EB 564

Master's Thesis

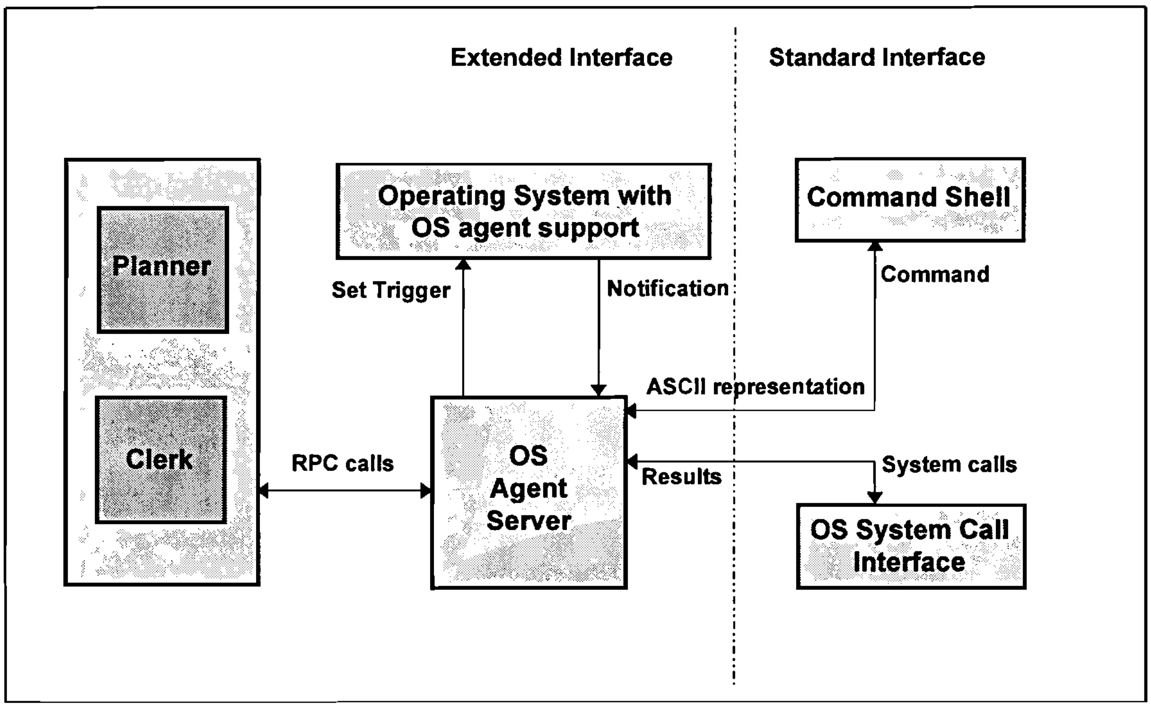
Agents in Operating Systems

Bob van den Heuvel (IT, 303852)
Eindhoven University of Technology

Coach: Ir. A.G.M. Geurts
Supervisor: Prof. Ir. M.P.J. Stevens

Faculty of Electrical Engineering,
Department of Digital Information Systems

November 1994 - June 1995



Agents in Operating Systems

'Since the beginning of this century, people have dreamed about the new companions that they might create with new technology. Some of those dreams are nightmares about malevolent computers enslaving mankind as techno-evolution catapults them far beyond our puny carbon-based brains. Most are wistful longings for new helpers, advisors, teachers, playmates, pets or friend. But all of the computer-based personae that weave through popular culture have one thing in common: they mediate a relationship between the labyrinthine precision of computers and the fuzzy complexity of man.'

Brenda Laurel,
excerpt from '*Metaphors with Character*'

(P) December 2, 1994 - June 6, 1995

Summary

At the University of Technology in Eindhoven, The Netherlands, the last six months in the curriculum of the Electrical Engineering study are reserved for a project supervised by members of the scientific staff. Upon completion of the project, a thesis has to be written reflecting the work the student has done during this time interval. Upon approval of the curriculum, the project and the thesis, the author will obtain the Dutch title Ir. (Ingenieur) (comparable to a Master Degree).

This is the thesis of such a project. The project bears the title "Agents in Operating Systems", the main goal of the project being the determination of what agents exactly are, how they are to be used (particularly in Operating Systems) and designed. The thesis is presented as a tutorial. The reader who is not completely familiar with terms such as artificial intelligence, learning processes and operating systems beforehand, will be supported with short overviews of these subjects. Because the subject of agents is so multi-faceted, that is, the subject is related to not only computer science, but also sociology, psychology and other disciplines, the author has chosen to include these overviews in the thesis.

Agents are smart pieces of software who can reflect and execute (successfully or unsuccessfully) the wishes of its creator, the creator being a robot, an operating system or even a computer user. The agent may be given a task, such as the automatic retrieval of documents, which, for example, are relevant to a project, that a computer user is currently working on, from any FTP site in the world. The agent tries to accomplish its given task completely independent of anything but its own internals.

The thesis starts off with a short introduction to artificial intelligence, explaining the position of agents in this field of science. Some examples are given in which agents may take a key role. Some properties of agents, such as their pro-activeness and their successfulness are discussed. Also, a definition of an agent is presented, hopefully satisfying all disciplines involved in the study of agents.

The actual construction of agents has also been studied: the specification and the implementation of agents and also the languages with which the agents can communicate with its creator or even other peers (a multi-agent system). We shall also take a look at the environment in which the agents may reside, in particular the operating system environment. Subjects such as how the requirements and the architecture should be like for

an environment in which agents are supported, are also discussed. The design of an agent operating system is clarified by introducing agents into an existing conventional operating system (UNIX, in this case).

So far, it is has been assumed that the agents possess knowledge, that was introduced upon their creation, with which to complete their (initial) task. The most important issue of an agent is however the ability to learn. An agent may even gain so much knowledge that it outgrows the knowledge beyond that possessed by its creator. We shall be examining some possibilities on how an agent can gain its knowledge. The main subject of this study will be reinforcement learning, in which the environment gives feedback to the agent by sending rewards or punishments according to the task the agent is currently executing. With an experiment of robots, which possess the ability to learn, the reinforcement learning process is clarified.

Lastly, some conclusions are drawn. By looking at the scarce material that is currently available on agents, the conclusion can be drawn that the main software houses keep their knowledge on agents to themselves. However, if we look at the agents that are supposedly implemented in programs like Microsoft's Excel 5.0 or Hewlett Packard's NewWave, it may be possible that they do not even possess such knowledge, because the 'agents' are mostly nothing more than hidden macro's and do not possess intelligence as such. One thing that is for sure is that there is a lot of profit involved with agents, so it is not that surprising that the software houses keep their knowledge secret, if they do possess it at all, that is.

From the material that was available, mainly other papers of colleagues throughout the world, it was clear that, although difficult to design, especially the learning processes, agents may revolutionize the way in which we use the computer. By introducing agents into an operating system the computer user does not need to have the knowledge of that operating system to operate his/her computer as is the case with conventional operating systems. If a user wants to free additional harddisk space, he simple creates an agent with this task, and the *agent* may then move files, the user probably does not want anymore, to another destination, or delete those files, after making a backup of them; the agent is supposedly reflecting the user's brain after all, so it will know whether the files are of interest or not. Also the management of a network system can benefit greatly from agents, such as agents who can automatically configure a network system or reduce overhead traffic between a source and destination node by handling this kind of information in the destination node itself.

Eindhoven,
June 6, 1995

Contents

List of Figures

List of Definitions

1. Introduction

1.1 Artificial Intelligence in short

The brain. Humankind has been intrigued by its most complex organ, which distinguishes humans from other beings, for centuries. The brain has been and still is the most researched part of the human body, yet the least understood. Now in this day and age we are trying to clone ourselves by electronic means via computers, also known as Artificial Intelligence.

Artificial Intelligence (AI) is currently almost three and a half decades in research. It's only now that we are beginning to grasp the full potential of AI and its effect on society as a whole. AI can be seen as part of the Information Revolution, which is currently well underway in its first stage. In retrospect, the Industrial Revolution would be considered a petty event indeed, when compared to the Information Revolution.

The object of AI, as it was introduced in 1961, was to enable a computer to perform the remarkable functions that are carried out by human intelligence. It was thought feasible that at the end of the 1960's computers would be available with the intelligence of the human species by the development of systems capable of playing games such as chess, proving of mathematical theorems, solving problems written in plain English, etc.

However, it was eventually realized that such objectives were not at all to be attained that soon, because of the rapid increase in data and procedures needed for such a level of intelligence. The computers were at that time simply not able to cope with such massive quantities of information, let alone the speed that was necessary to handle the information.

The introduction of the micro-processor computer resulted in an enormous boost of the possibilities in creating AI. However, computers nowadays still do not posses enough processor-speed and memory capacity to be able to emulate the human brain. Unfortunately this is not the only problem to tackle: we still do not understand how the human brain really works; how the different parts of the brain interact with each other. It was only recently discovered where in the brain the different processes, such as imagination and movement, take place and information, such as a person's character, is stored. But how these functions are established is still not understood. We do know however that neurons, certain chemicals and electricity are involved. Yet another problem arises to simulate the human brain, being that the brain usurps such a massive amount of energy for transformation into electricity (20% of the food we eat) that a laptop-computer would in comparison need half the output of a standard power-plant, if the computer were able to emulate the human brain with the current computer technology **[BBC]**.

Artificial Intelligence itself is literally still in its infancy, but with the tools available and the many researches that are currently being executed, it's probably not that far away from its adolescence stage. We must however first have to understand the human brain, how the different regions interact with each other, before we can develop even a remote resemblance of our brain with electronic means.

1.2 Distributed Artificial Intelligence

Most of the computer systems we use now do not operate as a stand-alone system. Most of them are connected to some kind of computer network such as a LAN or a WAN. We call these computer systems distributed systems; the resources of the network are *distributed* among its connected computers. Those systems require an operating system that can support the distribution of said resources.

Popular operating systems, which support this kind of sharing of resources, are UNIX, OS/2, Windows NT and Windows (for Workgroups). For any environment to be future-proof, one thing it must definitely have is support for multiple users distributed over multiple machines.

The arrival of distributed systems also meant a new discipline in the field of Artificial Intelligence: Distributed Artificial Intelligence (DAI). This kind of artificial intelligence is especially designed for cooperation with distribution supported environments. The world of DAI can be divided into three primary arenas:

1. ***Distributed Problem Solving (DPS)***: considers how the work of solving a particular problem can be divided among a number of modules, or 'nodes', that cooperate at the level of dividing and sharing knowledge about the problem and about the developing solution.
2. ***Parallel AI (PAI)***: is concerned with developing parallel computer architectures, languages and algorithms for AI.
3. ***Multi-agent systems (MAS)***: research is concerned with coordinating intelligent behaviour among a collection of (possibly pre-existing) autonomous intelligent

> 'agents'[1] how they can coordinate knowledge, goals, skills and plans jointly to take action or to solve problems. The type of the agents' environments DAI deals with are relatively simple and have low complexity in that they feature no 'noise' (i.e. distorted information received by the agent from the environment or corrupted information sent by the agent to the environment due to external influences) or uncertainty and can be accurately characterized. As we shall see in section 3.2, this is not at all true. The more agents we create in a certain environment, the noisier and more uncertain the environment will become.

This thesis is clearly not a review of DAI, although the material discussed in herein unarguably falls under the banner of DAI. The domain of classical DAI (mostly DPS and PAI) is avoided. For reviews of these areas, see **[Bond]**. In this report we shall study a part of the latter arena: agents.

The idea of agents is simple: create an electronic resemblance of the user, and let it make decisions (for the user) just as the user would do. An agent is in fact an electronic clone of the user's brain. It's thus an *agent* for the user, the reason why this term was chosen. Another often-used term, and probably the best, is Personal Digital Assistant (PDA). We can carry the train of thought even further by replacing the user in the above-mentioned sentence by an application, an operating system or even a robot, thus creating operating system agents.

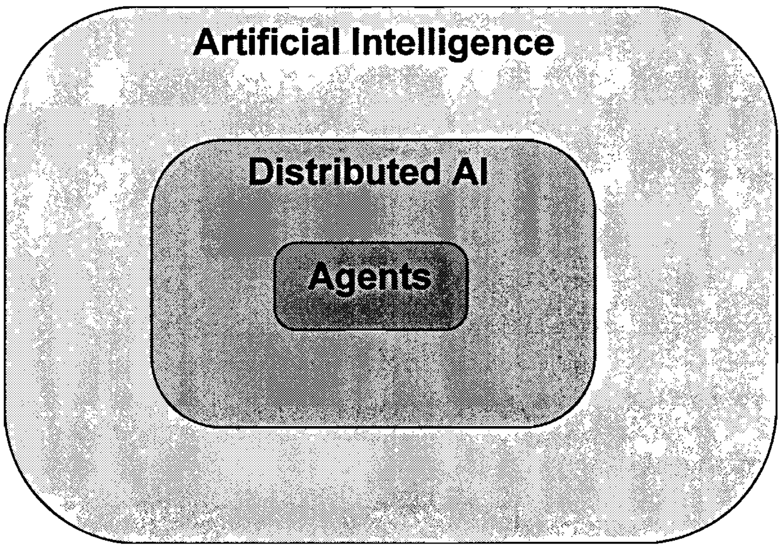


Figure 1-1: Placement of agents in the field of Artificial Intelligence

[1] For now consider an agent to be a computational process with a single locus of control and/or 'intention'.

In Figure 1-1 the placement of agents in the field of AI is shown. The presence of distributed intelligence does NOT imply that there must be a distributed system present. There may be multiple agents in some local, i.e., not distributed, system. Those agents however, can be considered distributed in that system, and have distributed intelligence, as such. This is the reason why agents are placed within the field of distributed intelligence in this figure.

Because of the restrictions mentioned earlier it should be obvious that agents as they are described above - clones of the human brain - will not be realized in the near future. However, if we let agents be 'clones' of some application or operating system instead of a computer user, it suddenly becomes feasible. Indeed, there are already applications and operating systems that do support some kind of agent-like software, for example Microsoft Excel 5.0's Wizard function and Hewlett Packard's NewWave **[Vizard]**. They are not intelligent as such - they often consist of nothing more than a couple of macro descriptions - but they are certainly the first step towards the ultimate goal of agents: to clone one's self (i.e. the computer user) or even go beyond the intelligence of the user by introducing learning processes to the agents.

1.3 Structure of this report

In chapter 2 agents will be discussed in general. Because agents have only recently emerged, several different definitions and interpretations can be found in the few articles that are now available. I have tried to combine, expand and polish several definitions in what I believe to be the best approach in defining agents and how they are to be interpreted. In the first section several implementation examples of agents are given followed by a discussion of the notion, different types and definition of agents. This seems a rather odd sequence, but it will become more obvious soon enough why this strategy was chosen.

We shall see that there are essentially two types: agents used in operating systems, i.e., agents working on a local environment (including robotics operation systems) and the agents used in a network, i.e., agents working in a distributed environment.

Agents that are essentially used in distributed environments will not be covered in this thesis, because our primary concern is the study of agents in a local environment.

In chapter 3 we embark on a more detailed look at the properties of agents, and how we might go about constructing them. We shall see that the construction of agents can be divided into three key issues: agent theories, agent architectures and agent languages. These issues will be explained separately in the first three sections of this chapter. This chapter is by no means a full account of a construction scheme for agents. Especially the first stage (agent theories) contains an enormous amount of behavioural science (psychology and sociology). As this is a technical thesis, the issue of the behavioural properties has been reduced to the bare minimum, but just enough to further embark on the subject of agents without much difficulty.

In chapter 4 the main subject will be discussed: Agents in Operating Systems. In the first section we shall take a short look at operating systems in general. Some terms, which are used throughout this chapter, are also presented. The introduction is followed by a description of an OS agent, including the advantages and drawbacks of script files versus OS agents and communication with OS agents. Also, an example of a framework for OS agents will be discussed. The framework described in this section is an already existing conventional operating system (UNIX), to which agent support has been added. The purpose as to why an existing operating system is chosen and not a newly designed agent system is also discussed.

In chapter 5 we shall be venturing on the path of learning capabilities of operating system agents. In the introduction we shall ask (and answer) ourselves the question whether it is possible to create a fully operable operating system (and agents) from a 'dumb' operating system (i.e., without any 'implemented' knowledge at the time of the operating system agent's creation). Some examples will be given as to why learning is so important to agents. An in-depth review of the most commonly used learning process, reinforcement learning, shall also be presented here. The learning process will be clarified by an experiment with robots, which possess some learning capabilities. Furthermore we shall look at the modeling of multi-agent domains, the Markovian Decision Process, and also its shortcomings in such environments.

Finally some conclusions will be presented in chapter 6.

2. Agents in general

2.1 What is an agent anyway?

Before actually describing what an agent exactly is and does, we shall first look at some examples of what agents might do in the environment they are residing in. So, consider these four events occurring in some near future:

1. In the country of Mithgar, the key air-traffic control systems fail to operate, due to some bad weather conditions. Luckily, the neighbouring countries Xian and Kelewan have computerized air-traffic control systems between themselves to track and deal with all affected flights, and the potentially disastrous situation passes without major incidents.
2. Upon logging in to your computer, you are presented with a list of e-mail messages, sorted in order of importance by your personal digital assistant (PDA). You are then presented with a similar list of news articles; the assistant draws your attention to one particular article, which describes a new viewpoint on work which is very close to your own. After an electronic discussion with a number of other PDA's, your PDA will have obtained that relevant technical report for you from an FTP-site, an HTTP-site etc., in the anticipation that it will be of your interest.
3. You are editing a file. Suddenly your PDA requests your attention: an e-mail message has arrived. It is from an important conference you recently sent a paper to. It contains a notification of that particular paper. Your PDA *predicts* that it will be of your interest, and you should be notified immediately. The message states that the paper has been accepted. Without any prompting, your PDA will start to arrange travel arrangements for that conference by consulting a number of databases and other network information resources. It will look for a flight from the nearest airport to the airport nearest the conference, taking a flight approximately twelve hours before the start of the conference, checking the availability of a room in a hotel near the World Trade Center, where the conference will be held. Upon completion of its quest the PDA will present you with a list with the available travel options. When one

arrangement is selected by the user, the PDA will arrange all the necessary bookings and give you an acknowledgement when everything is booked according to plan.

4. You are trying to format a group of spreadsheet cells or locations in a particular manner. If you are not adept at using the spreadsheet package, you might try to format each cell individually. The spreadsheet package detects your unproductive manner of formatting and notifies you that there is a faster way of formatting a group or it might even take the next steps automatically.

Although the present computer systems are not able to execute the processes (yet) as the given events, research is underway to let computer systems execute exactly as described above. The keyword here is *agents* (also called PDA's, Wizards, Coaches, automated assistants or Cyberbutler). For those anticipating a definition of an agent right away will have to exert some patience. It is very difficult to give a general definition of an agent because there are so many fields of science involved in the research of agents such as computer science, sociology and biology. Therefore only a *description* will be given now and towards the end of this section an attempt will be made to give a generally satisfying *definition* of an agent, when the reader will be more familiar with the subject matter.

An agent acts on behalf of his 'Master', the 'Master' being a human being, a computer process, an operating system or even other agents. The systems in which the latter agents reside are known as multi-agent systems. An agent is a piece of software which runs autonomous. It takes action which appropriately represents the interest of others; it must be robust and capable of securely handling private information (a heavy point of discussion, which we shall look at more closely in section 3.3.1); it tends to be highly active (most of the time it communicates with its environment, other agents or even human beings) and is an active part in the computational space, i.e. it reacts to and changes the overall system state.

The reader may wonder why a definition is so badly needed: after all, if many people are developing interesting and useful applications, it hardly matters that they do not agree on potentially trivial terminological details. This practice is however dangerous because unless this issue is not addressed, the term agent might become a so-called 'noise' term, subject to both abuse and misuse. This will in turn produce much confusion in the research community. I feel therefore the need to discuss this issue.

2.2 Notions of agents

There are globally two different approaches to be distinguished if we consider the term agent: one used by the non-AI scientists, such as computer scientists, which shall be called the weak notion of an agent (section 2.2.1), and the stronger, more contentious viewpoint of the AI scientists, which shall be called the strong notion of an agent (section 2.2.2). If we look at the subject matter studied in this thesis, we can conclude that the weak notion of an agent on itself is sufficient, but to give a complete picture of the notion of an agent, the strong notion will also be discussed.

2.2.1 Weak notion of Agents

The weak notion of an agent is perhaps the most general way in which the term agent is used to denote a hardware or, more usually, a software-based computer system enjoying some of the properties as described in section 2.1. The properties are:

- ***autonomy:*** agents can operate without direct intervention of humans, applications, operating systems and other such 'masters', and have some kind of persistent control over their actions and internal state.

- ***social ability:*** agents are most of the time communicating with its environment, human beings, other agents etc. The interaction between these entities is carried out via a communication language, called an agent communication language **[Genesereth]**.

- ***reactivity:*** agents can perceive their environment, such as the physical world, a user via a graphical user interface, a collection of other agents, INTERNET or possibly a combination of these. They respond timely to changes that occur in these environments.

- ***pro-activeness:*** agents do not simply act on changes in said environments, they exhibit goal-directed behaviour by taking the initiative.

The reader who is familiar with the UNIX environment may conceptualize the properties listed above as a UNIX-like software process. It is therefore not surprising that the weak notion of an agent is held high by the mainstream computer scientists. The agent is viewed as a self-contained, concurrently executing software process, that encapsulates some state and is able to communicate with other agents via message passing. People familiar with object-oriented programming (OOP) will immediately recognize this agent software process to be an object in an event-driven operating system. Indeed object-oriented programming seems to be an almost inevitable subject in the agent matter, as we shall see later on.

In the emerging discipline of *agent-based software* an agent is generally described as:

> 'Agents communicate with their peers by exchanging messages in an expressive agent communicating language. While agents can be as simple as subroutines, typically they are larger entities with some sort of persistent control.' **[Genesereth]**

Although this quotation states that agents do not have to be objects, but can also be implemented by conventional subroutines (procedures and functions), it is considered bad practice. Moreover, an object-oriented environment already has the necessary event-message control implemented, which is not the case for the conventional approach.

For softbots[2] a similar description is given:

> 'A softbot is an agent that interacts with a software environment by issuing commands and interpreting the environment's feedback. A softbot's effectors are commands (e.g. UNIX shell commands such as `mv` or `compress`) meant to change the external environment's state. A softbot's sensors are commands (e.g. `pwd` or `ls` in UNIX) meant to provide ... information' **[Etzioni]**

OOP seems to play a very important role in agent-programming, indeed. Agents seen in this way are very much like OOP-objects which can communicate via so-called event-messages with their peers. The only difference is that agents are pro-active, and OOP-objects are not: an OOP-object cannot make decisions for itself. It then seems almost unavoidable that agents have to be run in event-driven object-oriented operating systems (such as Windows). As we shall see later in section 3.3, there do exist operating systems with agents that do not support object-oriented programming but work with the conventional procedural structures, but the implementation of such a system will be much more costly than the object-oriented one, because an event-message system has to be implemented whereas in OOP environments, this message system already exists.

2.2.2 Strong notion of Agents

For scientists working with AI, the term 'agent' has a stronger and more specific notion than the one discussed in the previous section. This group of researchers believe an agent to be a computer system that, in addition to having the properties identified above, is either conceptualized or implemented using concepts that are more usually applied to human beings, such as mentalistic and emotional notions. The general properties in the strong notion of agents are:

- ***Successful***: An agent is successful if it accomplishes the specified task in the given environment.
- ***Capable***: An agent is capable if it possesses the effectors needed to accomplish the task.

[2] A softbot is a software agent used in a robot operating system.

- ***Perceptive***: An agent is perceptive if it can distinguish salient characteristics of the world that would allow it to use its effectors to accomplish the task.

- ***Reactive***: An agent is reactive if it is able to respond sufficiently quickly to events in the world to allow it to be successful.

- ***Reflexive***: An agent is reflexive if it behaves in a stimulus-response fashion.

Other possible properties might be ***mobility*** (the ability of an agent to move around an electronic network), ***veracity*** (the assumption than an agent will not knowingly communicate false information) and ***rationality*** (the assumption that an agent will act in order to achieve goals, and will not act in such a way as to prevent its goal being achieved) and many others, depending on how the framework will be defined. As we shall see in chapter 3, any formal definition of agent properties must include a framework for describing an agent, a task and an environment.

2.3 Different types of Agents

The class of agents can be divided into three different kinds, corresponding to their area of operation. These are:

1. Agents used on networks
2a. Agents used in Operating Systems
2b. Agents used in Robotics.

However, if the distinction is made on the operation area of the agent's master, it becomes very confusing in which group an agent belongs. Take for example an agent that filters e-mail: it immediately discards messages which (supposedly) aren't of any interest to the computer-user, and passes important messages through to the user. Is this a network agent (it is a recipient of network information) or is it an operating system agent (it helps preserve resources, in this case memory for buffering the e-mail information)? It is therefore more obvious to distinguish agents by the area in which they operate. Although this distinction is more obvious, oftentimes the agents are still separated by their master's operation area in the currently available literature. In this report we choose for the first approach. Note that the groups still remain the same in this latter case. Instead agents are now distinguished by the operation area of the agents themselves, not by the operation area of their master.

Also note that the agents do not differ in any way concerning the properties as mentioned in section 2.2. The only difference is that the agents in the first group act on a distributed system and that the agents in the second group act on a local system. We can therefore divide agents in another way: agents used in distributed systems and agents used in local systems. The (already vague) boundary between operating system agents and robot agents has then disappeared completely.

Other divisions, such as a division on properties of an agent, or a division on tasks of an agent, are also possible. These divisions are actually rarely used, so we stick to the division of agents into network agents and operating system/robot agents.

The agents in the first group are referred to in the literature as intelligent (network) agents. These kind of agents will not be discussed in this thesis. Because of time constraints the choice has been made to study agents in a local environment only. The reason the latter two are combined into one group is that the agents used in Robotics are essentially the same as the agents used in Operating Systems because the software on which the 'robots' operate *are* basically Operating Systems. The agents in the second group are often referred to as software assistants or *software agents*. In case of a robot existing as a piece of software only, the agents are referred to as *softbots*. These kind of agents, being the main subject of this thesis, will be discussed more elaborately in chapter 4.

2.4 Definition of an Agent

We are now finally in a position to give a generally satisfying definition of an agent. Neither is nor should this definition be the formal definition for the simple reason that there isn't one yet and there probably will never be one, because there are too many different sciences involved, each having their own interpretation of an agent.

Definition 2-1: An agent

An agent is an autonomous software component with some sort of persistent control, and behavioural properties, enabling it to accomplish a certain specified task (or set of tasks) on behalf of a master - being a computer-user, an operating system, a robot or an application - usually done by communicating with its environment, Master or other agent peers via the exchange of messages in an expressive *agent communication language* .

3. Constructing agents

Now that we have at least a preliminary understanding of what an agent is, we can embark on a more detailed look at their properties, and how we might go about constructing them. We can identify three key issues in the construction of agents:

1. ***Agent theories*** are essentially *specifications*: It addresses questions as: How are we to conceptualize agents? What properties should agents have? How are we going to formally represent and reason about these properties?
2. ***Agent architectures*** represent the move from specification to implementation. It addresses questions as: How are we to construct computer systems that satisfy the specified properties for agents? What software and/or hardware structures need to be used? What is an appropriate separation of concerns?
3. ***Agent languages*** are programming languages that may embody the various principles of agents. It addresses questions as: How are we to program agents? What are the right primitives for this task? How are we to effectively compile or execute agent programs?

These three issues will be looked at more closely in the next three sections. This chapter does not have the intention that the reader can create his/her own agents after reading these sections; it is intended as a global overview of which steps have to be taken in the construction of agents. We will also look at some currently available agent systems on the market examine them with the three stages of agent construction.

3.1 Agent Theories

In the *agent theories* stage, the agents are specified. When explaining human activity, we often make use of the so-called *folk psychology*. We often make statements like:

- Bob worked very hard because he *wanted* to possess a Master Degree.
- Mr. Smith recalculated his taxes, because he ***believed*** a miscalculation had been made.

By folk psychology human behaviour is predicted and explained through the attribution of *attitudes*. The most common attitudes are believing, wanting, hoping and fearing. These attitudes are often called *intentional* notions. If we were to attach these attitudes to software agents, we call the resulting agent system an *intentional system.*

3.1.1 Intentional systems

Intentional systems may have different 'grades'. A first-order intentional systems has beliefs and desires, but has no beliefs or desires about those beliefs and desires. A second-order intentional system does have beliefs and desires about beliefs and desires, both those of others and of its own. First-order intentional systems do not have this ability.

The question arises whether an agent does require the attribution of intentional notions. When we look at the ultimate role agents should fulfill in the near future, being an exact replica of the master, it must be obvious that this attribution is necessary and unavoidable. So, being an intentional system is a necessary condition for agenthood.

The intentional notions may be separated into two main classes of attitudes: *information attitudes* and *pro-attitudes:*

$$\text{information attitudes}\begin{cases}\text{belief}\\ \text{knowledge}\end{cases}\qquad \text{pro - attitudes}\begin{cases}\text{desire}\\ \text{intention}\\ \text{obligation}\\ \text{commitment}\\ \text{choice}\\ \text{.........}\end{cases}$$

In this light, information attitudes are related to the information that an agent has about the surroundings which it occupies (usually simply referred to as 'the world'). Pro-attitudes on the other hand guide the agent's action in its world. It seems reasonable that an agent must be represented by at least one information attitude and one pro-attitude. A possible combination of attitudes in these two classes is solely defined by the task(s) the agent must fulfill. Note that the two different classes are closely linked as a rational agent makes choices and forms intentions (pro-attitudes) based on the perception of the information of its world (information attitude).

3.1.2 Representing intentional notions

Intentional notions need some kind of representation in order to be understandable for both humans and machines. The best way to accomplish this is the use of first order logic. However, we need to make some small additional restrictions, because intention cannot be completely substituted by first order logic.

In the following classical example, we shall see what those restrictions imply. In ancient Greek and Roman pagan religion there was an almighty deity, Father of Time, Chronos. Chronos had one son named Zeus by the Greeks and Jupiter by the Romans. If we want to express that some ancient Greek named Plato believes that Zeus is the son of Chronos, we may represent this notion in first order logic as:

- *believe(Plato, Father(Zeus,Chronos))*

We also know that the 'constants' Zeus and Jupiter represent the same deity, only with different names, so we can state that:

- *(Zeus = Jupiter)*

If we substitute this statement in the believe-statement, which is allowed in first order logic, we get:

- *believe(Plato,Father(Jupiter,Chronos))*

Intuition however rejects this derivation as invalid, because believing that the father of Zeus is Chronos is *not* the same as believing that the father of Jupiter is Chronos. The problem that occurs here is known by the term *referentially opaqueness.* Intentions such as beliefs and desires are referentially opaque, i.e. they set up a context in which the standard substitution rules of first order logic do not apply anymore.

If we take the opaqueness of intentions into consideration, we can represent intentions completely with first order logic. We can extend the first order logic with certain operators to overcome the problem of opaqueness. How these operators are to be used is beyond the scope of this thesis. See **[Wooldridge, Chapter 2]** for more information.

3.2 Agent Architecture

We have already seen that an agent is an entity created to perform some task or set of tasks. Any property of an agent must therefore be defined in terms of the task and the environment in which the task is to be performed. The most basic question that can be asked is whether an agent achieves the task or not. Since success is the fundamental property for characterizing agents, some appropriate scale is needed for determining success.

The most simple implementation of that scale would be the binary scale: if the agent remains unsuccessful, this will be represented by the digit zero (0). Otherwise, in case of success, the digit one (1) will be used. The agent is now successful or not. A binary scale for success also leads to binary scales for the other properties such as capability. However, this will rule out all relative comparisons of the property capability among other agents because there are only two states defined: capable or not.

The scale is more useful when it is refined by choosing a numeric value to the quality of task achievement, a method common to artificial intelligence. Consequently, possible initial conditions and external influences can be taken into account to give a *measure* of the expected success of the agent.

To say that an agent is successful at a particular task in a particular environment, a framework is needed for specifying the agent, the task and the environment. The framework must contain sufficiently detailed descriptions to allow the distinction between successful and unsuccessful agents to be made. On the other hand, a framework must be general enough to allow a wide range of agents, tasks and environments to be specified. Therefore, in designing the framework, the range of tasks and environments needs to be considered. We can for example include tasks of achievement, maintenance and deadline, and no other tasks. We can choose between a static or a dynamic environment, or the allowance of other agents in the environment. The environment must certainly be agent-friendly, meaning that agents must be able to *react* and also *proact* to events. That is to say, they should be able to intervene *before* an action takes place, to help the user. Furthermore, how the interaction between the agents and the environment is to be represented has to be considered.

One goal of this specification is the ability to compare different agents performing the same task in the same environment. Clearly a distinction between the agent and the environment has to be made, if we are to compare agents. In general the boundary of those two are very clear: The environment is everything present before the agent was introduced, the agent is everything that was added, without changing the environment.

The most general representation of an agent framework is shown in Figure 3-1. The agent and environment in which it is situated, is clearly distinguished. It is already shown in the definition of an agent (see 2.4) that an agent must have some kind of persistent control. This part of persistent control of the agent is included in Figure 3-1.

The controller acts as the actual 'brain' of the agent. It perceives information from the environment and acts on this information by sending information back to the environment. The actual behaviour of the agent is therefore controlled by this part. The state of the agent is also maintained by the controller. Usually this state is some kind of memory.

The mechanism with which the agent communicates between the environment and the control part of itself has been separated here, but it remains part of the controller. Through this mechanism agents receive information from the environment via sensors, and reflect their wishes to the environment via effectors. Any details of the nature of the environment, structure and implementation of the agent and the agent's control have been abstracted away. The agent, for example, also contains some sort of storage of its basic behaviour set, in which the initial behaviour of the agent is defined.

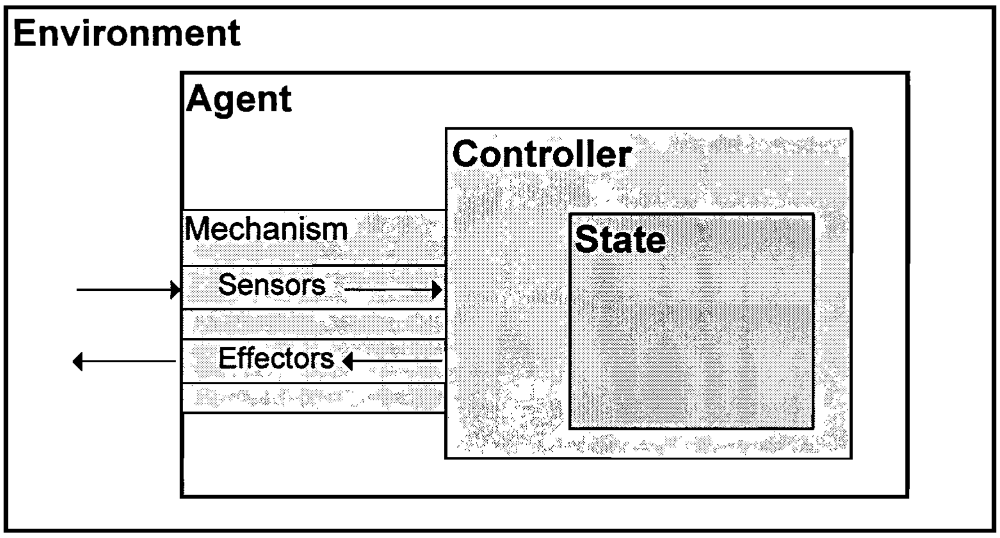


Figure 3-1: An agent system framework

The model as given in Figure 3-1 can be used for almost every agent system in existence, such as autonomous robots, knowbots[3]and operating system agents. The agent may have a physical presence (a robot) or just exist within the memory of a computer (a software agent). Oftentimes this boundary of environment and agent is very vague. Suppose an arm would fall from a robot, is this arm still part of an agent or is it now a part of the

[3]A knowbot is an agent that exists as a program running on a computer or computer network. It can perform tasks such as the retrieval of information from databases. An example is the World Wide Web worm, which uses this kind of bots.

environment? The control component could be implemented in software or hardware. By refining this abstract model, particular classes of agents can be specified.

Some examples of the classes that can be differentiated are:

- **Advisors** - agents who offer help and training
- **Guides** - agents who help in navigating databases
- **Servants** - agents who carry out tasks immediately
- **Representatives** - agents who work in a user's absence
- **Communicators** - agents who work with other users and agents

Traditionally, AI has concerned itself with complex agents in relatively simple environments, simple in the sense that they could be simply modeled and involved little or no 'noise' and uncertainty from the environment. However, the reactive and behaviour-based agent systems have become increasingly popular, placing the agents in a more complex, noisy and uncertain environment. Especially the multi-agent systems are to blame for introducing these environmental changes.

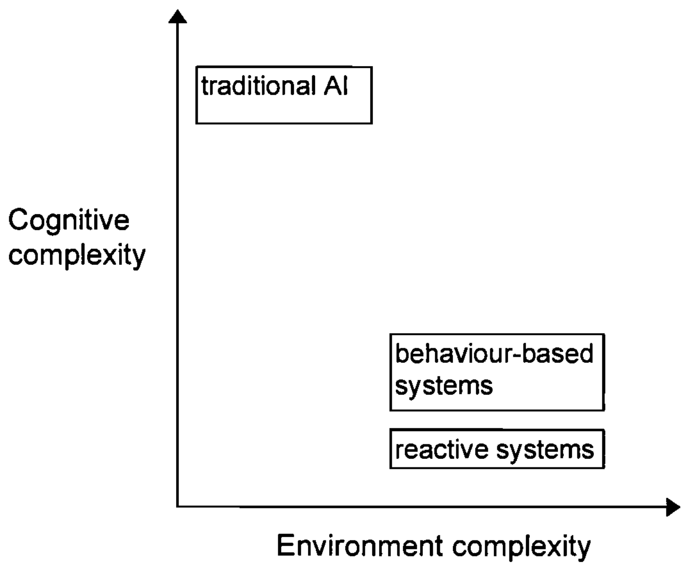


Figure 3-2: Relationship between cognitive and environmental complexity

The relationship between cognitive and environmental complexity is depicted in Figure 3-2. The position an agent system occupies is largely dictated by two factors:

1. The number of agents present in the system, and
2. the learning capabilities of an agent.

If we introduce more agents into a system, the environmental complexity increases. If we on the other hand introduce learning capabilities to the agents, which will be studied more closely in chapter 5, the cognitive complexity increases. By adjusting these two parameters we can create precisely the complexity of the agent system we want it to have. Under normal circumstances one couldn't care less whether the environment should be complex, or that the cognitive complexity should be as simple as possible. For research purposes however, the relationship between the number of agents and the environmental complexity, and the relationship between the learning capabilities and the cognitive complexity is very useful when experimenting with the behavioural characteristics of agents.

3.3 Agent communication languages

Any agent requires a way to interact with its environment by both observing (the sensors) and manipulating (the effectors) it. It requires some method of communication with the user or other agents. Since communication is the only means for an agent to complete its task, it is necessary that some kind of standardization is maintained when developing a language an agent can use for its communication purposes.

Communication language standards will help to create interoperable software by decoupling implementation from interface. Today, standards exist for a wide variety of domains. Electronic mail programs from different vendors can, for example, communicate with each other by means of a standard mail messaging system, called SMTP. Disparate graphics programs interoperate using standard formats like GIF and JPEG. Text formatting programs and printers use Postscript as a standard communication language.

However, if we would like to interoperate programs using different languages, problems may arise. Firstly, there can be inconsistencies in the use of syntax or vocabulary. One program may use a word or expression to mean one thing, while another program uses that same word or expression to mean something completely different. Secondly, there can be incompatibilities, meaning that programs may use different words or expressions to say the same thing.

Agent-based software engineering tries to eliminate these problems by mandating a universal communication language, in which inconsistencies and incompatibilities cannot exist. There are two popular approaches to the design of such a language: a procedural approach and a declarative approach.

It is not clear which of this two approaches will eventually succeed as the standard agent communication language. The declarative approach seems inevitable in the long run but because of the simplicity and familiarity, the procedural approach is likely to be more popular in the short term. The ultimate agent communication language may therefore end up more as a scripting language than ACL, a declarative communication language designed by ARPA which will be discussed in section 3.3.2.

3.3.1 Procedural communication language

A procedural approach is based on the fact that communication can be best modeled as the exchange of procedural directives. This approach is both simple and very powerful. They allow not only the transmission of individual commands but also entire programs and thus the ability to implement delayed or persistent goals of various sorts. Generally a procedural written language is directly and efficiently executable. However, the procedural method has some very annoying drawbacks. Firstly, devising procedures sometimes requires information about the recipient that may not be available to the sender. Secondly, procedures are unidirectional, but the agents' communications oftentimes should be bi-directional. Thirdly and most significantly, script languages are very hard to merge. This is not a problem when there is only communication on a one-to-one basis, but becomes more difficult when an agent receives multiple scripts from multiple agents that must run simultaneously and may interfere with each other.

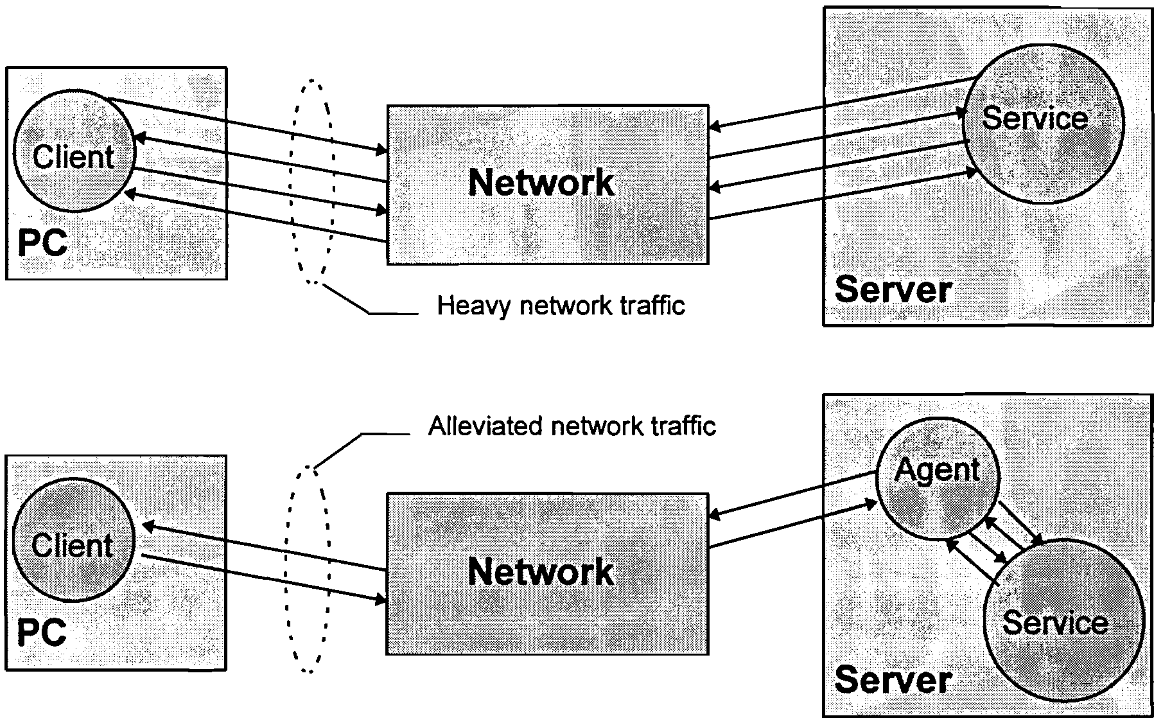


Figure 3-3: Reduction of network traffic with agents

The best known and used script language is General Magic's Telescript. The exact implementation of this language is, understandably, held secret by this company. Telescript is mostly used as an intelligent network agent communication language.

When information is to be transferred from a computer system A to another computer system B on a conventional network, the information usually has to be preceded by a request package from system A to system B, followed by the information. Oftentimes an acknowledgement has to be returned from system B to system A to tell that system that the information has arrived correctly or that the information has not been received correctly and that it has to be re-transmitted. Sometimes system B may need some preferences from system A, which is also sent over the network. As we can see, the overhead traffic is large.

Instead of packages of requests, permissions and data sent from A to B and vice versa, an agent, containing the messages, requests and preferences from system A, is sent on behalf of system A on the network to system B. Intermediate requests and answers do not have to travel over the network anymore, but can be dealt with in system B itself. After the agent has gained the answers to its queries, the agent returns to system A, thus saving a large amount of traffic, resulting in a saving of time, available bandwidth and cost. In Figure 3-3 we can see the reduction of network traffic when an agent, written in the Telescript language, takes the responsibility of the intermediate requests and responds.

Telescript can be compared to the Postscript language for printers **[Wayner]**. The Postscript language is much more flexible and efficient than sending simple bitmaps in terms of both size and speed. Most importantly Postscript is machine-independent. Telescript promises to bring the same interoperability to the networked world. General Magic already have a user interface on the market, called Magic Cap. When Magic Cap wants to communicate with the world it sends out Telescript agents. Another advantage of Telescript is that it can be implemented on every possible operating system.

Telescript consists of two large entities:

1. ***High Telescript***
2. ***Low Telescript***

High telescript is a modern, high-level object-oriented language. Its code is dynamically bound at run time and the Telescript engine handles garbage collection and memory management. When generated, High Telescript is sent to the locally residing Telescript engine which consists of a converter and the Telescript interpreter. The converter translates High Telescript into the low variety.

The fact that dynamic binding is necessary, is because Telescript must access both local and remote systems, having the engine handle the garbage and memory management for plugging security gaps. Agents resemble computer viruses very closely. Both are little programs that get to seize control of a foreign machine. The only real difference is that Telescript agents have to send invitations to a host system and can only execute on that system after presenting the correct credentials. The local Telescript engine decides how visiting agents can use local features like memory.

Low Telescript is a simple stack-based language, similar to Postscript, that runs on the Telescript interpreter. Its simplicity keeps the size of the interpreter down, minimizes the memory usage of agents and also makes the interpreter easy to port from one platform to another.

3.3.2 Declarative communication language

The declarative approach is based on the idea that communication can be best modeled as the exchange of declarative statements (definitions, assumptions etcetera). The declarative approach does not have the drawbacks, which are found in the procedural approach. This approach allows the language to be sufficiently expressive to communicate information of widely varying sorts; even procedures are included. It is very compact. A well-known declarative language, and rapidly becoming the standard for most agent-based systems, is the agent communication language designed by ARPA.

Researchers in the ARPA Knowledge Sharing Effort have defined the components of an Agent Communication Language (ACL) to consist of three clearly distinguishable parts:

1. **Vocabulary**
2. **Knowledge Interchange Format (KIF)**
3. **Knowledge Query and Manipulation Language (KQML)**

An ACL message can be constructed by using a KQML expression in which the 'arguments' are terms or sentences in the KIF format, formed with words from the ACL vocabulary.

The vocabulary of ACL is a large open-ended dictionary of words appropriate to common application areas. Each word has an English description for the use by humans in understanding the meaning of the word plus each word has formal annotations written in KIF for use by programs. The dictionary is open-ended allowing new words within existing areas and new applications to be added. ACL also allows the use of ontologies - different descriptions for describing the same thing - for any given area. An example is the ability to use polar coordinates, rectangular coordinates and cylindrical coordinates when describing 3D objects.

KIF is a prefix version of first order predicate calculus. An example of prefix calculus is for instance: *ab (meaning a times b). KIF contains various extensions to enhance its expressiveness. It provides for the encoding of simple data, constraints, disjunctions, negations, rules and so forth.

Suppose that, as an example of simple data structures in KIF format, a certain employer has to pay salary to his/her employees. Suppose these three sentences (more formal: tuples) are in the database:

- *(salary 366473 Woodridge 32000)*
- *(salary 325532 Aldridge 20000)*
- *(salary 547483 Widgets 45000)*

The first argument after the salary statement may be some internal identification code, the second the surname of the employee and the third the payment of that particular person.

A more complex expression is the following example:

- *(> (* (width chip1) (length chip1)) (* (width chip2) (length chip2)))*

to state that chip1 is larger than chip 2. KIF also supports logical operations. The complex sentence

- *(=> (and (real-number ?x) (even-number ?n)) (> (expt ?x ?n) 0))*[4]

asserts that the number obtained by raising any real-number ?x to an even power ?n is positive. One of the enhancements of KIF to increase its expressiveness is the use of the operators high-comma (`) and low-comma (,) and a related vocabulary. Take for instance the following sentence

- *(interested Peter `(salary ,?x ,?y, ?z))*

which asserts that agent Peter is interested in receiving triples in the salary relation. The use of commas signals that the variables should not be taken literally, but that the variables are parameters to the agent Peter. Without the commas, this sentence would say that agent Peter is interested in the sentence *(salary ?x ?y ?z)* instead of its instances. KIF may also be used as a script language via the *progn* statement. Take for example the sentence

- *(progn (fresh-line t) (print "Hello World!") (fresh-line t))*

This line ensures that there is a fresh line on the standard output stream, that "Hello World!" is printed on the same stream and that there will be a new fresh line. As can be seen, the semantics of the KIF core is similar to first-order logic calculation, despite the

[4] Names of variables begin with a question mark ('?') as in ?x and ?y

extensions (like the operators ` and ,) and restrictions to models that satisfy various axiom schematics (to give meaning to the basic vocabulary).

It is possible to design a complete communication framework with messages which conform to the KIF format only. This approach however would be very inefficient, because in this way each message must contain implicit information about the sender, the receiver, the time of message etc. The reason for this is the contextual independence of the KIF semantics. The efficiency can be enhanced by providing a linguistic layer in which this context is taken into account. This is the function of the KQML layer.

As used in ACL, KQML *messages* are similar to KIF *expressions*. Each message consists of a list of components in matching parentheses. The first word in the list represents the type of communication (e.g. reply, talk, ask, perform), the subsequent entries are KIF expressions appropriate to that communication, thus in effect the 'arguments'.

Intuitively, each message in KQML is one piece of dialog between the sender and the receiver, and KQML provides support for a wide variety of such dialog types. The simplest possible KQML dialog is for example

- from A to B: *(tell (> 3 2))*

to let agent B know that agent A thinks that 3 is greater than 2. In this case there is just one message; a simple notification. The sender simply conveys the enclosed sentence to the receiver. Generally, there is no expectation on the sender's part about what use the receiver will make of the information.

A more interesting dialog between two communicating agents is the sender requesting the receiver to execute an operation, which the receiver on its turn will reply with a 'satisfying' argument.

- from A to B: *(perform (print "Hello!" t))*
  from B to A: *(reply done)*

- from A to B: *(ask-if (> (size chip 1) (size chip2)))*
  from B to A: *(reply true)* (supposing that chip1 is indeed larger than chip2)

In the next example the sender asks the receiver to send a notification whenever the position of three chips are located with the subscribe command. The receiver sends three such sentences during some time interval, after which the sender will cancel its operation on the receiver with the unsubscribe command to let agent B know that further notifications are no longer needed.

- from A to B: *(subscribe (position ?c ?x ?y))*
  from B to A: *(tell (position chip1 8 10))*
  from B to A: *(tell (position chip2 8 46))*
  from B to A: *(tell (position chip3 10 34))*
  from A to B: *(unsubscribe (position ?c ?x ?y))*

KQML contains support for simple notifications, commands, questions and subscriptions. In addition it also provides support for delayed and conditional operations, request for bids, offers, promises etcetera.

Another fact which is worth mentioning is that KQML supports another layer besides the already discussed linguistic communication layer. It also contains a linguistic package layer to support the transmission of messages among agents operating in different processes. This layer inserts additional information that must be conveyed in communication protocols between distributed systems. Because of irrelevancy the details of this package layer will be left out of this thesis.

3.3.3 A comparison of procedural and declarative communication languages

Now that we have seen the differences between procedural and declarative communication languages, we can compare them to try to find the best approach, considering there is one, in which to implement an agent communication language.

It seems likely that the procedural approach will be the most popular implementation of an agent communication language in the short run as most development software and operating systems are still based on the procedural structure. The purchase of object-oriented software and the training courses for object-oriented programming, the programmers, who are used to procedural programming, have to undergo, will mean a large investment for most companies.

The probable initial success of procedural language can also be explained psychologically. If one has been in the habit of programming in procedural structures, it is very hard to suddenly drop this habit and become an object-oriented programmer. So the expectation is that procedural languages are likely to be more popular in the short run than the declarative languages.

However, if we look at the implementation of a procedural communication language we can expect a much larger cost than the implementation of a declarative one. In the procedural approach every agent must have the ability to create and understand messages. This messaging system however is initially not available and has to be developed separately whereas in the declarative systems the messaging system is already present, because of the object-oriented environment. Thus, the agents will not only be much larger in memory size, because of the implementation of a message 'interpreter' in every agent, the messaging system itself has to be developed.

Another drawback is that future revisions and enhancements of a procedural language will consume much more time and money because the actual source code probably has to be completely changed because of the changes which have to be made in not only the to be enhanced procedure and/or data-structure, but also other procedures making use of that particular procedure and/or data-structure.

In object-oriented programming however, the actual implementation of such data-structures is quite well bounded to only a small piece of source code, namely the data-structure's object code itself. Changes to a declarative language will therefore be much cheaper both time-wise and money-wise.

Another great advantage of a declarative communication language is the ability to implement procedural structures via the statement *progn.* On the other hand, declarative statements in a procedural language are not possible. The declarative approach is also written in a much more humanly understandable language, because it looks very similar to the representation of intentional notions by the use of first order logic, thus approximating our natural language very closely.

Also, the declarative communication language does not only closely resemble our natural language with its vocabulary but also with its grammar, which makes it easier for the user to use and understand. The procedural approach demands far more insight in the computer language itself. Indeed, we shall see in chapter 5 that the declarative communication language is much more suited for applying learning processes to agents than the procedural one.

While the procedural language seems more worthwhile on a short term, we eventually have to adopt the declarative approach because of its many advantages over the procedural approach, especially the ability to implement learning processes more easily. We may also expect some kind of blend (i.e. compromise) between the two approaches, which is already evident in the ability of running procedural statements in the declarative languages. We can therefore conclude that the declarative approach is the best approach in implementing an agent's communication language.

3.3.4 Multi-agent communication

Once we have the ability to use agents and the language to communicate with agents, there remains the question of how these agents should be organized to enhance collaboration. Two very different approaches have been explored:

1. ***Direct communication,***
2. ***Assisted coordination.***

With direct communication agents handle their own coordination, whereas with assisted coordination agents rely on special system programs to achieve coordination.

Direct communication has a big advantage in that it does not rely on the existence, capabilities, or biases of any other program. There are two main architectures to be distinguished in direct communication:

1. ***Contract-net,***
2. ***Specification sharing.***

In the contract-net approach agents in need of service distribute requests for proposals to other agents. The recipients of these messages evaluate those requests and submit bids to the originating agents. The originators decide on the bids which agents to task and then award 'contracts' to those agents.

In the specification sharing approach agents supply other agents with information about their capabilities and needs and these agents can use this information to coordinate their activities. Because specification mode decreases the amount of communication as opposed to the contract-net approach, it is often more efficient.

The disadvantages of direct communication are obvious: Direct communication's cost and the implementational complexity. If we take for example the Internet with millions of programs, the cost of broadcasting bids or specifications and the consequential processing of those messages will be phenomenal. Also each agent is responsible for negotiating with other agents and must contain all the code necessary to support this negotiation, resulting in a massive complex implementation.

To eliminate the above mentioned disadvantages, a new architecture has been developed based on the idea of assisted coordination. The system is often referred to as a *federated system*. A federated system with three machines, one with three agents and two with two agents apiece, is shown in Figure 3-4.

As can be seen from the diagram, agents *do not* communicate with each other directly, but with a system program, called a facilitator. The facilitators *do* communicate with each other. In a federated system, agents use ACL (oftentimes a restricted subset of ACL) to make their needs and abilities known to the local facilitator. In addition to this so-called meta-level information, they also send application-level information and requests to their facilitator and accept application-level information and requests from the facilitator in return.

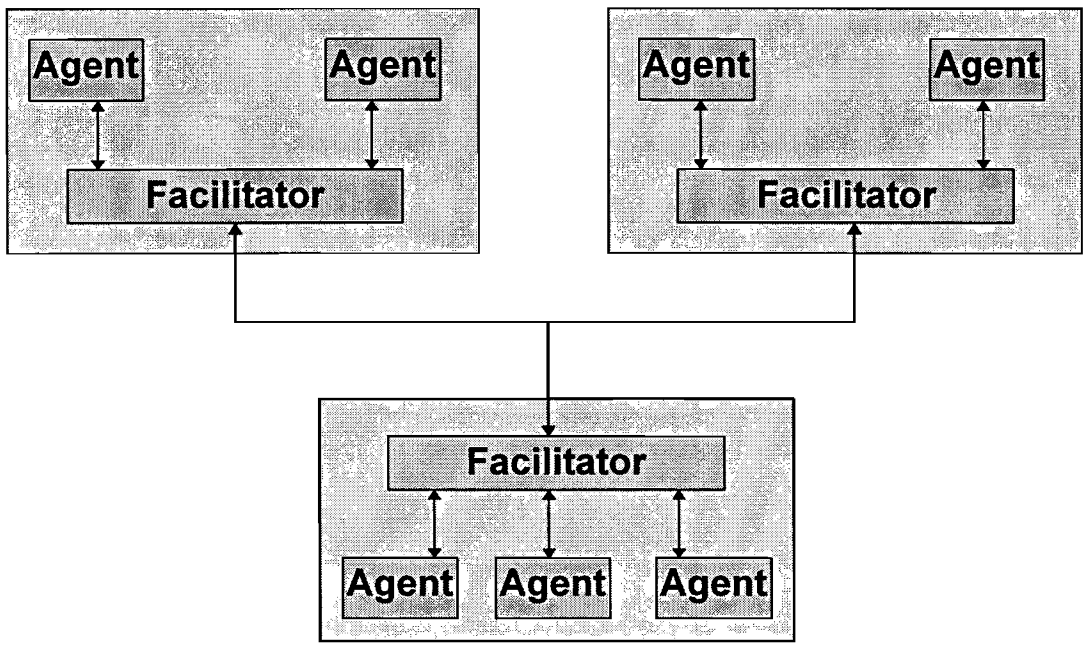


Figure 3-4: A federated agent system

A facilitator uses the information provided by the agents to transform the application-level information and route them to the appropriate facilitator of another agent. In effect, the agents form a 'federation' in which they give their autonomy up to their facilitator and the facilitators take the responsibilities for fulfilling their needs.

3.4 Requirements for an agent system

It is generally believed that agents will become the most profitable enterprise of the main software houses at the turn of the century. If a software house wants to act upon this assumption and develop an agent system, it has to have some requirements to fulfill: The basic requirements for a successful agent system implementation are:

1. The implementation of an environment which an agent can observe and affect.
2. The implementation of a flexible database system which can be effectively used by both the agent and the environment.
3. The implementation of some sort of support for a dialogue between user and agents, taking advantage of direct manipulation, natural language and all of the more conventional user interface components such as menus and buttons.

If we look at the current success of distributed systems, we can also include the recommendation that an agent system must support these distributed systems, resulting in multi-agent systems. This is however not a basic requirement, and thus not needed, but still recommended if it must contest with other vendor's agents systems.

4. Agents in operating systems and robotics

4.1 Introduction to Operating Systems

For the reader who is not thoroughly familiar with the concept of Operating Systems (OS), a very short overview will be presented in this section. In this chapter an OS agent system will be implemented in an existing operating system, requiring the user to have some familiarity with conventional operating systems.

If a computer user is working with an application, a 'ready-to-use' computer program, he or she usually does not have to have any knowledge of the computer hardware the application is running on. The application program contains the necessary code with which it can interact with the computer and its peripherals (see Figure 4-1).

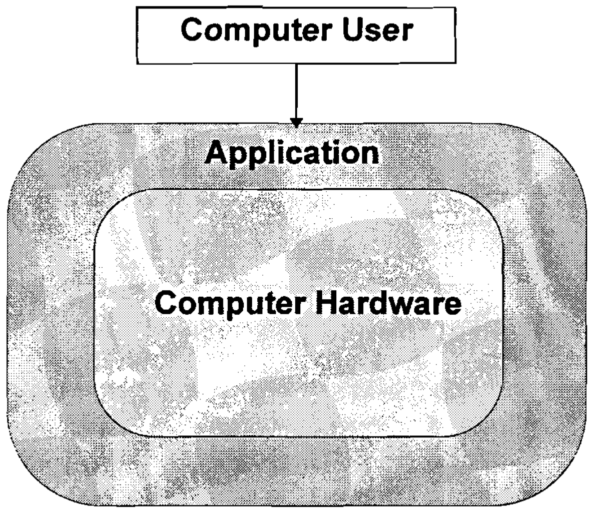


Figure 4-1: An application layer

Suppose we were to run several applications concurrently. In this case the system of Figure 4-1 would be very inefficient and on top of that also leading to dangerous situations:

1. The processor can only execute one instruction from one application at a time. To run more applications concurrently, a scheduling of processor time needs to take place.
2. Executing applications need memory space. The available memory space needs to be divided among the running applications in a fair way.
3. A peripheral may not be used by an application without claiming that peripheral for *its* use *only* in order to prevent a possible occurrence of deadlock.

Looking at the aforementioned three remarks we can conclude that some kind of control over the computer hardware is not a luxury, indeed. Furthermore, applications may contain code that is repeated in other applications, thus wasting valuable memory space. By implementing these common accessed parts of code in the sought-after control with appropriate rules on the usage, we may eliminate these problems. This control part, the control layer, is also known as an operating system.

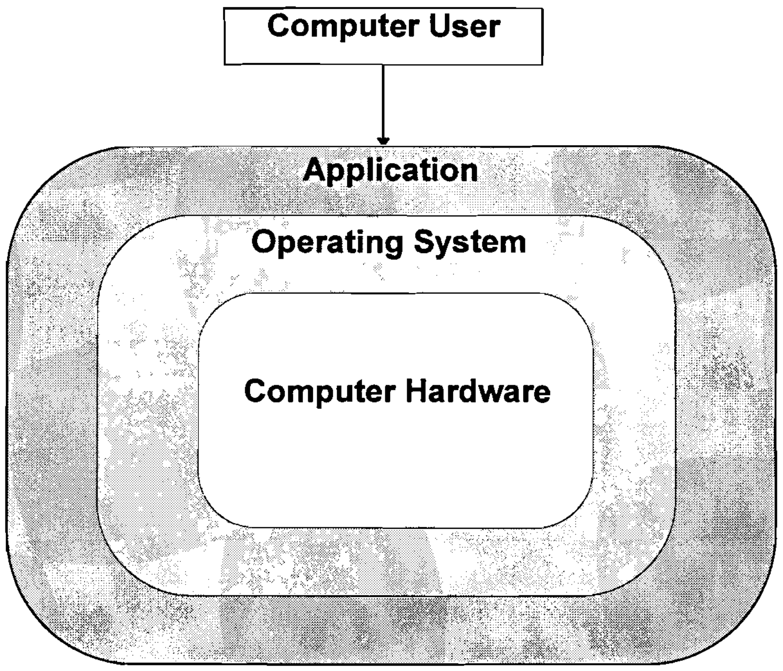


Figure 4-2: An operating system layer

An operating system may roughly be seen as an interface between a computer user's application and the computer hardware (see Figure 4-2). The operating system has essentially two main goals:

1. **Coordination** of the distribution of processor time, memory space and peripherals, and
2. **administration** regarding the common use of applications.

T. Hoare states the task of an operating system to be as follows:

> "The basic purpose of a computer operating system is to share the hardware which it controls among a number of users making unpredictable demands upon its resources; and its objectives are to do so efficiently, reliably and unobtrusively."

Before we proceed with the most important part of the operating system, we need to define some terms (see Definition 4-1).

Definition 4-1: Application, Program and Process

• ***Application***	:	- a (computer)program for the end-user.
• ***Program***	:	- the source code listing of an application
		- a description of the functions to be executed
• ***Process***	:	- a *running* program.

The innermost part of the operating system layer is called the nucleus. Its main goal is to create an environment in which *concurrent processes* can exist. This goal needs to be met regardless of the number of processes the system possesses. Thus, its main task will be the administration of processes, the insertion or deletion of processes and to assign these processes to the processor in some way. The nucleus has many more tasks but these are beyond the scope of our study.

All but the most primitive operating systems protect the operating system from being damaged by applications, by placing the operating system functions in a separate address space (or several address spaces) from the applications and by letting the applications run in an unprivileged mode of the processor, in which I/O and certain other instructions are NOT allowed. A processor can also run in a privileged mode, in which all instructions are allowed.

By letting each application run in a separate domain, with the memory-management unit set up in such a way that one application cannot access another application's data under any circumstance. Applications are run in an unprivileged mode of the processor, which is usually referred to as the *user mode*.

The part of the operating system that has access to the memory-management hardware and switches between the application's protection domains is referred to as the *kernel*. The kernel's code runs in a special, privileged mode called *supervisor mode* or *kernel mode*.

Until quite recently, the operating system was equivalent to the kernel, because all of the operating system ran in supervisor mode. This has changed with modern operating systems because of two reasons:

1. Operating systems are becoming very large which makes them increasingly hard to maintain as a single entity running in supervisor mode where bugs can create so much damage.
2. Distributed systems place different functions on different machines, so not all operating system functions are needed in every copy of the operating system.

The subject matter on operating systems presented in this section is sufficient enough to continue with the subject of OS agents. The reader who wants to know more about operating systems may find a large collection of well-written literature covering this subject. The most renown work in this area is undoubtedly Tannenbaum's book on operating systems, see **[Tannenbaum]**.

4.2 What does an OS agent do?

Since the birth of operating systems, its essential power has not changed that much: The form of the operating system interface may have changed considerably, the operating system's commands have remained nearly the same. To carry out a specific task a specific command has to be given, e.g. the command '*copy A B*' to copy a file A to destination B.

If we would like to carry out complex tasks, such as reducing disk utilization, the command set falls short. Since there is no command to reduce the disk utilization, some other way of executing this instruction is necessary.

The solution is usually to create a file consisting of a set of commands from the available command set that when executed sequentially carry out the specified task. This file is called a script file (UNIX) or batch file (DOS). Another solution may be the composition of commands directly from the command prompt by using 'pipes'. This approach however is very inefficient with frequent use because of its volatile form.

4.2.1 Script files vs. OS agents

The usage of script files is in itself a powerful and satisfying approach to solve the problem of accomplishing the more complex tasks. But there is one major drawback when working with script files: When the system layout changes, such as the addition of a printer, the relevant script files have to be rewritten to reflect the presence of the additional printer. Obviously, this is a very inefficient approach, indeed. Another (less severe) drawback of script files is that the user has to write and debug the 'programs' which should be considered as a real challenge to the average layman computer user.

Let us look at an example of a script file. At the department of Digital Information Systems we use an Apollo Domain UNIX operating system. A script file, called SHPQ, has been created to show the jobs queued on a laser-jet printer. The script file's contents is:

- *ls e:/ebq/sys/print/spooler.*

Clearly, this sentence just lists the jobs present in the printer spooler. Let us look at the knowledge of the operating system the user must have in creating this simple command. Firstly, one has to know that the printer 'spools' its jobs and that the print jobs are put in the e:/ebq/sys/print/spooler directory. A layman certainly could not have known this to be the case. Secondly, suppose that drive e has to be removed and the spooler is moved to drive d. This script file, and all other script files related to the printer spooler, have to be changed to reflect this removal. This simple example alone clearly illustrates the major drawbacks attached to script file.

Luckily there is one solution which does away with all the above mentioned drawbacks of script files and promises to finally revolutionize the power of an operating systems: operating system agents (OS agents). The user simply specifies a goal to accomplish, such as the reduction of disk utilization, to an agent and the agent itself decides how to accomplish that goal using its knowledge base of the system state and its commands. The agent dynamically synthesizes the appropriate command sequences, issues the required commands and system calls, handles errors and retries commands if necessary.

There are four main advantages when considering OS agent over script files:

1. The system chooses the most effective means of accomplishing a particular task, relying on commands or information that even the user may not be aware of.
2. If one method of accomplishing a task fails unexpectedly, the system can recover and try another, different method.
3. The language for specifying goals to the OS agents is system independent, as we have seen in chapter 3, making evolution or radical changes of the system transparent to the user.

4. Whenever the OS agent ‘knows’ about a new facility, that facility becomes instantaneously available to its planning process and is automatically invoked to satisfy relevant user requests.

There are however more advantages to OS agents. In section 4.3 we shall look at what these advantages are, when we have discussed the general framework for OS agents.

4.2.2 Communicating with an OS agent

Just as with general agents, we can divide OS agents into different classes. The chosen differentiation is however not unique as we may differentiate on different criteria and still come up with a plausible list of classes. The most obvious differentiation is that of the kind of request the user wants from an agent. With this differentiation we can, for example, construct the following classes:

- **Monitoring agents**: agents used to monitor certain events in a system. Examples: alerting the user that his disk utilization exceeds its pre-determined boundary; alerting the user that a colleague has logged on to the network or alerting the user that a document of his interest has been posted on some bulletin board.
- **Enforcement agents**: agents forcing the system to act according to some pre-determined constraints given to the agent by the user. Example: Keep all files in the *DOC* directory read-only.
- **Locating and Manipulating agents**: agents used to take actions on certain times or at certain sites. Examples: At midnight, compress files which have not been accessed for over a week, until 10Mb of free space is available; Print my paper on any available printer on floor 10, and, when successful, report back on which printer the paper was printed on.

These task classes combined do not represent the complete set of all agents, there are certainly more classes to be defined, such as scheduling agents. The point of the given classes here is to illustrate that we tell the agents *what* to accomplish and that we are not concerned with *how* the agents accomplish their tasks. The operating system interface should therefore be able to handle this kind of expressiveness. The language with which to ‘communicate’ with the operating system interface may be a natural language or another communication language such as ACL.

Let us look at the declarative communication language as discussed in section 3.3.2. To instruct an agent that it should maintain the disk utilization[5] of disk_1 below 75%, we could tell this to the agent as:

[5] Disk utilisation in this context means the part of the disk utilised by data

- *(maintain (disk.util.below disk_1 75%)).*

The agent may then use all techniques to his availability to keep the disk utilization below 75% by deleting files, moving files to another disk etcetera. *Maintain* is mainly used for constraints that must be fulfilled *all the time.* The *maintain* keyword is accompanied by one component:

- *(maintain <action>).*

Request on the other hand is used for actions that are usually applied only once. The *request* keyword has generally two accompanying components:

- *(request <action> :when <logical expression>).*

To tell an agent that the disk utilization of disk_1 must be brought down to 75%, when the current disk utilization is 85%, we may tell this to the agent as:

- *(request (disk.util.below disk_1 75%) :when (not(disk.util.below disk_1 85%)).*

We can also use variables. Suppose a user wants to be notified when another user, Peter, is logged onto machine B, we may tell this to the agent as:

- *(request (notify ?self) :when (active.on Peter ?machine_B)).*

The examples given here are all high-level commands, i.e. commands to protect the user from needing to understand the commands the computer's operating system itself uses (i.e. the command set from the operating system). So, the high-level commands have to be translated to commands which are understandable to the operating system (the low-level command set) .

4.2.3 Requirements for an OS agents

We are now able to define some requirements an OS agent should possess in order to accomplish the tasks as described earlier on.

1. *The agent must be able to execute OS commands, recover from any unexpected failures and , when necessary, automatically attempt alternative ways to complete its task.* For example, if a user was trying to send a file to some other user connected to the same network, and the network goes down, the agent must be able to restart the transmission when the network goes up again, or even attempt to retrieve the file from another source altogether.

2. *The synthesis of a sequence of OS commands, system calls etc. should be dynamic.* The synthesis must be dynamically established so that an agent can perform its task based on the *current* system configuration. Also the set of tasks the user may express is very large (potentially infinite), and would therefore be almost impossible to implement with static (i.e. hard-coding) synthesis.
3. *The agent must have to ability to adapt to differing conditions, resources and user tastes.* When an agent is busy with reducing disk utilization, one user may want all his Postscript dump-files to be deleted, while yet another user may want old files, i.e., not recently accessed, to be deleted instead of deleting his Postscript dump-files.

4.3 OS Agent's Framework

In order to develop agents in a certain environment, we need to define its framework first, as was discussed in section 3.2. The general framework for OS agents consists of two main modules **[Levy]**:

1. **The planner**, which synthesizes the from the user received requests of OS commands. Commands to be executed are sent to the clerk. This module is effectively the aforementioned (section 4.2.2) high-level to low-level translator. This module will be discussed in section 4.3.1.
2. **The clerk**, which handles all interactions between the OS agent and the environment. Actions include internode communication, command execution, status checking and so forth. This module will be discussed in section 4.3.2.

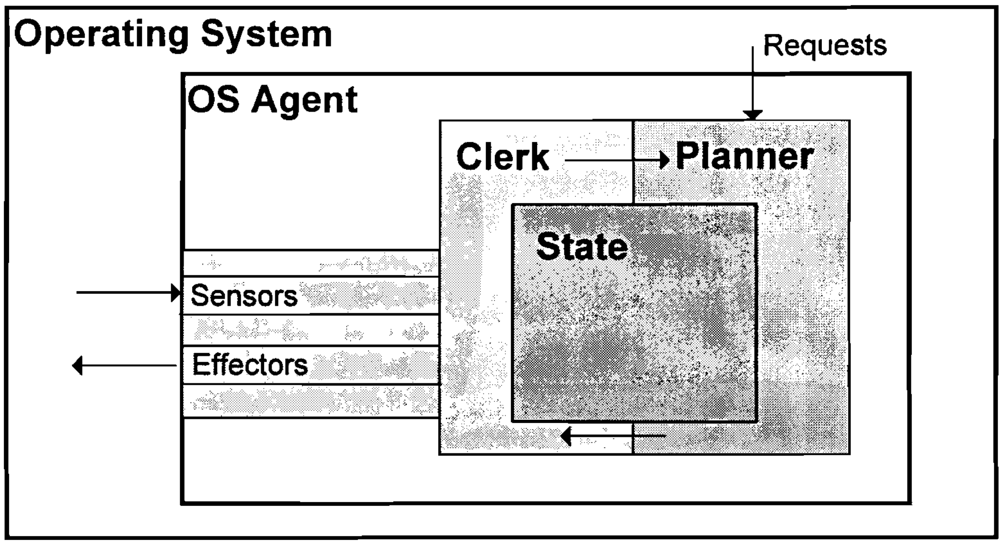


Figure 4-3: A general OS agent framework

If we look at the general agent framework from Figure 3-1, we can clearly see that the communication mechanism must be a part of the clerk. Because the clerk can also execute commands and does some status checking, it also covers another part of the controller besides the communication mechanism.

The planner consists of the rest of the controller not already contained by the clerk. Both clerk and planner use the state module. If we translate this description into a pictogram like the general agent system of Figure 3-1, we get the following figure of an OS agent (Figure 4-3):

We shall now take a more in-depth look at the planner and clerk of an OS agent system as described by **[Levy]**.

4.3.1 The Planner

The planner's main task is the translation of high-level commands (the user's requests) to low-level commands (the operating system's command set). This translation is carried out by mapping the high-level commands into a sequence of low-level commands. The models with which this mapping takes place may be provided by the system manager or may be automatically learned by the machine itself through the incorporation of machine learning algorithms, another study field in Artificial Intelligence which will be treated in section 5.

The planner's goal is thus trying to satisfy the user's request. The planner's goal is effectively a conjunction of atomic propositions, called the *subgoals*. In order to satisfy the user's request all the subgoals have to be satisfied, in this case only can the main goal be satisfied.

Given a goal, the planner dynamically synthesizes a sequence, called a *plan*, of OS commands from its goal and invokes the clerk to execute the plan. To dynamically generate a plan, the planner has to represent the available OS commands. This representation ought to answer to (at least) two fundamental questions about each command:

1. **Under what conditions will the command execute successfully?** The command will execute successfully if a set of necessary conditions for including the command in the plan, which are referred to as *preconditions*, is met.
2. **What is the effect of executing the command?** After executing (most) commands, the system state will almost certainly be changed. These effects are referred to as *postconditions*.

The planner creates a data structure representing its plan, containing all unsatisfied subgoals, which initially are all subgoals corresponding to the original plan. The planner then chooses one subgoal to satisfy and searches for an action with such a postcondition that it is equal to the condition after the subgoal is satisfied. If such an action can be found, its preconditions are added to the list of subgoals to be satisfied and the action is inserted into the plan. The planner then chooses another subgoal and repeats the process all over until all subgoals have been satisfied.

There are however some points that the planner should take into account when carrying out the iteration:

1. **Keep a model of the system's state**. Some subgoals may already be satisfied by the current system state or by actions already inserted into the plan.
2. **Necessity of backtracking**. Some subgoals cannot be executed because the environment has changed. For example, if the plan is currently in an FTP shell, the system cannot carry out normal OS commands, but only commands belonging to the set of commands of FTP itself and vice versa.

The planner is finished when each action's preconditions are satisfied, which brings the system to a state where the input goal is satisfied. The planner contacts the clerk to execute the plan it has created.

4.3.2 The Clerk

When the planner decides to execute a command, it sends a message to the clerk detailing the command and its arguments. After execution, the clerk notifies the planner whether the execution of the command was successful or not, and, when necessary, accompanied by what information was obtained (e.g., a list of file names in case of an `ls` command). If an execution fails it is up to the planner to correct the problem by choosing an alternative command or by entering a debugging mode in which the planner tries to determine the source of the error and to prevent it from happening again in the future.

The system-dependent information is stored in the clerk, thus keeping the rest of the OS agent invariant across different operating systems[6]. An OS agent may be forced to wait for an external event (i.e., an event occurring outside of the agent's control). In this case it is up to the clerk to detect this event and notify the planner about it.

If we were to implement the clerk just as we have studied so far, we would inevitably get into trouble the portability between different operating systems is considered because the

[6] This is of course not completely true, because the planner's command models varies from one system to another.

clerk contains machine specific information for its operation (e.g., how to delete a file). We would therefore like the clerk to operate machine independent. To solve this problem we can let the clerk send Remote Procedure Calls (RPC) to an operating system *server*. In turn, the OS agent servers interact with the operating system through standard system calls and commands where possible, and through extended mechanisms and abstractions where necessary.

By defining a clerk/server interface based on RPC, and locating in the server all of the operating system interface functions, we support both distribution and heterogeneity: distribution because a clerk can communicate with both local and remote servers in the same way; heterogeneity because the RPC interface is standardized and machine independent. In the next section we shall be looking at an example of such an OS agent system.

4.4 An example of an OS agent system

After having discussed a general OS agent's framework in the previous sections, it is now time to consider an example of such an OS agent system. The most obvious implementation would be to rely solely on an existing operating system command interpreter such as UNIX. Unfortunately, this approach has two major drawbacks:

1. Not all information concerning the operating system's state is available through the command interface, and
2. certain kinds of constraint and monitoring requests will require repeated polling at the command level, resulting in a dramatic (and oftentimes unnecessary) message overhead when such polling processes are being carried out on a network, especially when several OS agents are requesting the same information from one site.

The reason why a conventional operating system is chosen over an OS agent system, which is designed from scratch, is threefold:

1. The reader is probably more familiar with a conventional operating system, thus achieving a better understanding of the concept of OS agent systems.
2. OS agent can be reproduced more cheaply on existing operating systems; for the time being that is. A user does not have to buy and adapt to a new operating system supporting OS agents.
3. Agents can be studied more closely by using a conventional operating system instead of using an OS agent system, which are intuitively more complex, resulting in a larger research 'arena'.

Before proceeding, lets recapitulate the previous sections: the planner creates a plan on how to fulfill the request given by a user. The plan is transferred to the clerk which issues appropriate RPC calls to an OS agent server. In turn, the server issues machine specific commands to the operating system itself. By choosing a conventional operating system we can readily construct a diagram of the complete OS agent system (see Figure 4-4).

In Figure 4-4 we can see a dividing line between the standard interface (i.e., the conventional operating system) and the extended interface (i.e., the additions necessary to support OS agents in the conventional operating system). Let us now look at how the system parts interact with each other.

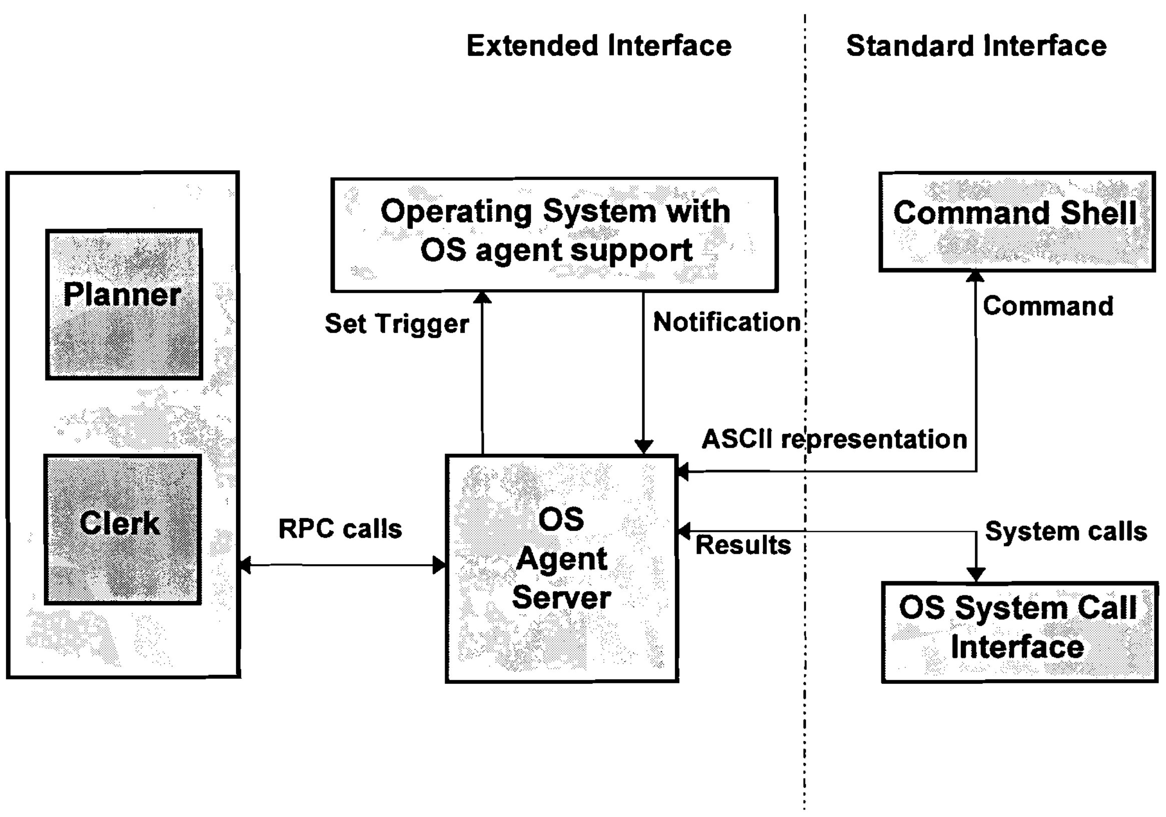


Figure 4-4: Structure of the OS agent

The system completely relies on events. An event, in an operating system's context, signifies a change of state within the system, such as a user entering a character from a keyboard, a modification of a file, the invocation of a kernel procedure etcetera. In the design the *types* of events, useful for the purpose of an OS agent, are determined a priori (e.g., 'file modification' is an event type). These set of event types are expected to change slowly over time. On the other hand, event *instances* of interest, such as the modification of a *particular* file, are subjected to change constantly.

The interaction between planner, clerk and OS agent server has already been explained above. But how does the OS server interact with the standard operating system interface? The OS agent server translates the clerk requests into a set of event notification demands that the kernel is expected to fulfill. The server may then indicate its interest in a specific event instance by 'setting a trigger' (see the arrow labeled Set Trigger in Figure 4-4). By setting the trigger, the operating system and the OS agent enter into a contract, whereby the system guarantees to notify the server each time that particular event instance occurs.

Because the kernel has to signal these occurrences of events as fast as possible, and communication between the clerk and the server might cause an undesirable waiting period, the utilization of network and processor resources are consequently kept to a modest level.

We can also limit the number of notifications of OS agent servers by additional constraints that have to be satisfied before notification occurs. We can, for example, confine the interest of the server to be notified only if a file size grows (or decreases) beyond a specified threshold, instead of requesting to be notified every time a file is modified. These additional constraints are known as 'predicates', which are Boolean expressions that evaluate to true or false.

To recapitulate, the event specification by OS agents is facilitated by three mechanism:

1. Defining a relatively static set of event types.
2. OS agents servers can dynamically express an interest in specific instances of these events, depending on the requests that the OS agent servers receive from their clerks.
3. Inclusion of a mechanism for patching small server-supplied code sequences into the kernel to signal specific event occurrences.

5. An agent's learning process

5.1 Introduction to learning processes

The agents and operating systems we have discussed so far have all had some kind of basic knowledge. When we want, for example, the disk utilization to be reduced below a certain threshold, the agent 'knows' that files may be deleted or moved to another disk in order to reduce the disk utilization. In some way the agent has gained that knowledge. The knowledge may have been given in a - for the agent - passive way, by the operating system or by the computer user, or in an active way, in which the agent learned that deletion and moving of files equals reduction of disk utilization during its lifetime, which is of course the most interesting case for our study.

There are many things an agent can learn, but not many ways in which it can learn it. The following categories can be classified when considering what an agent can learn:

- learning declarative knowledge
- learning control
- learning behaviours
- learning to select behaviours and actions

Learning declarative knowledge contains only a small field to cover. The only declarative knowledge the agents have to learn to date are maps of the environment. Maps and environments are closely tied to action in the 'world', which is why they are the primary type of declarative knowledge so far used in agents. Note that not all maps and environments are explicit and declarative per se: when they are tied to act*ing* and interact*ing* in the 'world', agents can learn procedural knowledge. So, the difference between declarative knowledge and procedural knowledge, being the actions themselves versus the acting , is fairly obvious.

Learning control deals with learning the forward or inverse model of the system. A forward model provides predictions about the output expected after performing an action in a given state. The inverse model provides an action, given the current state and desired output. The control of an agent has already been dealt with in section 3.2.

Learning new behaviours deals with the problem of acquiring strategies for achieving particular goals. Learning new behaviours is learning *how* to do something, while on the other hand selecting behaviours is learning *when* to do something. In section 5.1.2 we shall study behaviour learning in more detail. Before this, however, we shall look at some possibilities that learning processes could introduce in (agent supporting) operating systems.

5.1.1 The quest for knowledge

A most interesting question arises when we look at the active way of obtaining knowledge: Does the agent really need the intelligence at its time of creation in order to operate correctly, i.e., undergo a learning process with a minimal baggage of basic knowledge, and be able to accomplish its task which was set out for it, or can it learn the necessary basic knowledge to accomplish the task by a trial and error method?

Taking the train of thought on this concept even beyond the subject of agents: Do we really need an operating system with its 'intelligence' already implemented or, to put it in another way: Will it be possible to have a 'dumb' operating system, without any knowledge whatsoever, to be able to operate like an ordinary operating system, just by gaining all its knowledge using the process of trial and error?

Obviously, the latter question must be answered negatively. There should always be some knowledge implemented in order to start to learn from some, for the agent or operating system unknown, knowledge and thus to gain new knowledge. The system must certainly know beforehand how to interpret, store and administer new, unknown knowledge. We shall see shortly the reason why the answer to the latter question *has* to be negative.

Is it really relevant or even necessary to consider the question whether an operating system can learn some new knowledge? We could create a practical case, in which machine learning may be worthwhile and show that the question asked is relevant and necessary.

Suppose some manufacturer of Interactive Compact Disc (CD-I) players has players in several price ranges, each with different specifications, the more expensive, the more features the player will have. A cheap player won't play CD-I movie discs while a more expensive player does play them. Internally the 'operating system' of the player with the most features must be specified, verified and implemented. The cheaper players' operating system can then be obtained by stripping the fully specified operating system of the appropriate functions, according to their specifications, leading to a relatively impractical design method, especially when upgrading of the operating system is considered.

However, we could design an 'operating system' that could 'learn' whether some features - such as the ability to process moving pictures, to display the playing time etc. - are available or not and operates according to the features found. If this were indeed the case, it would suffice to design only one 'operating system' for the entire range of players, making the designing stage much more practical and faster to complete than designing separate operating systems for differently specified CD-I players.

Most importantly however, is that the agent operating system is dynamically configurable. For example, a consumer might want to upgrade his cheap CD-I player model by buying a movie module. He only has to insert it into the player in order to be able to play the CD-I movie discs, without the need to replace or re-configure the operating system; the agent will detect the presence of the new module and alerts the operating system of the fact that a movie module is now present. With the conventional design method, this is not possible. However, the manufacturer may, for instance, implement dip-switches into the cheaper models, making not only the designing stage more complex, but the user himself has to alert the player of the new addition of a module with the dip-switches.

Another field of work where such 'adaptable' operating systems would be very much wanted and needed is the field of robotics. One could create a universal operating system that may operate on all, similarly tasked, robots. Suppose we have two robots, one immobile and one mobile robot. Each robot has to examine some previously manufactured metal ring, for the connection of water pipes in housings, for structural faults. It has to take the rings from some storage box, examine the ring, and store it after examination in a box labeled unusable or useable.

If the first storage box is empty, or one of the latter storage boxes becomes full, the immobile robot must alert someone or something mobile to either get a new box of rings or replace the full box with an empty one. The process of examination must therefore be halted until this action has occurred. The mobile robot however may retrieve a new box or replace the full boxes itself. It does not need to wait for a person or other robot to take the appropriate action with the boxes.

The operating systems of the two robots aren't that much different, the only difference being the alerting of someone and the placement/removal of the storage boxes. We could choose one operating systems for both robots and let the operating system itself adapt to the (im)mobility of the robot. Later on, the robot can always be replaced by its counterpart, without the need to replace the operating system altogether.

Here we have presented two examples which clearly shows the advantages of implementing the ability of gaining knowledge by learning processes. The learning process has two main purposes:

1. It simplifies the built-in knowledge (the CD-I example).
2. It simplifies the adaptation to external and internal changes, if adaptation was available in the first place (the Robotics example).

We will term the ability to cope with changes in the environment as *adaptability*. With the help of adaptability, agents can deal with 'noise' in their internal and external sensors and with inconsistencies in the behaviour of the environment and other agents. All creatures, whether natural or artificial (e.g,. the agents themselves), fail at their tasks under certain conditions. The purpose of learning then, is to reduce this set of conditions, and thus reducing the chance of failure. The definition for learning in general is presented in Definition 5-1.

Definition 5-1: Learning

The successful attempt to reduce a certain set of conditions, under which tasks may fail, thus reducing the chance of failure.

Adaptability, however, does not necessitate learning. Many species are genetically equipped with elaborate 'knowledge' and abilities, from very specific (the ability to record and utilize celestial maps), to the very general (language, motor control). Birds, for example, do have the capability to adapt, without learning. They simply adapt by travelling to warmer regions when the seasons are getting colder and return when the cold periods are over, without ever having learned this. This capability however is not a feature of inanimate objects, such as an operating system.

This also confirms the answer to our previous question - whether it is possible to create a fully operable operating system from a 'dumb' operating system by using learning processes - to be unfeasible because operating systems do not posses 'genes' as such. The basic knowledge needed by the operating system to operate appropriately can therefore be seen as the 'genes' of the operating system. We must bear in mind, however, that genetic code is finite. The adaptability, without learning processes, is then also limited.

Having shown the need for some basic knowledge in order to start an (artificial) learning process, we now focus our attention back on agents. The learning process by the trail and error method mentioned earlier, is also known as *Reinforcement Learning*. It is based on the interactions of agents and its environment.

In reinforcement learning approaches the agent gains its knowledge from externally created 'rewards' and 'punishments', according to the task the agent is currently executing. Thus reinforcement learning is a class of the learning methodologies in which agents learn from feedback of the environment. The feedback is interpreted as positive or negative scalar reinforcement. The goal of learning is to maximize positive reinforcement (the 'rewards') and/or minimize negative reinforcement (the 'punishments') over time.

5.1.2 A robotics learning experiment

The above-mentioned reward and punishment system may be clarified by an experiment that Maja Mataric has carried out at MIT Artificial Intelligence LAB **[Mataric1 & 2]**. The reason why this experiment is mentioned in this thesis is because:

1. Mataric introduced *learning capabilities* to the robots, and
2. the *techniques* used by the robots to learn certain things via a reward and punishment system, could be used for agents with learning capabilities.

So, when the word 'robot' is mentioned in this section the reader is encouraged to interpret it as the word 'agent', during any interaction with the environment or other robots.

Mataric created 20 identical mobile little robots with the ability to locate and move ice pucks. The robots are able to locate other robots via infra-red sensors and radio communication by broadcasting their position relative to two stationary beacons. She created a small field of 12.5 by 5 meters in which the center was marked as Home (see Figure 5-1). The robots' task was to deliver the pucks, which were spread out over the field, to the area marked home. By placing a camera above the field the movement of the robots could be precisely analyzed. Note that the robots were not *programmed* to deliver the pucks to home, but were *told* to do so. They had to solve the problem on how to deliver the pucks to home themselves.

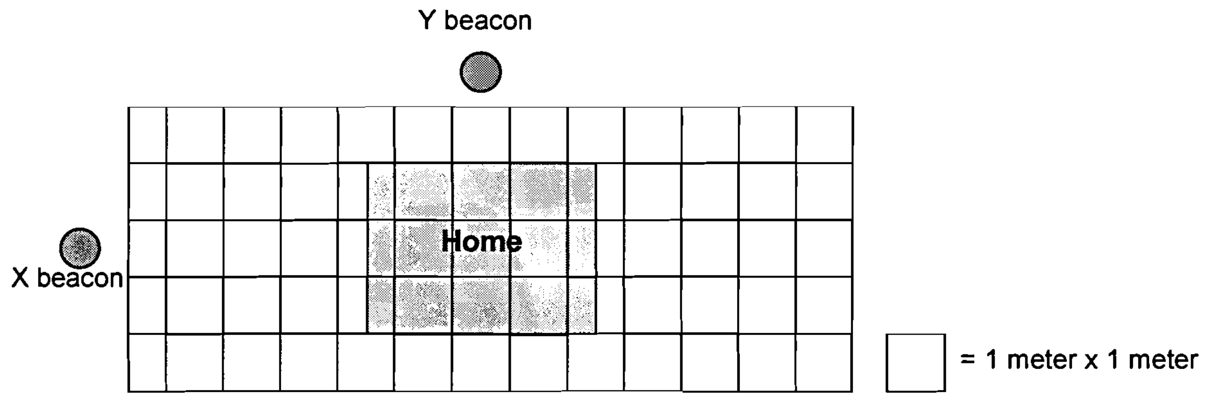


Figure 5-1: The experiment field

This model relies wholly on the principles of *basic behaviour*, a means to combine the constraints from the robot (read: the agent), such as the mechanical and sensory characteristics, and the constraints from the experiment field (read: the agent's environment), such as the information the robot can obtain from the experiment field. Mataric postulated that, for each domain, a set of behaviours can be found that are basic in that they are *required* for generating other behaviours, as well as being the set the robot needs to reach its goal. The basic behaviours cannot be reduced any further to each other.

The basic behaviour set Mataric used continued the following behaviours:

- **safe-wandering:** minimize the collision between robots,
- **following:** minimize interference from other robots by structuring movement of any two robots,
- **homing:** enable the robot to proceed to a particular location, in this case the home field,
- **aggregation:** gathers the robots, and
- **dissipation:** scatters the robots

Other behaviours like grasping and dropping could be included, but they are not relevant to the interaction between environment and other robots, they are only relevant on the individual robot level, so these behaviours will not be included in the minimized behaviour set used in this experiment.

The basic behaviours are intended as building blocks for achieving higher-level goals. The behaviours are embedded into an architecture that allows two types of combination:

1. **direct**, by summation of different basic behaviours into a new behaviour (see Figure 5-2), or
2. **temporal**, by switching between different basic behaviours, thus creating a new behaviour (see Figure 5-3)

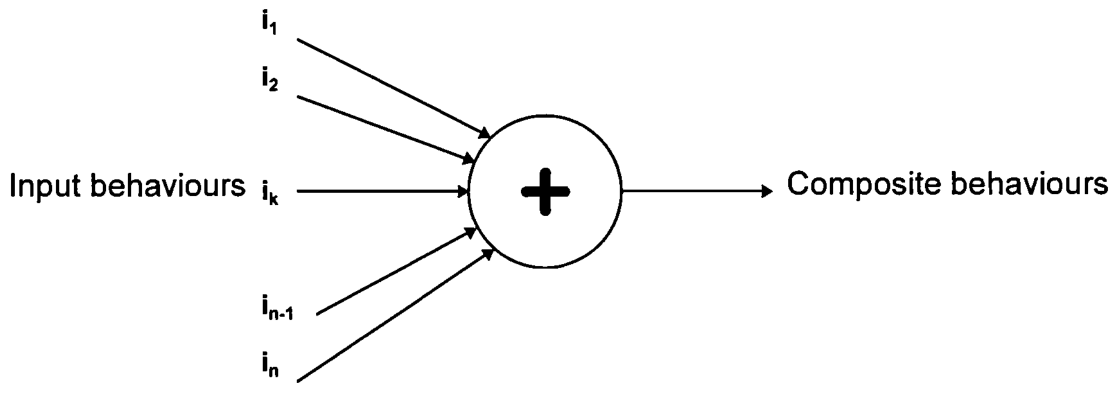


Figure 5-2: Direct behaviour combination

Flocking behaviour can be simulated by summation of safe-wandering, aggregation and homing behaviours, while foraging (collecting pucks and moving them to home) may be simulated by switching between safe-wandering, dispersion, following and homing behaviours.

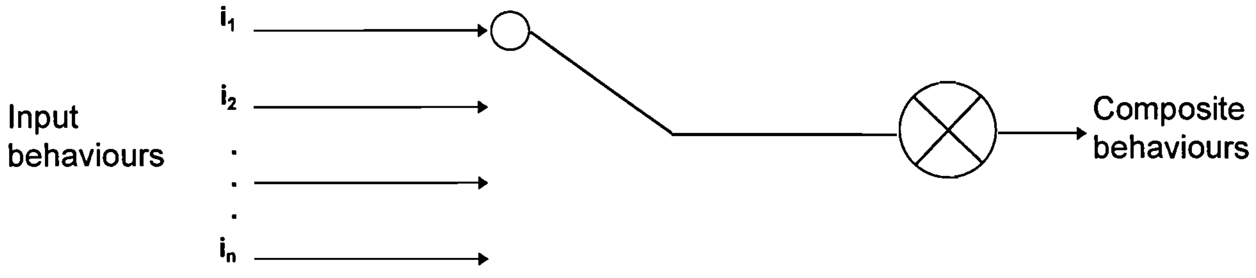


Figure 5-3: Temporal behaviour combination

Once the basic behaviour set is established, it can be implemented using a variety of algorithms. The algorithms are all verified on their correctness by carrying out 50 experiments per basic behaviour. Although proving that one robot acts formally correct is difficult, group behaviour of the ensemble of robots can be evaluated more easily.

The experiment proved that behaviours can be successfully implemented by using the appropriate algorithms, and that higher-level behaviours can be created by combining behaviours, be it direct- or temporal-wise, from the basic behaviour set. The results leading to this conclusion are too extensive to reproduce in this thesis, so the interested reader is referred to **[Mataric2]** for a full account on her experiment.

In this experiment the higher-level behaviours are determined before the experiment itself, i.e., the robot 'knows' how to behave to accomplish his given task because the higher-level behaviours are already implemented by direct or temporal combination of the appropriate basic behaviour algorithms. However, we want the basic behaviours to be automatically combined into higher-level behaviours, via the learning process already mentioned earlier: unsupervised reinforcement learning based on the robots' (or agents') interaction with the environment. In this situation only can we truly speak of a learning robot.

To summarize the experiment in so far, we can conclude that in order to create intelligent agents that use learning processes, we have to specify, implement and verify a basic behaviour set, from which the agents should create new higher-level behaviours. Because we want the agents to combine the different basic behaviours on their own, we have to create the ability for the agent to do so. A useful approach is the reinforcement learning process.

5.2 Reinforcement learning

Knowing the unavoidable need for automatic combination of basic behaviours rather than manually feeding the already 'computed' combinational behaviours to the robots, Mataric enhanced her experiment by applying external rewards and punishments on the robots, thus implementing a learning process.

In the framework she reformulated the terms *state* and *action* as respectively *behaviour* and *condition*, the reason being that state as a monolithic descriptor of the robot and its environment did not scale up to the multi-robot domain used in the experiment, given the continuous and discrete aspect describing the robot (velocity, infra-red sensors, radio data) and the existence of many other robots in the environment.

Even for the simplest of agents, a monolithic descriptor of all state properties is prohibitively large. Furthermore, atomic actions are too low level and have effect too unpredictable and noisy that they are not useful to be applied to any learning algorithm. She defined behaviour and condition as:

Definition 5-2: Behaviour and condition

- **Behaviour:** A control law that achieves goals but hides low-level control details.
- **Condition:** The necessary and sufficient subsets of state required for triggering the behaviour set. Conditions are many fewer than states, thus greatly reducing the robot's learning space and speeding up any reinforcement learning algorithm.

Reinforcement learning in situated domains can now be defined as:

Definition 5-3: Reinforcement learning (in situated domains)

Reinforcement Learning in situated domains can be defined as learning the conditions necessary and sufficient for activating each of the behaviours in the repertoire such that the agent's behaviour over time maximizes received reward.

5.2.1 Accelerated learning

The amount and quality of the reinforcement determines how quickly the agent will learn the correct policy to complete its given task. In general, reinforcement learning can be accelerated in two ways:

1. by building more information, and
2. by providing more reinforcement to the agent.

In nondeterministic, uncertain worlds, learning in a limited time interval requires shaping of the reinforcement in order to take advantage of as much information as possible. To aid a learner in a dynamic, noisy and nondeterministic environment, the reinforcement learning process can be enhanced with the introduction of:

1. **heterogeneous reward functions** that partitions the task into subgoals, thus providing more immediate reinforcement, and
2. **progress estimators** that are functions associated with particular conditions which provide some metric of the learner's performance. Progress estimators are also know as the internal critics.

Figure 5-4 shows the results of a foraging task implemented with three different algorithms over twenty different trials. The first algorithm, called Q-learning, is a standard reinforcement learning algorithm. The robot receives a reward whenever a puck is delivered at the home region. The graph shows that about 27% of trials the robot learned the correct policy.

The second algorithm implemented the heterogeneous reward functions, in which also subgoals, like grasping a puck, dropping a puck and reaching home, are rewarded. We can see clearly that robot learns to accomplish its task much faster. In about 50% of twenty trails the robot was able to learn the policy correctly.

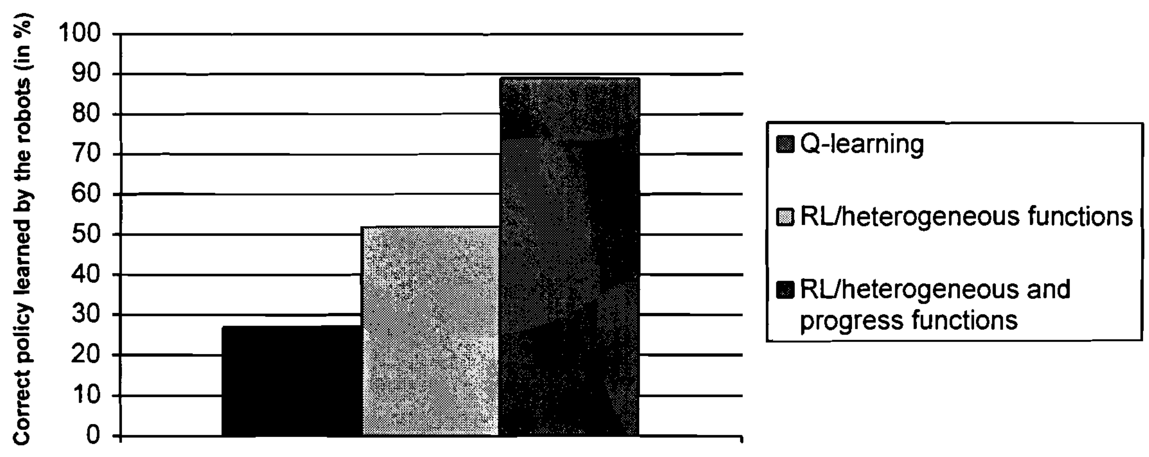


Figure 5-4: Performance of three reinforcement learning strategies

The third algorithm included not only the heterogeneous function but also the progress estimator functions. One estimator monitoring the progress in getting away from an intruder and a second estimator monitoring progress toward home.

The two progress estimators were found to be sufficient for making the given learning task possible and for consistent and almost complete learning performance. Disabling either one of the estimators disabled the robots from learning the complete policy.

5.2.1.1 Heterogeneous reward functions

Reward functions with a single high-level goal intuitively require a large amount of intermediate reinforcement in order to aid the agent in learning. The more subgoals created from the high-level goal, the more (intermediate) reinforcement can be applied, and thus the faster the learner converges to its correct policy.

It was already mentioned that agents in situated domains usually maintain multiple concurrent goals, which can be achieved and maintained by using behaviours as the basic unit of control. A task in such a situated domain can then be represented with a collection of these concurrent goal-achieving behaviours. Reaching a goal generates an event that provides primary reinforcement to the learner.

The general form of such an event-driven reinforcement function can be represented with:

$$R_e(c,t) = \begin{cases} r & \text{if the event } E \text{ occurs} \\ 0 & \text{otherwise} \end{cases}$$

Event-driven reinforcement for any event E is a function of the set of conditions c and time t. The received reinforcement r may be positive or negative.

A general heterogeneous reward function has the following form:

$$R_e(c,t) = \begin{cases} r_{E1} & \text{if event } E1 \text{ occurs} \\ r_{E2} & \text{if event } E2 \text{ occurs} \\ . & \\ . & \\ . & \\ r_{En} & \text{if event } En \text{ occurs} \\ 0 & \text{otherwise} \end{cases}$$

The complete reward function is a sum of inputs from the individual event-driven functions. In case multiple events occur simultaneously, appropriate reinforcement for all of them is received from multiple sources.

Lets illustrate a heterogeneous function with an example for the robotics experiment from section 5.1.2:

- A robot receives reward R_a whenever it avoids an obstacle, and reward R_h whenever it reaches home.
- The corresponding reward function appears as follows:

$$R(c,t)=\begin{cases} r_a & \text{if an obstacle is avoided} \\ r_h & \text{if home is reached} \\ 0 & \text{otherwise} \end{cases}$$

- If the robot happens to be avoiding an obstacle and reaches home at the same time, it receives reinforcement from both sources concurrently:

$$R(c,t) = r_a + r_h$$

As can be seen from the example above, each of the heterogeneous reward functions provides a part of the structure of the learning task, thus speeding up the learning process.

5.2.1.2 Progress estimator functions

The progression of most goals can be measured immediately as few tasks need to be defined as long sequences of behaviours without any feedback. Progress estimators use domain knowledge to measure progress during a behaviour and, if necessary, to trigger behaviour termination. Intermittent reinforcement can be provided by estimating the agent's progress relative to its current goal and weighting the reward accordingly. The following are two general forms of progress estimator functions:

$$R_p(c,t)=\begin{cases} m & \text{if } c \in C' \wedge \text{progress is made} \qquad (m > 0, C' \subset C) \\ n & \text{if } c \in C' \wedge \text{no progress} \qquad (n < 0, C' \subset C) \end{cases}$$

$$R_s(c,t)=\begin{cases} i & \text{if } c \in C' \wedge \text{progress is made} \qquad (i > 0, C' \subset C) \\ j & \text{if } c \in C' \wedge \text{regress is made} \qquad (j < 0, C' \subset C) \\ 0 & \text{otherwise} \end{cases}$$

C is the set of conditions, and *C'* is the set of conditions associated with the given progress estimator, i.e. those conditions for which the given progress estimator is active. R_p is a two-valued function that monitors only the presence and absence of progress, while R_s is a three-valued function that monitors the presence and absence of progress, as well as negative progress or regress.

Lets consider the following example of a progress estimator function using feedback as a learning signal:

- The robot's task is to learn to take pucks home.
- Having found a puck, the robot can wait until it accidentally finds home and then receives a reward.
- Alternatively, it can use a related subgoal, such as getting away from the 'food'/puck pile, for feedback.
- In such a scheme, the longer the robot with a puck stays near the 'food', the more negative reinforcement it receives.
- This strategy will encourage the behaviours that take the robot away from the food, one of which is *homing*.

The reward functions to accomplish this task can be readily made with the above-mentioned formulas.

5.2.2 The Markovian Decision Process

The most commonly, but not exclusively, used model of reinforcement learning is the so-called *Markovian Decision Process* (MDP) model. The model contains conditions as stated in Definition 5-4.

Definition 5-4: Markovian Decision Process conditions

1. The agent and the environment can be modeled as synchronized finite state automata.
2. The agent and the environment interact in discrete time intervals.
3. The agent can sense the state of the environment and use it to make actions.
4. After the agent acts, the environment makes a transition to a new state.
5. The agent receives a reward after performing an action.

While many interesting learning domains can be modeled using MDP, situated agents learning in a nondeterministic, uncertain environment, which an operating system certainly is, do not fit this model. Most RL models are based on the assumption that the agent and its environment are always in a clearly-defined state that the agent can sense. In situated domains however, the world is continuous and only partially observable instead of the in MDP assumed readily prelabeled states and a readily and consistently accessible world.

The reason why states are not clearly-defined is because the agent's collection of properties may contain discrete states, while other properties are continuous. The descriptor of all state properties will then be quickly too extensive to work with, even for small and simple agents. As was mentioned earlier, the notions of state and actions can be replaced by the notions of behaviour and conditions, resulting in a substantially smaller set of properties.

The observability is only partial because sensors, however complex they are, have limited abilities and cannot provide descriptions of the world as we perceive it to be. Also, the sensor's world of perception is quite smaller than ours and oftentimes fixed on a certain subspace of the world. So, the sensors return a simplified property such as presence of and distance to objects within its fixed sensing region. This collapse of multiple states into one results in partial observability, i.e. there is a many-to-one mapping of the world and the internal states. This inability to distinguish different states makes it difficult or even impossible for a learning algorithm to assign the appropriate utility to actions associated with such states.

From the previous paragraph we can conclude that modeling of dynamic, nondeterministic worlds, which most multi-agent systems are, to be very difficult, indeed. Oftentimes such a world is still modeled using the MDP model, be it slightly adjusted. We see this practice often in other disciplines as well, such as queuing problems, and contains mostly allowable differences with the real-world model. We have to bear in mind however that the agents' sensors will probably introduce an even lesser accurate representation of reality than the model itself does. The sensors are the most difficult part of the agent to model to reflect the agent's world.

From this viewpoint, reinforcement learning algorithms have a general form. In
Figure 5-5 this general form is represented in a block graph. Note that the state and action have re-appeared in the ideal MDP model. The letters in this graph have the following meaning:

- I : the internal state
- a : the current action
- s : the current world state
- F : an evaluation function mapping I and s into a
- r : reward for executing action a in world state s
- U : an update function mapping I, s, a, r into I

We can see that the internal state I encodes the information the learning algorithm saves about the world, most commonly in the form of a table maintaining state and action data. The update function U adjusts the current state based on the received reinforcement, and maps the current internal state, input, action and reinforcement into a new internal state. The evaluation function F maps an internal state and an input into an action based on the information stored in the internal state. Different RL algorithms vary in their definition of U and F.

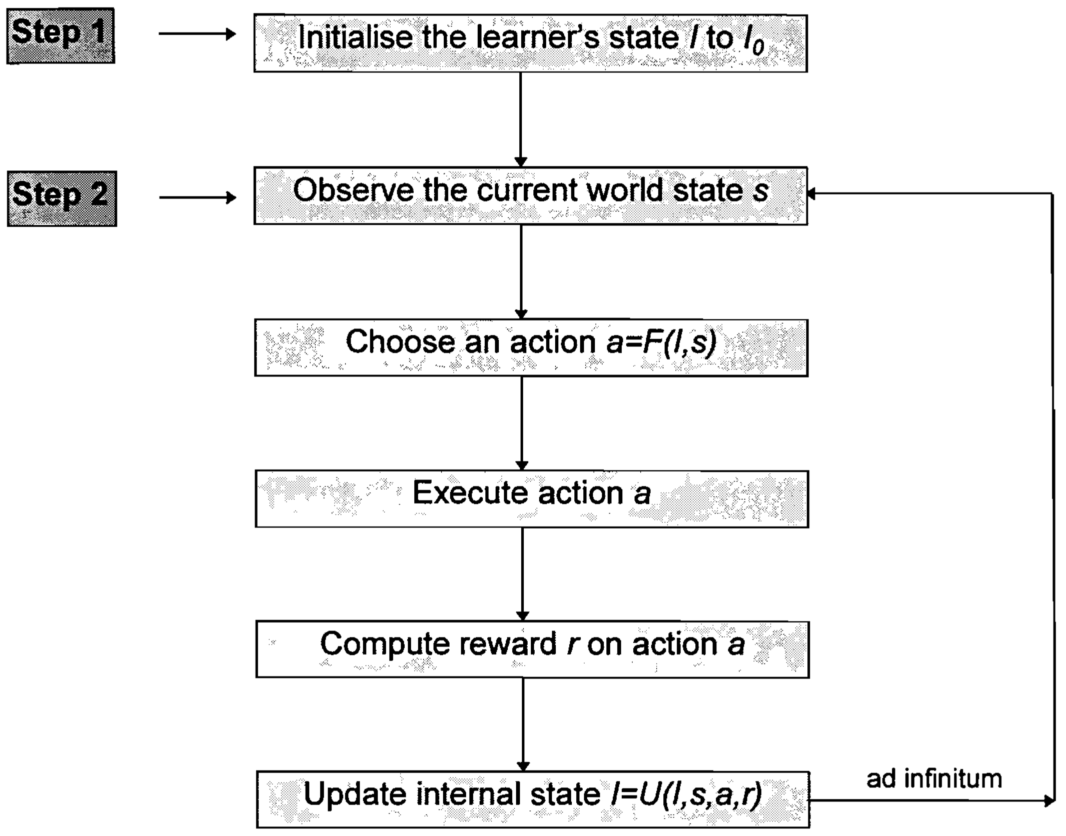

Step 1
Initialise the learner's state I to I_0
Step 2
Observe the current world state s
Choose an action $a=F(I,s)$
Execute action a
Compute reward r on action a
Update internal state $I=U(I,s,a,r)$
ad infinitum

Figure 5-5: General form of a reinforcement learning algorithm

The mainstream RL algorithms are based on the *temporal differencing (TD)* class, meaning that TD methods deal with assigning credit or blame to past actions by attempting to predict long-term consequences of each action in each state. Assigning delayed reward or punishment is considered to be the most important and difficult problem in reinforcement learning. Temporal credit is assigned by propagating the reward back to the appropriate previous state-action pair. So, temporal differencing methods are based on *predicting* the expected value of *future* rewards for a given state-action pair, while assigning credit is locally based on the difference between *successive* predictions. The aforementioned reward functions determine how credit is assigned.

Provable convergence of RL algorithms based on temporal differencing and other related learning strategies based on dynamic programming is asymptotic and requires infinite trials. Generating a complete policy, however incorrect, requires time exponential in the size of state space, and the optimality of that policy converges in the limit as the number of trials approaches infinity. Even with only ten-bit states, this translates into hundreds of thousands of trials, so even in ideal Markovian worlds the number of trails required for learning is prohibitive, even for the smallest state spaces. In situated learning problems still another problem arises, because with insufficient reinforcement the learner may fail to converge (i.e., complete its policy), as we saw earlier in Figure 5-4, in which case with only standard reinforcement (Q-learning) the robots failed to grasp their correct policy 3 out of 4 times.

Returning to the subject of temporal credit assigning, we can expect that the more delayed the reward is, the more trials the learning algorithm requires and consequently the longer it takes to converge. Algorithms using immediate reinforcement naturally learn the fastest.

Rewards are commonly used in only two extreme ways: immediate or very delayed. It is not common practice to use rewards 'almost' immediate or only slightly delayed. In situated domains, however, rewards do tend to fall in between the two popular extremes, providing some immediate rewards, plenty of intermittent ones, and very few very delayed ones. Delayed rewards, and even more so *impulse reinforcement*, delivered only at the single goal, are prohibitively difficult and slow in use, although they do eliminate the possibility for biasing the learning.

The overcome these problems progress estimators were introduced. Instead of giving a reward at a goal's 'end', intermittent estimates of progress are sent as an intermediate reward. Of course the danger of incorrect, inconsistent or internally biased estimations does exist, but with careful application of progress estimators it can speed up the learning algorithm considerably, as was already shown in Figure 5-4.

Luckily impulse reinforcements are very rare in situated domains, thus reducing the danger of prohibitively slowing the learning process down. They are rare because agents in situated domains usually have more than one single goal. Some of those multiple goals are maintained concurrently, while others are achieved sequentially. A robot participating in the foraging task, for example, maintains a continues low-level goal of collision avoidance, while at the same time keeping a minimal distance from other robots in order to minimize interference, may attempt to flock and may be heading home with a puck.

Most RL models require that the learning problem be presented as a search for a single goal optimal policy, so that it can be specified with only one global reward function. To overcome this problem we could keep track of the multiple goals in a situated domain framework by using separate state spaces and reinforcement functions for each goal and merge them later on into one state space and reward function.

We have seen in this section that there are two main problems when standard MDP models are applied to multi-agent domains (thus also operating systems) are considered:

1. The state space is prohibitively large, and
2. delayed reinforcement is insufficient for learning certain tasks, such as the foraging task in our robotics example of section 5.1.2. Unless appropriate progress estimators are used, such learning tasks will never converge.

For the modeling of such domains, MDP models can be obtained empirically for each system, but it remains difficult to prove the correctness of the model.

6. Conclusions

In the last section of this thesis, some conclusions will be drawn on the subjects presented in the previous sections and on the project as a whole. In the preliminary stage of the project itself - searching for useable material, such as books and reports, concerning agents - it became almost immediately clear that either the main software houses withhold their knowledge on agents from the public or the development of agents is still in its infancy.

If we look at some articles, such as **[Vizard]**, in which Microsoft, currently the largest software house worldwide, claims to have build agents into programs like Excel 5.0 and Word 6.0, it seems that the development of (intelligent) agents is indeed still far off. The so-called agents Microsoft claims to have implemented are not intelligent at all: They are nothing more than macro-ish written pieces of software. Excel, for example, can direct the user what to do next when creating a sheet. If the user has an empty sheet in front of him, the 'agent' states that formulas or text may be entered in column x and row y. Obviously this is hardly an intelligent task for an intelligent agent, let alone the notion that an 'agent' is completely obsolete in this case: a macro could have done the same thing with less fuss for the programmers.

The material that *was* found were merely some reports from colleagues throughout the world, mainly from other universities. The articles found in computer magazines were hardly worthwhile, because they kept the reader largely in the dark on how agents actually work. Most of these authors did not tell their stories beyond some vague stories what agents could do for the computer society as a whole. So, the main information had to come from the reports from other universities, all retrieved from internet. The trouble with this kind of information found on the Internet is that the information should be treated with the utmost care, because the information contained therein does not necessarily have to be correct.

It was immediately noticed that the subject of agents is very much multi-faceted. Not only the discipline of computer science is involved, but also disciplines like sociology, psychology (to study the behavioural patterns of agents) and biology. This has the disadvantage that there is not a general definition available for an agent because every discipline sees an agent differently. Hopefully the definition presented in this thesis (Definition 2-1) is satisfying to everyone in the different disciplines involved by combining the different interpretations into this single definition.

When agents will become available to the masses, it would certainly revolutionize the way in which we are going to use our computers. Computer systems, including network systems, may be configured by agents without the intervention from a human system operator. Tedious and repetitive tasks may be executed by agents rather than the computer user him/herself, alleviating the stress upon him/her.

The most important properties of an agent were found to be its *autonomy*, its *reactiveness*, its *pro-activeness* and its *successfulness.* For any formal definition of agent properties to have validity, a framework has to be defined, describing the environment in which the agent is going to reside and the task it has to accomplish in this environment.

The construction of agents must be executed according to a three phase plan:

1. creating the specifications,
2. creating an implementation from the specifications, and
3. creating a communication language with which the agent can communicate with its surroundings, including other peer agents.

Currently the communication languages are available in two kinds: conventional procedural languages and declarative languages, of which the declarative language clearly has the preference, because of its power to cooperate with learning processes unconditionally. Due to the 'habits die hard' syndrome we can expect, however, that the communication language to be used in the future contains a mixture of both procedural and declarative properties.

In order for the implementation of an agent system to be successful it has to comply to the following requirements:

1. The implementation of an environment which an agent can observe and affect.
2. The implementation of a flexible database system which can be effectively used by both the agent and the environment.
3. The implementation of some sort of support for a dialogue between user and agents, taking advantage of direct manipulation, natural language and all of the more conventional user interface components such as menus and buttons.

For OS agents the following requirements should be met:

1. The agent must be able to execute OS commands, recover from any unexpected failures and, when necessary, automatically attempt alternative ways to complete its task.

2. The synthesis of a sequence of OS commands, system calls etc. should be dynamic.
3. The agent must have the ability to adapt to differing conditions, resources and user tastes.

Furthermore, it is possible to create an agent system within an existing conventional operating system, like UNIX or Windows, by defining and implementing an extended interface with OS agent servers. A new agent operating system is naturally preferred, but not necessary. The disadvantages of agent operating systems created on top of a conventional operating system are that

1. not all information of the operating system's state is available, and that
2. repeated polling at the command level is sometimes necessary, resulting in a dramatic message overhead.

Agents will never be powerful if they do not have the capability to learn. Learning is a key feature to the success of agents. The learning process of an agent is classified into four different categories, differentiated by the kind of information learned:

1. learning declarative knowledge
2. learning control
3. learning behaviours
4. learning to select behaviours and actions

The most useful approach to learning was found to be *Reinforcement Learning*. Punishments and rewards are sent by the environment to an agent according to the correctness of the action the agent previously had executed. The goal of the learning system is to maximize positive reinforcement ('rewards') and minimize negative reinforcement ('punishments').

By appropriately applying heterogeneous functions and progress estimators, the learning process can be enhanced in such a way that the true policy the agent needs to learn converges. If these additions are not used, there is a chance that the agent will never learn the correct policy to accomplish its task, because of the dynamic, noisy and nondeterministic environment, which a multi-agent environment is.

The Markovian Decision Process (MDP) was shown to be the most used model for reinforcement learning. Because of a large state space in multi-agent domains the terms *state* and *action* have to be reformulated into *behaviour* and *conditions* respectively, thus reducing the 'state space' (i.e., 'behaviour space') considerably. The MDP model may be adjusted to conform to these changes, but this process is very difficult, because it has to be determined empirically.

Agents need some kind of basic behaviour set from which other behaviours can be created in order to start a learning process. These new behaviours may be created by either directly combining or switching between basic behaviours. Selecting the right basic behaviour set is the key to a successful policy learning. With a wrongly selected behaviour set, the correct policy may never be learned, or another irrelevant policy may be learned.

Looking back on the project as a whole, it can be concluded that agents do have the potential to create a new revolution in computer science, if used appropriately. However, we must never forget the danger agents may bring about. A serious attempt to prevent the misuse of agents has to be made in order for agents to become fully accepted. Agents do have the power to invade one's privacy, mainly because it acts almost in the same way a virus does. Because of the early stages in which the development of agents currently resides, this problem may be dealt with appropriately, soon enough.

A first step to a solution may be in the way Telescript is designed, in which an agent has to ask permission before it may act on a computer system or other peer agents that do not belong to its original creator's. However, a great disadvantage of Telescript is that every (computer) system that wants to participate in the agent capabilities must have a system-dependent interpreter built into it.

Protecting one's data from abusive agents should therefore have the highest priority in agent research, otherwise agents will never become accepted in the world called information technology.

Although the attempt has been made to give a complete-as-possible overview of the current standings in agent development in the available period of time, there is always something that has been overlooked, intentionally left out, or newly developed. This is, obviously, inherent to a project like this. There are a lot of reports floating about on the Internet on the subject of agents. Most reports, however, do require the user to have some basic knowledge on agents already. With this report as a general reference, all documents should be understandable to most of the interested people.

I part with some very eloquently spoken four short, but wise, words from the coach upon my asking him whether my project was now really finished: "Nothing is finished, ever", he replied. How true, indeed...

References

[BBC]	Broadcasted on 27 - 31 December 1994, BBC1 *BBC Christmas Lectures 1994 on the Human Brain* Lectures by Standford University, London: BBC Press, 1994.
[Bond]	Ed. by A. H. Bond & L. Gasser *Readings in Distributed Artificial Intelligence* San Mateo: Morgan Kaufman Publishers, 1988
[Coen]	Coen, M. H. *Sodabot: A Software Agent Environment and Construction System* Massachusetts Institute of Technology, June 1994. A.I. technical Report 1493.
[Etzioni]	Etzioni, O. & N. Lesh & R. Segal *Building softbots for UNIX* Software agents - Papers from the 1994 Spring Symposium, Technical Report SS-94-03, p. 9-16, AI Press.
[Genesereth]	Genesereth, M. R. & S. P. Ketchpel *Software Agents* Communications of the ACM, Vol. 37 (1994), No. 7, p. 48-53.
[Geurts]	Geurts, A.G.M. *Besturingsprogrammatuur voor Digitale Systemen I* (Dutch) University of Technology Eindhoven, December 1993 Reader nr. 5735
[Goodwin]	Goodwin, R. *Formalizing Properties of Agents* School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, May 1993. Technical Report CMU-CS-93-159.

[Levy] Levy, H. M. & R.B. Segal & O. Etzioni & C. A. Thekkath
OS Agents: Using AI Techniques in the Operating Environment
Department of Computer Science and Engineering,
University of Washington, Seattle, USA, April 1993
Technical Report 93-04-04.

[Mataric1] Mataric, M. J.
A Distributed Model for Mobile Robot Environment-Learning and Navigation
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 1990
AI Technical Report 1228

[Mataric2] Mataric, M. J.
interaction and Intelligent Behaviour
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology, May 1994
AI Technical Report 1495

[Shirai] Shirai, Y. & J. Tsujii
Artificial Intelligence: concepts, techniques and applications
Bath: Pitman Press, 1982

[Tannenbaum] Tannenbaum, A. S.
Operating Systems Concepts
London: Prentice Hall Int., 1987

[Vizard] Vizard, M.
Excel upgrade adds agents
Computerworld, Vol. 27 (1993), No. 41, p. 4

[Wayner] Wayner, P.
Agents away
Byte, Vol. 19 (1994), No. 5, p. 113-118

[Wood] Wood, A.
Desktop Agents
School of Computer Science,
University of Birmingham, Birmingham, UK, April 1993.

[Wooldridge] Wooldridge, M. & N. R. Jennings
Intelligent Agents: Theory and Practice
Department of Computing,
Manchester Metropolitan University, Manchester, UK and
Department of Electronic Engineering,
Queen Mary & Westfield College, London, UK, 1994.