builds a joint interpretation from the inputs. A finite-state approach is used to multimodal integration. A three tape device are used, where the first represent the speech stream, the second the gesture symbols, and the third their combined meaning. This three tape finite-state device takes speech and gestures as input by using the first to tapes, and then writes out a multimodal meaning by using the third tape. This technique enable users to interact freely using speech alone, pen alone, or dynamic synchronized combinations of speech and pen.

Chapter 4

The overall architecture

This approach to multimodal input involves modalities for command input and location input. These modalities could both be user input and context input. The overall architecture will be described in this chapter where the agents and the blackboard will be presented.

The overall architecture is based on input agents and a common blackboard for sharing input information. As figure 4.1 shows, four main parts exist in the current architecture. This could easily be extended in the future if desired. I will give a brief overview in this section, and will present each part in more detail later on.

Agents are responsible for collecting information directly from the user, and interpret this input so it could be used to decide what to do. In the current architecture, the user could mainly give location information and command information. One or more LocationAgents are responsible for collecting the location information. The second type of information which is possible to get from the user, is command information. This shows what the user wants to do, while the location information shows what objects the user wants to perform that action on. Each input modality is an own agent that get the necessary information from the user, which means that the agents which are used for collecting location information extends the LocationAgent.

These agents which communicate directly with the user, is responsible for analyzing this information and send it to a common blackboard. This blackboard is used for storing input information which later is used in the multimodal integration part.

The last part of the architecture is the FusionAgent. This is responsible for listening for changes in the blackboard, and collect this information. A listener which listen for command information is used, and as soon as one of the CommandAgents writes some command information on the blackboard, this agent reads the command and all location information available on the

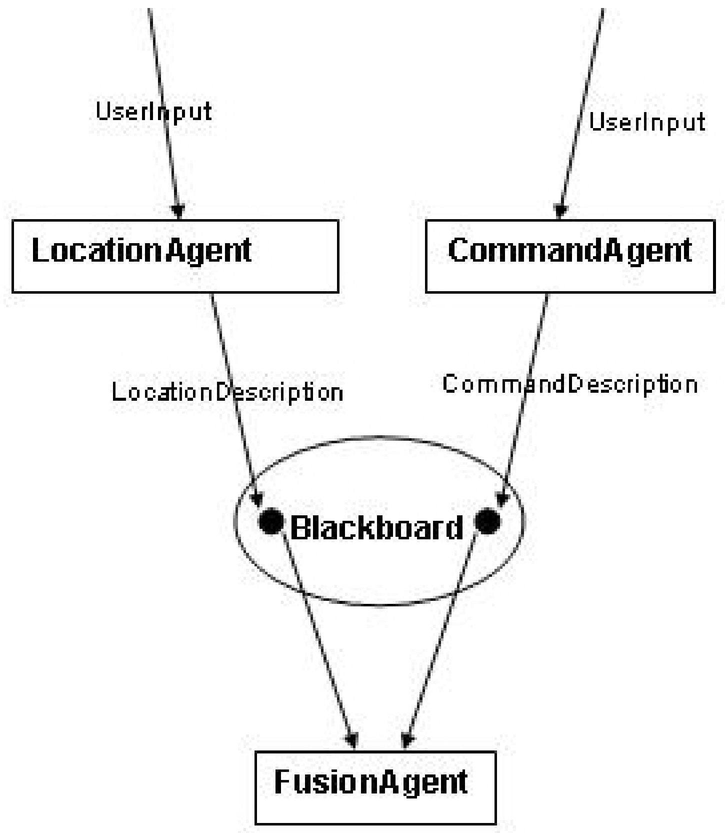


Figure 4.1: The overall architecture

blackboard. This input information has to be analysed and given one common meaning for an action to take place. That is typically to combine one piece of command information with one piece of location information and decide what action to be done with what object.

4.1 The multimodal input process

The overall goal is to collect all available information from the user and combine this information to decide one common action. Four main steps takes care of this as shown in figure 4.2.

The first step is input collection from the user. This input is then analysed and sent to a common workspace. The integration process is responsible for collecting the information from the different input modes, and merge them

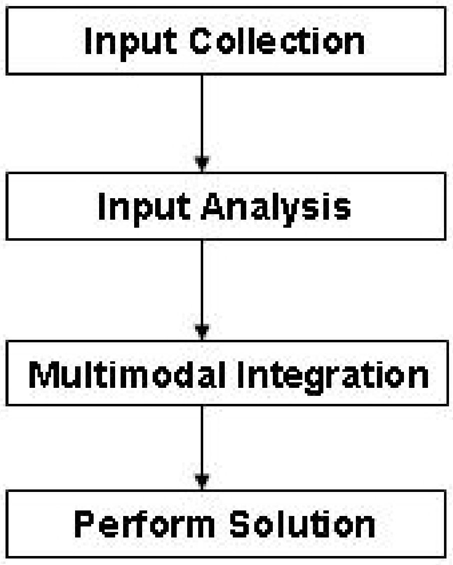


Figure 4.2: The processes for integrating several input modalities to one meaning

to one common meaning. From this common meaning, an action can be performed. These processes are: Input collection, input analysis, multimodal integration, perform solution.

Input Collection

The input collection is the process of collecting information from the user. The LocationAgent and the CommandAgent are responsible for this, and are in direct contact with the user. Input are collected dependent of input modalities that are used in the system. For the current architecture, this is restricted to modalities for collecting command information and location information.

Input Analysis

As the information is collected, this information has to be analysed to be used further on. Location information usually contains getting coordinates, and in the analysis process this is interpreted to refer to specific objects. Each coordinate could refer to several objects, and all this information has to be stored. This information is interpreted by using some knowledge model of the environment that are used. When speech is collected, it uses the speech grammar, and listen for special sentences. From these sentences, the meaning of what has been said have to be extracted. As this analysis is done, the information is sent to the blackboard. The LocationAgent and the CommandAgent are responsible for this process.

Multimodal Integration

This process is the main part of merging several information elements into one common meaning. By collecting all available information from the blackboard, it uses late semantic fusion to decide what action should take place. As mentioned in chapter 2, late semantic fusion have an advantage when the different input modalities should be independent of each other. The output of this process is what should be done with which object. The FusionAgent is responsible for performing this process. This process is discussed in detail later on.

Perform Solution

After it has been decided what should be done, this process is responsible for performing exactly this action. Since input is the focus of this thesis, little work have been done on this part.

4.2 Blackboard

The communication between the agents that collects input and the FusionAgent, is through a blackboard architecture. An advantage with this architecture is that the different knowledge sources (agents) do not need to know about each other and work independently of each other. However, they need a common structure to represent the knowledge. A frame structure are used for representing the knowledge, and two structures exist depending on which input type are collected; location or command.

Figure 4.3 show the representational structure for location and command information. Each input agent that collects information from the user, will collect and analyse this informtation and store it on the blackboard. The agent then will use one of these representation structures when the information is stored on the blackboard depending on which modality it refers to. CommandAgents will use the command structure, and store information about the specific command, which type of input modality, additional information, and the time of command completion. Input time and additional information are not mandatory information, but the other three needs to

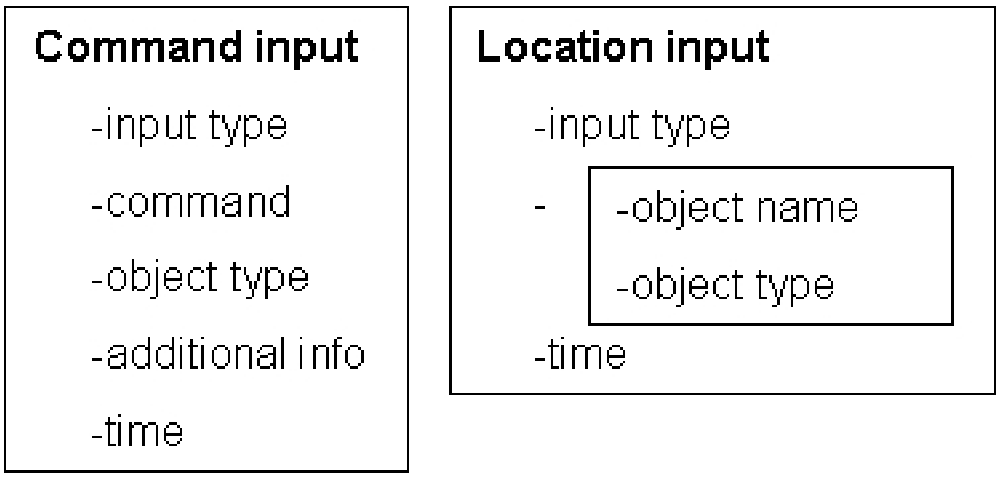


Figure 4.3: Representation structure of input knowledge

be given. The LocationAgent collects information about potentially location objects. This is stored in a structure with information as type of input modality, a list of location names and types, and the time of completion.

As input information are collected and analysed into the form of this frame structure by either the CommandAgent or the LocationAgent, this information can be sent to the blackboard. The current structure of the blackboard allows each input modality only one instance of information on the blackboard at each time, which means only the most recent information from each input modality agent exist on the blackboard. As soon as a new command input is placed on the blackboard, the FusionAgent is notified, and will at that time collect all existing information on the blackboard, and try to find a joint multimodal interpretation.

4.3 The Agents

As mentioned before, there are agents that are responsible for collecting input from the user and there are an agent responsible for fusing all the collected input to one meaning. I will present 4 different agent classes that makes the overall system, and I will present the architecture on these agents. The agents are: InputAgent, CommandAgent, LocationAgent, and FusionAgent.

4.3.1 The agent architecture

InputAgent

The basic agent is the InputAgent and all others agents extends this. This basic architecture consists of four different modules: a module for communication with the blackboard, knowledge models, a reasoning engine, and a module which control the behaviours.

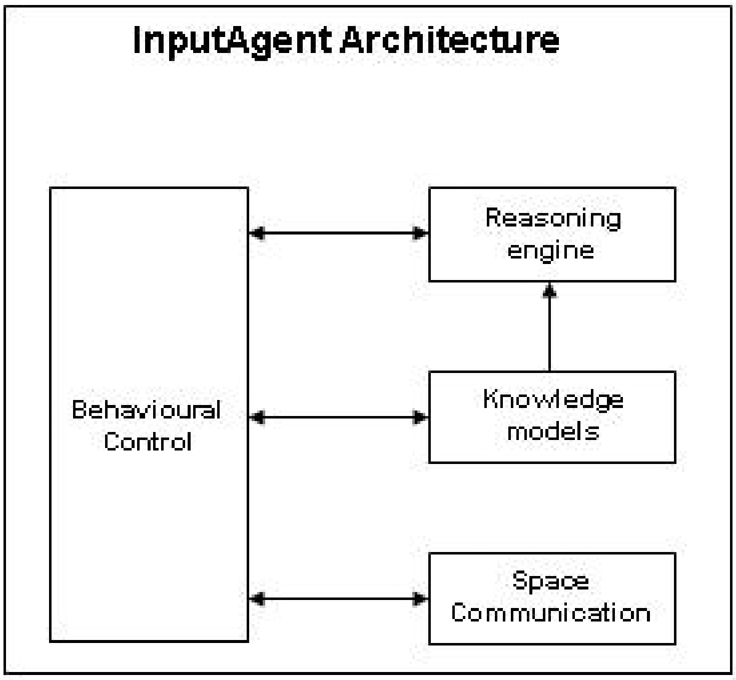


Figure 4.4: The basic agent architecture of the InputAgent

The module which control the behaviour, the behavioural control module decides which actions should take place at each time, and this decision is based on information about the environment and the external input the agent gets at each time.

The space communication module takes care of communicating with the blackboard. The InputAgent contains all necessary protocols which is needed for space communication, and alle other agents extends this. This communication is used for sharing input information.

The knowledge models get information about the environment which is useful in the process of analysing input modalities and in the process of multimodal integration. External models about the environment is stored, and is used to decide which commands and location objects that can be integrated to be used together.

The fourth module is the reasoning engine. This is the one which is responsible of reasoning over the knowledge the agent has at each time. As the behaviour control decides what should happen, this module knows how to do it.

Command Agents

The CommandAgent extends the InputAgent, and the architecture is about the same. It has in addition an extra module, the User Interaction Module. This senses the information from the user, and this depends on which input modality used for this agent.

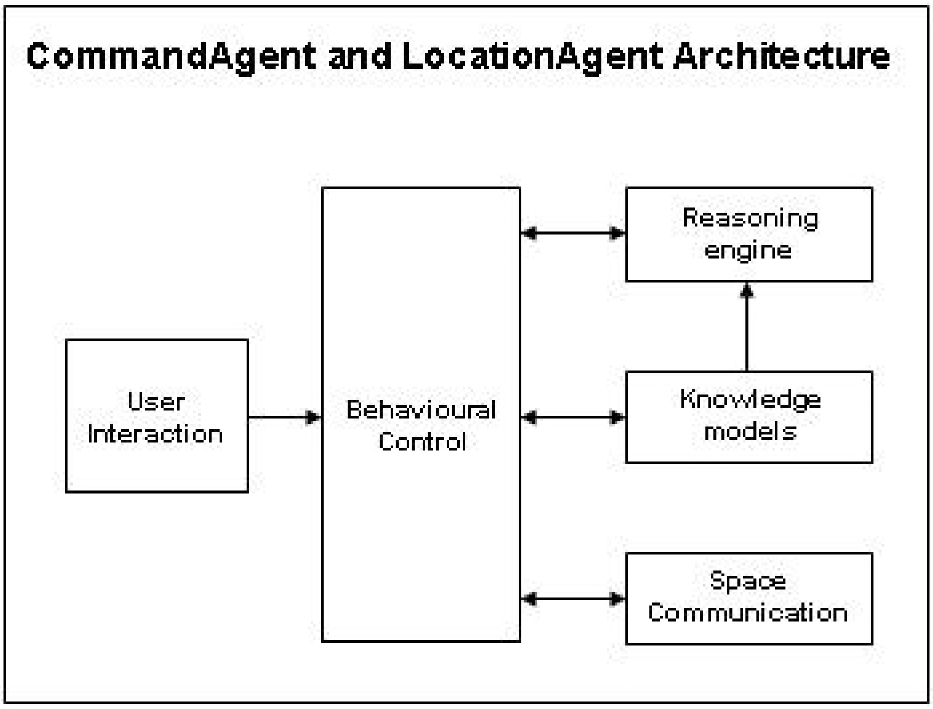


Figure 4.5: The architecture of the CommandAgent and LocationAgent

The task of the CommandAgent is to listen for command input from the user, analyse this input and send it to the common space. The command agent's process diagram is shown in figure 4.6.

The first step for the CommandAgent is to listen for input. This is dependent on which input modality is used, and in the next chapter when the implementation is described, an approach where speech and text are

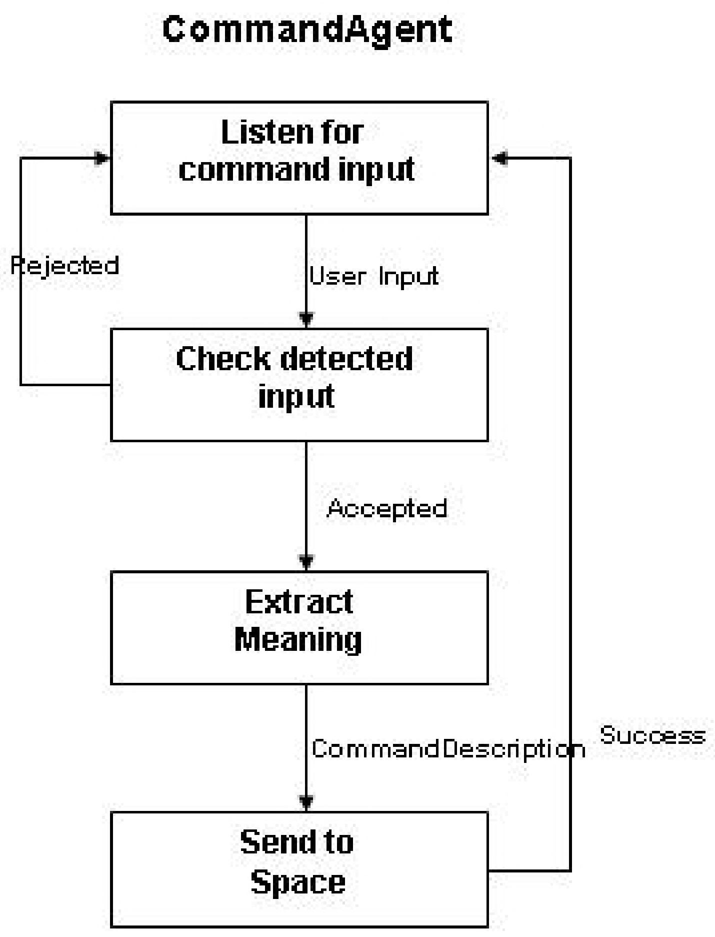


Figure 4.6: Command Agent Process Diagram

used will be described in detail. When an user input is detected, this input has to be checked for being an accepted input. When the input is accepted as an allowed sentence, the meaning can be extracted. When comes to the CommandAgent, this is to make the CommandDescription from the detected input. As the CommandDescription is finalized it is sent to the common blackboard.

Location Agents

The LocationAgent extends the InputAgent, and the architecture is the same as the command agent shown in figure 4.5. The main task of the Location-Agent is to get the location coordinates from the user, and then find what objects the user potentially refer to.

The task of the LocationAgent is to collect the coordinates the user sould refer to depending on the chosen input modality, analyse this information to find which potential objects these coordinates refer to,and then send relevant information to the space. The LocationAgent's process diagram is shown in figure 4.7.

As an input is detected and the coordinates are collected, the agents converts these coordinates to a list of potential objects. This list contains every object that has coordinates from the knowledge base that matches the incoming coordinates. If a match is found, the name and type of the object is stored with a time stamp in a LocationDescription entry object. As this process is finished, this information is sent to the common blackboard for further use by sending the LocationDescription.

Fusion Agent

The role of the fusion agent is to make a single meaning from the input information from the command agents and the location agents. Figure 4.8 shows the process diagram of this agent. This agent listen to the blackboard, and when a new command description object is placed on the space by a CommandAgent, the fusion agent will be notified and it will start to collect all location information from the space together with the command information.

As the agent has collected all available information from the blackboard, the integration process will start. First of all it has to decide which modalities to use in the integration process. The command information is of course the one that was notified about, but the location information has to be sorted in a priority list. The time stamp will be used for this. A hysteresis value has to be decided, and by using the time stamp of the command information, the decision can be made if map information or position information should be

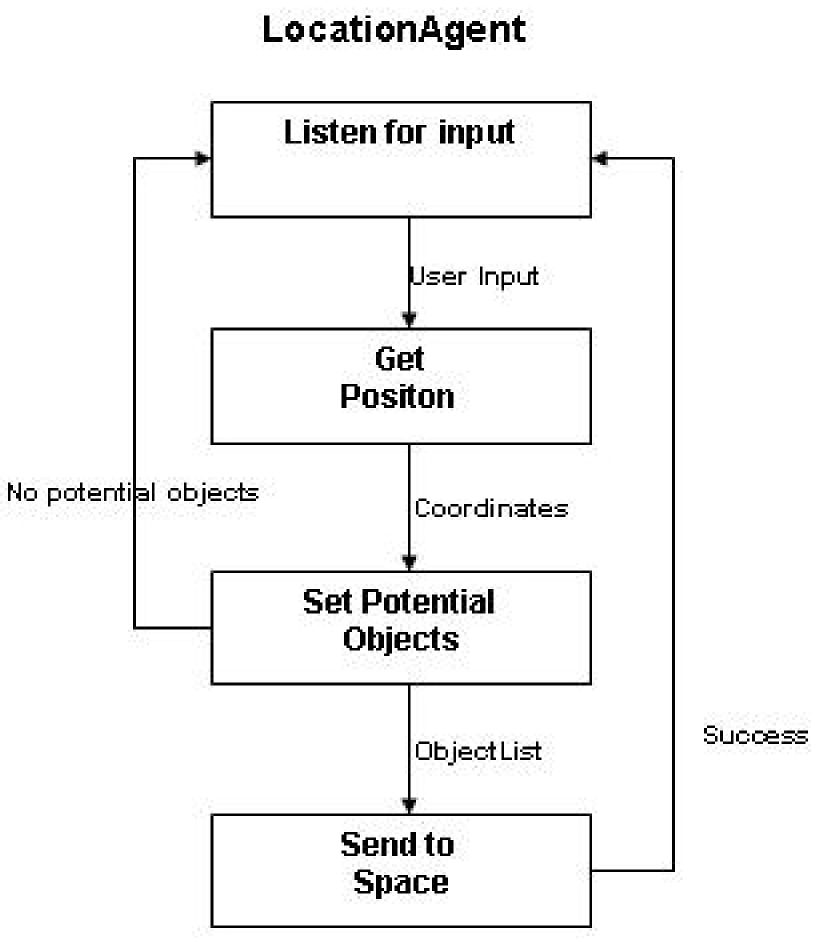


Figure 4.7: LocationAgent Process Diagram

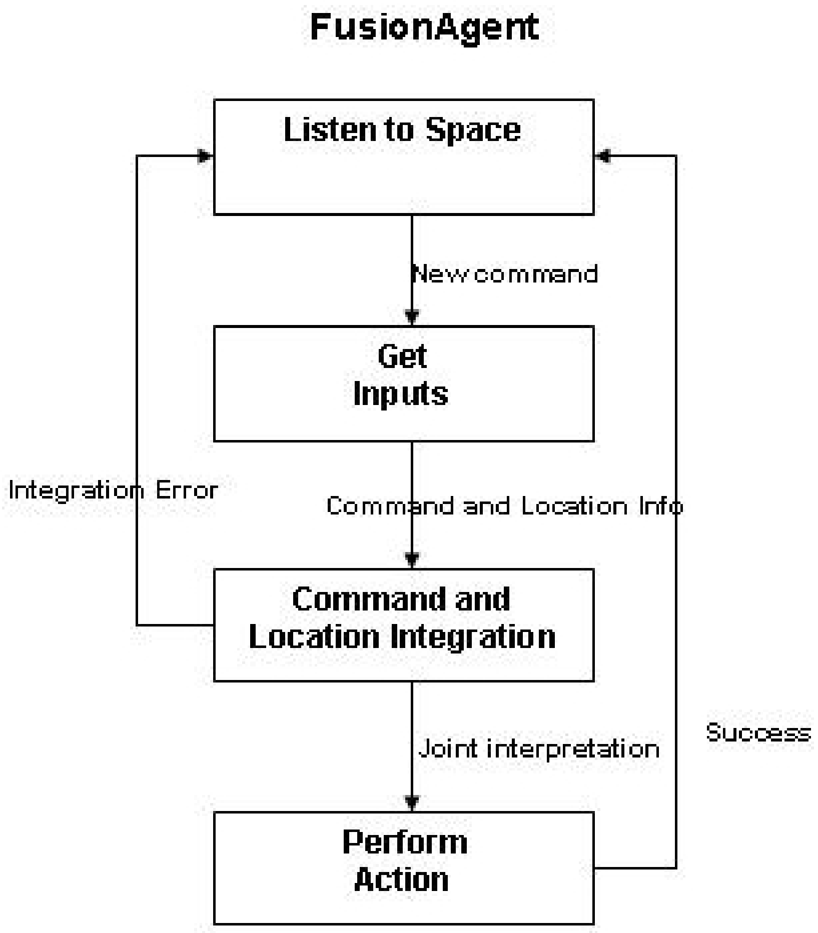
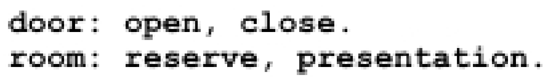


Figure 4.8: FusionAgent Process Diagram

used. If the user points on the map while given a command, it is likely that those two operations is given closely in time, and this time difference is the hysteresis value set. This is what was mentioned as temporal compatibility in an earlier chapter. If command and location information are not closely with respect of time, the physical position of the user is considered. This physical position does not always change for a time, and should not be that time dependent. From this information, a priority list will be made, and the one on the top will be considered first. If it is not possible to get a successful result from this, the next modality on the list will be considered.

As the modalities that should be integrated are decided, the integration process could start. The integrator has currently information about possible objects from the LocationAgent, and the most recent command from the CommandAgent. How this process works, is discussed in detail in next chapter when the integration algorithm is presented, but I'll give a brief description here anyway.

If object information is in addition given from the command information, the integration process is a bit easier. This object could then can be matched against the objects from the location information. If object is not given from the command, external knowledge about the environment has to be used. Figure 4.9 shows how this information is stored externally:



```
door: open, close.
room: reserve, presentation.
```

Figure 4.9: Example of the knowledge which shows possible commands on each object

The FusionAgent get this knowledge at startup, and the fusion agent then has knowledge about all possible types of objects in the environment, and which commands that can be performed on them. As the FusionAgent shall integrate information containing a command and possible locations, this knowledge has to be used to get the coupling between command and object. By using this knowledge and the information from the relevant input modalities, probably one common meaning is found. If not, an error occurs, and either new information has to be found, or the next on the priority list is used.

When a joint integration is found, the agent has information about what to do. It is then ready to perform this action. This is out of the scope of this thesis, and it should probably send this information further on to an another agent which contain information about activator. For now, a simulated action will be performed in the last process.

4.4 Multimodal integration

The process of multimodal integration should merge the information that comes from the different input modalities, and get one common meaning. This common meaning should contain what specific object the user refer to, and what should be done with this object.

As discussed earlier, there are in general two main methods to solve this, early feature-level fusion and late semantic fusion. In this architecture, the late semantic fusion is chosen. This approach is favourable when the input modalities are different and complementary as command information and location information are. By using semantic fusion, the different modalities work independently and the different recognizers could be trained individually. When using the semantic fusion method, it is much easier to extend the architecture to involve other modalities, or to change one modality. If a better speech recognizer shows up, and is preferable to use in this system, this can easily be changed without consider doing anything with the other modalities.

Semantic fusion requires a common meaning representation for the different modalities, and in this architecture a frame-based structure has been chosen. A frame represent objects with a set of attribute/value pairs. These is implemented in the form of the entries that is placed on the space, one for command information and one for location information. The CommandDescription and the LocationDescription is discussed in detail earlier, and consists all relevant information which each agent gets from the user when it is placed on the common space.

The goal with the multimodal integration process is to find one common meaning which specify one specific object and what should be done with this object. That means that we need both information from the CommandDescription and the LocationDescription. These descriptions is the frame-based structures which has to be merged together based on their attribute values.

As a CommandDescription is placed on the space, the FusionAgent is notified, and it reads this CommandDescription and all LocationDecription present on the space. By using the time attribute, the LocationDescriptions are sorted in a priority list. If a LocationDescription which refers to a map input is placed on the space within a certain time interval, this will get highest priority, if not the physical location will get highest priority. The relevant information when merging two input modalities together, will be the command and eventually the object type if given from the CommandDescription, and the list of object names and types from the LocationDescription. This list of objects is sorted by the LocationAgent, and when the LocationDescription is sent to the space, this list will be sorted with the most likely object on top.

To find a match, we start by traversing the list of object names and types from the LocationDescription. If a object type is given from the command information, it will try to find a match from the object type attribute from the location attribute. If a match is found, this will refer to a specific object name, and the FusionAgent has reached its goal and found a specific object and what it should do.

The object type is often not referred to from the command information. In these cases, the matching process is more complex. A grammar is made which shows all possible object types, and which commands can be performed on them. The algorithm start again by traversing the list of object name and types from the location information. For each object the grammar will be used to find potential commands which can be performed on it. If one of these commands matches the present input command from the CommandDescription, a match is found and the goal is once more reached.

4.5 Knowledge representation

External knowledge about the environment has to be used both when analysing user input and in the process of multimodal integration. Relevant information is about available objects in the environment, where these objects are located, and which actions can be performed on them.

In the process of collecting location information from the user, some predefined location based knowledge has to be used. This knowledge has to contain information about available objects in the environment, and where they are located. Figure 4.10 shows to the left how this information is represented. Each object represented on the map or in the physical environment is given some attribute/value pairs. An unique object name is used to identify each object, and this could for example be the name of a room as "kitchen". A set of object types is used to identify which type object it is, and this could be "room". The coordinates is limited to a square where the four points are identified, x minimum, x maximum, y minimum, and y maximum. This knowledge is used to determine which objects it could be referred to as an (x,y) coordinate is found. By matching this coordinate to this information, potential referring objects is found.

To the right on figure 4.10, a knowledge model that determine what action can be performed on each type of object is shown. The attribute commands is a list of commands, and one object type can contain several commands. This

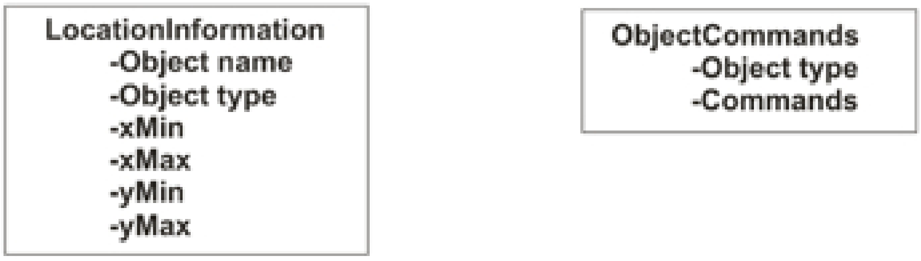


Figure 4.10: Knowledge about location and commands

knowledge is important in the multimodal integration process when matching command information and location information, and this knowledge is the link between commands and location.

This knowledge is stored external and permanent, and the agents get this knowledge at start up. Some dynamic knowledge are also present, this is the knowledge that should be integrated, and which comes from the user. This is the knowledge about which action the user wants to do, and to which objects. This knowledge is represented as frames, and placed on the blackboard. As shown earlier on in figure 4.3, the command information show information about which commands the user wants to do, and sometimes to which object type that action should be performed on. The location information holds information about which objects that are potential matches with respect to the coordinates the user should refer to.

Chapter 5

Implementation

A prototype of the architecture presented in the previous chapter has been implemented, and in this chapter the current implementation will be described. The FusionAgent, the CommandAgent, and the LocationAgent are all implemented to work with a blackboard. In the current implementation, two agent are implemented to represent different modalities for both the CommandAgent and the LocationAgent. These are respectively the SpeechAgent and the TextAgent for the CommandAgent, and the PositionAgent and the MapAgent for the LocationAgent. These agents and how they communicate will be discussed in detail in this chapter.

The chapter starts with describing the implementation of the blackboard, and the communication between the agents collecting information from the user and the FusionAgent. Then each agent are described in detail, and at last the fusion algorithm.

5.1 Blackboard

The architecture used to implement the blackboard struckture is space-based, and is called JavaSpace. As described in chapter 2, this is a Java based implementation with entry objects placed on the space.

As mentioned in the previous chapter, a common representational structure is used to represent the location and command input. A frame structure was used. This could refer to the entry objects that are placed on the JavaSpace by each agent, and which again are collected by the FusionAgent. The entry objects that refer to the frame structures described in chapter 4 are showed in figure 5.1.

The communication between the agents and the JavaSpace will be described in detail later in this chapter, but as described in chapter 2, four

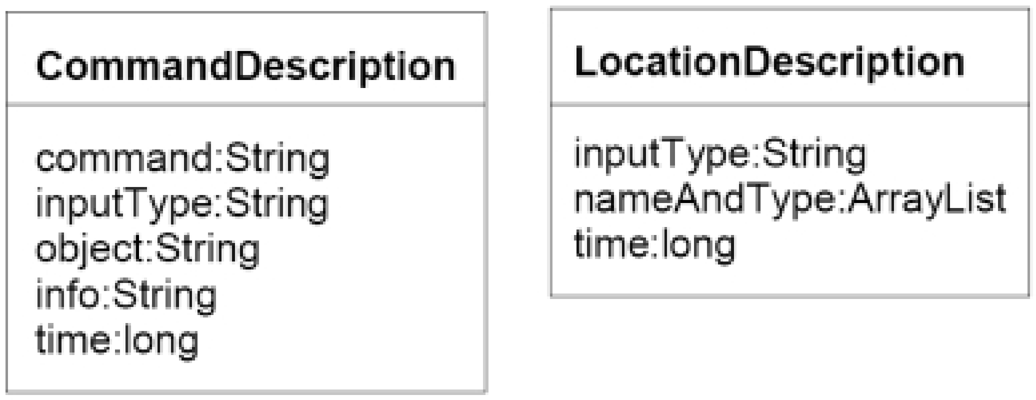


Figure 5.1: Two type of entry objects, one for command information and one for location information

commands are possible: write, take, read and notify. By using these commands on the LocationDescription and the CommandDescription, and easy method of communication is provided.

5.2 Agents

Figure 5.2 shows the agent hierarchy with respect of inheritance. This shows each implemented agent, and each agent will be discussed in detail in this section.

5.2.1 Agent communication

A space-based model is used for the FusionAgent to collect the information from the CommandAgent and the LocationAgent. Figure 5.3 shows how this space communication works.

The figure shows the most important components, and which commands are used. The agents SpeechAgent and TextAgent extends the CommandAgent, and the MapAgent and the PositionAgent extends the LocationAgent. To communicate with the space, entry object is made. As described in the previous chapter, the CommandDescription and LocationDescription represent respectively command information and location information from the user. How the different agents works and how the information is collected and used will be presented in the following section. In this section the communication between the agents through the JavaSpace will be presented.

Command agents collect information about which command should take place. This command is stored in an entry called CommandDescription

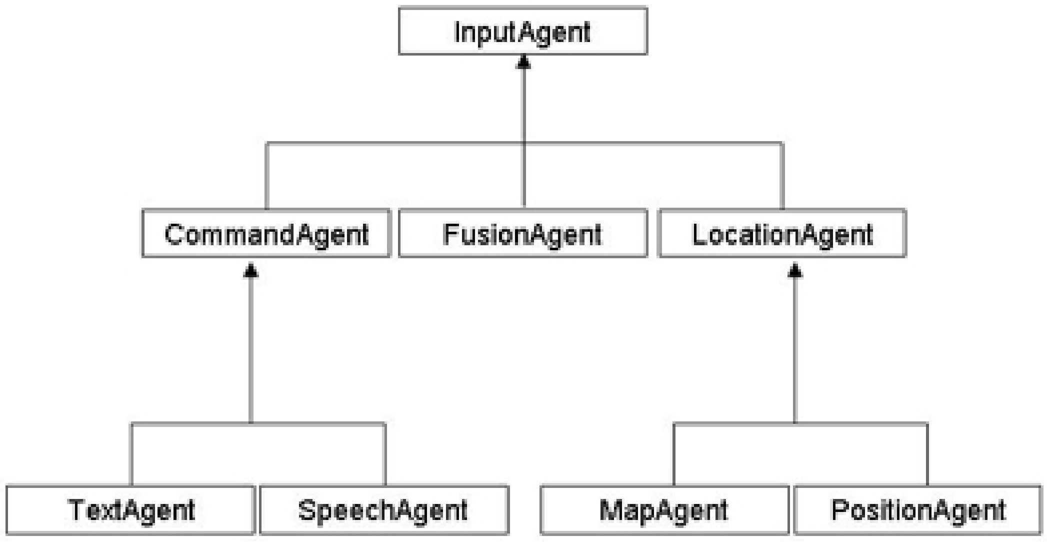


Figure 5.2: The figure shows inheritance in the agents

shown in figure 5.1. When all information is collected, and the Command-Description entry is made. To make sure the FusionAgent will get the last entry object placed on the space, only one entry can be present on the space from each input modality at each time. The agent will then look for other entries placed on the space by the same agent by searching for inputType. If an entry is found, this will be deleted from the space by using the take command, and the new one is written to the space.

The location agents collects location information from the user. In the current implementation, this is either the users present physical location through the PositionAgent, or location information which refers to pointed coordinates on a map through the MapAgent. This information is stored in a LocationDescription entry as shown in figure 5.1. When all location information is collected, and this LocationDescription is made, the entry is written to the space. Again old entries are first deleted using the take operation from the space before a new one is written as shown in figure 5.3.

In the current situation, up to four entry objects could be placed on the JavaSpace, hence two LocationDescriptions and two CommandDescriptions. Each of the CommandDescriptions contains exactly one command each, while each of the LocationDescriptions could contain several location objects.

The FusionAgent is the one responsible for collecting the information from the blackboard and integrate it into one joint interpretation and decide which command should take place. A listener is implemented for the FusionAgent to listen to the JavaSpace for changes. As figure 5.3 shows,

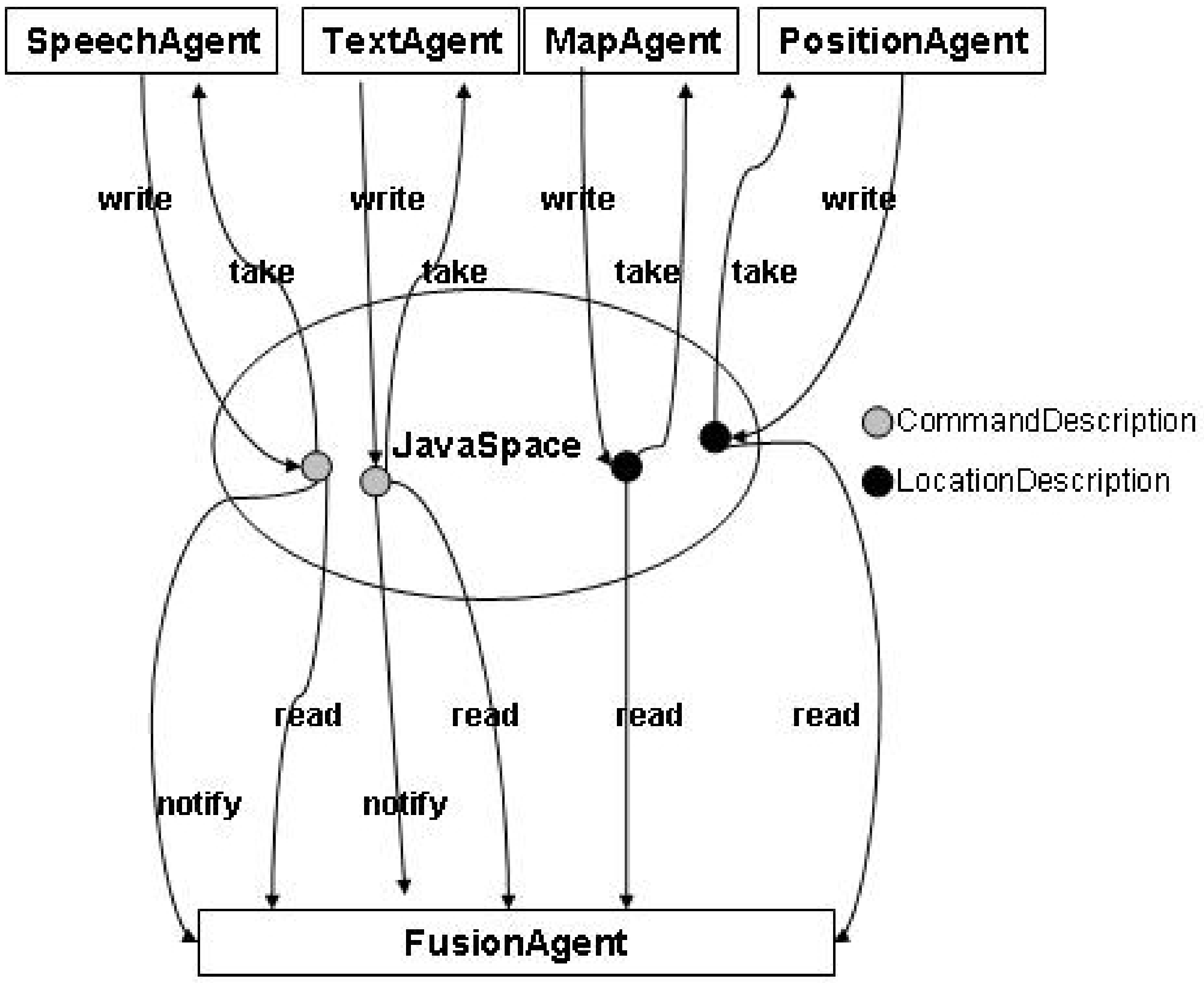


Figure 5.3: Space Communication

this uses the notify operation to listen for CommandDescriptions. As one of the CommandAgents writes a CommandDescription on the JavaSpace, the FusionAgent is notified and reads this CommandDescription and all of the existing LocationDescriptions on the space. The FusionAgent then has both command information and location information that is collected and it should hopefully be capable of finding a joint interpretation. This process is described in detail in the next section.

5.2.2 InputAgent

This is the top agent in the hierarchy showed in figure 5.2. The other agents inherits this agent's methods and behaviours. Figure 5.4 shows the implementation of the InputAgent and its Behaviours.

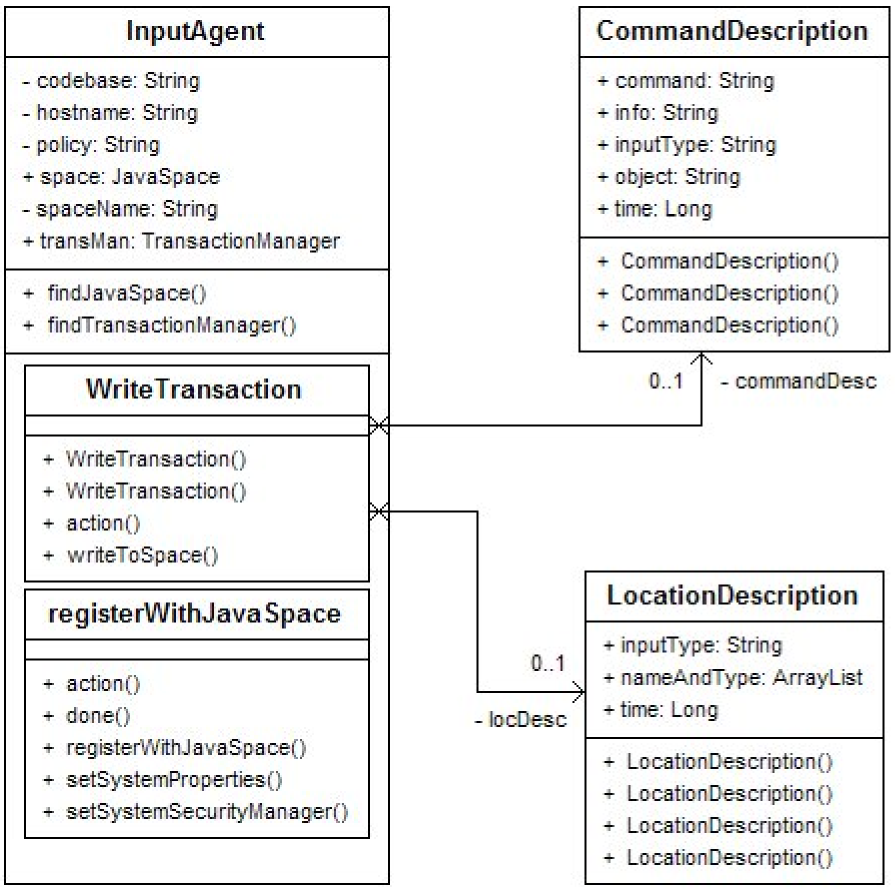


Figure 5.4: The input agent with its behaviours

The main task of this agent is to utilize the connection to the JavaSpace.

Therefore it has information about the JavaSpace, and it has behaviours for finding and connecting to a JavaSpace at startup and methods to update the JavaSpace.

The InputAgent includes two behaviours. Both behaviours extends the Jade OneShotBehaviour and are managed through each agents behavioural module. The behaviour registerWithJavaSpace is responsible for connecting to the specified URL where the JavaSpace exist, find the Lookup service and finding the named JavaSpace and a TransactionManager. Transactions are used when entries are passed on between the agents and the JavaSpace, and the TransactionManager are used to control these transactions. The InputAgent holds this information after it is found.

The second behaviour implemented in the InputAgent, is the WriteTransaction behaviour. This is responsible to update the JavaSpace with the new information that is provided by the detected input. As seen from figure 5.4, this behaviour is connected to the entries CommandDescription and LocationDescription. The WriteTransaction writes the new description on the JavaSpace and deletes eventual old description provided by the same agent. The process of taking an old entry from the space, and writing a new one, is formed as a transaction.

5.2.3 CommandAgent

Two agents are implemented for command input. These are the TextAgent and the SpeechAgent. As shown in figure 5.2 these agents inherit the CommandAgent, which again inherit InputAgent. Figure 5.5 shows the implementation of the input agents that is responsible for collecting command input.

The CommandAgent contains methods for making the entry CommandDescription based on the information from the user, and then writing the CommandDescription to the JavaSpace by using the WriteTransaction behaviour from the InputAgent.

SpeechAgent

The SpeechAgent listen to speech input from the user, analyse this and send relevant information to the blackboard. Figure 4.6 showed the process diagram for the CommandAgent, and the figure showed that four processes were idetified to accomplish the goal of the CommandAgent.

The SpeechAgent listen for speech input from the user. IBM ViaVoice is used for speech recognition. This is a speech engine that converts speech to text. The Java Speech API is used to get access to the IBM ViaVoice speech

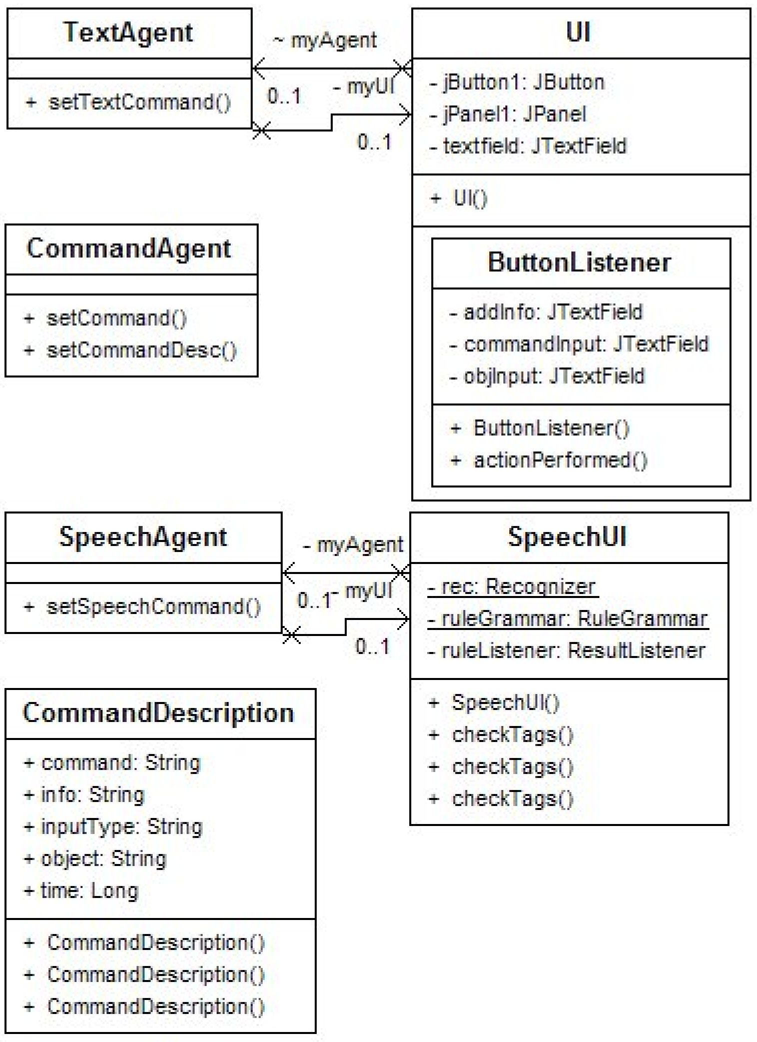


Figure 5.5: The agents that is used to collect command information from the user

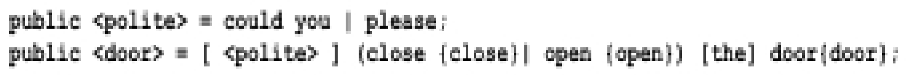
recognizer. To define the words that a user can say, a grammar has to be defined. The basic functionality of this speech recognizer is grammar management, and result production when something that matches the grammar is spoken.

The speech engine is a state system where each state defines a particular mode of operation. A speech engine must be in one of four possible allocation modes: DEALLOCATED, ALLOCATED, ALLOCATING_RESOURCES, DEALLOCATING_RESOURCES. For a recognizer to listen for incoming speech, it has to be in the ALLOCATED state. When the recognizer is allocated, several choices of states has to be done. It could either be PAUSED or RESUMED. For the recognizer to receive input, it has to be resumed. If it is paused, it is the same as turning of the microphone. The second state system of an allocated recognizer, is FOCUS_ON or FOCUS_OFF. This indicates if a specific recognizer instance has the speech focus. This is especially important when more than one application share the same underlying recognition. The third state system of an allocated recognizer, indicates the current recognition activity. Three states are possible: LISTENING, PROCESSING, and SUSPENDED. When listening, it listen for incoming audio, but has not detected speech yet. When processing, it process incoming speech that may match an active grammar. As suspended, it is temporally suspended while grammars are updated. In this state, the audio input is buffered and processed once the recognizer returns to listening and processing states.

As the recognizer is in LISTENING mode, and incoming speech is detected, it changes to PROCESSING mode. When incoming speech is processed, it check for a match with active grammars. If a match is found, a result is created. This result is what provide the application with input. It is from this result the meaning should get extracted. As long as in the PROCESSING state, a result can be updated. When a recognition is completed, it changes to SUSPENDED state. Then a result finalization event is performed to indicate that the result has got all the information it should. After the suspended mode, the recognizer goes back to LISTENING mode.

The grammar design is a very important part of the SpeechAgent. This grammar is in Java Speech Grammar Format, JSGF. This specify what can be said, and this has to consider objects in the context, and which commands can be done on these. A rule grammar is used here, and is therefore provided by the application. The grammar is needs to have information about the environment, and the grammar could be on the form:

The grammar defines the set of tokens the user can say, and how these tokens could be spoken. In the rule grammar used, the rules are defined by tokens that is relevant for this example. When using this rule grammar, the recognition process is constrained, and error rate should be notable reduced.



```
public <polite> = could you | please;
public <door> = [ <polite> ] (close {close}| open {open}) [the] door{door};
```

Figure 5.6: An example of a grammar in JSGF

As the recognizer detect incoming speech that matches an active grammar, the application is provided with a recognition result by the recognizer. A result listener is used to receive events when a result matches the grammar. A result could either be accepted or rejected. As a result is accepted, an event is issued in the resultListener, and what should take place could be specified in the application. If the recognizer is not confident that it has recognized a result correct, the result will be rejected. But even if a result is accepted, it is not sure that the recognition is correct. Misrecognition could still happen, and that is were multimodal interfaces could help in reducing the error rate.

As a result is accepted, the meaning has to be extracted. In this case this will be a command, and in some cases a object and some extra information about what should be performed. When this information is found, the entry which is placed on the blackboard is made, in this case the CommandDescription.

The class SpeechUI shown in figure 5.5 is used to start the speech recognizer and listen for new input. The speechUI uses a rule grammar and a rule listener. The rule grammar specifies what can be said, and is discusses in an earlier chapter. The rule listener listen for speech input that matches the rule grammar, and when the user has spoken an allowed sentence, the rule listener notify this.

As the user speaks a sentence that matches the grammar, the meaning of it has to be extracted for it to be possible to make a CommandDescription of it. To do this, tags are used in the grammar. As a possible command is spoken during the sentence, a tag is placed after this command. When a sentence is accepted, the tags that are in this sentence are stored in an array. If the sentence "please close door" from figure 5.6 was spoken, the array has the two tags "close" and "door" as elements. This makes it possible to extract the relevant information from a sentence. This information are then stored in the CommandDescription as the values of the attributes command and object. The same method is used for eventually extra information. The result listener uses these tags when the SpeechAgent's setSpeechCommand() is called. The methods for setting the CommandDescription and the WriteTransaction behaviour are inherited from the CommandAgent and the InputAgent.

When using the speech recognizer, a few steps has to be done in the implementation. At first a recognizer has to be created. Language and user can be chosen here. Since different users can have their own profiles in ViaVoice, it could be an advantage with respect to recognition accuracy to teach ViaVoice how you speech. If these are not specified, default values will be used. After a recognizer is created, it has to be put in the ALLOCATING state. The next step is to load and enable grammars, and in this application one external file is loaded to use a rule grammar. After the grammar is loaded and enabled, a listener has to be attached, and a rule listener is here connected to the recognizer. For the recognizer to start listen, it has to be set in the FOCUS_ON state and RESUME. When this is done, it should start listening for speech input.

TextAgent

The TextAgent is responsible for collecting textual input from the user. A class called UI is used for managing the Graphical User Interface. The GUI contains three textfields and a button. The current implementation do not support natural written language, which means the command information has to come through the three textfields. The CommandDescription shows what relevant information should be collected. The information that could be collected through the GUI is the command, the object and some extra information that could be relevant, for example if a room should be reserved, this extra information could be at which time. The time is collected when the button is clicked and the CommandDescription is ready to be written to the JavaSpace.

At startup, the agent's behaviours are started. For the TextAgent, only the registerWithJavaSpace that is inherited from the InputAgent is started up. A Graphical User Interface which listen for user inputs is also started.

As said, a Graphical User Interface is used to collect command information. The user writes the information in the textfields, and at least the command field got to contain something. The other two fields are optional. A ButtonListener is used to decide when the command information is finalized, and as soon as the user clicks the button, the text that is written in the textfields are collected and TextAgent's setTextCommand() is called. This uses the methods inherited from the CommandAgent which sets the CommandDescription based on the textual input, the agent type, and the time. When the CommandDescription is finalized, the WriteTransaction behaviour is executed.

5.2.4 LocationAgent

Figure 5.7 shows the LocationAgent, and figure 5.7 the agents that inherit it, and the other classes that is used in the process of collecting location information from the user.

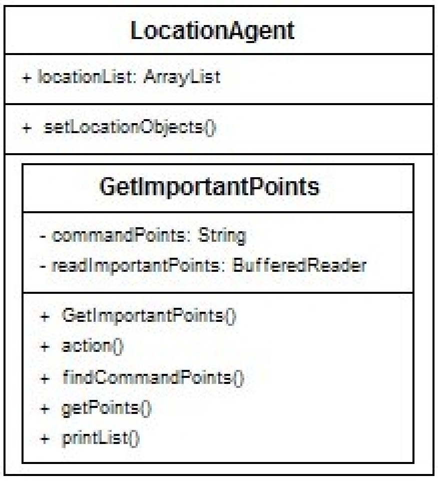


Figure 5.7: The LocationAgent

The LocationAgent has one behaviour implemented, the GetImportant-Points behaviour. This is responsible for getting the knowledge about the environment from an external source, and in this implementation the knowledge is stored in a text file. This information will be loaded to the Location-Agent at startup, and contain information about which objects that exists in the environment, and for which coordinates it is accessible.

The LocationAgent also contain a method for setting which objects a specific input coordinate refers to. As coordinates are found either from input from a map or a physical position, the knowledge about the environment is used to decide which objects these coordinates could refer to.

Two agents are implemented for input collection of location information from the user, and these are the MapAgent and the PositionAgent. Both agents implements the methods and behaviours from the InputAgent and the LocationAgent as shown in figure 5.2.

To decide which objects each coordinate refers to, some knowledge about the world has to be used. This refers to points on the map or in the physical environment, and tells which coordinates refers to which objects. This information is stored in an external file, and is imported to the LocationAgent when started up. An example of this could look as figure 5.9.

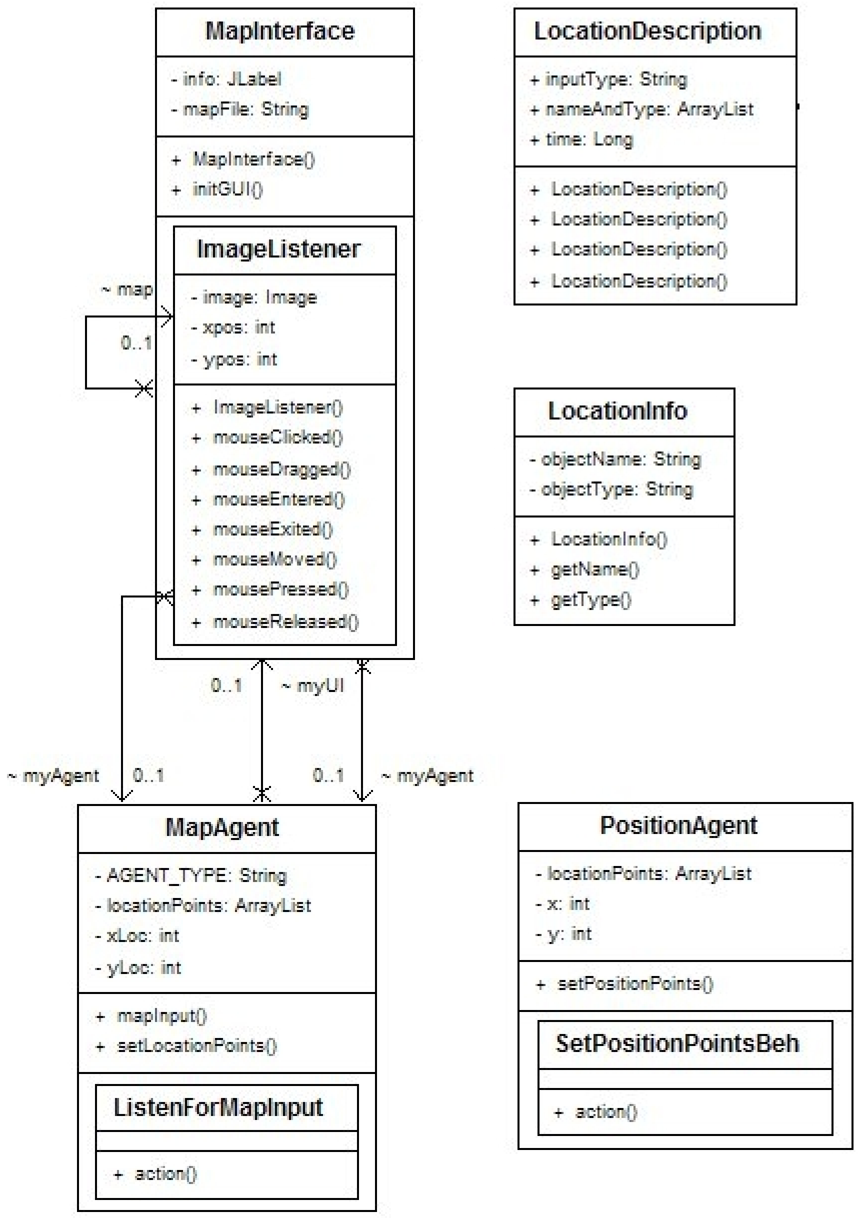
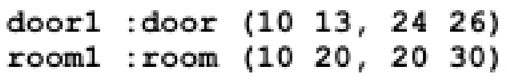


Figure 5.8: Location input



```
door1 :door (10 13, 24 26)
room1 :room (10 20, 20 30)
```

Figure 5.9: Example of location points

This is on the form objectName :objectType(xminimum xmaximum, yminimum y maximum). If the coordinates for a user would be (12,25), then this has to be checked with this information, and in this example both door1 and room1 would be potential objects. All potential objects has to be considered and sent further to the common space. At this form, each object is in a rectangle of coordinates.

MapAgent

The MapAgent uses a Graphical User Interface formed as a map for user input. This is the MapInterface. An ImageListener is used to detect which objects the user points to. In the current implementation, these coordinates registers one specific object, and therefore reacts on a mouse click. As the user clicks the mouse on the map, the horisontal and vertical position is stored, and the setLocationPoints() method to the MapAgent is called. The potential objects that the user refers to are then stored as a LocationDescription.

The attributes that are stored in the entry LocationDescription is inputType, nameAndType, and time. The inputType is the input modality, and is in this case maplocation. The nameAndType attribute is a list which contains an object name and an object type.

To decide which objects the coordinates could refer to, setLocationPoints() are used. By using the LocationAgent's locationList which contains the information about the environment, and the setLocationObjects() method, a match will be searched for. Each object in the locationList contains their location, and by searching and try to match the location coordinates against the locationList, it should find the potential objects that coordinate could refer to.

The MapAgent uses the behaviour GetImportantPoints inherited from the LocationAgent to get the knowledge about the environment and how the coordinates are related to the map. The attribute locationList is used to store this information in the agent. The MapAgent has a behaviour called ListenForMapInput which is responsible for listening to input from the user. This behaviour will start the MapInterface and listen for map input from the user. As a map input is detected, the x and y coordinates are stored and setLocationObjects inherited from the LocationAgent is called. This uses the locationList to decide which objects potentially refers to the coordinates, and the LocationDescription is made using this information. As the MapAgent has finalized the LocationDescription, it can be written to the JavaSpace by using the WriteTransaction

PositionAgent

The PositionAgent is not fully implemented as intended. This agent should collect the user's physical position by using the Cordis RadioEye. This is a network positioning system that finds the computers physical location. In this implementation there have been some problem with the connection to the RadioEye, and the implementation is not finished with respect to this. This agent is now simulated to contain a static physical location. This is to show how the fusion process works when more than one LocationAgent exists in the system.

The PositionAgent has a behaviour called SetPositionPointsBeh which should listen for inputs from the RadioEye, and get the eventual new position and then made a new LocationObject if there is any change in potential referring objects.

The GetImportantPoints inherited from the LocationAgent is also used here to get external knowledge about the environment and make the locationList. This should be the same environment, but the coordinates in the physical environment is probably not the same as in the map, so these should be interpreted to refer to the same object.

The registerWithJavaSpace and the WriteTransaction behaviours inherited from the InputAgent, has the same functionality as described in the previous three agents.

5.2.5 FusionAgent

The current implementation of the fusion agent is implemented to integrate location input and command input which is found through the previously described input agents. As figure 5.2 shows, the FusionAgent inherit the InputAgent. As mentioned earlier, a late semantic fusion method is used, and frames are used as representational structure to decide semantic compatibility.

Figure 5.10 shows the FusionAgent with its methods and behaviours. Two behaviours are implemented, SetPotentialLocationObjects and SpaceListener. The behaviour control module first starts the registerWithJavaSpace inherited from the InputAgent. After the connection and services related to the JavaSpace is started, the SetPotentialLocationObjects starts. This behaviour gets external knowledge about which commands can be performed on which objects. The agent gets this knowledge stored in a list called objectCommands. The last behaviour started at startup is SpaceListener. This is responsible to listen for input from the blackboard JavaSpace.

For the notify() method to listen for new input to the JavaSpace, a lis-

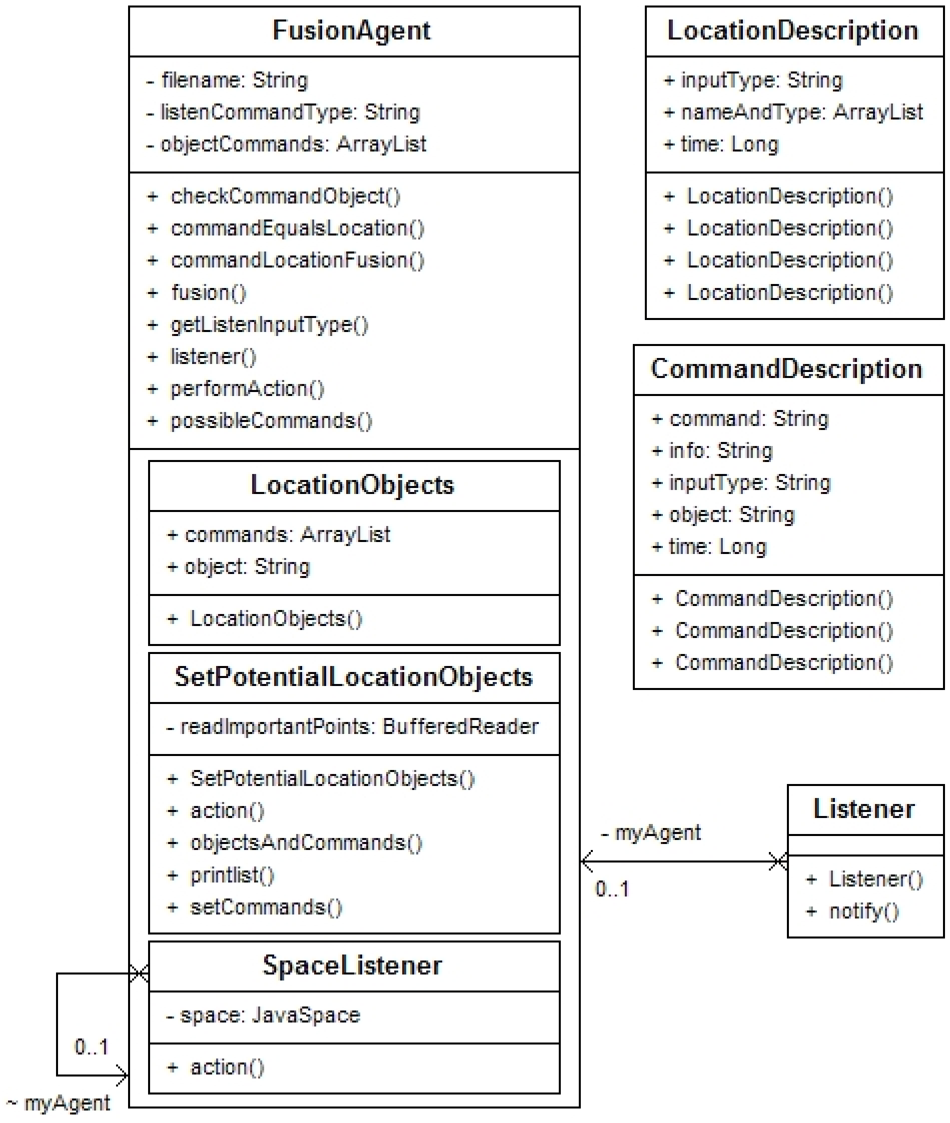


Figure 5.10: FusionAgent

tener class has to be used. This describes what the agent should listen for. In this implementation, the FusionAgent listens for a new CommandDescription. The listener will then collect this new command input together with all LocationDescriptions that exists on the JavaSpace.

As new command input is detected, and the location input is collected, the FusionAgent starts the fusion process. The location information are at first sorted as there probably will be two input modalities, map and position. To sort them, the time value will be used, and a hysteresis value on the map input decides which should be preferred. In the present implementation, this hysteresis value is 2 seconds, that means if there is between 0 and 2 seconds difference in time of the completion of the command input and the map input, the map input will be preferred, if not the position input will be preferred. This sorted list are used in the command-location fusion. If a match is not found in the preferred location input, a match will be searched for in the next modality in the list. There is no specific reason why 2 seconds are chosen for this other than it seems to work as desired. To find the best value, an extensive test should be done on this, but this was not a high priority in this work.

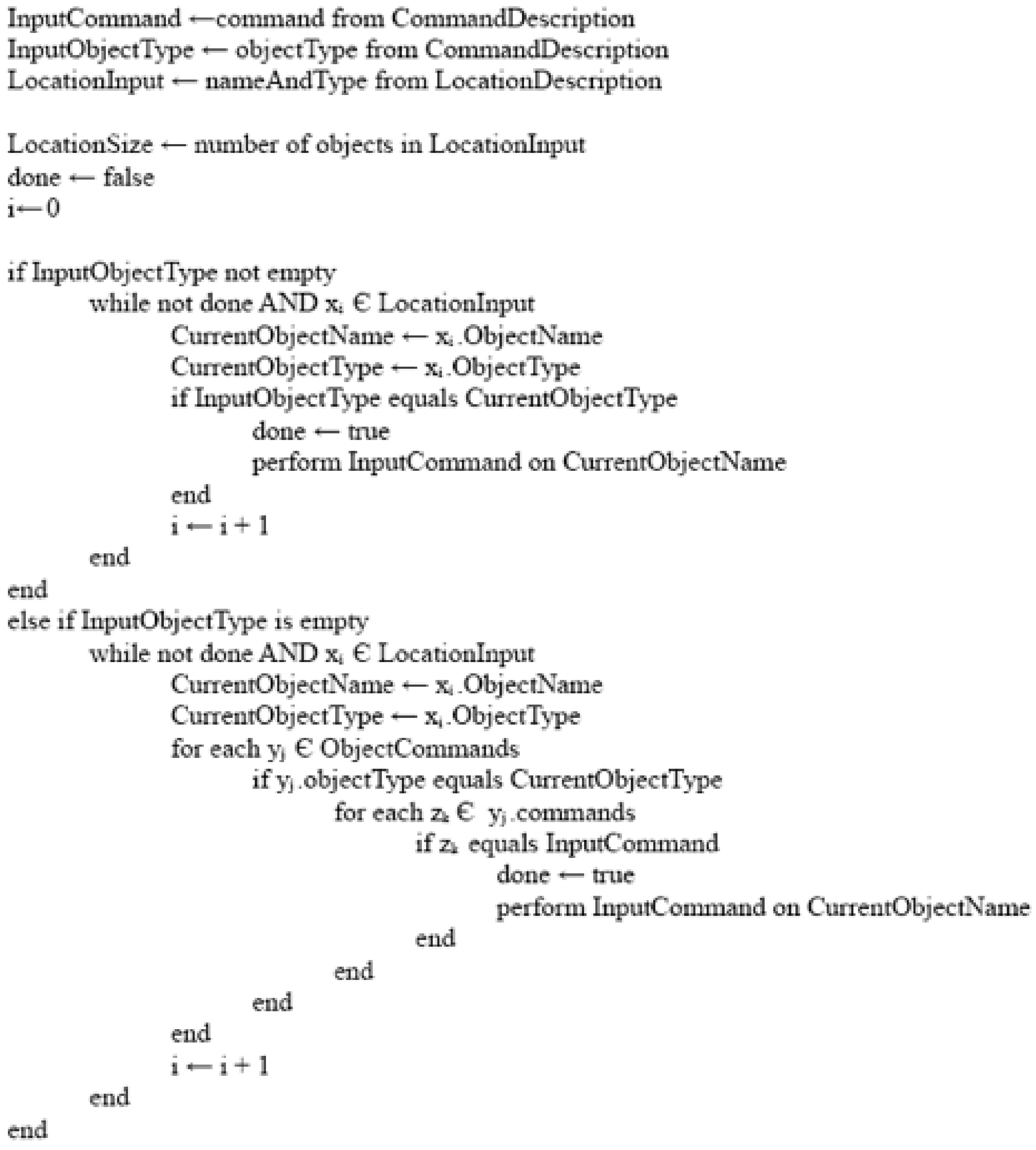
5.3 Fusion algorithm

The fusion algorithm is responsible for multimodal integration of a location base information and command based information. This algorithm is a part of the FusionAgent, and is called through the execution of the *commandLocationFusion()* method.

The algorithm get as input the command and object type which comes from the commandDescription from the blackboard in addition to the list of potential location objects. It also uses the information about all possible objects in the environment, and which commands can be performed on these objects.

At first it does a check if the object type is given as input from the command description. If this object type is given, a loop will be started. This will go trough each element of the list of potential objects from the location input until an eventual match is found. Each of these elements consists of an object name and an object type. Each elements object type will be matched against the object type given from command description. If this matches, the algorithm is done and an action is performed. If a match is not found, an recognition error occurs.

If the command description is not given from the first check, a more complex method has to be used to find a match. This uses a set of 3 nested loops. The first loop goes through each element from the list of potential objects from the location input until a match is eventually found. As before, these elements has a object name and a object type. The next loop will go through each elements form the list of all possible object types in the environment. If one of these object types matches current object type from the location input, then check the commands. If the input command matches one of the possible commands, the a match is found. If the match is found, the loops ends, and a action can be performed.



```
InputCommand ← command from CommandDescription
InputObjectType ← objectType from CommandDescription
LocationInput ← nameAndType from LocationDescription

LocationSize ← number of objects in LocationInput
done ← false
i ← 0

if InputObjectType not empty
    while not done AND $x_i$ ∈ LocationInput
        CurrentObjectName ← $x_i$.ObjectName
        CurrentObjectType ← $x_i$.ObjectType
        if InputObjectType equals CurrentObjectType
            done ← true
            perform InputCommand on CurrentObjectName
        end
        i ← i + 1
    end
end
else if InputObjectType is empty
    while not done AND $x_i$ ∈ LocationInput
        CurrentObjectName ← $x_i$.ObjectName
        CurrentObjectType ← $x_i$.ObjectType
        for each $y_j$ ∈ ObjectCommands
            if $y_j$.objectType equals CurrentObjectType
                for each $z_k$ ∈ $y_j$.commands
                    if $z_k$ equals InputCommand
                        done ← true
                        perform InputCommand on CurrentObjectName
                    end
                end
            end
        end
        i ← i + 1
    end
end
```

Figure 5.11: Fusion algorithm

Chapter 6

Demonstration

An example is implemented to demonstrate the current implementation of the architecture. The four input modalities speech, text, map and physical position is implemented in a fictitious environment.

6.1 The example

The example is located in a floor with a few offices, meeting rooms and student workplaces. In addition to these rooms, windows, curtains and doors are given as objects which are possible to perform actions on. In this section each implemented input modality will be presented together with the scenarios the system are briefly tested in to show how the implementation works.

6.1.1 Map location

Figure 6.1 shows the map used in the implementation. The map shows the objects that are used in this example. Four different object types are identified; room, door, curtain and window. Each instance of an object type has an unique object name, for example office1 and office2.

As the user points somewhere on the map, the coordinates are detected. The MapAgent needs some information about where each object are located to transform these coordinates to objects. For this example, a list of each object has been made with information about which object type it is and it's coordinates. Figure 6.2 shows how the information in this external file are stored with information about the object name, object type, x minimum, x maximum, y minimum and y maximum. These information are loaded to the MapAgent at startup and are used in the input analysis.

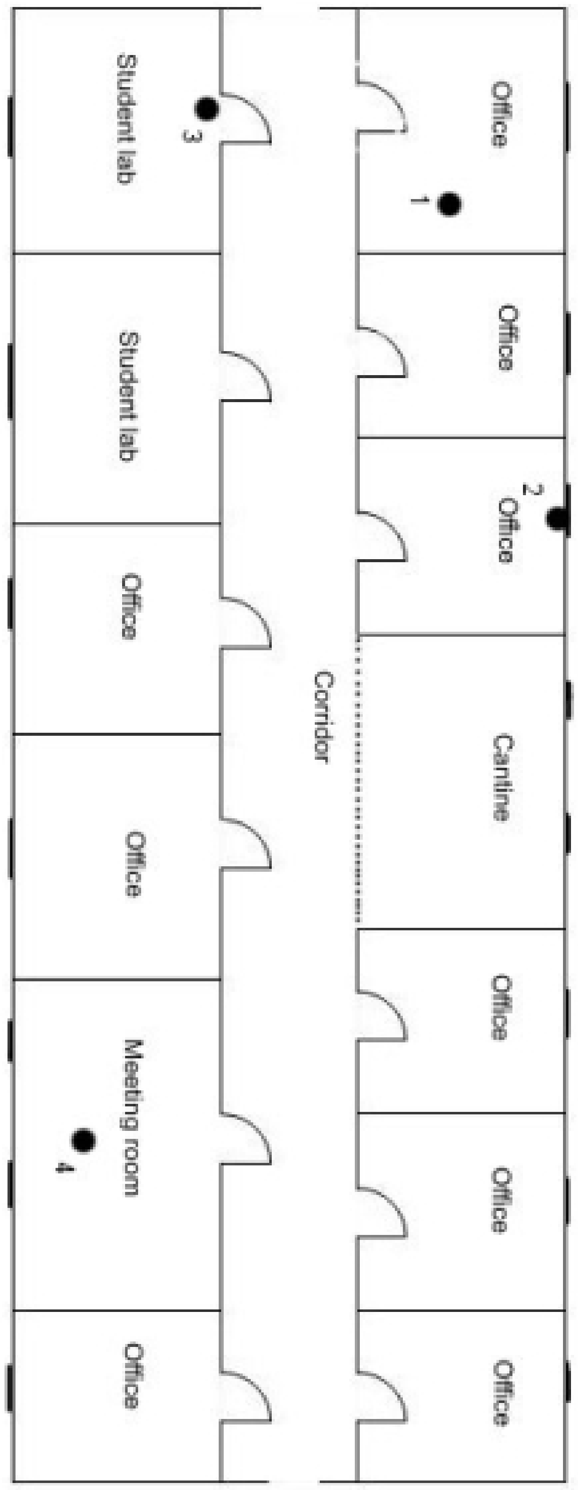
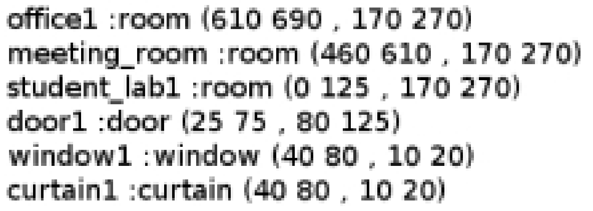


Figure 6.1: The map used in the example



```
office1 :room (610 690 , 170 270)
meeting_room :room (460 610 , 170 270)
student_lab1 :room (0 125 , 170 270)
door1 :door (25 75 , 80 125)
window1 :window (40 80 , 10 20)
curtain1 :curtain (40 80 , 10 20)
```

Figure 6.2: Example of how the location information is stored externally

The four point drawn into the map, just refers to the 4 scenarios, and will be discussed later.

6.1.2 Physical position

As discussed earlier, this should use the Cordis Radioeye to find the users physical position. This should use the same map, but the physical environment would not use the same coordinates for the same objects since the map is scaled down. The difference has to be found and the coordinates could be transformed in a way that the map location and the physical position fit each other. Since the implementation of the PositionAgent is not finished, the PositionAgent holds a static position.

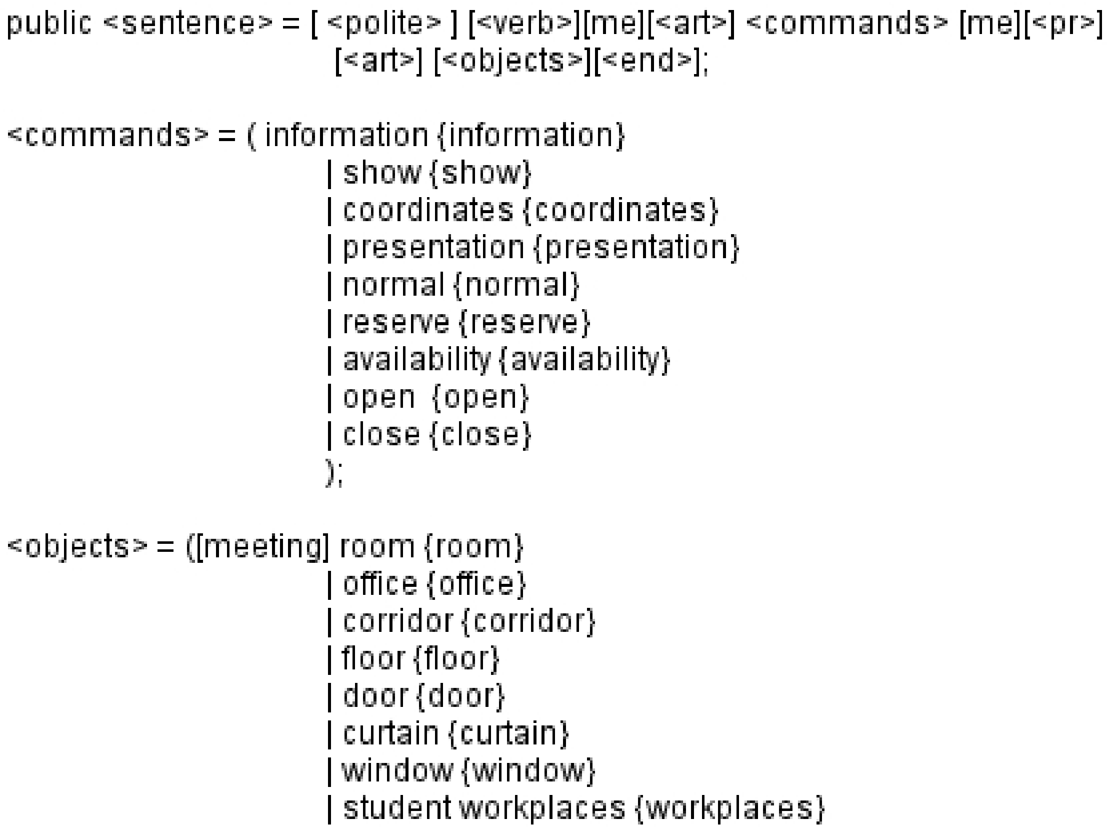
6.1.3 TextAgent

In the current implementation, the text agent do not support natural language, and is used only for keyboard input to fill in the necessary parts in the CommandDescription. It is implemented as a GUI with 3 textboxes; one for command, one for object, and one for additional information.

6.1.4 Speech command

Speech is the primary command input, and as discussed earlier, the IBM ViaVoice provide the recognizer. For the application to understand what is said, the grammar design is important. An example of the speech grammar is shown in figure 6.3. This show a sentence that could be spoken, and some commands and location objects that can be used in that sentence.

This grammar just tells which words that are allowed, and not allowed combination with respect of combination of location objects and commands. This will be checked in the part of the fusion. For example the sentence "reserve door" is allowed, but do not make any sence. Anyhow, a command



```
public <sentence> = [ <polite> ] [<verb>][me][<art>] <commands> [me][<pr>]
                        [<art>] [<objects>][<end>];

<commands> = ( information {information}
                        | show {show}
                        | coordinates {coordinates}
                        | presentation {presentation}
                        | normal {normal}
                        | reserve {reserve}
                        | availability {availability}
                        | open  {open}
                        | close {close}
                        );

<objects> = ([meeting] room {room}
                        | office {office}
                        | corridor {corridor}
                        | floor {floor}
                        | door {door}
                        | curtain {curtain}
                        | window {window}
                        | student workplaces {workplaces}
```

Figure 6.3: Speech grammar

could not be done since the fusion agent has the information about which commands can be performed on which objects.

6.1.5 The scenarios

Four scenarios have been made to show how the system works. This is tested by using all modalities, and I will present the analysis of it in this subsection.

The only modality that could be trained, is speech. For me as the user, I have trained it as much as is recomended in the IBM ViaVoice user guide, and it should therefore work satisfactory. Since not beeing a native english, this could decrease the quality of the recognition. For a neutral user which has not trained the speech recogniser, the recognition rate would probably be lower.

Issues that is important when showing how the system works are how accurate the speech recognizer are, the different location modalities, and the priority of the location input with the 2 seconds rule for the map input.

As said, four scenarios are made to test the system. The four point marked on the map, should be the coordinates it should be referred to in the four scenarios. Points 1, 2 and 4 will be referred to by clicking the on the map, while point 3 should refer to the present physical location. The commands which should be performed is:

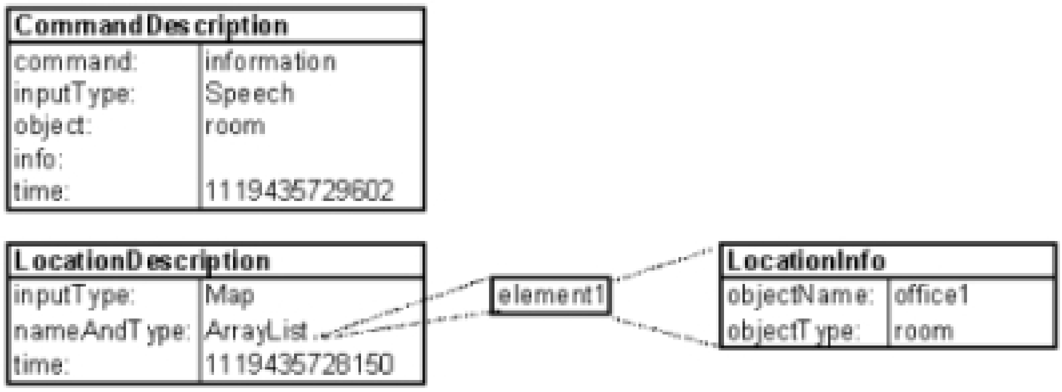
- Scenario 1: I want some information about the room called office1.
- Scenario 2: First I want to close the window, and the close the curtains.
- Scenario 3: I want to close the door where I am.
- Scenario 4: I want to reserve the meeting room at 3 o'clock.

Scenario 1: For this scanario, map and speech modalities are combined. The example is performed by clicking on the map at point 1 while speaking the following sentence: "Could you give me some information about this room".

From the SpeechAgent, the sentence "Could you give me some information about this room" is a match from the grammar. From the grammar, the input to the application will come from the tags as "information" and "room". In the making of the CommandDescription, the command will be set to "information", and the objectType will be set to "room".

The MapAgent is responsible for the other input modality, and collects the location input and makes the LocationDescription. When the mouse is clicked somewhere on the map in point 1, the coordinates are found. In this example point (103,71) is clicked. By using the knowledge about the map environment, and traversing the list of where the different map objects are located, it is found that office1 is a potential match.

Figure 6.4 shows the CommandDescription and the LocationDescription that is written to the space. As the figure shows, the time difference between the map input and the command input is less than two seconds, which means the map input is a priority for location input.



CommandDescription	
command:	information
inputType:	Speech
object:	room
info:	
time:	1119435729602

LocationDescription	
inputType:	Map
nameAndType:	ArrayList
time:	1119435728150

element1

LocationInfo	
objectName:	office1
objectType:	room

Figure 6.4: Scenario 1

When finding the joint interpretation for this input, the inputs must be merged. The command is ”information”, and ”room” is given from the CommandDescription. Since object type is given, the elements in the name-AndType list are checked for a matching object type. As seen, ”room” is the objectType of ”office1”, and the match is found. The joint interpretation for these inputs are ”information office1”.

Scenario 2: In this scanario I want to test both speech and text modalities together with the map. At first I speak ”close” while chosing point two, then I write ”close curtain” and click on 2.

Figure 6.5 shows the CommandDescription and the LocationDescription that is written to the JavaSpace after the speech and map input are collected and analysed.

The time difference are about 0,6 seconds, and once more the map input are chosen as location input. This time only the command are given from the speech input. To find a potential location match, the knowledge about which commands are allowed on the different objects are used. In this case, 3 types of objects are potential from the map input; room, window and curtain. By using this knowledge, the command ”close” could be used both for window

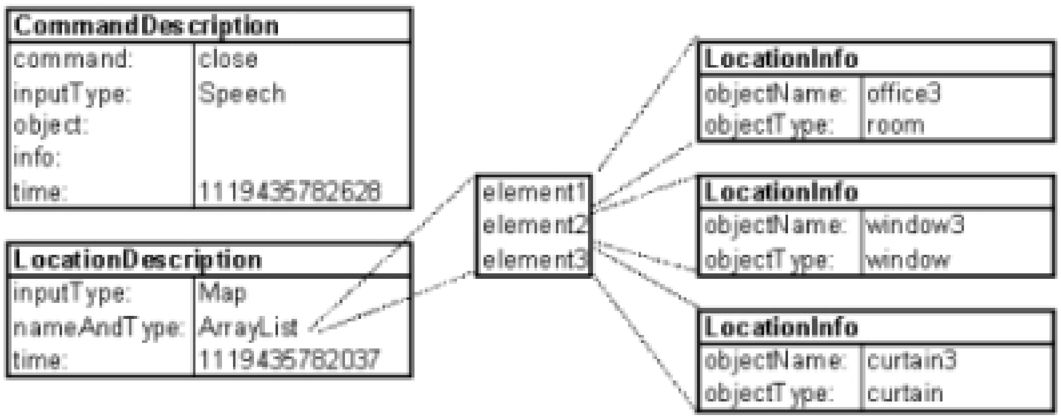


Figure 6.5: Scenario 2

and curtain, and since window are preferred, this will be used. The joint interpretation will be "close window3".

The second part of this scenario are to open the curtain. The only difference in the LocationDescription will be the time attribute. This will change since the map is clicked again closely in time to the new CommandDescription. The CommandDescription will change it's inputType to "text", while the object attribute will be given the value "curtain". In this case the joint interpretaion will be "close curtain".

The second part of this scenario is given to show that the object have to be given form the command input if the curtain are going to be chosen since it has lower priority than window. If this is not given, the highest priority will be chosen since both could use the close command.

Scenario 3: For this scenario, I just speak "close" once more without clicking the map. This time it should close the door close to point 3.

Just as in scenario 2, the command "close" are spoken. However, in this scenario it will not be followed by a map location within two seconds. The difference compared with scenario 2, is that since the difference in time between map and speech input are more than 2 seconds, the position input will have highest priority. Figure 6.6 shows the CommandDescription and the LocationDescription.

The PositionAgent gets the positions where the user is located. In this case, the user is at point 3, and this could refer to both student_lab1 and door7. When the user speaks the command "close" and the command is not followed by any map location input within 2 seconds time difference, the last known position of the user will be used.

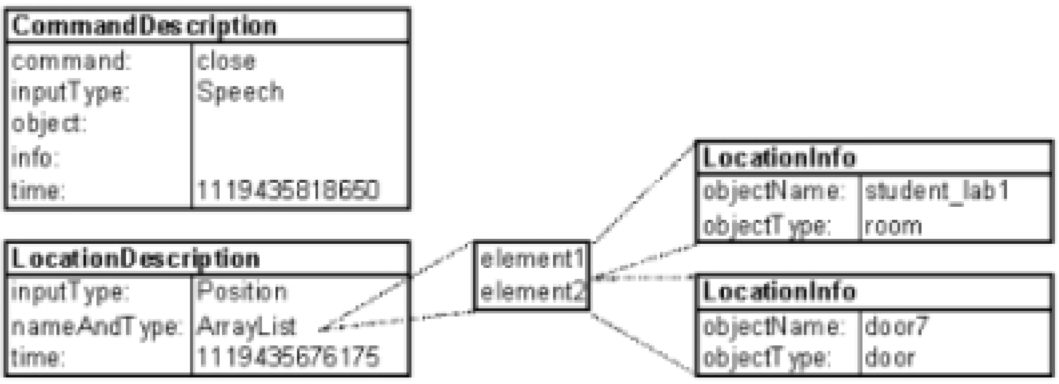


Figure 6.6: Scenario 3

The process of finding a joint interpretation works just like in scenario 2. Since the object is not given from the ocmmand input, the knowledge about the environment has to be used. In this example the command "close" could refer to "door", but not to "room". This means the joint interpretation would be "close door7".

Scenario 4: This scenario once more combine speech with map location. I click on point 4 while speaking the sentence "could I reserve this room at three".

This scanario is made to illustrate the use of the extra information. In this case the user wants to reserve the meating room at 3 o'clock. Figure 6.7 shows the CommandDescription and the LocationDescription after input analysis. Once more the map location is chosen since the time difference is about 0,7 seconds.

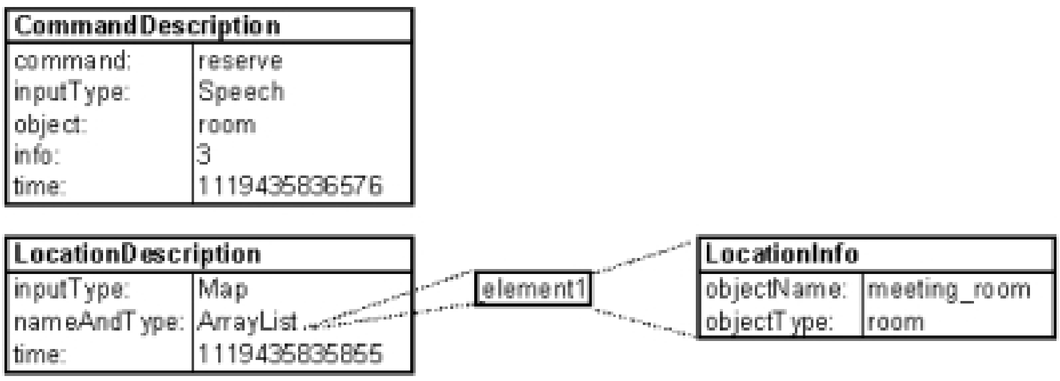
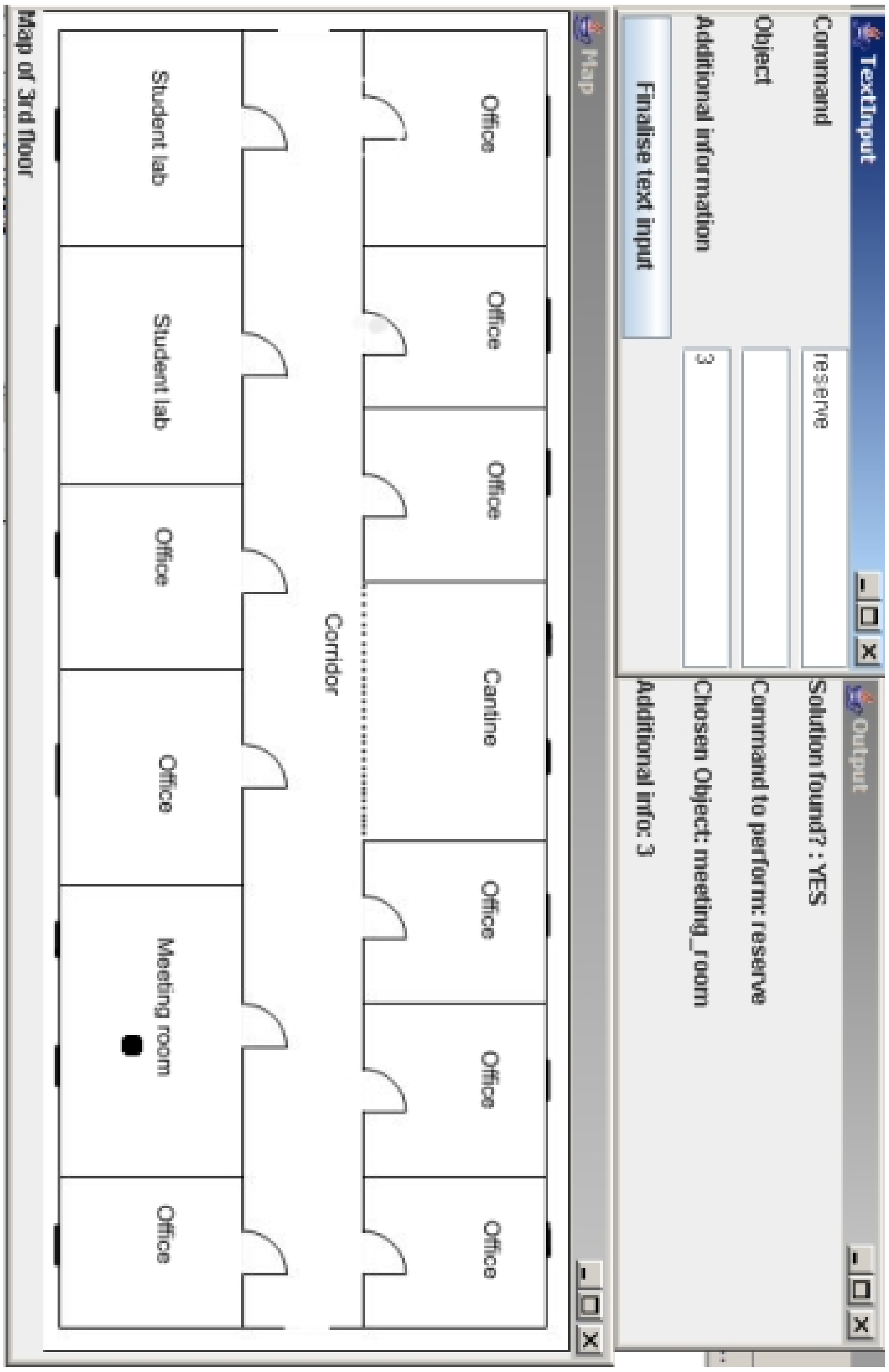


Figure 6.7: Scenario 4

In this case, the object is again given from the command input, and this matches the object type given from the location input. This makes the matching process easy, and the joint interpretation is "reserve meetin_room 3".

Visual example

Figure 6.8 shows the graphical interface. The three agents TextAgent, MapAgent, and the FusionAgent has a GUI, while the SpeechAgent and the PositionAgent do not. The figure shows an example where the user want to reserve the meeting room at three o'clock. In this case, the TextAgent and MapAgent are used as input sources. The user gives the command "open", and the additional information "3" in the text GUI, and clicks on the map GUI where at the black circle. The FusionAgent's output GUI shows that a solution is found, and that the meeting_room should be reserved at 3.



TextInput

Command: reserve

Object:

Additional information: 3

Finalise text input

Output

Solution found? : YES

Command to perform: reserve

Chosen Object: meeting_room

Additional info: 3

Figure 6.8: The graphical test interface

Chapter 7

Conclusion

7.1 Discussion

In this work an architecture is proposed of the input part of a multimodal interface which combine command and location information. The goal has been to design a general framework where it should be easy to add new input modalities, and to implement a prototype which illustrate the use with a few input modalities. Towards reaching this goal, work have been done in the following areas:

- specifying the requirements for this approach.
- a literature survey of relevant theory like intelligent interfaces and agents.
- a study of previously designed system.
- designing the overall architecture.
- implementation of a prototype with input modalities as speech and text for command information and map and position for location information.

The current architecture is general, and could easily be extended with other modules without doing to many changes to the rest of the system. If a new command modality is added, this should probably lead to no changes other than adding the new agent. However, adding a new location modality would probably lead to small changes in the FusionAgent. The current architecture is not dependent of specific input modalities as long as there are modalities for both command and location. If modalities other than these are going to be added, a few changes has to be done with the FusionAgent.

One goal behind this work was to find an alternative human-cumputer interface to the traditional ones to improve usability by using multimodal input. To draw a conclusion on this, it should be extensively tested, and this is out of the scope of this thesis. However, a few comments can be said on this part. A problem with the current architecture where speech and text input are used as input for command, is that the commands are invisible for the users. A menu system could be used in addition for command input, but the users should probably know what they want to do, and that means it is important that the speech grammar is designed in a flexible way where the user has freedom to say the same things in different way. It can be very frustrating if the cumputer never understand what you try to say. This type of input has as discussed earlier the advantage over menu-based input that the command could be easier available. Advantages by using multimodal input was discussed in chapter 2.3.5, and could include flexibility, availability, adaptability, efficiency and lower error rate. The flexibility with respect of choosing input modalities should be a an advantage with this architecture. The user should be free to choose the modality it want as long as it combines a command and a location input. Advantages with respect of availability should also be seen in this architecture. For example could users with handicaps use speech to open doors. The architecture provide adaptability in a way that it is possible to change modalities for different situations, for example speech should not be used in noisy environment, and other modalities as text should then be used. Since the input collection is done by independent agents which works on their own, the efficiency should be increased since they could work in parallel. As said earlier, speech is a modality where the error rate is a problem. By using multimodal input, the error rate should be reduced as other modalities helps in the interpretation, and they should get the strengths from several parts. However, there is still a lot of work to do, and the error rate is still higher than in traditional interfaces.

The proposed architecture fulfills the requirements specified about independent input modalities. Each modality works independently of each other, but it is necessary to have a common method of representing the analysed information. A frame structure is used to fulfill this requirement where a set of attributes are given some values. Different structures are used to represent command and location. As soon as the different input agent have collected and analysed new input information, it should easily be accessible for the agent that is responsible for the fusion of the different input modalities. A blackboard model is used in this approach, where each input agent writes it results to the blackboard, and the fusion agent collect this information when needed. By using this information, the fusion agent should hopefully find the joint interpretation and decide which action should take place and to which object.The last requirement was about mobility, and both Jade and JavaSpace are implementations that support distibuted computing and mobility.

The external knowledge about the environment the system work in is a very important aspect for the system to work as desired. The knowledge needs to be specified externally and be loaded into the system when the specific environment are used. In this prototype, the required knowledge are stored in external files, and contain information about which objects exist in the environment, where they are located, and which commands are allowed to be performed on them.

This is where this work stops. How the action should be performed is not considered. I have not had any focus on the output, but there should probably be some activators which could be triggered when the action is decided.

7.2 Future work

The most important elements are implemented, but there are still a lot of work that could be done, and should be done if the system should be useful. I will discuss some of these in this section.

As just mentioned, this work stops after the input is collected and the multimodal integration is done. In the future, this could be extended to focus on the action part and the output.

Multimodal integration: The method used for multimodal integration works well for the current implementation. However, a few things should be considered. The hysteresis value mentioned earlier for choosing either map input or physical location should be tested to find the best value. For now, this is set to two seconds. The map location and the command input should be temporal compatible which means they should be closely in time. As mentioned, the value of two seconds are chosen and it seems like an potentially good value, but an extensive test should be done to find the best value.

Input modalities: It is possible to add new modalities for location and command input, and this should be easy in the current architecture without too many changes in the current system. Several other modalities were mentioned in chapter 2, and a new agent could be added to the system for each new modality. An another possibility could be to extend the current architecture to get input other than location and command. In this case a new frame structure has to be made which suits this type of input, and changes needs to be done in the FusionAgent to consider this new modality. New input types could for example be modalities that identfy the user, and the system imediately knows who the user is and information about the user could be used.

It could be useful to extend the location information to contain more than just specific objects in a specific point. The input information could be extended to find information from regions or greater areas by marking a whole area. In this case it is necessery to detect input for more than just one point, and register all points where the pointing device is moved. This could make it possible to find information about an whole area.

Another aspect with the location input is that the position agent do not work as desired as it just refers to a static position right now. An another effort to make the Cordis RadioEye work should be done. Trouble with the connection to the RadioEye is the reason for not working now, and with more available time, this should get priority. In this case, it is also necessery to make a knowledge model where the coordinates to the map and to the physical position co-ordinates.

As a location agent detect an input with coordinates, it should be analysed by using the knowledge about the environment to find potential objects it is referred to. As it is now, this list of potential object are sorted as in the knowledge base. A solution here could be by giving them some weights by deciding how close they are to the object, or by using some other models, for example the state of the object or by studying some specific patterns.

More work could be done to the SpeechAgent as well. Currently there exist a problem with the position of the tags. As mentioned earlier, as a spoken sentence match an allowed combination of words from the grammar, tags with the keywords are extracted and sorted in an array. Since it is listed in an array on position , this could be a potential source of error. As spoken input is accepted, the SpeechAgent get the list of tags sorted with command in the first position, location object in the second, and the additional information in the third position. A problem occurs if the location object is not spoken, but the additional information is, then would the SpeechAgent believe that the additional information was a location object since the tag is in position two. Several solution could be possible to solve this problem, and one solution could be to mark the tags with type. An example could be to just use use a "cmd_" in the beginning of a command tag like cmd_open. In this case, the array of tags has to be searched and the different tags is placed in their right positions.

Another aspect with speech, could be that the speech recognizer did not recognize the word correctly. The recognizer could have several interpretations of the spoken word, and this is stored in a n-best list. If the user speaks the word "test", this could easily be interpreted as "best". If the recognizer rate "best" highest, a misrecognition occur. By using the n-best list, there could be possibilities to make a list of possible commands, like for the location information with the objects.

Knowledge representation: For this system to be more general, there should be a better way to represent knowledge about the environment. An ontology could be developed to make a better understanding of the domain knowledge. Using a database would probably be a better method for storing the information used in the process of analysing input and in the fusion process. Knowledge about the environment is important when analysing input, and there should be a better method of knowledge representation when using larger and more general environments than the one that have been used in this example.

As mentioned, there could be a possibility to extend the architecture to contain information about users, and a user model should be made in this case. As mentioned in earlier chapters, the different input modalities could be individual trained, and for this to be useful, it is necessary to know about the user. In the present implementation, the speech recognizer are possible to train, but since the current approach do not hold user information, this user model used by the speech recognizer are stored in IBM ViaVoice, and the speech agent use the default user set by ViaVoice.

Language independency: Since the current implementation of the blackboard model is implemented as a JavaSpace, there is a need for a Java implementation of each agent. This is currently no problem, since Jade is used to implement the agent, and Jade is implemented through Java. In most cases this should be no problem, and if an input modality which is using an another language than Java, the agent could still normally be implemented by using Java and Jade. However, if there should be a reason to implement an agent in another freamwork than jade, and in an another language than Java, there should be no problem since jade supports the FIPA standard. A new agent could be implemented that should be responsible for communicating with the JavaSpace on the input side. As long as these agents use the same standard when communicating, they should be language independent. If non-jade implemented agents send their messages to the jade implemented agent in a common communication language, this agent could send it further on to the JavaSpace.

Appendix A

Running the example

To run the example, work on setting up the computer has to be done before it can run. This appendix will discuss what need to be done. First of all, the content on the CD should be copied to the computer since writing to the disk is necessary for the JINI services.

Three .bat files need to be started in this order: setCP.bat, startJavaSpace.bat, and startfusion.bat. The setCP.bat set the necessary classpath, the startJavaSpace.bat starts the JINI and JavaSpace services and will be discussed later in this appendix, and startfusion.bat starts the current implementation of the agents.

The current implementation of the architecture discussed in this thesis are in the fusioninput.jar file on the CD.

Java

First of all, Java needs to be installed on the machine. During this work, version JDK 1.5.0 were used.

IBM ViaVoice and Speech for Java

For the SpeechAgent to work, it is necessary to have IBM ViaVoice installed on the computer with the IBM JavaSpeech implementation of the JavaSpeech API. The IBM ViaVoice application is not on the CD, but the IBM Java Speech package is in the directory CD/ibmjs, and this needs to be installed after the IBM ViaVoice application is installed. However, it should be possible to run other agents than the SpeechAgent without ViaVoice installed.

JADE

The Jade package is on the CD, and is set in the classpath in the setCP file.

JINI/JavaSpace

The whole JINI package version 1.2.1 is on the CD. To set up and run the JavaSpace service could lead to some work. However, the files on the CD should do the set up, but sometimes it could be necessary to do individual changes. 5 services need to be started: A HTTP server, RMI activation daemon, a lookup service, a transaction manager and the JavaSpace service. The five .bat files for doing this is in directory CD/JavaSpaces, and the startJavaSpace.bat in CD directory starts all these. The codebase that are set for the http server leads to the directory CD/JSLib. If any problems should occur, the website "The Nuts and Bolts of Compiling and Running JavaSpaces Programs" [27] should be helpful.

Appendix B

Java-Code

Constants: Contains the constants, and changes to environment should be done here.

```
/** Constants.java
 * @author Ove-Andre
 *
 *
 */
public class Constants {
        //JavaSpaces
        public static final String POLICY =
                "C:\\jini1_2_1_001\\example\\lookup\\policy.all";

        public static final String CODEBASE = "http://localhost:8080/";
        public static final String HOSTNAME = "jini://localhost/";

        public static final String SPACENAME = "JavaSpaces";

        //Files
        public static final String MAP_FILENAME = "points.txt";
        public static final String OBJECT_AND_COMMANDS = "object_command.txt";

        final static String MAPFILE = "kart.JPG";

        //Agent types

        public static final String LOCATION = "Location";
        public static final String SPEECH = "Speech";
        public static final String MAPLOCATION = "MapLocation";
        public static final String TEXT = "Text";


}
```

InputAgent: This is the top agent in the hierarchy where the connection to the JavaSpace is handled.

```
/* InputAgent.java
 * Created on 23.feb.2005
 * @author Ove-Andre
 */

import jade.core.Agent;

//JS
import java.rmi.RMISecurityManager;
import net.jini.core.discovery.LookupLocator;
import net.jini.core.lease.Lease;
import net.jini.core.lookup.ServiceID;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceMatches;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.transaction.Transaction;
import net.jini.core.transaction.TransactionFactory;
import net.jini.core.transaction.server.TransactionManager;
import net.jini.lease.LeaseRenewalManager;
import net.jini.space.JavaSpace;
import net.jini.core.entry.Entry;
import net.jini.lookup.entry.*;

//Jade
import jade.core.behaviours.OneShotBehaviour;
import jade.core.behaviours.Behaviour;

import java.util.ArrayList;

public class InputAgent extends Agent{

//      JS

        private static Class[] name = null;
        private static ServiceTemplate st = null;
        private static ServiceItem si = null;
        private static ServiceItem[] services = null;
        private static LookupLocator lookupLocator= null;
        private static ServiceRegistrar registrar = null;
        private static ServiceID id = null;
        public JavaSpace space = null;
        public TransactionManager transMan = null;

        String policy = Constants.POLICY;

        String codebase = Constants.CODEBASE;
        String hostname = Constants.HOSTNAME;

        String spaceName = Constants.SPACENAME;

        /**
         * Finds the lookuplocator on the specified host
         * @param hostname
         * @throws Exception
         */
        public void setLookupLoc(String hostname) throws Exception{
                lookupLocator = new LookupLocator(hostname);
                String host = lookupLocator.getHost();
                int port = lookupLocator.getPort();
                System.out.println("Lookup Service: jini://" + host + ":" + port);

                registrar = lookupLocator.getRegistrar();
                id = registrar.getServiceID();
                System.out.println ("Lookup Service ID: " + id.toString());
    }

        /**
         * Find a named javaspace
         * @param spaceName
         * @return javaspace
         */
        public void findJavaSpace(String spaceName){

                Entry [] attr = new Entry[1];
                attr[0] = new Name(spaceName);

                Class[] type = new Class[]{JavaSpace.class};
                ServiceTemplate template = new ServiceTemplate(null,type,attr);
        ServiceMatches matches = null;

        try{
                matches = registrar.lookup(template,Integer.MAX_VALUE);
                if (matches.totalMatches == 0){
                        space = null;
                }else {
                        space = (JavaSpace)matches.items[0].service;
                        ServiceID id;
                        System.out.println("Found JS: "+ matches.items[0].serviceID);

                }
        }catch (Exception e){
                e.printStackTrace();
                space = null;
        }

}

/**
 * Find a transaction manager
 * @return transactionmanager
 */
public void findTransactionManager(){
        Class[] type = new Class[]{TransactionManager.class};
        ServiceTemplate template = new ServiceTemplate(null,type,null);

        ServiceMatches matches = null;

        try{
                matches = registrar.lookup(template,Integer.MAX_VALUE);
                if (matches.totalMatches == 0){
                        transMan = null;
                }else {
                        transMan = (TransactionManager)matches.items[0].service;
                        ServiceID id;
                        System.out.println("Found TS: "+ matches.items[0].serviceID);

                }
        }catch (Exception e){
                e.printStackTrace();
                transMan = null;
        }

}

/**
 *
 * Behavior which manage contact with JavaSpace
 */
class registerWithJavaSpace extends Behaviour{
        private String hostname;
        private String spaceName;
        private String policy;
        private String codebase;
        private boolean done = false;


        public registerWithJavaSpace(Agent ag,String hostname, String spaceName,
                        String policy, String codebase){
                super(ag);
                this.hostname = hostname;
                this.spaceName = spaceName;
                this.policy=policy;
                this.codebase=codebase;
        }

        /**
         * set system properties as where to find the policy file
         * and where to find the downloadable codebase
         */

        public void setSystemProperties(String policy, String codebase){
                        System.setProperty("java.security.policy",policy);
                        System.setProperty("java.rmi.server.codebase",codebase);
        }

        /**
         * set the securitymanager if not already set
         *
         */
        public void setSystemSecurityManager(){
                if (System.getSecurityManager() == null){
                        System.setSecurityManager(new RMISecurityManager());
                }
        }

        public void action(){
                try{
                        setSystemProperties(policy,codebase);
                        setSystemSecurityManager();
                        ((InputAgent)myAgent).setLookupLoc(hostname);
                        ((InputAgent)myAgent).findJavaSpace(spaceName);
                        ((InputAgent)myAgent).findTransactionManager();
                        if (((InputAgent)myAgent).space == null){
                                System.out.println("Find JS SPACE NULL");
                        }else System.out.println("Find JS SPACE");
                }catch (Exception e){
                        e.printStackTrace();
                }
                done = true;
        }

        public boolean done() {
                return done;
        }
}

/**
 *
 * @author Ove-Andre
 *
 * Bahaviour which is responsible to update JavaSpace
 */
class WriteTransaction extends OneShotBehaviour{
        private JavaSpace space;
        private TransactionManager transMan = null;
        private String agentType;

        /**
         *
         * @uml.property name="commandDesc"
         * @uml.associationEnd multiplicity="(0 1)"
         */
        private CommandDescription commandDesc;

        /**
         *
         * @uml.property name="locDesc"
         * @uml.associationEnd multiplicity="(0 1)"
         */
        private LocationDescription locDesc;

        public WriteTransaction(Agent ag, String agentType, CommandDescription inputCommand){
                super(ag);
                this.agentType = agentType;
                this.commandDesc = inputCommand;
                this.locDesc = null;
        }

        public WriteTransaction(Agent ag, String agentType, LocationDescription inputLoc){
                super(ag);
                this.agentType = agentType;
                this.commandDesc = null;
                this.locDesc = inputLoc;
        }

        public void writeToSpace(){
                try{
                        Transaction.Created trans = TransactionFactory.create(transMan, Lease.FOREVER);
                        Transaction txn = trans.transaction;

                        LeaseRenewalManager leaseMgr = new LeaseRenewalManager();
                        leaseMgr.renewUntil(trans.lease,Lease.FOREVER, null);

                        String out=myAgent.getLocalName()+" OUTPUT ";

                        System.out.println("AGENT "+agentType);

                        //Write Command
                        if (commandDesc != null && locDesc == null){
                                CommandDescription result;
                                CommandDescription temp = new CommandDescription(agentType);
                                while ((result = (CommandDescription) space.takeIfExists(temp, txn, 1000))
                                                != null){
                                        System.out.println("OLD DESCRIPTION " + result.inputType +" "+
                                                result.getCommand() +" " +result.getObject() + " " +
                                                result.info+ " "+ result.getTime());

                                }
                                System.out.println("WRITE: "+commandDesc.getInputType()+" "+
                                                commandDesc.getCommand()+ " "+ commandDesc.getObject()+
                                                " "+ commandDesc.getTime() +" " + commandDesc.info);
                                space.write(commandDesc,txn,Lease.FOREVER);
                        }

                        //Write LocationInfo
                        if (commandDesc == null && locDesc != null){
                                LocationDescription result;
                                LocationDescription temp = new LocationDescription(agentType);
                                while ((result = (LocationDescription) space.takeIfExists(temp, txn, 100))
                                                != null){

                                        ArrayList rlist = result.getNameAndType();
                                        int sizeR = rlist.size();
                                        out = "";
                                        for (int i = 0; i < sizeR; i++) {
                                                LocationInfo rloc = (LocationInfo)rlist.get(i);
                                                out = out + i +" "+rloc.getName()+ " "+ rloc.getType()+ " ";
                                        }
                                        System.out.println("OLD DESCRIPTION" + result.getInputType()+
                                                        result.getTime()+ out);
                                }

                                space.write(locDesc, txn, Lease.FOREVER);
                                ArrayList list = locDesc.getNameAndType();
                                int listSize = list.size();
                                String locInfoString = "LocationInfo size: "+ listSize + " ";
                                for (int i = 0; i < listSize; i++) {
                                        LocationInfo locI = (LocationInfo)list.get(i);
                                        locInfoString = locInfoString + i + " Name: "+ locI.getName()+
                                                                        " Type: "+locI.getType() + " ";

                                }

                                System.out.println("WRITE: InputType: " + locDesc.getInputType()+ " " +
                                                locInfoString+ locDesc.getTime());
                        }

                        txn.commit();
                        leaseMgr.remove(trans.lease);
//                      return true;
                }catch(Exception e){
                        e.printStackTrace();
                        //return false;
                }
        }

        public void action(){
                this.space = ((InputAgent)myAgent).space;
                if (space == null){System.out.println("NO SPACE");
                }
                this.transMan = ((InputAgent)myAgent).transMan;
                writeToSpace();
        }
}
}
```

CommandAgent: Makes the CommandDescription

```
/*CommandAgent.java
 *
 * Created on 20.mai.2005
 *
 */

import java.util.ArrayList;

public class CommandAgent extends InputAgent{

        //Sets the entryobject CommandDescription and writes to JavaSpace
        public void setCommand(String command, String object, String info, String agentType){
                CommandDescription id = setCommandDesc(command,object,info, agentType);
                this.addBehaviour(new WriteTransaction(this,agentType,id));
        }

        //Makes a CommandDescription from the commandinput
        public CommandDescription setCommandDesc(String command, String object, String info, String agentType){
                ArrayList sl = new ArrayList();
                long timeLong = System.currentTimeMillis();
                Long time = new Long(timeLong);
                CommandDescription sd = new CommandDescription(agentType, command, object,info,time);

                return sd;
        }
}
```

CommandDescription: Holds information about the input command.

```
/* CommandDescription.java
 * Created on 20.apr.2005
 */
import net.jini.core.entry.Entry;

public class CommandDescription implements Entry{
        public String inputType=null;
        public String command= null;
        public String object= null;
        public String info = null;
        public Long time = null;

        public CommandDescription(){
        }

        public CommandDescription(String objectType){
                this.inputType = objectType;
        }

        public CommandDescription(String objectType, String command,
                        String object, String info, Long time){
                this.inputType = objectType;
                this.command = command;
                this.object = object;
                this.info = info;
                this.time = time;
        }

        public String getInputType() {
                return this.inputType;
        }

        public String getCommand() {
                return this.command;
        }

        public String getObject() {
                return this.object;
        }

        public void setInputType(String type) {
                this.inputType = type;
        }

        public void setCommand(String command) {
                this.command = command;
        }

        public void setObject(String object) {
                this.object = object;
        }

        public void setTime(Long time) {
                this.time = time;
        }

        public Long getTime() {
                return this.time;
        }
}
```

SpeechAgent: Responsible for handling speech input

```
/* SpeechAgent
 * Created on 26-Aug-2004
 *
 * @author oveandre
 *
 */

public class SpeechAgent extends CommandAgent {

        private SpeechUI myUI;
        private static final String AGENT_TYPE = Constants.SPEECH;

        protected void setup()
        {
                System.out.println("Start SpeechAgent");
                myUI = new SpeechUI(this);
                this.addBehaviour(new registerWithJavaSpace(this,hostname,spaceName,policy,codebase));
        }

        public void setSpeechCommand(String command, String object, String info){
                setCommand(command, object, info, AGENT_TYPE);
        }
}
```

SpeechUI: Listen for speech input

```
/* SpeechUI.java
 * Created on 23-Aug-2004
 *
 * @author oveandre
 *
 */

import javax.speech.*;
import javax.speech.recognition.*;
import java.io.FileReader;
import java.util.Locale;

public class SpeechUI {
        static Recognizer rec;
        static RuleGrammar ruleGrammar;

        private SpeechAgent myAgent;

        //ResultListener, uses the grammar and listen for Accepted results
        ResultListener ruleListener = new ResultAdapter(){
                //RESULT_ACCEPTED
                public void resultAccepted(ResultEvent e) {
                        FinalRuleResult r = (FinalRuleResult) (e.getSource());
                        String tags[] = r.getTags();

                        int i = tags.length;
                        if (i == 1){
                                checkTags(tags[0]);
                        }if (i==2){
                                checkTags(tags[0],tags[1]);
                        }if (i==3){
                                checkTags(tags[0], tags[1],tags[2]);
                        }
                }
        };

        public void checkTags(String command){
                System.out.println("kommando: "+ command);
                myAgent.setSpeechCommand(command,"","");
        }

        public void checkTags(String command, String object){
                System.out.println("kommando: "+ command + "Object: "+ object);
                myAgent.setSpeechCommand(command,object,"");
        }

        public void checkTags(String command, String object, String info){
                System.out.println("kommando: "+ command + "Object: "+ object + "Extra info: "+ info);
                myAgent.setSpeechCommand(command,object,info);
        }

        //Initiate Speech recognizer
        public SpeechUI(SpeechAgent a){
                myAgent= a;
                try{
                                //Create a recognizer
                                rec = Central.createRecognizer(new EngineModeDesc(Locale.ENGLISH));

                                //start up recognizer(State allocate)
                                rec.allocate();

                                //load and enable grammar
                                String GrammarName = "Speech.gram";
                                FileReader reader = new FileReader(GrammarName);
                                ruleGrammar = rec.loadJSGF(reader);
                                ruleGrammar.setEnabled(true);

                                //attach a resultlistener
                                rec.addResultListener(ruleListener);

                                //commit the grammar
                                rec.commitChanges();

                                //Request focus (focus_on) and start listening (state resume)
                                rec.requestFocus();
                                rec.resume();


                } catch (Exception e){
                                e.printStackTrace();

                }
```

```
          }
}
```

TextAgent: Responsible for text input.

```
/* TextAgent.java
 * Created on 20.mai.2005
 *
 * @author Ove-Andre
 *
 */

public class TextAgent extends CommandAgent {

        private UI myUI;

        private static final String AGENT_TYPE = Constants.TEXT;

        protected void setup()
        {
                System.out.println("Start TextAgent");

                //Register with the blackboard
                this.addBehaviour(new registerWithJavaSpace(this,hostname,spaceName,policy,codebase));

                //initiate GUI and listen for input
                myUI = new UI(this);
                myUI.show();
        }

        //Accepted textInput found
        public void setTextCommand(String command, String object, String info){
                setCommand(command, object, info, AGENT_TYPE);
        }
}
```

UI: Listen for text input

```java
/* UI.java
 * Created on 10-Sep-2004
 *
 * @author Ove-Andre
 *
 */
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class UI extends JFrame{

        //GUI
        JPanel jPanel1 = new JPanel();
        JButton jButton1 = new JButton();
        JTextField textfield = new JTextField(20);

        TextAgent myAgent;

        //CONSTRUCTOR
        public UI(TextAgent ag) {
                myAgent = ag;
                jbInit();
        }

        //INIT GUI
        private void jbInit(){
                setTitle("TextInput");
                setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                Container guiBeholder = getContentPane();
                LayoutManager layout = new GridLayout(4,2,5,5);
                guiBeholder.setLayout(layout);
                JButton knapp = new JButton("Finalise text input" );
                JLabel cmd = new JLabel("Command");
                JLabel obj = new JLabel("Object");
                JLabel addit = new JLabel("Additional information");
                JTextField commandInput = new JTextField(15);
                JTextField objInput = new JTextField(15);
                JTextField addIn = new JTextField(15);

                guiBeholder.add(cmd);
                guiBeholder.add(commandInput);
                guiBeholder.add(obj);
                guiBeholder.add(objInput);
                guiBeholder.add(addit);
                guiBeholder.add(addIn);
                guiBeholder.add(knapp,BorderLayout.SOUTH);
                ButtonListener knappelytteren = new ButtonListener(commandInput, objInput, addIn);
                knapp.addActionListener(knappelytteren);
                pack();
        }


        //Sends information about the command as the button is pressed to the TextAgent
        class ButtonListener implements ActionListener{
                JTextField commandInput;
                JTextField objInput;
                JTextField addInfo;


                public ButtonListener(JTextField commandInput, JTextField objInput, JTextField addInfo){
                        this.commandInput = commandInput;
                        this.objInput = objInput;
                        this.addInfo = addInfo;
                }

                public void actionPerformed(ActionEvent hendelse){
                        String commandIn = commandInput.getText();
                        String objIn = objInput.getText();
                        String info = addInfo.getText();
                        System.out.println("Du trykket p knappen  " + commandIn);

                        myAgent.setTextCommand(commandIn, objIn, info);
                }
        }



}
```

LocationAgent: Manages the location input and makes the LocationDescription

```
 /* LocationAgent.java
 * Created on 24.jan.2005
 */

import jade.core.behaviours.*;

import java.io.*;
import java.util.*;
import java.net.URL;
import java.net.HttpURLConnection;
import net.jini.core.entry.Entry;
import net.jini.core.lease.Lease;
import net.jini.core.transaction.Transaction;
import net.jini.core.transaction.TransactionFactory;
import net.jini.lease.LeaseRenewalManager;
import org.xml.sax.*;
import org.xml.sax.helpers.XMLReaderFactory;


public class LocationAgent extends InputAgent{

        public ArrayList locationList;

        public LocationDescription setLocationObjects(int xLoc, int yLoc, String agentType){
                ArrayList l = locationList;

                String name = "";
                String type = "";
                int xmin = 0;
                int xmax = 0;
                int ymin= 0;
                int ymax= 0;
                ArrayList obj = new ArrayList();


                for (int i = 0; i < l.size(); i++){
                        ArrayList el = new ArrayList();
                        el = (ArrayList) l.get(i);
                        name = (String) el.get(0);
                        type = (String) el.get(1);
                        xmin = Integer.parseInt((String) el.get(2));
                        xmax = Integer.parseInt((String) el.get(3));
                        ymin = Integer.parseInt((String) el.get(4));
                        ymax = Integer.parseInt((String) el.get(5));

                        if (xLoc >= xmin && xLoc <= xmax && yLoc >= ymin && yLoc <= ymax){
                                System.out.println(name + " er aktuell");
                                LocationInfo locInfo = new LocationInfo(name,type);
                                obj.add(locInfo);
                        }
                }
                long timeLong = System.currentTimeMillis();
                Long time = new Long(timeLong);
                LocationDescription locObj = new LocationDescription(agentType, obj,time);
                return locObj;
        }

        //Reads locationinformation about the environment from a textfile
        public class GetImportantPoints extends OneShotBehaviour {

                        private FileReader readFile;
                        BufferedReader readImportantPoints;
                        String commandPoints;
                        private String filename;
                        private String agentType;


                        /**
                         * Read one line at the time from the textfile and stores the names and the points in an
                         * ArrayList
                         *
                         */

                        public GetImportantPoints(String filename, String agentType){
                                this.filename = filename;
                                this.agentType = agentType;
                        }

                        public ArrayList findCommandPoints(){
                                ArrayList points = new ArrayList();
                try{
                    readFile = new FileReader(filename);
                    readImportantPoints = new BufferedReader(readFile);
                    commandPoints = readImportantPoints.readLine();

                    while (commandPoints != null) {
                        if (commandPoints.length()>0){
                            points.add(getPoints(commandPoints));
                        }
                        commandPoints = readImportantPoints.readLine();
                    }
                    readImportantPoints.close();
                }catch (IOException e){
                    e.printStackTrace();
                }
                return points;
            }

            /**
             * Setting the name and the points of the ArrayList of Points
             * @param s
             * @return
             */
            public ArrayList getPoints(String s){
                int startLocPoints;
                int startType;
                int locChord;
                String name;
                String type;


                ArrayList list = new ArrayList();

                startLocPoints=s.indexOf("(");
                startType = s.indexOf(":");

                name = s.substring(0,startType-1);
                type = s.substring(startType+1,startLocPoints-1);
                list.add(name);
                list.add(type);
                boolean substr = false;
                String out = "";
                for (int j = startLocPoints; j < s.length(); j++) {
                    char chr = s.charAt(j);
                    if (chr != '('&&chr !=')'&&chr!=','&&chr!=' '){
                        if (!substr){
                            out = "";
                            out = out + chr;
                            substr = true;
                        }else out = out + chr;

                    } else if (substr){

                        list.add(out);
                        substr = false;
                    }
                }
                return list;
            }

            /**
             * Printing the ArrayList of Points
             * @param l
             */
            public void printList(ArrayList l){
                for (int i = 0; i < l.size(); i++) {
                    ArrayList el = new ArrayList();
                    el = (ArrayList) l.get(i);
                    String out = "";
                    for (int j = 0; j < el.size(); j++) {
                        out = out +" "+ el.get(j);
                    }
                    System.out.println("Points: "+ out);
                }
            }
            public void action(){
                locationList = findCommandPoints();
            }
        }
}
```

MapAgent: Responsible for map input

```
/* MapAgent.java
 * Created on 01.apr.2005
 *
 */

import jade.core.behaviours.OneShotBehaviour;
import jade.core.behaviours.SequentialBehaviour;

import java.util.ArrayList;

public class MapAgent extends LocationAgent{

        MapInterface myUI;

        private String filename = Constants.MAP_FILENAME;
        ArrayList locationPoints = new ArrayList();
        int xLoc;
        int yLoc;
        String AGENT_TYPE = Constants.MAPLOCATION;


        protected void setup(){
                System.out.println("MapAgent");


                SequentialBehaviour jsBeh = new SequentialBehaviour(this);
                jsBeh.addSubBehaviour(new registerWithJavaSpace(this,hostname,spaceName,policy,codebase));
                jsBeh.addSubBehaviour(new GetImportantPoints(filename, AGENT_TYPE));
                jsBeh.addSubBehaviour(new ListenForMapInput());

                this.addBehaviour(jsBeh);
        }

        public void mapInput(){
                myUI = new MapInterface(this);
        }

        public void setLocationPoints(int x, int y, String agentType){
                LocationDescription locationDesc = setLocationObjects(x,y,agentType);
                this.addBehaviour(new WriteTransaction(this, agentType, locationDesc));
        }

        class ListenForMapInput extends OneShotBehaviour{
                public void action(){
                        ((MapAgent)myAgent).mapInput();
                }
        }
}
```

MapInterface: Listens for map input

```
/* MapInterface
 * Created on 01.apr.2005
 *
 */

import java.awt.*;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

import javax.swing.JPanel;
import javax.swing.*;

public class MapInterface extends JFrame{
        String mapFile = Constants.MAPFILE;
        ImageListener map;
        JLabel info;
        MapAgent myAgent;

        public MapInterface(MapAgent ag){
                myAgent = ag;
                initGUI();
        }

        protected static ImageIcon createImageIcon(String path){
                return new ImageIcon(path);
        }

        public void initGUI() {
        setDefaultLookAndFeelDecorated(true);

        //Create and set up the window.
        setTitle("Map");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel jpanel = new JPanel(new GridLayout(1,1));
            map = new ImageListener(createImageIcon(mapFile).getImage(), myAgent);
            map.setName("Map");
            jpanel.add(map);
            info = new JLabel("Map of 3rd floor");
            setPreferredSize(new Dimension(1000, 550));
            add(jpanel, BorderLayout.CENTER);
            add(info, BorderLayout.PAGE_END);
        pack();
        setVisible(true);
    }

        //Listens for inputs from the map
        class ImageListener extends JComponent implements MouseListener{
                Image image;
                int xpos;
                int ypos;

                MapAgent myAgent;

                public ImageListener(Image image, MapAgent myAgent) {
                        this.image = image;
                        this.myAgent = myAgent;
                        addMouseListener(this);
                }

                public void mouseDragged(MouseEvent e){
                        xpos = e.getX();
                        ypos = e.getY();
                        System.out.println("DRAG"+xpos + " "+ ypos);
                }

                public void mouseMoved(MouseEvent e){
                        xpos = e.getX();
                        ypos = e.getY();
                        System.out.println("MOVE"+xpos + " "+ ypos);

                }

                public void mouseClicked(MouseEvent e) {
                        //Since the user clicked on us, let's get focus!
                        requestFocusInWindow();
                        xpos = e.getX();
                        ypos = e.getY();

                        System.out.println(xpos + " "+ ypos);
                        myAgent.setLocationPoints(xpos, ypos, myAgent.AGENT_TYPE);
                }
        public void mouseEntered(MouseEvent e) { }
        public void mouseExited(MouseEvent e) { }
        public void mousePressed(MouseEvent e) {
                xpos = e.getX();
                ypos = e.getY();

        }
        public void mouseReleased(MouseEvent e) { }



        protected void paintComponent(Graphics graphics) {
                Graphics g = graphics.create();



        if (image != null) {
                g.drawImage(image, 0, 0, this);
        }

                g.setColor(Color.BLACK);

                g.drawRect(0, 0, image == null ? 125 : image.getWidth(this),
                                image == null ? 125 : image.getHeight(this));
                g.dispose();
        }



        }





}
```

PositionAgent: Responsible for position input

```
/* PositionAgent.java
 * Created on 24.jan.2005
 * @author oveandre
 *
 */


import jade.core.behaviours.*;
import java.util.*;
import java.net.URL;
import java.net.HttpURLConnection;

import net.jini.core.entry.Entry;
import net.jini.core.lease.Lease;
import net.jini.core.transaction.Transaction;
import net.jini.core.transaction.TransactionFactory;
import net.jini.lease.LeaseRenewalManager;

import org.xml.sax.*;
import org.xml.sax.helpers.XMLReaderFactory;

public class PositionAgent extends LocationAgent{

        //CONSTANTS

        private static final String AGENT_TYPE = Constants.LOCATION;

        //VARIABLES

        private String filename = Constants.MAP_FILENAME;
        ArrayList locationPoints = new ArrayList();
        int x = 56;
        int y = 177;

        protected void setup(){
                System.out.println("PositionAgent");


                SequentialBehaviour jsBeh = new SequentialBehaviour(this);
                jsBeh.addSubBehaviour(new registerWithJavaSpace(this,hostname,spaceName,policy,codebase));
                jsBeh.addSubBehaviour(new GetImportantPoints(filename, AGENT_TYPE));
                jsBeh.addSubBehaviour(new SetPositionPointsBeh());
                this.addBehaviour(jsBeh);
        }

        public void setPositionPoints(){
                LocationDescription locationDesc = setLocationObjects(x,y,AGENT_TYPE);
                this.addBehaviour(new WriteTransaction(this, AGENT_TYPE, locationDesc));
        }

        class SetPositionPointsBeh extends OneShotBehaviour{
                public void action(){
                        setPositionPoints();
                }
        }
}
```

LocationDescription: Holds the location input

```
/* LocationDescription.java
 * Created on 20.apr.2005
 *
 *
 */

import net.jini.core.entry.Entry;
import java.util.ArrayList;

public class LocationDescription implements Entry {
        public Long time= null;
        public String inputType= null;
        public ArrayList nameAndType = null;
        public LocationDescription(){
        }

        public LocationDescription(String objectType){
                this.inputType = objectType;
        }

        public LocationDescription(String objectType,ArrayList list){
                this.inputType = objectType;
                this.nameAndType = list;
        }

        public LocationDescription(String objectType, ArrayList list , Long time){
                this.inputType = objectType;
                this.nameAndType = list;
                this.time = time;

        }

        public String getInputType() {
                return this.inputType;
        }

        public ArrayList getNameAndType() {
                return this.nameAndType;
        }

        public Long getTime() {
                return this.time;
        }

        public void setInputType(String type) {
                this.inputType = type;
        }

        public void setNameAndType(ArrayList nameType) {
                this.nameAndType = nameType;
        }

        public void setTime(Long time) {
                this.time = time;
        }
}
```

LocationInfo: LocationDescription has a list of object names and type, this list contains the LocationInfo.

```
/**LocationInfo.java
 *
 */

import java.io.Serializable;

public class LocationInfo implements Serializable{
        String objectName;
        String objectType;

        public LocationInfo(String name, String type){
                this.objectName = name;
                this.objectType = type;
        }

        public String getName(){
                return objectName;
        }

        public String getType(){
                return objectType;
        }

}
```

FusionAgent: Responsible of fusing the different input modalities to one meaning

```
/*FusionAgent.java
 * Created on 17-Feb-2005
 *@author oveandre
 */


import java.io.BufferedReader;
import java.io.FileReader;
import java.util.ArrayList;
import net.jini.space.JavaSpace;
import net.jini.core.lease.Lease;
import jade.core.behaviours.*;

public class FusionAgent extends InputAgent{
        String filename = Constants.OBJECT_AND_COMMANDS;
        ArrayList objectCommands;
        String listenCommandType;

        private OutputGUI myGUI;


        //Setup fusion agent
        protected void setup(){


                System.out.println("FusionAgent");
                System.out.println(codebase + " "+ hostname);

                SequentialBehaviour jsBeh = new SequentialBehaviour(this);
                jsBeh.addSubBehaviour(new registerWithJavaSpace(this,hostname,spaceName,policy,codebase));
                jsBeh.addSubBehaviour(new SetPotentialLocationObjects(filename));

                jsBeh.addSubBehaviour(new SpaceListener());

                this.addBehaviour(jsBeh);

                myGUI = new OutputGUI(this);
                myGUI.show();
        }

        /**
         *
         * @param inputCommand: Command from CommandAgent
         * @param list: List of possible locations
         * @param state: Which element of the list to focus on
         */
        public void fusion(CommandDescription inputCommand, ArrayList list, int state){
                LocationDescription location = new LocationDescription();
                int listSize = list.size();

                if (listSize > 0 && state < listSize){
                        location = (LocationDescription)list.get(state);
                }
                commandLocationFusion(inputCommand, location, list ,state);
        }

        //Check if object is given from the commandAgent
        public boolean checkCommandObject(String commandObj){
                boolean com =true;
                if (commandObj.equals("")){
                        com = false;
                }
                return com;
        }

        //Check if object given from command input equals object type from given locationobject.
        public boolean commandEqualsLocation(String command, String location){
                boolean equals = false;
                if (command.equals(location)){
                        equals = true;
                }
                return equals;
        }

        //perform output
        public void performAction(String command, String object, String addInfo){
                System.out.println("Perform action: "+ "Command: "+ command+ " Object: "+ object);
                myGUI.setSolution("YES");
                myGUI.setCMD(command);
                myGUI.setObject(object);
                myGUI.setAddInfo(addInfo);
        }



        //Checks if given command is alowed for this location object.
        public boolean possibleCommands(String command, String locType){
                int objSize = objectCommands.size();
                boolean found = false;
                int i =0;
                while (!found && i < objSize){
                        LocationObjects potentialObjectCom = (LocationObjects)objectCommands.get(i);
                        String potentialObject = potentialObjectCom.object;
                        if (potentialObject.equals(locType)){
                                ArrayList commands = potentialObjectCom.commands;
                                int commandsSize = commands.size();
                                int j = 0;
                                boolean foundCommand = false;
                                while (!foundCommand && j < commandsSize ){
                                        String potentialCommand = (String)commands.get(j);
                                        if (potentialCommand.equals(command)){
                                                foundCommand = true;
                                                found = true;
                                        }
                                        j++;
                                }
                        }
                        i++;
                }
                return found;
        }

        /**Fusion of command and location modalities
         *
         * @param inputCommand: input from CommandAgent
         * @param location: Input from LocationAgent
         * @param priList: sorted list of possible locationinputmodalities
         * @param state: Which locationinputmadality to check.
         */
        public void commandLocationFusion(CommandDescription inputCommand, LocationDescription location,
                        ArrayList priList, int state){
//              Command INPUT
                String commandObject = "";
                String command = "";
                String commandInfo = "";

                if (inputCommand!=null){
                        if(inputCommand.getCommand()!=null){
                        commandObject = inputCommand.getObject();
                        command = inputCommand.getCommand();
                        commandInfo = inputCommand.info;
                        }
                }

                //Location INPUT
                ArrayList locationInput = new ArrayList();
                if (location!=null){
                        locationInput = location.getNameAndType();
                }

                int locSize = locationInput.size();
                boolean done = false;
                int i = 0;

                //If objecttype is speeched
                if (checkCommandObject(commandObject)){
                        while (!done && i < locSize ){
                                LocationInfo loc = (LocationInfo)locationInput.get(i);
                                String locName = loc.getName();
                                String locType = loc.getType();
                                if (commandEqualsLocation(commandObject, locType)){
                                        if(possibleCommands(command, locType)){
                                                done = true;
                                                performAction(command, locName, commandInfo);
                                        }
                                }
                                i++;
                        }
                }

                //If objecttype is not speeched
                else{
                        while (!done && i < locSize){
```

```
                    LocationInfo loc = (LocationInfo)locationInput.get(i);
                    String locName = loc.getName();
                    String locType = loc.getType();
                    if (possibleCommands(command, locType)){
                            done = true;
                            performAction(command,locName, commandInfo);
                    }
                    i++;
            }
    }

    if (!done){
            state++;
            if(state<priList.size()){
                    fusion(inputCommand, priList, state);
            }else {
                    myGUI.setSolution("NO, Recognition error");
                    myGUI.setCMD("");
                    myGUI.setObject("");
                    myGUI.setAddInfo("");
            }
    }
}

public String getListenInputType(){
        return listenCommandType;
}

//Listens to JavaSpace for a new CommandDescription
public void listener(){
        try{
                Listener listener = new Listener(space,this);
                System.out.println("Fusion Listener");


                CommandDescription temp = new CommandDescription();
                System.out.println("LISTENED COMMAND" + temp.getInputType());
                listenCommandType = temp.getInputType();
                space.notify(temp, null, listener, Lease.FOREVER, null);
        }catch (Exception e){
                e.printStackTrace();
        }
}


class SpaceListener extends OneShotBehaviour{
        JavaSpace space;

        FusionAgent myAgent;

        public void action(){
                listener();
        }
}

class SetPotentialLocationObjects extends OneShotBehaviour{
        private FileReader readFile;
        BufferedReader readImportantPoints;
        private String filename;

        /**
         * Read one line at the time from the textfile
         *
         */

        public SetPotentialLocationObjects(String filename){
                this.filename = filename;
        }

        public ArrayList objectsAndCommands(){
                String objectCommand;
                ArrayList commands = new ArrayList();
                try{
                        readFile = new FileReader(filename);
                        readImportantPoints = new BufferedReader(readFile);
                        objectCommand = readImportantPoints.readLine();
                        while(objectCommand!=null){
                                if (objectCommand.length()>0){
                                        commands.add(setCommands(objectCommand));
                                }
                                objectCommand = readImportantPoints.readLine();
                        }
                }catch(Exception e){
                        e.printStackTrace();
                }
                return commands;
        }

        public LocationObjects setCommands(String fileInput){
                int startCommands = fileInput.indexOf(":");
                ArrayList commands = new ArrayList();
                String object = fileInput.substring(0,startCommands);
                boolean stringEnd = false;
                boolean substr = false;
                String out = "";
                int j = startCommands+1;
                while (!stringEnd){
                        char chr = fileInput.charAt(j);
                        if (chr=='.'){
                                stringEnd = true;
                                commands.add(out.substring(1,out.length()));

                        }else if (chr==','){
                                commands.add(out.substring(1,out.length()));
                                out = "";
                        }else{
                                out = out + chr;
                        }
                        j++;
                }
                LocationObjects loc = new LocationObjects(object, commands);
                return loc;
        }

        public void printlist(ArrayList list){
                int listSize = list.size();
                for (int i = 0; i < listSize; i++) {
                        String out = "";
                        LocationObjects obj = (LocationObjects)list.get(i);
                        out = "Object: "+ obj.object + " Commands: ";
                        ArrayList commands = obj.commands;
                        int commandSize = commands.size();
                        for (int j = 0; j < commandSize; j++) {
                                out = out + commands.get(j);
                        }
                        System.out.println(out);
                }
        }
        public void action(){
                objectCommands = objectsAndCommands();
        }
}

class LocationObjects{
        public String object;
        public ArrayList commands;

        public LocationObjects(String object, ArrayList commands){
                this.object = object;
                this.commands = commands;
        }
}


}
```

Listener: Listen for changes on the JavaSpace

```
/**Listener.java
 *
 */

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.space.JavaSpace;
import java.util.ArrayList;

public class Listener extends UnicastRemoteObject implements RemoteEventListener  {
    private JavaSpace space;
        private FusionAgent myAgent;

    public Listener(JavaSpace space, FusionAgent ag) throws RemoteException {
        this.space = space;
        myAgent = ag;
    }
    //notifies when new CommandDescription is placed on JavaSpace, collects
    //information from JavaSpace and sends to fusion
    public void notify(RemoteEvent ev) {
        CommandDescription command = new CommandDescription();
        LocationDescription location = new LocationDescription(Constants.LOCATION);
        LocationDescription maplocation = new LocationDescription(Constants.MAPLOCATION);

        try {
                CommandDescription commandResult =
                (CommandDescription)space.read(command, null, 10000);
                LocationDescription locationResult =
                (LocationDescription)space.readIfExists(location, null, 200);
                LocationDescription maplocationResult =
                (LocationDescription)space.readIfExists(maplocation, null, 200);

                long speechTime = 0;
                        long posTime = 0;
                long mapTime = 0;

                if (commandResult!=null){
                        System.out.println("COMMANDNOTIFY " + commandResult.getInputType());
                        speechTime = commandResult.getTime().longValue();
                }
                if (locationResult!=null){
                        System.out.println("LOCATIONNOTIFY " + locationResult.getInputType());
                        posTime = locationResult.getTime().longValue();
                }
                if (maplocationResult!=null){
                        System.out.println("MAPLOCATIONNOTIFY " + maplocationResult.getInputType());
                        mapTime = maplocationResult.getTime().longValue();
                }
                boolean map = false;
                if (speechTime!=0 &&mapTime != 0){
                        long difference = java.lang.Math.abs(speechTime - mapTime);
                        if (difference <=2000){
                                map = true;
                        }
                }
                ArrayList priorityList = new ArrayList();
                if (map){
                        System.out.println("MAPPRI");
                        priorityList.add(maplocationResult);
                        priorityList.add(locationResult);
                }else{
                        System.out.println("PosPRI");
                        priorityList.add(locationResult);
                        priorityList.add(maplocationResult);
                }
                int state = 0;
                myAgent.fusion(commandResult, priorityList, state);

        } catch (Exception e) {
            e.printStackTrace();
        }

        System.out.println("NOTIFY FINISHED");
    }
}
```

OutputGUI: Show output and action that should happen in a GUI

```
/*
 * Created on 30.jun.2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */

import javax.swing.*;

import java.awt.*;
import java.awt.event.*;

public class OutputGUI extends JFrame {

        JLabel solution = new JLabel("Solution found? : ");
        JLabel cmd = new JLabel("Command to perform:");
        JLabel obj = new JLabel("Chosen Object");
        JLabel addInfo = new JLabel("Additional Info: ");


        FusionAgent myAgent;

        public OutputGUI(FusionAgent ag) {
                myAgent = ag;
                jbInit();
        }

        private void jbInit(){
                setTitle("Output");
                setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                Container guiBeholder = getContentPane();
                LayoutManager layout = new GridLayout(4,2,5,5);
                guiBeholder.setLayout(layout);
                guiBeholder.add(solution);
                guiBeholder.add(cmd);
                guiBeholder.add(obj);
                guiBeholder.add(addInfo);

                pack();
        }

        public void setSolution(String s){
                solution.setText("Solution found? : "+ s);
        }

        public void setCMD(String s){
                cmd.setText("Command to perform: " + s);
        }

        public void setObject(String s){
                obj.setText("Chosen Object: "+s);
        }

        public void setAddInfo(String s){
                addInfo.setText("Additional info: "+ s);
        }

}
```

SpeechGrammar: the speech grammar shows which combination of words are allowed to speech

```
grammar Speech;

<polite> = (could you | could I | please) [please];


<commands> = ( information {information}
                                        | show {show}
                                        | coordinates {coordinates}
                                        | presentation {presentation}
                                        | normal {normal}
                                        | reserve {reserve}
                                        | availability {availability}
                                        | open  {open}
                                        | close {close}
                                        );

<objects> = ([meeting] room {room}
                                        | office {office}
                                        | corridor {corridor}
                                        | floor {floor}
                                        | door {door}
                                        | curtain {curtain}
                                        | window {window}
                                        | student workplaces {workplaces}
                                        );


<verb> = get | give | show | set | check;

<art> = the | this | some ;

<pr> = of | about;

<end> = in detail | <clock>;

<pron> = in;

<clock> = [at] ( one {1}
                                | two {2}
                                | three {3}
                                | four {4}
                                | five {5}
                                | six {6}
                                | seven {7}
                                | eight {8}
                                | nine {9}
                                | ten {10}
                                | eleven {11}
                                | twelve {12}
                                );

public <sentence> = [ <polite> ] [<verb>][me][<art>] <commands> [me][<pr>] [<art>] [<objects>][<end>];

public <sentence2> = [ <polite> ] [<verb>] [<art>] [<objects>] [<pron>] <commands> [mode];
```

ObjectCommand: Possible commands on the objects

```
door: open, close.
room: reserve, presentation, information, show, coordinates, normal, availability.
curtain: open, close.
window: open, close.
```

Points: Where the objects are located

```
office1 :room (0 125 , 0 125)
office2 :room (125 210 , 0 125)
office3 :room (210 300 , 0 125)
office4 :room (435 520 , 0 125)
office5 :room (520 610 , 0 125)
office6 :room (610 690 , 0 125)
office7 :room (250 345 , 170 270)
office8 :room (345 460 , 170 270)
office9 :room (610 690 , 170 270)
meeting_room :room (460 610 , 170 270)
student_lab1 :room (0 125 , 170 270)
student_lab2 :room (125 250 , 170 270)
corridor :room (0 610, 125 170)

door1 :door (25 75 , 80 125)
door2 :door (150 200 , 80 125)
door3 :door (225 275 , 80 125)
door4 :door (450 500 , 80 125)
door5 :door (530 580 , 80 125)
door6 :door (625 675 , 80 125)
door7 :door (35 85 , 145 185)
door8 :door (160 210 , 145 185)
door9 :door (270 320 , 145 185)
door10 :door (370 420 , 145 185)
door11 :door (510 560 , 145 185)
door12 :door (625 675 , 145 185)

window1 :window (40 80 , 10 20)
window2 :window (150 180 , 10 20)
window3 :window (230 255 , 10 20)
window4 :window (320 340 , 10 20)
window5 :window (380 405 , 10 20)
window6 :window (460 490 , 10 20)
window7 :window (550 570 , 10 20)
window8 :window (630 655 , 10 20)
window9 :window (50 80 , 260 270)
window10 :window (160 200 , 260 270)
window11 :window (270 295 , 260 270)
window12 :window (380 415 , 260 270)
window13 :window (475 495 , 260 270)
window14 :window (540 565 , 260 270)
window15 :window (635 660 , 260 270)

curtain1 :curtain (40 80 , 10 20)
curtain2 :curtain (150 180 , 10 20)
curtain3 :curtain (230 255 , 10 20)
curtain4 :curtain (320 340 , 10 20)
curtain5 :curtain (380 405 , 10 20)
curtain6 :curtain (460 490 , 10 20)
curtain7 :curtain (550 570 , 10 20)
curtain8 :curtain (630 655 , 10 20)
curtain9 :curtain (50 80 , 260 270)
curtain10 :curtain (160 200 , 260 270)
curtain11 :curtain (270 295 , 260 270)
curtain12 :curtain (380 415 , 260 270)
curtain13 :curtain (475 495 , 260 270)
curtain14 :curtain (540 565 , 260 270)
curtain15 :curtain (635 660 , 260 270)

canteen : canteen (300 435 , 0 110)
```

Bibliography

[1] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa. Jade, a white paper. *exp*, 3(3), 2003. http://exp.telecomitalialab.com/.

[2] J. Bers, S. Miller, and J. Makhoul. Designing conversational interfaces with multimodal interaction. *DARPA Workshop on Broadcast News Understanding Systems*, 1998.

[3] R. A. Bolt. Put-that-there: Voice and gesture at the graphics interface. *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, 1980.

[4] L. Boves and E. den Os. Multimodal multilingual information services for small mobile terminals (must), 2001.

[5] A. Cheyer and L. Julia. Multimodal maps: An agent-based approach. In *Multimodal Human-Computer Communication*. Springer-Verlag, London, UK, 1998.

[6] P. Cohen, M. Johnston, D. McGee, S. Oviatt, J. Pittman, I. Smith, Chen. L., and J. Clow. Quickset: Multimodal interaction for distributed application. *Proceedings of the fifth ACM international conference on Multimedia*, 1997.

[7] D. D. Corkill. Collaborating software: Blackboard and multi-agent systems & the future. In *Proceedings of the International Lisp Conference*, New York, October 2003.

[8] D. Gelernter. Generative communication in linda. *ACM Transactions on programming languages and systems*, 7(1), January 1985.

[9] M. A. Grasso, D. S Ebert, and T. W. Finin. The integrality of speech in multimodal interfaces. *ACM transactions on Computer-Human Interaction*, 5(4):303–325, 1998.

[10] D. Groome. *An introduction to cognitive psychology. Processes and disorders.* Psychology press, 1999.

[11] T. G. Holzman. Computer-human interface solutions for emergency medical care. *Interactions*, 1999.

[12] A. K. Jain and A. Ross. Multibiometric systems. *Communications of the ACM*, 47(1), 2004.

[13] M. Johnston, S. Bangalore, G. Vasireddy, A. Stent, P. Ehlen, M. Walker, S. Whittaker, and P. Maloor. Match: An architecture for multimodal dialogue systems. *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, 2002.

[14] R. Kurzweil. When will hal understand what we are saying? computer speech recognition and understanding. In D. G. Stork, editor, *Hal's Legacy: 2001's Computer as Dream and Reality.* MIT Press, 1998. http://mitpress.mit.edu/e-books/Hal/.

[15] H. Lieberman and T. Selker. Agents for the user interface. In Jeffrey Bradshaw, editor, *Handbook of Agent Technology.* MIT Press, 2002.

[16] 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, 1999. http://www.ai.sri.com/ cheyer/papers/oaa.pdf.

[17] M. T Maybury. Intelligent user interfaces: An introduction. In M. T. Maybury and W. Wahlster, editors, *Readings in Intelligent User Interfaces*, pages 1–13. Morgan Kaufmann Publisher, 1998.

[18] J. G. Neal, Z. Thielman, Z. Dobes, S. M. Haller, and S. C Shapiro. Natural language with integrated deictic and graphic gestures. In *Proceedings of the 1989 DARPA Workshop on Speech and Natural Language*, 1989.

[19] H. S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(3), October/November 1996.

[20] H. S. Nwana, L. Lee, and N. R. Jennings. Co-ordination in software agent systems. *BT Technology Journal*, 14(4), October 1996.

[21] S. Oviatt. Multimodal interfaces. In *Handbook of Human-Computer Interfaces.* Lawrence Erlbaum, New Jersey, 2002.

[22] S. Oviatt. Advances in robust multimodal interface design. *Computer Graphics and Applications, IEEE*, 23(5), 2003.

[23] S. Oviatt. User-centered modeling and evalutation of multimodal interfaces. *Proceedings of the IEEE*, 91(9), September 2003.

[24] L. Rabiner and B. Juang. An introduction to hidden markov models. *ASSP Magazine, IEEE*, 3(1), 1986.

[25] S. J. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

[26] W. Wahlster. User and discourse models for multimodal communication. In J. Sullivan and S. Tyler, editors, *Intelligent User Interfaces*, pages 45–67. ACM Press, New York, 1991.

[27] The nuts and bolts of compiling and running javaspaces programs. Web Page. http://java.sun.com/developer/technicalArticles/jini/javaspaces/.

[28] Adaptive agent architecture. Web Page. http://chef.cse.ogi.edu/AAA/.

[29] Fipa. Web Page. http://www.fipa.org.

[30] Galaxy communicator. Web Page. http://communicator.sourceforge.net/.

[31] Jade. Web Page. http://jade.tilab.com.

[32] Javaspaces service specification. Web Page. http://www.jini.org/nonav/standards/davis/doc/specs/html/jsTOC.html.

[33] Java speech api programmer's guide. Web Page, 1998. http://java.sun.com/products/java-media/speech/forDevelopers/jsapi-guide/index.html.

[34] Grammar format specification. Web Page, 1998. http://java.sun.com/products/java-media/speech/forDevelopers/JSGF/index.html.

[35] Ibm viavoice. Web Page, 1998. http://www-306.ibm.com/software/voice/viavoice/.

[36] M. Wooldridge. *An Introduction to MultiAgent Systems.* John Wiley & Sons, Inc., 2002.