Multi-agent Stratego

Caspar Treijtel

Delft University of Technology

Faculty of ITS
Department of Mediamatica
Knowledge Based Systems
Delft University of Technology
October 12, 2000

Abstract

The field of multi-agent systems is an active area of research. One of the possible applications of a multi-agent system is the use of distributed techniques for problem solving. In stead of approaching the problem from a central point of view, a multi-agent system can impose a new mode of reasoning by breaking the problem down in a totally different way.

In this report we investigate a distributed approach to playing Stratego. The individual pieces of the Stratego army are represented by computational agents that each have their own field of perception, evaluation and behavior.

A first prototype of a framework has been built that consists of a simulation environment for the agents and an implementation of the agents evaluation function. The agents have a rule engine that generates behavior that is a resultant of the environment in which they live. This report presents a result of playing the game using agents against a human player.

Contents

Acknowledgements

This report describes the master's thesis that I have done in the past nine months. A master's thesis is the final stage in the doctorate program to become an engineer. The project has been done in the group Knowledge Based Systems, headed by prof.dr. H. Koppelaar. A lot of people have helped me in the process.

Specifically I would like to thank drs.dr. L.J.M. Rothkrantz for his guidance during the work. Although he is a busy man, he can always find ways to be available when needed. Also special thanks go to my colleague Lein Ton for helping me a lot with the work but also for being a great colleague to work with. Furthermore I would like to thank Natasja Roodbergen, she has been a great support by participating in the process of the investigation. She has played against the prototype that has been developed and was open to discussions about the investigation. Finally I would like to thank the following people who also were very supportive. These are Michiel van der Weiden and Martijn Krenn for their interests and commitment to the project, David Lindeijer for sharing his expert knowledge on software engineering, Boi Sletterink for helping me with all kinds of technical problems and offering solutions, Gertjan Eggenkamp for reading the report and providing me with comments, Andriy Levytskyy for helping me out with printing the report, and last but not least my family for providing me with all the support I needed.

Chapter 1

Introduction

This report describes a first attempt to play the Stratego game with multiple agents. The Stratego game is a board game where two players battle each other with their armies of pieces. The object of the game is to capture the enemy flag, by moving pieces towards the enemy and try to capture enemy pieces. An interesting property of the game is that the information the players have is incomplete, because the identity of the opponent's pieces is concealed until exposed by battles between pieces.

Our motivations for using the multiple agent approach are as follows. When we consider a human society from a central point of view we see that it is a very complex system. A possible attempt to understand the complex behavior of a human society may be considering it as a system that is made up of individuals that each have their own characteristics, behavior patterns and interactions with each other. It it the sum of all the local actions and interactions that constitutes the overall behavior of the society. In other words, we can understand this complex system by considering it in a distributed fashion.

We suspect that the distributed approach of taking a local point of view in stead of a central point of view, can not only be used to understand complex systems but may also be used to solve complex problems. This investigation is an attempt to support this hypothesis by considering the Stratego game. Specifically we want to investigate whether a distributed way of playing this game will provide us with a means to break down the complexity of playing it.

We believe that the Stratego game can serve as an example for supporting our hypothesis, because of the characteristics of the game. The game brings about a high complexity when seen from a central point of view. This is a direct consequence of the fact that during the greater part of the game, both players have incomplete knowledge of the board situation. Our approach will be an attempt to handle the game's complexity by using the distributed decision making at the level of the Stratego pieces.

The complexity of the game can best be recognized by attempting to design a computer algorithm that approaches the game the same way human players do. When a human player plays Stratego, the human takes a central point of view of the game. Up to a certain level every human can learn to play the game. We assume that the human brain can somehow handle the complexity of the game by making up tactics and strategies, form hypothesis and go after their intuition. Depending on the level of play, some of the abilities of the human players may be implicit knowledge. Only experienced players will have formulated some specific rules that can be applied to play the game. But we presume that a part of the knowledge necessary to play the game comes natural to the human brain and may be difficult to specify.

The structure of this report is as follows. At first Chapter 2 discusses some theories with respect to agents and multi-agent systems, the main tools of the investigation. Also an overview will be given of similar distributed approaches to problem solving. Following, in Chapter 3 the report gives an overview of the Stratego game, its rules, how to play it and finally some game theory that will provide us with some insights into the complexity of the game. Chapter 4 builds further on the distributed approach by discussing a design for an agent army playing Stratego. Chapter 5 presents a number of ideas on the agents' internals by combining both knowledge about agents and about the game. We have implemented a prototype for playing the game which will be outlined in Chapter 6. In Chapter 7 we will discuss the results from the investigation. Finally, Chapter 8 contains some conclusions and recommendations.

Chapter 2

Multi-agent systems

This chapter provides some background material of the investigation. The relatively new field of research called *multi-agent systems* is described. But first an introduction is given to the concept of an intelligent agent.

2.1 Intelligent agents

In the computer science literature a lot of papers, reports and books contain the word “agent”. Apparently it has become a very popular word of describing systems and software. But there seem to be a lot of disagreements as to what the characteristics of these systems and software are that constitute an agent. In this Section we will review some of the main interpretations of the agent-concept.

Before we start comparing definitions and interpretations we need to have an understanding of what types of agents we will discuss. The Collins English dictionary gives a rather broad definition of the word agent:

```
Agent(n.) -
  1. A person who acts on behalf of another person, group,
     business, government etc.
  2. A person or thing that acts or has the power to act.
  3. A substance or organism that exterts some force or effect.
  4. The means by which something occurs or is achieved.
  5. A person representing a business concern
     (i.e. a travelling salesman).
```

According to this characterization virtually any system can be classified as an agent. Thus our agent definition needs to be more specific, to characterize the kind of agents in which we are interested. Our interpretation will be in the context of agents that are used in the computer science literature. These have two main characteristics, a level of *intelligence* and *autonomy*.

Agents can be classified in robotic agents and software agents. Although sometimes these types of agents are taken together in the discussions, some researchers in the field (e.g. Steels 1995) only focus on building robotic agents, whereas others only speak of software agents (Maes 1995).

According to Russel & Norvig (1995), an agent is anything that can be viewed as *perceiving* its environment through *sensors* and *acting* upon that environment through *effectors*. See Figure 2.1 for a schematic view of an agent interacting with its environment.

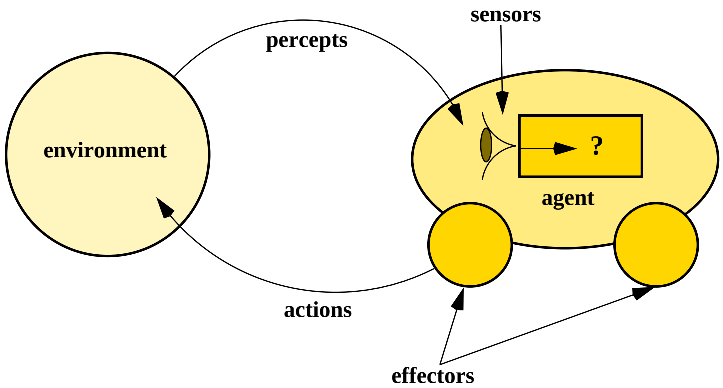


Figure 2.1: Agents interact with environments through sensors and effectors

Viewed this way the agent is a mapping from percepts to actions, that can influence the environment in which the agents live. But also this definition may include a rather broad range of software because it heavily depends on what is exactly meant with the environment in which the agent lives. According to this definition every program would be an agent if the notions of environment, sensing and acting are not restricted (Franklin & Graesser 1996).

In Russel & Norvig, the intelligent agent's properties described are rationality and autonomy. For an agent to act rational it should do the 'right' thing, where the right thing is to be interpreted as the thing that would make it most successful. In order to know what to do, the agent should have its own performance measure to be able to maximize its performance. A performance measure that is sufficient in making the agent rational depends both on the evidence provided by the agent's perception and the (built-in) knowledge the agent has. Note that performing actions in order to obtain useful information is an important part of rationality. The agent has to try to make the percept sequence as rich as possible to increase its understanding of the environment it is in.

Maes defines autonomous agents to be "computational systems that inhabit some complex, dynamic environment, sense and act autonomously in this environment, and by doing so realize a set of goals or tasks for which they are designed" (Maes 1995). Maes distinguishes between three types of agents. Firstly we have the autonomous robot that inhabits a real physical environment. Alternatively there are 2D or 3D animated agents that inhabit simulated physical environments. Finally, so-called knowbots, software agents or interface agents are disembodied entities that inhabit the digital world of computers and computer networks (Maes 1994).

We would like for the agent to be as autonomous as possible, but what exactly does autonomy mean for agents? Russel & Norvig state that a system is autonomous to the extent that its behavior is determined by its own experience. Therefore a "complete autonomous" system would be able to survive even without any initial knowledge, but requiring this of a system would be far too stringent. It is reasonable to provide an artificial intelligent agent with some initial knowledge, as well as the ability to learn.

According to Steels (1995), the term autonomy essentially means "be relatively independent of". For an agent to be independent this would mean that it would be able to cope with unforeseen changes in its environment without the need for help from its designer. A clear distinction can therefore be made between the concepts of autonomy and automaticity.

An automatic system is something whose behavior you can fully predict as soon as you know its internal basis of decision making. An autonomous system on the other hand is a system "which makes up its own mind". Autonomous systems are *self-controlling* as opposed to being under the control of an outside agent. This means that the agent must have relevant self-knowledge and motivations to be able to control itself (McFarland 1994).

The agent as defined by Steels has three essential properties. First of all the agent is a system. This means a set of elements having a particular relation among themselves and with the environment. Second, an agent performs a particular function for another agent or system. A third essential property is that the agent is capable to maintain itself. In other words it can remain viable in the environment in which it lives. Maintaining its own viability is an important part of the agent's "self-interest". Steels states that some of the software agents reported on in the literature do not have any self-interest whatsoever. An example of an attempt to construct autonomous robotics agents can be found in (Steels 1997). Here an investigation is described that should make a robotic agent remain viable in an ecosystem in which there are increasing pressures.

Some researchers have tried to define what it is that characterizes intelligence. In (Keijzer 1999), human intelligence is not only seen as the cognitive ability to reason but also the human talents of dealing with the environment in which we live. Keijzer states that intelligence can only be understood as an interaction between processes in the brain, the physical ability to act and environmental conditions. In other words, intelligence is *situated* and *embodied*. Brooks' robots, that are built using the *subsumption architecture* (see Section 2.1.1) are examples of an interactive approach. Here the causality of intelligence is viewed as the interaction between the brain (or possibly the complete nervous system), the body with all kinds of sensors / actuators and the environment. The brain depends on its body and environment, to which it delegates much of its capacities to solve problems.

2.1.1 Subsumption architecture

Since the middle of the eighties a new approach in the field of robotics began to be accepted called *behavior based robotics*. The new approach to building robots was very different from the traditional way of modeling robots. The traditional approach is based upon a strict functional decomposition of modules whereas behavior based modules are designed with the notion of directly forming a part of the overall behavior of the robot. The behavior modules may have their own perceptual, modeling and planning requirements.

The subsumption architecture designed by Brooks (1986) is a clear example of behavior based architectures. In this architecture the modules represent task achieving behaviors in stead of information processing modules in the traditional approach, see Figure 2.2.

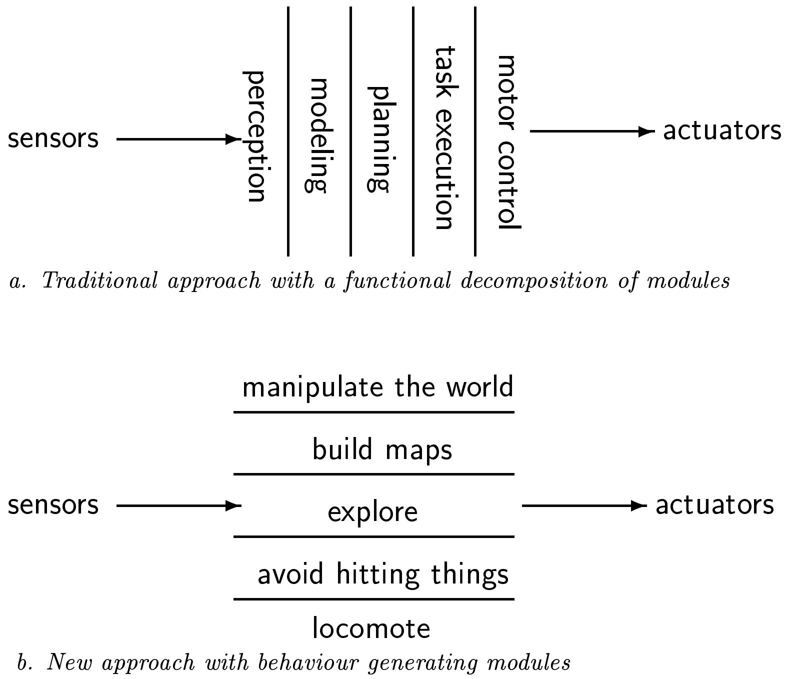


Figure 2.2: The contrast between the (a) traditional approach and the (b) behavior based approach

The architecture consists of a number of layers that produce behavior of their own, possibly using other layers in the process. A great advantage of this type of architecture is that the system can be extended easily by adding layers, some of which may even have conflicting behaviors. It is the sum of the total set of behaviors that constitute the overall behavior of the system.

The operation of the system is reminiscent of Marvin Minsky's approach described in his book *The society of mind*. He describes the operation of the brain in terms of an ensemble of *agencies*, each responsible for a simple functionality. The agencies communicate among themselves to reach a "decision", and the total (emergent) effect is essentially the operational mind (Minsky 1985).

2.2 Multiple cooperative agents

So what about multi-agent systems? A multi-agent system is a system in which multiple agents are working together, possibly in a distributed context. There are a number of reasons to distribute artificial intelligence. A lot depends on the nature of the problems that are to be solved, or the topics that are to be investigated. Because of the possible use in multiple domains, the essentials of multi-agent systems can best be understood by considering the areas of application. What are they good for?

Ferber (1999) has made a classification of the possible areas of application as shown in Figure 2.3. In the following paragraphs an outline of each of these five categories of applications is given.

Problem solving. This concerns the use of software agents, brought into action to accomplish tasks that are of use to humans. These software agents are computing agents and have no real physical structure. Ferber discriminates between 'distributed solving of problems' and 'solving of distributed problems'. The first concerns a problem solving where the expertise to solve the problem is distributed among agents, i.e. the agent system comprises of a number of agent-specialists. The latter deals with problems that are themselves distributed. Typically, in these applications multiple agents of identical skills are used.

Distributed techniques for problem solving are sometimes used for problems where the domain is not distributed nor is the expertise. Yet sometimes a multi-agent system can dictate a different point of view of the problem, which might make it possible to break down the problem to an easier way to solve.

Multi-agent simulation. Theoretical models of the surrounding world are sometimes used in simulations to explain or forecast natural fenomena. In contrast with the traditional analytical models that are used, the multi-agent approach to modeling is conceptually different. In stead of creating the model from a central viewpoint, individuals are directly represented along with their behavior and interactions. This way the modeler expects to see *emergent* behavior patterns arising during simulations.

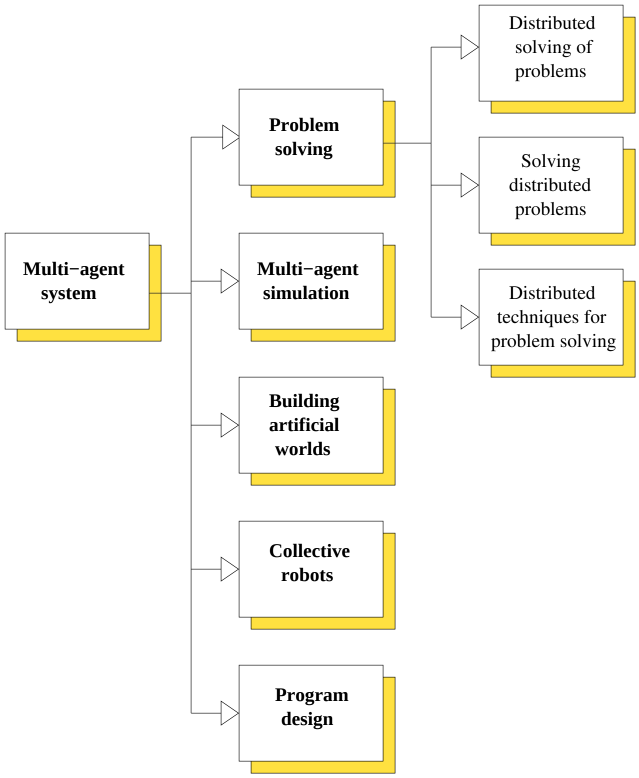


Figure 2.3: A classification of the various types of application for multi-agent systems

Building artificial worlds. A big part of research efforts on agents is the construction of synthetic worlds. These are worlds made up by the designer for investigating the use of agents. In this case the agents are purely computational agents. These worlds are specifically constructed in order to investigate agent-concepts, interactions amongst agents or to gain an understanding of the influence of behavior on the regulation of a society.

Collective robotics. An active area of research is the construction of multiple physical robots situated in a shared environment. Research interests can be to build autonomous robots or a study of interaction and cooperation between robots.

Program design. According to Ferber multi-agent systems can be used for *kenetic* program design. The ambition of this concept of designing is to be able to create distributed systems and software that operate with great flexibility and ability to adapt to the environment.

Chapter 3

Playing Stratego

This chapter covers the basics of playing the Stratego game. First some general things about the game are described, the layout of the board, number of pieces and rules of the game. After this general introduction a somewhat more elaborate description of the game follows by using some example scenarios. Of course we are also interested in how a computer would play Stratego. The last Section gives an overview of some game theories that can be used by computer algorithms.

3.1 The game

The game of Stratego was designed by Milton Bradley company in 1961. It is a military strategy game for two players fighting each other on a board of squares. Each player has an army of pieces that each represent a military man or a military object (a bomb or a flag). Every piece has a certain rank that is used to determine the outcome of battles between pieces.

The goal of the game is to capture the enemy's flag. At the start of the game all of the ranks of enemy pieces are unknown. The players have to come up with a strategy to find out the positions and identities of enemy pieces, including the flag.

Upon starting the game, the players position their armies. They are completely free to position their pieces within their side on the board, but every square can contain only one piece. The first thing that both players have to do is think carefully about the initial strategic positions of their pieces. The initial positioning is very a important part of the strategics of the game. The strength of the army is to a considerable extent result of the initial positioning.

The simulation of the Stratego game described in this report is a slightly modified version of the original game. The board consists of six rows of eight columns of squares on which two armies begin playing with sixteen pieces each. See Figure 3.1 for a schematic view of the board and pieces.

Figure 3.1: A schematic view of the Stratego board

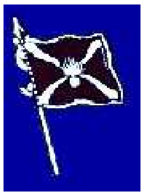
The figure shows a board situation where both players have positioned their armies and are ready to move.

3.1.1 Rules of the game

As mentioned before the identity of any enemy piece is unknown at the start of the battle. When a piece attacks an enemy piece, the ranks of both pieces determine the outcome of the battle. The higher rank wins, which means that the looser is removed from the board. When the pieces share the same ranks both are removed. However there are two exceptions. Any piece attacking a bomb looses except for the miner who can dismantle bombs. The second exception concerns the spy. This piece can attack and defeat the general, but when the general attacks the spy the general wins.

A movable piece can move only one square at a time, either forward, backward, to the left or to the right. It attacks an enemy piece by trying to move to an already occupied square on the board. Two kinds of non-movable pieces exist which are bombs and the flag.

3.1.2 Description of the pieces

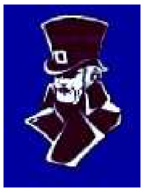


The flag. This is the most important piece of the Stratego army, because once attacked the game is lost. Every enemy piece can beat the flag. It is the first non-movable piece and it has rank 0.

A sensible position for the flag will be somewhere on the first row of the board. One can choose to put the flag in the corners or behind the lakes. However an alternative may be to place the flag directly behind the lake on the second row; as this is an unusual position, it may be difficult for the enemy to discover it.

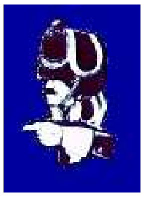
The bomb. The bomb is lethal for every enemy piece except for the miner, who can dismantle bombs. The bomb is the second non-movable piece and has rank 1.

The bombs are very suitable for protecting important pieces in the army, such as the flag. They should therefore be positioned near those important pieces, but care has to be taken not to position them somewhere where they can be an obstacle for friendly pieces.



The spy. The spy is a piece with a low rank, rank 2. Every enemy piece attacking it wins, or when attacked by an enemy spy a draw occurs. However, the spy has an important quality. Upon attacking the enemy general the spy wins.

Therefore the spy is a poor defender, but can be a vicious attacker when used at the right moment. A good starting position may be somewhere protected by strong pieces. This increases the chances of survival throughout the game until needed in the end-game. For the spy it is essential to conceal its rank as long as possible.

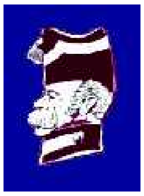


The scout The scout is piece with rank 3, which is mostly used to reveal enemy ranks at the cost of its own life. In the game there four scouts for this purpose. Note that in the original Stratego game the scout can move multiple squares in one move. We have chosen not to follow this rule because we do not want a move to reveal the rank.

Most profitable positions may at the front line. But some of them may want to be used later in the game and thus positioned somewhere initially 'safe'. The most optimal way for a scout to function is to discover important pieces of the enemy army, and that may only be possible later in the game.

The miner. The miner is the only piece able to dismantle enemy bombs. It has rank 4, which means it can also capture enemy scouts, but all other pieces with higher rank are lethal. Two miners are available at the start of the game.

It is important to position the miners somewhere where they can move easily. When for example an enemy bomb has been discovered, one of the miners has to come into action. Preferably with an easy way to get there. However, since there are only two of them and their rank is quite low, they also need a bit of protection.



The sergeant. The sergeant typically belongs to the middle-ranks of the Stratego army. It has rank 5, all lower ranks—except for the bomb of course—can be captured and all higher ranks are lethal. The initial Stratego army has two sergeants.

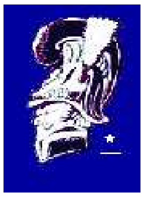
The sergeants are probably at their best during the mid-game. They can capture enemy scouts and miners and of course the enemy spy. The sergeant does not have a very important role in the army and therefore its concealing its rank as long as possible is not really necessary.

The lieutenant. With rank 6, the lieutenant is the first 'higher rank' in the army. The Stratego armies each have one lieutenant. Care should therefore be taken not to reveal its rank when unnecessary. However a role in the army may be to capture as much as enemy pieces as possible, because of its rank which is between low ranked pieces (that can be captured) and higher ranks (which are more important in the end game).

A good starting position may be somewhere at the front line, when the lieutenant is to be offensive, or more at the back where the lieutenant can play a more defensive role.

The captain. The captain has rank 7. With this rather strong rank the captain might want to conceal its rank for a while. It will probably come in handy in the end-game, if the captain is to be used for defenses. Also the captain can take offensive actions, however it should beware of the enemy major and general, and therefore await his chances.

A good position may be somewhere near important pieces for a defense. For example to defend the spy and miners. Later on in the game it can move towards the enemy if necessary.



The major. The major has rank 8, which makes it the second most powerful piece of the Stratego army. Its only threat is the enemy general. Therefore it may be a valuable attacker when the enemy general has been exposed or better yet, captured.

The major's starting position is preferably somewhere in the back, where it can defend lower ranks and wait for a possible attack when the risks can be estimated well. The major should think twice before attacking pieces that haven't moved yet, they may be bombs.

The general. The general is the most powerful piece in the Stratego army, with rank 9. It can only be captured by the enemy general (a draw), by the enemy bombs upon attacking them and by the spy attacking the general. It should never attack pieces that haven't moved yet, because they may be bombs. Preferably the general's rank is to be concealed as long as possible. Once the rank has been revealed, the general has to beware the enemy spy luring for an attack.

The position of the general should be somewhere in the back, close to the flag. The general will probably not be used until the end-game and therefore it may remain quiet for a while.

3.2 Analysis of the game

Unfortunately there have not been any studies about Stratego. Still we want to learn more about the game's complexity. As a human we have some ideas on how to play the game Stratego, but to extract or to make the human knowledge explicit is a difficult task. In this section we will discuss thoughts on the game's strategic, tactical and operational aspects. Following, we will give some example scenarios to illustrate the game in Section 3.2.2.

3.2.1 General considerations

We can make a distinction between different stages in the game. These are the *opening*, *mid game* and *end game*.

Opening

The influence of the initial positions of the pieces is to a considerable extent determinant for the game's outcome. The players' strategies will be closely related to the initial positions. Upon starting the game both players have to develop their pieces, in order to create an optimal space to move. After developing the pieces—and in the process having had some battles—the game gradually progresses to the mid game stage.

Mid game

During the mid game stage, some space has been created by the battles between pieces that have been held. This means that both players can begin to think about strategies, as the identity of some of the enemy pieces will have been discovered.

End game

When most of the pieces of the armies have been captured, the game will go into the final stage. The end game is typically less complex because much of the incomplete information has been discovered and there are less possible scenarios to be considered. Both players will probably be able to form strong hypotheses about important enemy positions, including the flag positions.

3.2.2 Example scenarios

We hope to gain more insight in the Stratego game by analyzing some example scenarios. In the first scenario the importance of the spy will be emphasized. Here the blue general threatens the red flag, but the flag is safe because the spy is covers the square that the blue general needs to cross (see Figure 3.2).

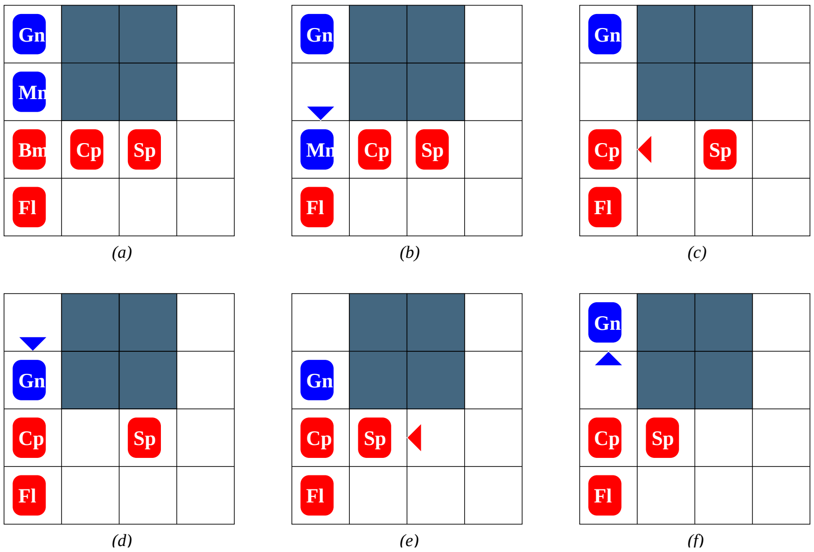


Figure 3.2: Value of the spy

Description of the scenario in Figure 3.2:

(a) Starting position.

(b) The blue miner attacks the red bomb, red's defense is under pressure.

(c) The red captain has to capture the miner in order to protect the flag.

(d) The blue general moves forward to threaten the captain.

(e) The red spy moves to the left, here it is a potential danger for the blue general.

(f) The blue general has to retreat, to avoid being killed by the spy, who is covering the red captain.

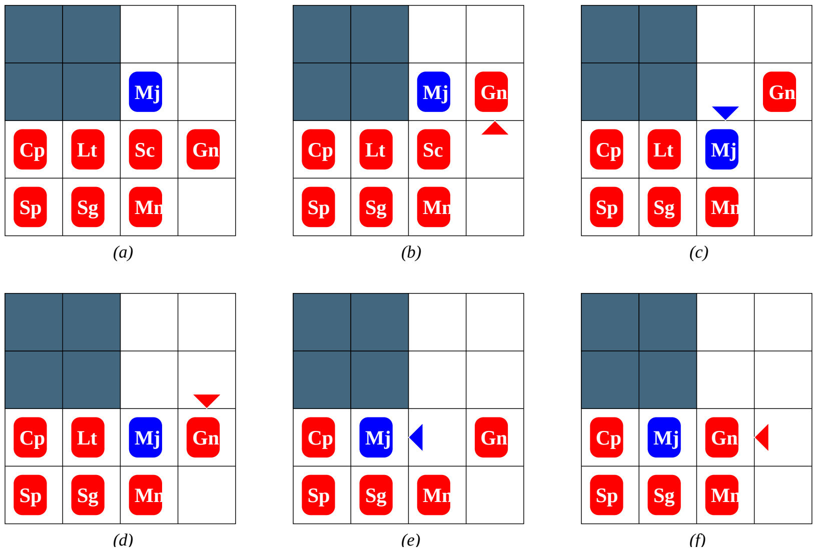


Figure 3.3: Potential threats

In the next scenario the red player makes a poor move by putting the general forward. The red army's defense has been taken down completely, and the blue major gratefully takes advantage of it.
Description of the scenario in Figure 3.2:

(a) The blue major has taken a dangerous position for red.

(b) The red general moves forward, in an attempt to threaten the blue major.

(c) The blue major attacks.

(d) The red general wants to defend, he moves backward.

(e) The blue major strikes again.

(f) The red general chases the blue major.

3.3 Game theory

In the past a lot of research has been done on playing games. This Section discusses some general theories on this subject, to understand more about game playing. Specifically we are interested in theories on two-person games. Examples of studied games are Chess and Go, which are perfect-information games which means that both players have all information on the board-situation. However Stratego is typically a game with *incomplete* information, but this discussion may serve as an illustration of the complexity of it.

A game can be represented by a *game tree*. The tree's nodes represent positions, which are connected by branches that represent moves. The root node of the tree is the starting position of the game and the leaves of the tree are terminal positions. Each level in the tree is called a "ply" introduced by Arthur Samuel in his paper on Machine Learning (Samuel 1959).

The theories on game playing all try to search the game tree to find the best possible move that can be played on each move. Therefore a subset of the game tree is visited during 'thinking,' which is called the *search tree*. Ideally the search tree contains a path from our move to a terminal position where we have won. The difficulty is to find this path, as the game tree grows exponentially with increasing levels. For a uniform game tree that has a branching factor of W, the level at a depth of D ply from the root node contains W^D positions to be evaluated. As an example, Chess has an average branching factor of 35 (Marsland 1986). Because of the large number of positions it is not possible to search until a mate position is encountered, but instead the tree is searched until a maximum depth has been reached, which is called the *horizon*.

Typically not all the leaves of the search tree are *quiescent* positions. Quiescent positions are positions that can be assessed accurately because of the lack of moves that can change a position drastically, like complex captures. Some leaves of the tree will have to be examined deeper for a decent evaluation. The *horizon effect* is a problem with search trees where some positions are not quiescent because of inevitable losses that are postponed by the insertion of delaying moves. A quiescence search can help to see beyond the horizon but still all searches exhibit this problem.

The minimax principle

A standard type of search tree is called the *minimax* search tree. Because the game tree cannot be searched exhaustively, the tree is searched up to a certain depth. In a minimax tree the leaves of the tree are assigned values that are evaluations of the corresponding positions. These values are propagated up the tree according to the *minimax principle*. This principle states that at levels where we (max) are to move, the highest value of the child nodes is propagated. At levels where the opponent (min) is to move the lowest value of the child nodes is propagated.

This way the values of the internal nodes are determined in a bottom-up fashion until the root node has been reached. The resulting root node's value is the best value we can expect to achieve assuming best play of both sides. The path from the root node to the best value is called the *main variation*.

Alpha-Beta pruning

The search algorithm described above can be made more efficient. To avoid having the need to visit *all* possible nodes until a certain depth has been reached, the alpha-beta algorithm can be used. Using this algorithm a part of the minimax tree can be *pruned*, using a lower (α) and upper (β) bound on the expected value of the tree. α is the minimal value that max is guaranteed to achieve and β is the maximal value that max can hope to achieve. From min's point of view β is the worst value that min is guaranteed to achieve. So the actual minimax value lies inside the α–β interval. This implies that values of positions that are known to be outside the α–β interval can be safely ignored. We can say that they are not on the main variation, without knowing the exact value of the position.

Alpha-Beta Enhancements

The α–β algorithm can be enhanced in a number of ways. Theoretical advances have been made by Judea Pearl, who designed the Scout algorithm (Pearl 1980). This algorithm is comparable to the *minimal window search techniques* that have been developed in the late seventies. The basic idea behind these methods is that it is cheaper to prove a subtree inferior, than to determine its exact value. Much depends on the type of game we are playing because every game has its own type of tree structure. For example Chess can give a rather 'bushy' search tree with which significant improvements can be made by using minimal window techniques (Marsland 1983). But for random game trees it is known that refinements are asymptotically equivalent to the simpler α–β algorithm (Marsland 1986).

The order in which the search tree is constructed is very important, because it determines the number of cut-offs that can be made. Therefore a number of techniques have been proposed that are called *move ordering mechanisms*. For example, Gillogly states that "since the refutation of a bad move is often a capture, all captures are considered first in the tree, starting with the highest valued piece captured" (Gillogly 1972). A more recent technique has been introduced by Schaeffer, called the *history heuristic*. This technique "maintains a history for every legal move seen in the search tree. For each move, a record of the move's ability to cause a refutation is kept, regardless of the line of play" (Schaeffer 1983). A history table is kept and each time the available moves are re-ordered so that the most successful ones are tried first.

Another example of an improvement technique is called *progressive and iterative deepening*. The idea of this technique is to build the search tree using iterations. At each iteration the tree is expanded with one ply, retracing the best path from the previous iteration. This technique has the following advantages:

- At every moment during search some best move so far will have been found, therefore the algorithm can safely search until time is up.
- At each iteration results of previous iterations can be used, this will make the algorithm more efficient because a preliminary order of search can be applied.

Also the use of *transposition tables* makes search more efficient. Previous search results can be stored in a large direct-access table. When a certain position is reached again, the corresponding table entry serves three purposes (Marsland 1986). Firstly, it may be possible to use the table score to narrow the (α–β) window bounds. Secondly, the best move that was found before can be tried immediately (re-ordering). Finally, the primary purpose of the table is to enable recognition of move transpositions that have lead to a position (subtree) that has already been completely examined. In this case there is no need to search again.

Chapter 4

Design

4.1 Requirements

This section discusses the requirements of our Stratego agent. These requirements will be discussed in terms of the agent's environment (Section 4.1.1), its desired functionality (Section 4.1.2), the communication scheme (Section 4.1.3) and the way decisions are to be made (Section 4.1.4).

4.1.1 The agent's environment

At the basis of the design of our Stratego agent are the ideas discussed in chapter 2. The agent is considered as a relation between percepts and actions, which influence the external world. See Figure 4.1 for a schematic diagram of the agent.

This diagram conceptually resembles the agent diagram in chapter 2. The agent's sensors pick up visual percepts from neighboring squares and sonic percepts from fellow agents in the army. Two kinds of effectors are available, a pair of virtual legs to move along the board and a message passing mechanism to be able to communicate with fellow agents. Note however that the virtual legs can only cause an action when the corresponding agent has the right to move. Therefore we consider a proposal to move to a certain square as a sonic effector, i.e. sharing its intention with its fellow agents.

Our agent lives in a world populated by two kinds of Stratego pieces, fellow agents of its own army and enemy pieces. These enemies can only be considered as possibly dangerous moving objects, with which no interaction other than battles can be held. From the point of view of the agent the enemy pieces can be either lethal for the agent or killed by the agent.

With the fellow agents in the agent's world a more delicate interaction can be achieved. In the Stratego game friends do not battle each other and therefore the interaction comprises of an information exchange about the world in which they live. Ideally the interaction among fellow agents adds an important added value to the distributed decision making of the agent

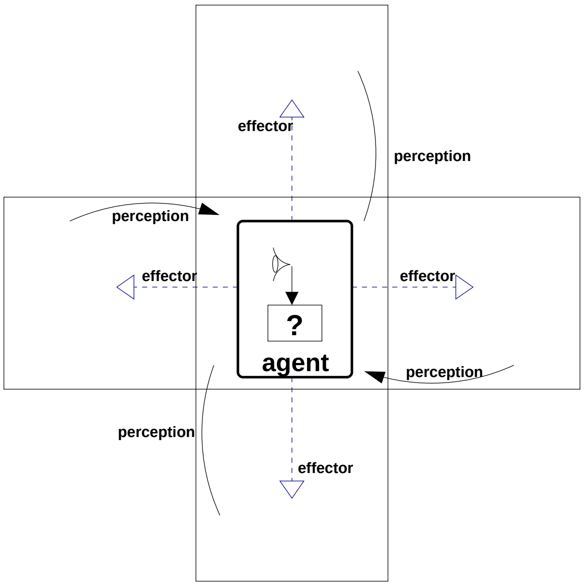


Figure 4.1: The environment in which the agent playing Stratego lives

army. This communication scheme we propose will be discussed further in Section 4.1.3.

4.1.2 The agent's functionality

In designing the agents we want to make use of the fact that each piece in the Stratego army has a certain dedicated role. These roles originate from their specific ranks and the rules of the Stratego game. For some pieces this goal can be quite explicit. For example, the miner's primary goal is to dismantle enemy bombs. All pieces have secondary goals as well however, of which possibly the most important one is to stay alive. The scout however forms a clear exception to this general goal, as the scout's primary goal might be to discover the identity of enemy pieces—possibly at the cost of its own life.

We propose to define some degrees of freedom in our model of the agent that will allow us to experiment with different types of agents in the Stratego army. Specifically we define for each agent:

- The agent's perception range. Depending on the agent's role in the army the perception will be a diamond of range one to five, or an $n \times n$ square of fields. Important pieces will have wider perceptions.

- The agent's 'reactive' behavior. For every agent we define four elementary behaviors that are executed following a reaction in various situations. These behaviors are:

1. *Attack*—attack an enemy piece that is situated within attacking range of the agent (at a distance of one square).
2. *Flee*—move away from an enemy piece that is situated within attacking range of the agent. Ideally the agent moves to a square that is in the opposite direction of the enemy piece.
3. *Wander*—random walk.
4. *Stay*—do nothing.

- The agent's 'cognitive' abilities, like for example evaluate situation, compute optimal next move, form hypotheses, make plans.

Emotional and motivational states

One of the cognitive abilities of human beings is to experience, show and assess emotions. A lot of research is being done to investigate the nature of emotions. Some researchers try to model them as part of computational systems (e.g. Cañamero 1997, McCauley & Franklin 1998). Cañamero reports on an experiment to implement an autonomous creature that shows various learning and problem-solving capabilities. The implementation uses motivations and emotions as a basis for intelligence of the creature.

According to McCauley & Franklin emotions give us humans the ability to make an almost immediate assessment of situations. They believe that autonomous agents may produce 'rational' behavior when they are being enriched by the ability of experiencing emotions.

Using an architecture as designed by McCauley & Franklin or Cañamero is however beyond the scope of this investigation. But we do want to acknowledge different emotional and motivational states that the agent undergoes, in order to have more degrees of freedom in the input-output relation of the agent.

In our design emotion is modeled as follows. Emotions are related to parameter settings regarding the agent's perception and behavior. For example, if an agent gets upset, afraid or stressed we shrink his field of perception (tunnel view). And if the agent is angry we increase the possibility to attack.

4.1.3 Communication among agents

We propose to apply two levels of communication. One is communication by means of a blackboard that can be written to and read from by every agent. The blackboard is a container of all information of the board situation that is available. This way all agents can rely on the fact that their field of perception is in accordance with the current board-situation. The information on the blackboard strictly contains information about the board status.

Additionally the agents can use an asynchronous message passing structure. Agents can send and receive messages to each other containing information about the Stratego battlefield. The communication structure allows sending messages to all other agents, sending messages to agents of a certain rank or sending messages to specific agents.

The contents of messages can either be known facts, hypotheses or requests. Examples of messages can be notifying all agents of capturing an enemy piece, communicating the known location of an enemy bomb, etc. The flag may send a 'request for help'-message to all others when it perceives an enemy approaching.

4.1.4 Decision making

Because of the fact that only one piece can move at a time, a mechanism has to be found that decides which agent is allowed to move. We propose three possible ways of implementing this mechanism:

- Based on a random pick, where at each move a random choice will be made of the agent that is allowed to move. It is possible however that the chosen agent does not want to move. In this case another agent will be randomly chosen. When none of the agents want to move, one of them will be forced to move.

- Based on scores, where each agent evaluates its current situation and assigns scores to preferences of moving. A higher score will indicate a stronger desire to move and the agent with the highest score will be allowed to move.

- Based on hierarchy in the Stratego army. Because we are speaking of an army, a way of organizing it may be a strict hierarchy based on ranks. In this scheme the highest rank in the army may decide which agent will be allowed to move.

4.2 An architecture the Stratego agent

For our Stratego agent we propose to define a three layered architecture, with the layers *sensors*, *evaluation* and *effectors* as indicated in Figure 4.2. These layers relate sensor inputs to actuator outputs.

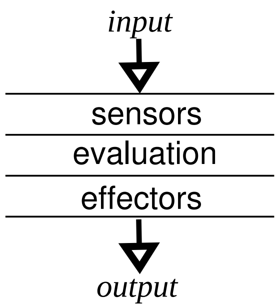


Figure 4.2: Three layered model for the Stratego agent

The actual relation between percepts and actions takes place in the evaluation layer. There are various possibilities for filling in the evaluation layer. In this report two different strategies will be discussed. These are the traditional approach to model cognitive systems and a model that is based on the behavior based approach designed by Brooks (see Chapter 2 for a discussion of Brooks' work).

4.2.1 The traditional approach

The traditional approaches to models of cognitive systems is based upon a strict functional decomposition of modules. These approaches result in so called *sense-model-plan-act* frameworks. The cognitive system contains a number of modules that are built on top of each other, each performing a dedicated function as a part of the system.

One characteristic of these types of frameworks is that every module has a specific function that uses input from the module before it. See Figure 2.2a for a schematic view of this type of framework. In this Figure the 'planning'-module uses data from the 'modeling'-module as input, because the system can only build up a plan after making a model of the environment where it is currently located.

When applied to the Stratego agent, the traditional framework takes the form as indicated in Figure 4.3. The three layers sensors, evaluation and effectors are influenced by the motivational and emotional states that the agent undergoes.

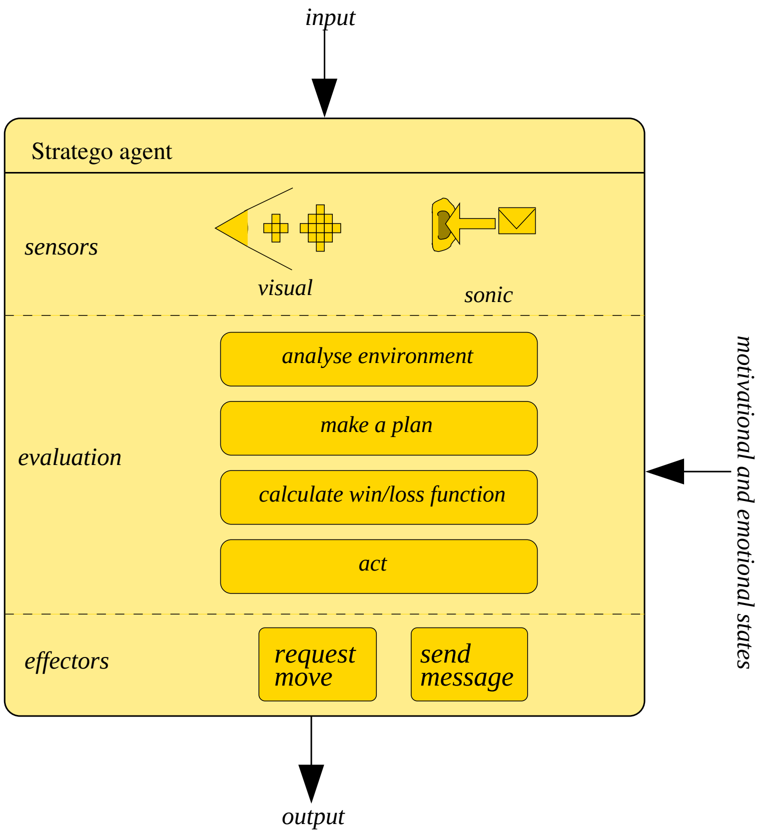


Figure 4.3: Traditional approach to modeling the Stratego agent based a functional decomposition of modules in the evaluation layer

4.2.2 The behavior based approach

The behavior based approach has the advantage that new modules with new behaviors can be added to the system quite easily. Also the architecture allows for a combination of modules that may be based on each other or that may be conflicting among each other. It is imaginable that some goals of Stratego agents may very well be conflicting. The architecture of the behavior based approach seems to be very appropriate for our notion of the Stratego agent, in the sense that for each goal we are able to add a separate behavior module. The three layers sensors, evaluation and effectors are influenced by the motivational and emotional states that the agent undergoes.

As is the case in the subsumption architecture, these modules operate in a considerable autonomous way. The modules shown in Figure 4.4 are some behaviors that apply to a piece in a Stratego environment. Depending on the situation at hand, one of the behaviors has the overhand and dictates

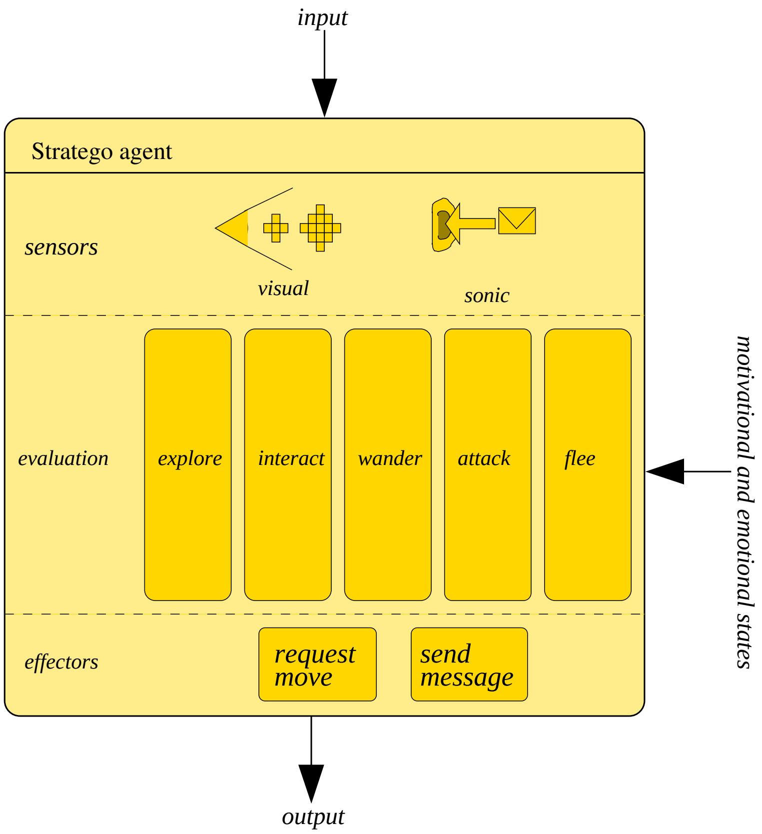


Figure 4.4: Behavior based model of the Stratego agent based on separate behavior modules in the evaluation layer

the overall behavior of the agent. In the Figure some behaviors can be viewed as *primitive behaviors* and some as *cognitive behaviors*. The two behaviors *attack* and *flee* are primitive behaviors that are intrinsic to the agent's character. In some situations for example the agent might become scared of being killed. In that case the behavior flee takes over control and the agent tries to run away from danger.

The cognitive behaviors *explore* and *interact* are more delicate behaviors. The explore behavior for example makes the agent want to move along the board in order to take a strategic position or discover enemy positions. The interact behavior is a behavior that is a relation between percepts (either visual or sonic) and attempting to communicate information across the army.

Chapter 5

Knowledge of the agents

This chapter focuses on the knowledge level of the individual agents. Since the agents represent pieces of the Stratego army, we want them to express behavior that can be seen as 'rational' from their point of view. In other words, we want them to express behavior that will make the agents successful in achieving their goals. Our approach is based on a rule-set that explicitly defines what to do for a number of situations. Section 5.2 gives an elaborate discussion of a rule-set of one of the agents, the miner. But first Section 5.1 discusses the concept of rule-based systems in general.

5.1 Rule-based systems

Rule-based systems, also called production systems, form a well-known architecture for implementing systems based on artificial intelligence techniques. The heart of a rule-based system consists of a database, a rule base and an inference engine. These components interact with the external environment through a perception of the environment and an execution to influence the environment (see Figure 5.1).

The database contains a representation of the state of the environment in *asserted facts*. Upon perceiving the environment the agent asserts corresponding facts in the database. The rule base consists of a set of rules, each of which map a specific state in the environment to one or more actions the agent performs. The rules take the following form:

```
if <list of conditions> then <list of actions>
```

where `<list of conditions>` is associated with asserted facts in the database and `<list of actions>` are actions that may update other facts in the database or influence the external environment. The connection between the facts in the database and the rules in the rule base is made by the inference engine. Upon assertion of facts, the inference engine considers all rules in the rule base. When a state of the world matches a rule, the rule is said to be *fired*.

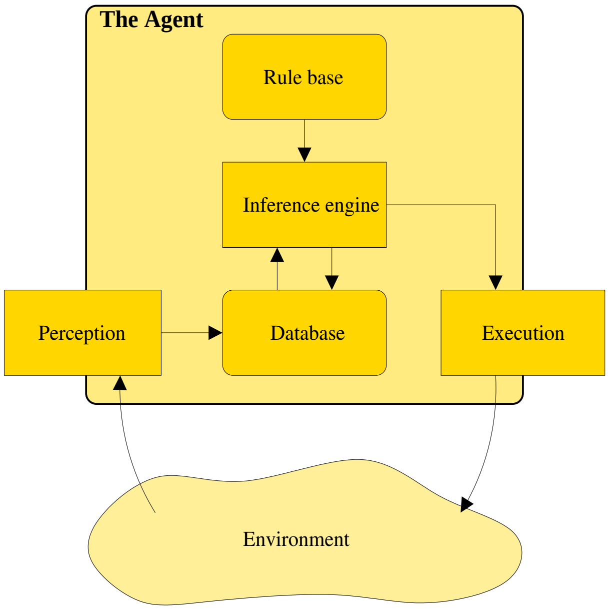


Figure 5.1: An agent based on a rule-based system

5.2 Rules for the agents' behavior

For each of the Stratego agents we have defined a set of rules that specify the behavior, according to the current situation of the agent. We call these rule-sets preference rules, since they indicate preferences to exhibit behavior rather than performing explicit actions.

In this section we will describe a *preference* rule-set for each agent in the Stratego army. Every rule in the set defines several conditions to activate the rule and a preference that is expressed upon activation. The use of preferences in stead of actions in the rules arises from the desire to allow separate behaviors to be activated simultaneously. In Section 5.2.1 an example of a preference rule-set will be given.

5.2.1 Preference rules for the agents

Here we will give some example preference rules for the agent with rank 4, which is the miner. Its complete rule-set, along with the rule-sets of the other agents can be found in Appendix A. Since the miner has a dedicated role within the Stratego army, its behavior has to be somewhat cautious.

The miner has 29 preference rules, which take the following conditions into consideration:

- *Enemy bombs captured*—when all enemy bombs are captured there will be no more bombs to be dismantled and the miner will become less cautious.
- *I have moved*—when this condition is not met, the miner will be less eager to move since it does not want the enemy to know it is a movable piece.
- *My rank revealed*—when this condition is not met, the miner will be less eager to attack pieces because it will try to conceal its rank as long as possible.
- *Enemy at distance 1 or 2*—an enemy piece is spotted within range 1 or 2. Possible types of enemies are:
  1. U: an enemy piece with unknown rank.
  2. H: an enemy piece with a higher rank.
  3. L: an enemy piece with a lower rank.
  4. Bm: an enemy bomb!

Note that in the rule-set tables in the appendix an empty field means a 'don't care,' which means that the corresponding condition does not influence the firing of the rule. Below we will explain some example rules from the miner table.

Preference rule 1

This rule will fire the preference **attack** when the following conditions are met:
→ *enemy bombs captured*,
→ *I have moved*,
→ *my rank revealed* and
→ *enemy with unknown rank present at distance 1.*
In this case the miner is not very cautious because it does not need to dismantle bombs anymore, the enemy already knows that it is not a movable piece and its rank is already known.

Preference rule 13

This rule will fire the preference **flee** when the following conditions are met:
→ *I have moved*,
→ *my rank revealed*,
→ ***not*** *enemy bombs captured* and
→ *enemy with unknown rank present at distance 1.*

Here the miner has a preference for fleeing, because it still has to dismantle bombs and the enemy knows the rank of the miner.

Preference rule 22

This rule will fire the preference **stay** when the following conditions are met:
→ ***not*** *I have moved*,
→ ***not*** *my rank revealed*,
→ ***not*** *enemy bombs captured* and
→ *enemy with higher rank present at distance 1.*
Here the miner dares to stay in spite of the fact that it can be attacked by the enemy. The reason for staying is the fact that the miner has not moved yet and its rank is unknown. The miner wants to prevent the enemy from knowing that it is movable and the enemy piece may be careful.

Preference rule 27

This rule will fire the preference **attack** if and only if an enemy bomb has been spotted at distance 1. It is the task of the miner to dismantle bombs, therefore it will attack.

Chapter 6

Implementation

In this chapter we will describe an implementation for a prototype called Stratesys, as an acronym of the words Stratego expert system shell. The implementation has been done using the object-oriented programming language Java2. At this moment a first prototype of Stratesys has been developed, in which the basic functionalities of our design have been implemented.

6.1 An implementation for the agents

In the current version of the Stratesys we have implemented an agent type that is based on production systems, as described in Section 5.1. For the communication among agents and the agent rule-engine we have used the JavaSpaces™ Technology and an expert system shell called Jess, respectively. These will be described in Section 6.1.2. But first we will review some alternative approaches to implementing agent systems.

6.1.1 Existing agent frameworks

A lot of effort is being put into building frameworks that allow for construction of distributed systems. Some of them are specifically meant for building multi-agent systems, like for example JAT*Lite* from the Stanford university and the Aglets software kit from IBM. These two are described below.

JAT*Lite*

JAT*Lite* stands for Java Agent Template *Lite*. It is a package of Java classes and programs that allow users to create quickly new systems of software agents that communicate over the Internet in order to perform a distributed computation (Jeon, Petrie & Cutkosky 2000).

The architecture of JAT*Lite* consists of a number of layers built on top of each other. These layers are the AbstractLayer, BaseLayer, KQMLLayer, RouterLayer and the ProtocolLayer (see Figure 6.1).

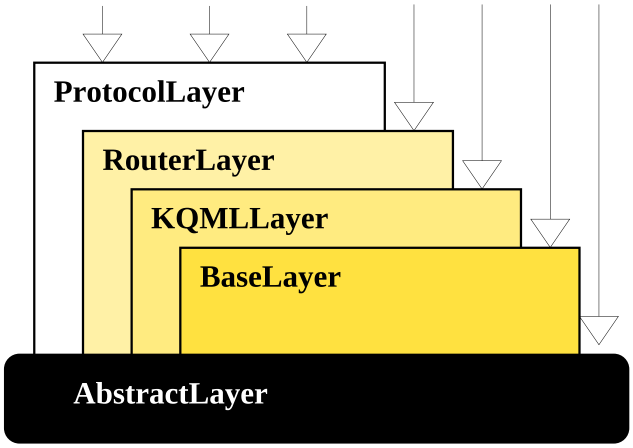


Figure 6.1: JAT*Lite* layered architecture

- The **AbstractLayer** provides the collection of abstract classes necessary for the JAT*Lite* implementation.

- The **BaseLayer** provides basic communication based on TCP/IP and the AbstractLayer. There is no restriction on the message language or protocol.

- The **KQMLLayer** provides for storage and parsing of KQML messages.

- The **RouterLayer** provides name registration and message routing and queuing for agents. All agents send and receive messages to the Agent Message Router (AMR) which forwards them to their named destinations.

- The **ProtocolLayer** provides several open Internet protocols such as FTP and SMTP.

The advantage of this type of layered architecture is that the programmer can access the framework from any layer, depending on the needed functionality. For example, the RouterLayer provides the AMR containing functionality for automatic routing of messages and looking up agents by name.

Special protocols exist that allow for communication between agents. Agents can send and receive typed messages to each other in a predefined Agent Communication Language (ACL). An example of such a language is KQML. The JAT*Lite* architecture provides support for this language in the KQMLLayer.

KQML is a communication language and protocol developed by the Knowledge Sharing Effort (KSE) consortium. It is both a message format and a message-handling protocol to support run-time knowledge sharing among agents. Some of the key features are:

- KQML messages are opaque to the content they carry, they communicate an 'attitude' about the content (assertion, request, query).
- The language primitives are called *performatives*, which define the permissible actions (operations) that agents may attempt in communicating with each other.
- An environment of KQML speaking agents may contain special agents called *facilitators*. These agents provide administrative services to the other agents.

KQML is a first attempt for a communication language offering primitives that have a meaning outside the context of a specific application or representation. The language should be disassociated from interaction and transport protocols and should be concerned only with attitudes about the content. For more information about KQML see (Finin, Labrou & Mayfield 1997).

Aglets

The Aglet Workbench made by IBM provides an applet-like programming model to build Java-based mobile agent systems. The aglet is an acronym for *agile* applet. It is a reflection of the Java applet model. According to Lange & Oshima (1998), the Java language and specifically the Java applet model have some shortcomings when it comes to building Java-based mobile agent systems:

- There is inadequate support for resource control. This means it can be difficult to control the resources consumed by agents.
- There is no protection of references. There is no way for the agent to directly monitor and control which other agents are accessing its methods.
- There is no object ownership of references. This implicates that we cannot explicitly 'void' an agent, it will remain running until no other object has a reference to it and it is being removed by the garbage collector.
- There is no support for preservation and resumption of the execution state. Therefore, for an agent to properly resume a computation on a remote host, it must rely on internal attribute values and external events to direct it.

The Aglet Workbench was designed to benefit from the agent characteristics of Java while overcoming some of the mentioned deficiencies. The basic elements of the workbench are a set of abstractions. They are aglet, proxy, context, message, future reply and identifier:

Aglet A mobile Java object that visits aglet-enabled hosts in a computer network. It is autonomous and reactive.

Proxy A representative of an aglet. It serves as a shield for the aglet that protects the aglet from direct access to its public methods.

Context A context is an aglet's workspace. It is a stationary object that provides a means for maintaining and managing running aglets in a uniform execution environment where the host system is secured against malicious aglets.

Message An object exchanges between aglets. It allows for synchronous as well as asynchronous message passing between aglets.

Future reply Used in asynchronous message-sending as a handler to receive a result later asynchronously.

Identifier Bound to each aglet, globally unique and immutable throughout the lifetime of the aglet.

The aglet's fundamental operations are creation, cloning, dispatching, retraction, activation and deactivation, disposal and messaging. An aglet can be brought to life by being instantiated from scratch (creation) or by being copied from an existing aglet (cloning). We can also get rid of an aglet (disposal). An aglet is actively mobile when it pushes itself from its current host to a remote host (dispatching). An passively mobile aglet is being pulled from its current host by another host (retracting). Aglets can go to sleep (deactivation) and be woken up again (activation). Finally multiple aglets can exchange information to accomplish a given task (messaging).

6.1.2 The rule engine

Here we will discuss the details of our implementation of the Stratego agents. As described in Chapter 5, we have chosen to use a production system to fill in the agents' evaluation layer. In this section we will describe an implementation for the preference rules.

Upon each move of Stratesys, all agents evaluate their situation and express their desire to act (or not to act). Because of the fact that only one agent can move at a time, Stratesys has to pick one and allow it to move. This is done according to a weighted function that takes into account all desires of the agents.

The agent's rule engine has been implemented using the notion of separate behavior patterns. The implementation is therefore a reflection of the behavior based model of the agent as described in Chapter 4 (see Figure 4.4). The reason for applying this type of model is the fact that the agent's behavior can be explained as being a resultant of all separate behaviors. In other words, we can say that the agent shows *emergent* behavior that is caused by the sum of all separate behaviors.

A great advantage of this type of emergent behavior is that the agent comes somewhat closes to our notion of an autonomous system. The agent's perception, goals, motivations, etc. all influence the agent's actions. This means we can define different types of rules for the agent that may harmonize or conflict with each other. The overall behavior of the agent will be determined as the sum of all separate behaviors.

Another advantage of using this approach is the ease of adding new modules for additional behavior patterns, where modules—as is the case in the subsumption architecture—can inhibit each other to form more complex behavior.

Jess - Java Expert System Shell

Since the Stratesys system was developed using Java, a natural choice for an expert system shell is the Java expert system shell (Jess). Jess is a rule engine and scripting environment written entirely in Java. It was originally inspired by the CLIPS expert system shell, but has grown into a complete, distinct Java-influenced environment of its own. Because of its complete implementation in Java, the rule-engine can be easily embedded within the Java simulation environment. For detailed information about Jess see (Friedman-Hill 2000).

For every agent we have implemented the behaviors as described in Chapter 4, which are attack, flee, wander and stay. Additionally, we have added some extra behaviors that apply to specific situations. These are:

1. *Attack-general*—a specific rule for the spy. When the spy sees the enemy general within attacking range, this behavior will make the spy attack it.

2. *Attack-bomb*—a specific rule for the miners. Upon seeing an enemy bomb within attacking range, the miner will be eager to attack it.

3. *Avoid*—a rule that is applied for all agents except the miners, because it is used to avoid enemy bombs.

In the prototype, all agents have a visual perception in the form of a diamond of five squares wide. See Figure 6.2 for a picture of an agent (miner) in its environment. The miner sees an enemy piece with unknown rank (north square) and an enemy scout (north-east square). The agent also

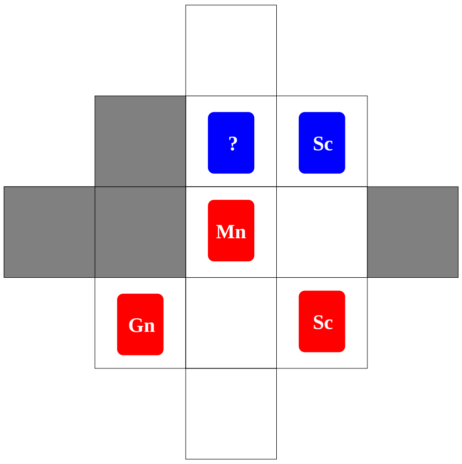


Figure 6.2: The miner in its environment

sees some fellow agents, the general and a scout (south-west and south-east respectively).

In the current implementation of the rule engines, the evaluation consists of a mapping from enemy locations to a desire to move (for each direction) or to stay, expressed in *scores*. In this specific example, the miner may want to flee from the unknown enemy. But it also sees an enemy scout which can be beaten. Therefore in this particular case the miner's behavior will be a mixture of the desire to flee and attack and wander:

$$\begin{array}{ccc} & -150 & \\ 75 & 0 & 60 \\ & 225 & \end{array}$$

The scores indicated above express relative desires to go or stay. Negative scores mean that the agent does not want to go in the corresponding direction. In the example the scores are a resultant of the behaviors flee, attack and wander. The fleeing behavior is due to the enemy with unknown rank. Since the miner is a somewhat cautious agent, the score to move backward is largest and the miner will decide to go backward. The wander behavior is a less important behavior, which is used to express the general desire to move around the board. It has contributed to the scores with minor additions to each direction, the forward direction being the preferred wander-route.

Communication structure

We have implemented a communication structure using the JavaSpaces™ Technology (Freeman et al. 1999). This technology allows for numerous applications that run in a distributed context. Several programs can make use of the JavaSpace-service, which runs within a local area network. See Figure 6.3 for a picture with two JavaSpaces and several 'Dukes' interacting with it.

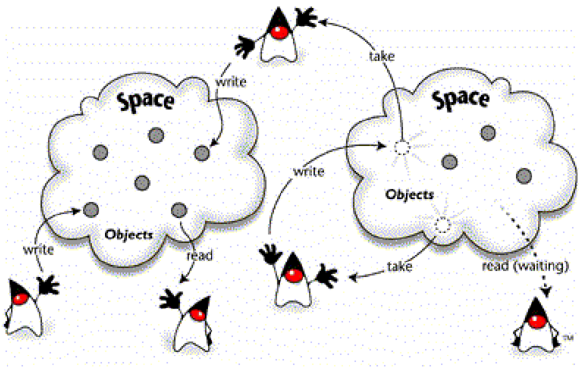


Figure 6.3: The JavaSpaces Technology

One of the possible applications of this technology is an implementation of a robust message-passing system. In our agent-based application, the Dukes in the picture can be viewed as Stratego agents that use the space to communicate with each other. We have called our agent communication structure the 'AgentSpace.'

An advantage of using a JavaSpace-service is its ease of use. We have implemented three types of messages, which are messages to all other agents (broadcast), messages to all agents with a specific rank and messages to specific agents. Although the communication structure has been built, in the prototype it is not yet actually used by the agents. An example of a first step to make use of messages can be notifying all other agents of a captured enemy piece, or a loss of a friendly piece.

6.2 The simulation

In this section we will review the details of the simulation environment. The Stratesys framework allows for different types of players to play against each other. It has three modes of operation, which are human versus agents, agents versus agents and human versus human. To make the latter mode possible the framework has been set up as a Client-Server model. One or two Clients connect to the Server to request a game of Stratego.

All functionality regarding the simulation of the agents is placed in the Client program. This will be discussed in Section 6.2.1. Section 6.2.2 discusses the implementation of the communication between the Clients and the Server.

6.2.1 Simulating the agent's environment

All agents have the three layers sensors, evaluation and effectors. See Figure 6.4 for a schematic view of the agent. Here we can see each layer containing the agent's internals. It also shows the objects it is related to in its environment.

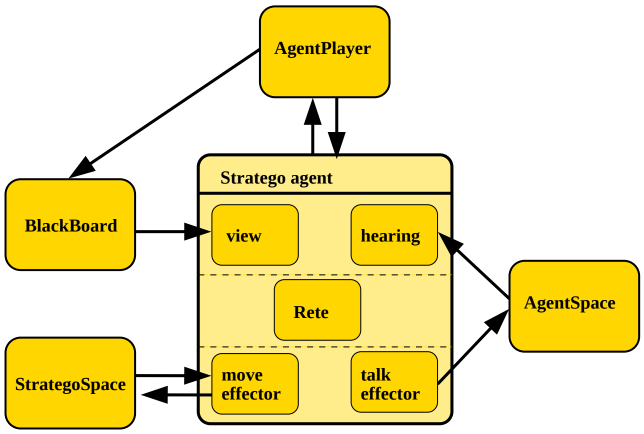


Figure 6.4: Schematic view of the agent's implementation

The *AgentPlayer* functions as a representative of the Stratego army formed by agents. It is responsible for creating all agents upon start-up, initializing them and positioning them on the Stratego board. Currently, the positioning is fixed but can be made more flexible by extending the functionality of the AgentPlayer.

Also, the AgentPlayer is responsible for maintaining information on the *BlackBoard*. This is an object that continuously reflects the actual situation on the Stratego board, the way the AgentPlayer sees it. In other words the AgentPlayer keeps positions of all pieces and where possible fills in missing information concerning enemy ranks.

The *AgentSpace* is the agent's interface to communication with its fellow agents (a JavaSpace-service). It is read from by the *Hearing* object en sent to by the *Talk effector*. At the moment, the Hearing object only listens for messages that are addressed to all agents in the army (broadcast). However the AgentSpace allows for an easy addition of messages to agents with a specific rank and messages to specific agents.

The *View* object provides the agents with visual perception. It is actually an accurate copy of a small part of the BlackBoard. It continuously checks for recent changes on the BlackBoard, and updates itself whenever necessary.

The StrategoSpace is the communiation medium for the Client and the Server. This will be described below.

The agent's evaluation cycle

The agent's lifecycle can be viewed as a number of states and transitions. In Figure 6.5 an automaton is drawn with its states and transitions. The most important state in the automaton is the *Evaluate* state. Here, the Rete algorithm is applied using the percepts that have been received. If the evaluation leads to an action, it will cause a transition to the *Sleep* state. Currently the action that has been implemented is sending a move request to the Server, waiting for an answer and waiting for new percepts.

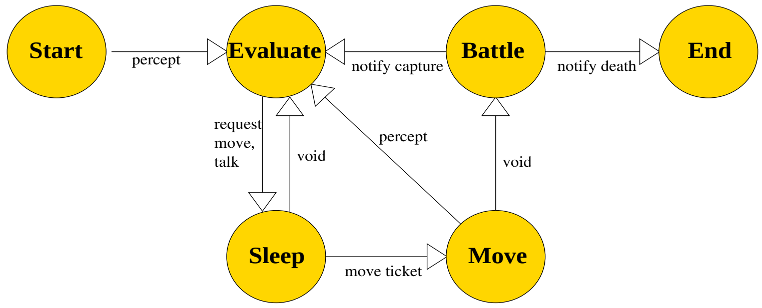


Figure 6.5: An agent's evaluation cycle

In the *Move* state a piece can do an actual move. From the move state there are two possible transitions to other states. When a move to an empty square was done the agent perceives some changes in its environment and evaluates them. The other possibility is a battle with an enemy piece. In the *Battle* state the agent either wins and notifies all fellow agents of the capture or the agent looses and notifies its death.

6.2.2 The Client-Server model

Since we wanted to be able to play human versus human games, we have created two programs that implement a Client-Server model. The Client is the main Stratesys program. The Server runs in the background, continuously listening for Clients to connect. See Figure 6.6 for a schematic view of the Client and the Server.

The communication between the Clients en the Server has been implemented by a JavaSpace-service called 'StrategoServiceSpace.' Using the space the Clients and the Server can exchange information by reading from and writing messages to the space. The communication scheme that we have used for this is described in Section 6.2.3.

The Client

The Client is the main Stratesys program. At startup, a window is positioned on the screen with an empty Stratego board. The possible ways of

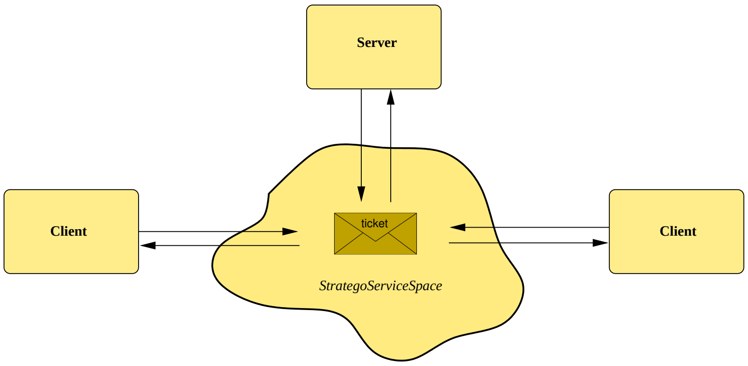


Figure 6.6: The Client-Server model

playing the game are a human player playing against an agent army and two human players playing against each other.

At startup, the players will be registered with the Server. The human playing Stratego can position his army by clicking on the squares of the board. A pop-up menu will appear that will allow the player to choose a piece. In Figure 6.7 we can see the Client window, where the red player has already positioned its army. It is now waiting for the blue player.

When all pieces have been positioned, the players can begin to move their pieces. By using mouse-clicks on the squares of the board the human player can select pieces to move. For clarity concerning the situation on board we have chosen to implement the use of animation for each moving piece. When a correct move has been requested the board draws an animation of the moving piece from the initial position to its destination.

Depending on the type of game that is played, one or two Player objects will be created. Only in the human versus human mode will the Client create one Player object. Naturally this implicates the necessity of another Client in the network. In the other modes of operation, only one Client is used which runs both Player objects.

The Player object is schematically shown in Figure 6.8. The Player object has both references to its own Pieces and to EnemyPieces. The EnemyPieces are actually only 'dummy' Pieces that are a visual representation of the actual enemy pieces.

In the figure, the arrows represent messages exchanged across the objects. From the Player object to the Server and back are messages to register the player with the board. Messages from Player to Pieces concern position and move messages. The same applies to the messages sent from the Pieces to the Server and back.

The EnemyPieces however only receive messages from the Server and relay them to the Player object. This is because of the fact that these objects are only visual representations, as mentioned before.

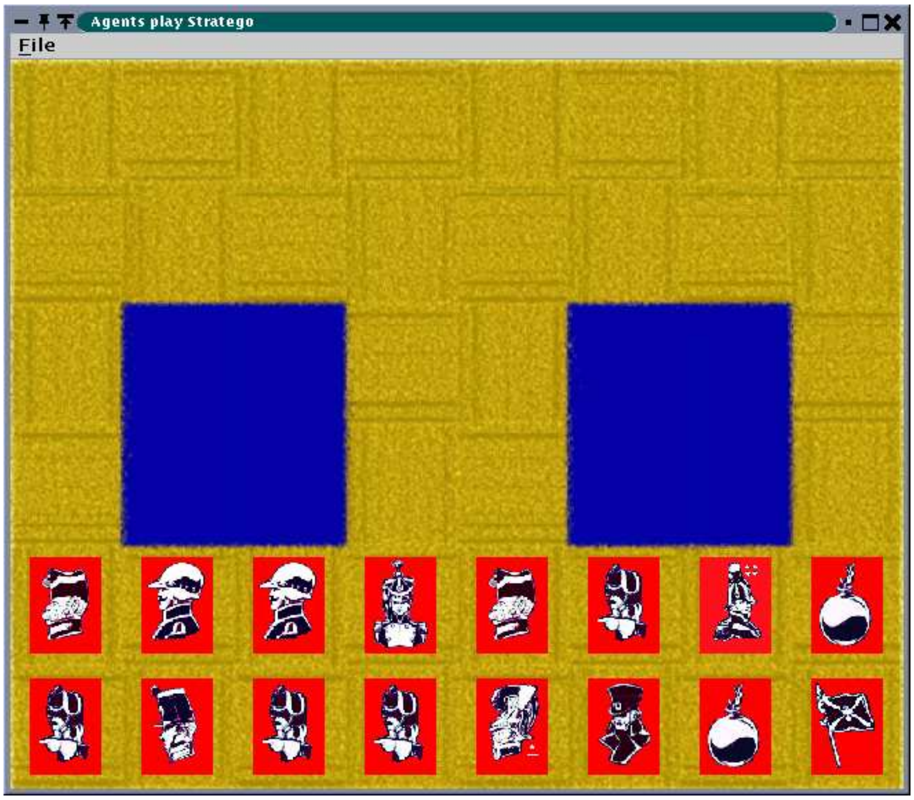


Figure 6.7: The Client window

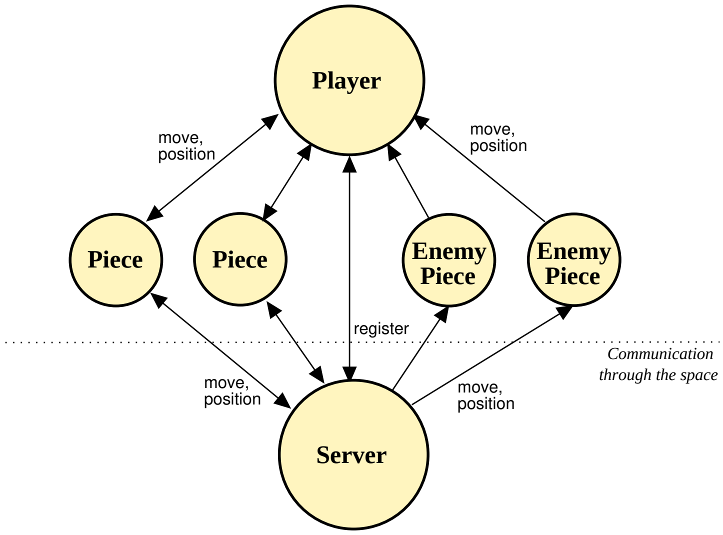


Figure 6.8: The Player object

The Server

Our implementation of the Server can accept two players wanting to play Stratego. These players can reside in one Client program or two. The latter case is only for human versus human games. After the game is over, the Server will wait for new requests for playing.

The Client-Server communication consists of four phases. These are *registration*, *positioning pieces*, *moving pieces* and *notifying a game over*. For each of the phases we have defined specific messages which we call *tickets*. Tickets are sent as requests and received as answers to that request. The idea behind the concept of a ticket is that a ticket gives a piece the right to position itself somewhere or move to a certain square.

Upon starting the game, the Client creates one or two Player objects, depending on the type of game that is played. The Players send a *RegistrationTicket* to the Server to register. After sending the ticket, they will receive an answer with information about the registration (successful or not).

When two players have registered to the Server, they can position their pieces. For each piece to be positioned a *PositionTicket* is created and sent to the Server. The Server checks to see if the requested positions are valid, and sends the tickets back with this information.

6.2.3 Specification of the tickets

In the following figures the tickets for each phase is shown, specifying the data that is sent back and forth.

```
recipient: <name of the server>
sender   : <name of the player>
valid    : not used
error    : not used
uniform  : <color>
           preference for color
```

```
recipient: <name of the player>
sender   : <name of the server>
valid    : <true / false>
           indicates succes / failure
error    : <exception message / null>
           reason for failure
uniform  : <color>
           uniform to play with
```

Figure 6.9: Registration tickets are used to register a player with the Server

```
recipient: <name of the server>
sender   : <identification of the piece>
valid    : <true / false>
           true -> position me
           false -> remove me
error    : not used
position : <requested position / null>
rank     : <rank of the piece>
```

```
recipient: <identification of the piece>
sender   : <name of the server>
valid    : <true / false>
           indicates succes / failure
error    : <exception message / null>
           reason for failure
position : <requested position / null>
rank     : not used
```

Figure 6.10: Positioning tickets to request initial positions on the board

```
recipient: <name of the server>
sender   : <identification of the piece>
valid    : not used
error    : not used
move     : <requested move>
battle   : not used
```

```
recipient: <identification of the piece>
sender   : <name of the server>
valid    : <true / false>
           indicates succes / failure
error    : <exception message / null>
           reason for failure
move     : not used
battle   : <battle info / null>
```

Figure 6.11: Requests to the Server for moving pieces

```
recipient: <name of the server>
sender   : <name of the player>
valid    : not used
error    : not used
message  : <surrender / draw>
```

```
recipient: <name of the player>
sender   : <name of the server>
valid    : not used
error    : not used
message  : <winner / looser / draw>
           winner sent to the winner
           looser sent to the looser
           draw sent to both players
```

Figure 6.12: One of the players can issue a 'game over', by surrendering or notifying a draw

Chapter 7

Testing the prototype

7.1 Example test-runs

In this section we will consider two situations for the miner. We will assume that the miner is in the environment as indicated in Figure 7.1.

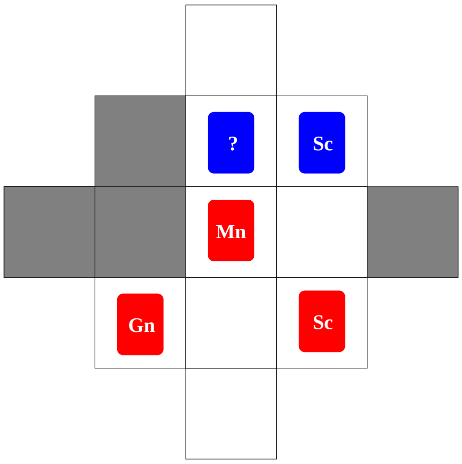


Figure 7.1: The miner's environment

First, we will try the case where the miner has not moved yet and its rank is still unknown. The Jess output gives us:

```
f-51    (enemy-unknown north-east)
f-52    (enemy-unknown north)
f-53    (stay)
f-54    (update-scores 250 0 0 0 0)
f-55    (wander)
f-56    (update-scores 0 50 25 25 10)
For a total of 57 facts.
```

```
Score for staying: 250
Score for moving forward: 50
Score for moving left: 25
Score for moving backward: 25
Score for moving right: 10
```

Here we see that the miner really wants to stay. This is mainly because of the fact that it has not moved yet. It assumes that the enemy will be less eager to attack since the miner has not moved yet and its rank is still.

Second we will assume that the miner has moved, but its rank is still unknown. This leads to the result:

```
f-51   (i-have-moved)
f-52   (enemy-unknown north-east)
f-53   (enemy-unknown north)
f-54   (flee north)
f-55   (update-scores 0 -200 50 200 50)
f-56   (wander)
f-57   (update-scores 0 50 25 25 10)
For a total of 58 facts.
Score for staying: 0
Score for moving forward: -150
Score for moving left: 75
Score for moving backward: 225
Score for moving right: 60
```

Now the miner wants to flee from the unknown piece in front of it. Therefore the backward direction has the largest score.

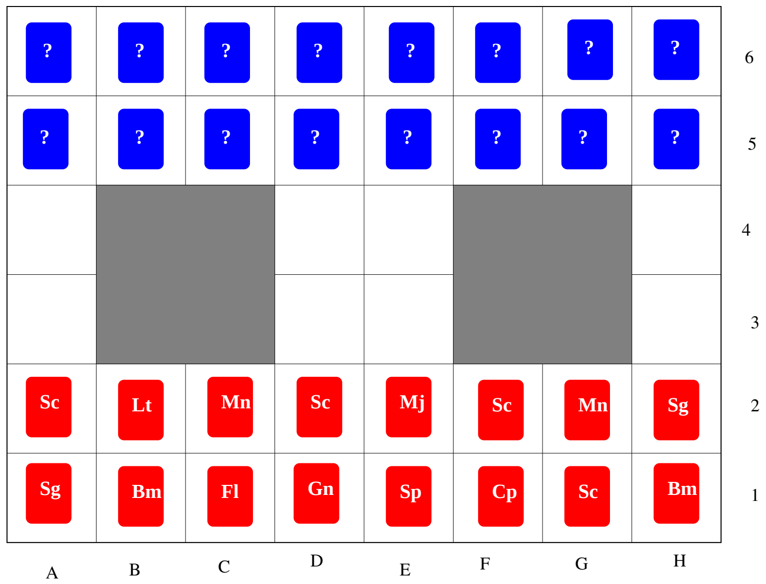


Figure 7.2: Starting positions

7.2 Playing the game

In this section we will review a game played between the Stratesys agents and a human player. We will review it from the point of view of the Stratesys program. In the experiment, the positioning of the Stratesys army was programmed to be fixed. See Figure 7.2 for the board with starting positions.

The game was won by the human player. But as we shall see, is was a game with opportunities for both players. Espcially during the opening and mid-game stages the agents (Stratesys) were playing convincing. Unfortunately, the end-game still is difficult for the current implementation of Stratesys.

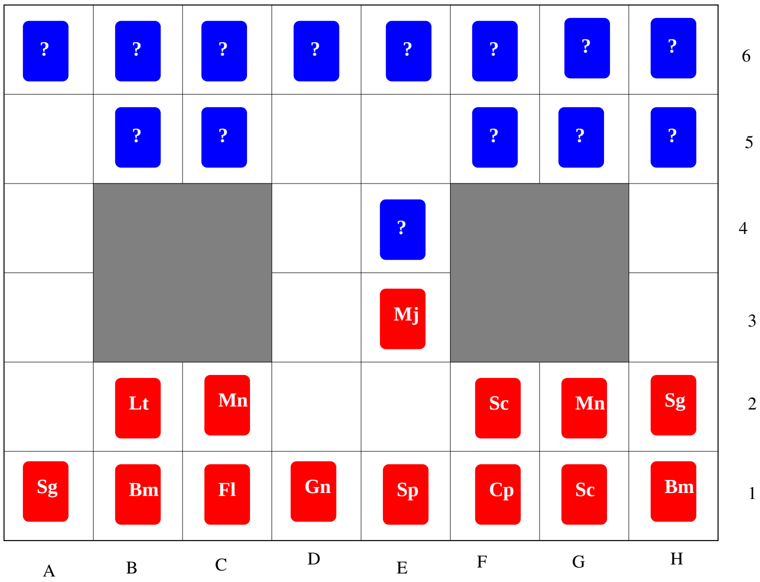


Figure 7.3: Move 6

Move	Stratesys	Human	Comment
1	d2-d3	a5-a4	
2	h2-h3	e5-e4	
3	d3-d4	d5xd4	Enemy scout attacks scout, a draw.
4	e2-e3	a4-a3	
5	h3-h2	a4-a2	Enemy scout attacks scout, a draw.
6	e3xe4	e5-e4	Stratesys major attacks human major, a draw

```
f-51 (i-have-moved)
f-52 (enemy-unknown north)
f-53 (attack north)
f-54 (update-scores 0 200 -50 -200
-50)
f-55 (wander)
f-56 (update-scores 0 50 25 10 25)
For a total of 57 facts.
Score for staying:  0
Score for moving forward:  250
Score for moving left:  -25
Score for moving backward:  -190
Score for moving right:  -25
```

In the following board position the major at e3 is chosen to do a move (see Figure 7.3). It decides to attack the enemy piece at e4. This decision is based on the scores as shown in the jess output to the left.

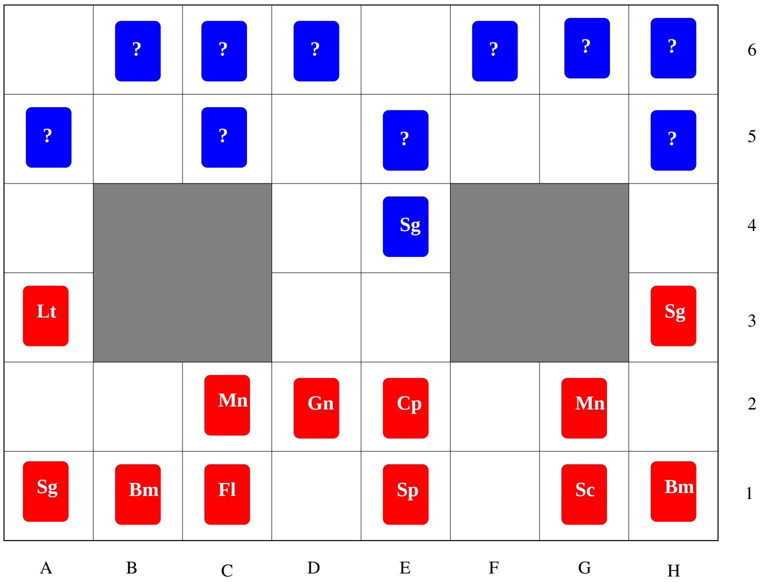


Figure 7.4: Move 16

Move	**Stratesys**	**Human**	**Comment**
7	f2-e2	a5-a4	
8	b2-a2	a4-a3	
9	a2xa3	e6-e5	Stratesys lieutenant beats human scout
10	e2-e3	e5-e4	
11	f1-f2	h5-h4	
12	f2-e2	f5-e5	
13	e3xe4	h4-h3	Human sergeant beats Stratesys scout
14	h2xh3	g5-h5	Stratesys sergeant beats human scout
15	d1-d2	b5-a5	
16	g1-f1	a5-a4	

```
f-51 (wander)
f-52 (update-scores 0 50 25 25 10)
For a total of 53 facts.
Score for staying:  0
Score for moving forward:  50
Score for moving left:  25
Score for moving backward:  25
Score for moving right:  10
```

Here the scout at g1 gets to move. It decides to move to f1 because of its wander behavior. As we can see in Figure 7.4, the only free square is to the left.

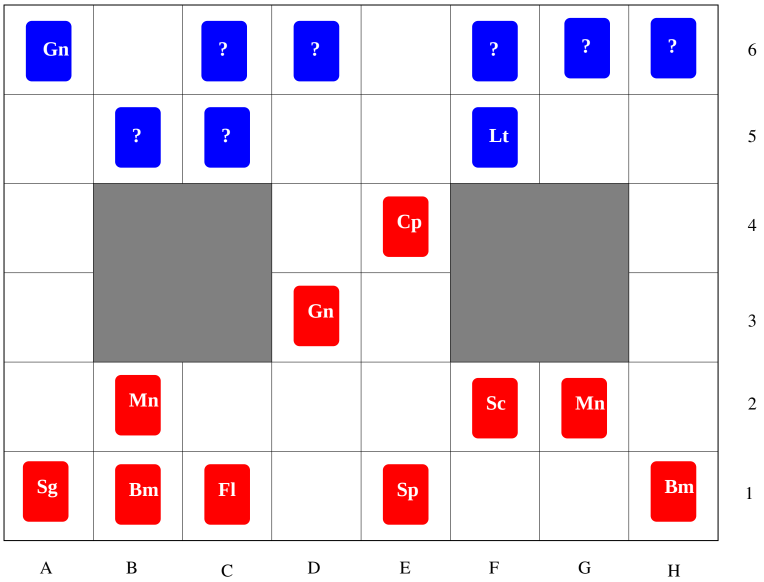


Figure 7.5: Move 25

Move	**Stratesys**	**Human**	**Comment**
17	a3xa4	h5-h4	Stratesys lieutenant beats sergeant
18	h3xh4	h4-h5	
19	a4-a5	h5-g5	
20	a5-a6	g5-f5	
21	f1-f2	b6xa6	Human general beats Stratesys lieutenant
22	d2-d3	e5-d5	
23	e2-e3	c5-b5	
24	e3xe4	d5-c5	Stratesys captain beats human sergeant
25	d3-d4	c5-d5	

```
f-51 (wander)
f-52 (update-scores 0 50 25 25 10)
For a total of 53 facts.
Score for staying:  0
Score for moving forward:  50
Score for moving left:  25
Score for moving backward:  25
Score for moving right:  10
```

The general is going to move from d3 to d4. This is due to the wander behavior, there are no enemies in the neighborhood (see Figure 7.5).

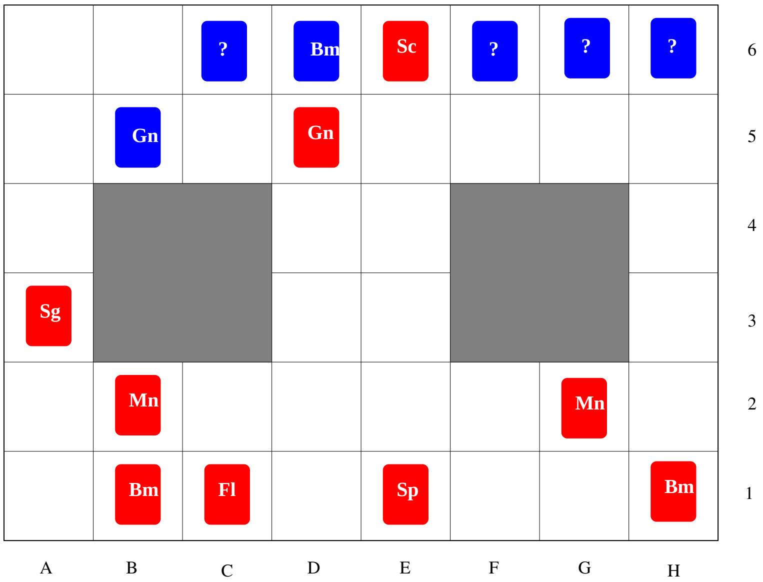


Figure 7.6: Move 35

Move	**Stratesys**	**Human**	**Comment**
26	e4-e5	d5xe5	Human captain attacks Stratesys captain, a draw
27	f2-e2	b5-a5	
28	e2-e3	a5-a4	
29	e3-e4	a4-a3	
30	a1-a2	a3xa2	Stratesys sergeant beats human miner
31	e4-e5	a6-a5	
32	e5-e6	f5-e5	
33	a2-a3	e5-d5	
34	d4-d5	a5-b5	
35	d5xd6	b5-c5	Human bomb beats Stratesys general

```
f-51 (i-have-moved)
f-52 (my-rank-revealed)
f-53 (enemy-general west-west)
f-54 (enemy-unknown north-west)
f-55 (enemy-unknown north)
f-56 (attack north)
f-57 (attack north-west)
f-58 (update-scores 0 75 75 -75 -75)
f-59 (update-scores 0 200 -50 -200 -50)
For a total of 62 facts.
Score for staying:  0
Score for moving forward:  325
Score for moving left:  50
Score for moving backward:  -250
Score for moving right:  -115
```

The general decides to attack! This is risky, but since its rank is alrey revealed it is more likely to attack. Unfortunately, this is a bomb (see Figure 7.6).

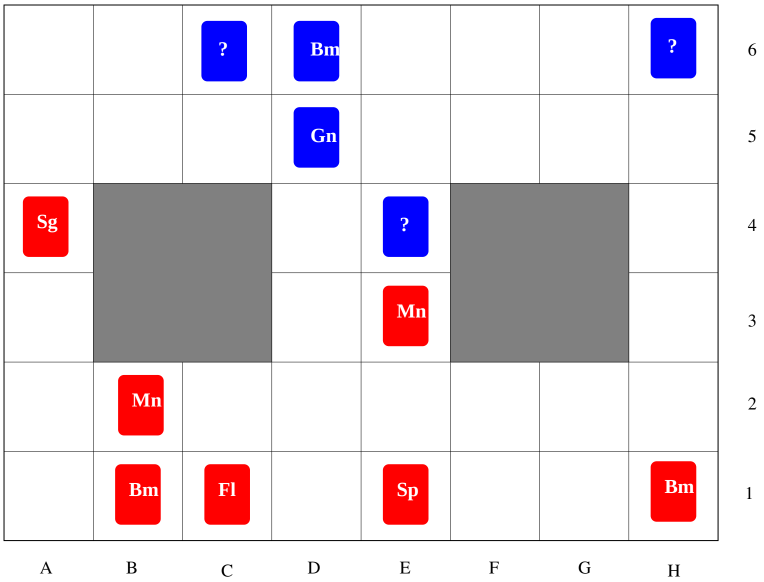


Figure 7.7: Move 46

Move	**Stratesys**	**Human**	**Comment**
36	g2-f2	f6-f5	
37	e6-e5	f5-g5	
38	a3-a4	c5-d5	
39	e5-f5	g5-h5	
40	f5-g5	h5-h4	
41	g5xg6	h5-h5	Stratesys scout beats human spy
42	g6xh6	h5-g5	Human bomb beats Stratesys scout
43	f2-e2	g5-f5	
44	e2-e3	f5-e5	
45	c2-b2	e5-e4	
46	b2-a2	d5-c5	

```
f-51 (i-have-moved)
f-52 (wander)
f-53 (update-scores 0 50 25 25 10)
For a total of 54 facts.
Score for staying:  0
Score for moving forward:  50
Score for moving left:  25
Score for moving backward:  25
Score for moving right:  10
```

The miner goes to the left. Its wander behavior has equal scores for both left and right and a random pick is performed (see Figure 7.7).

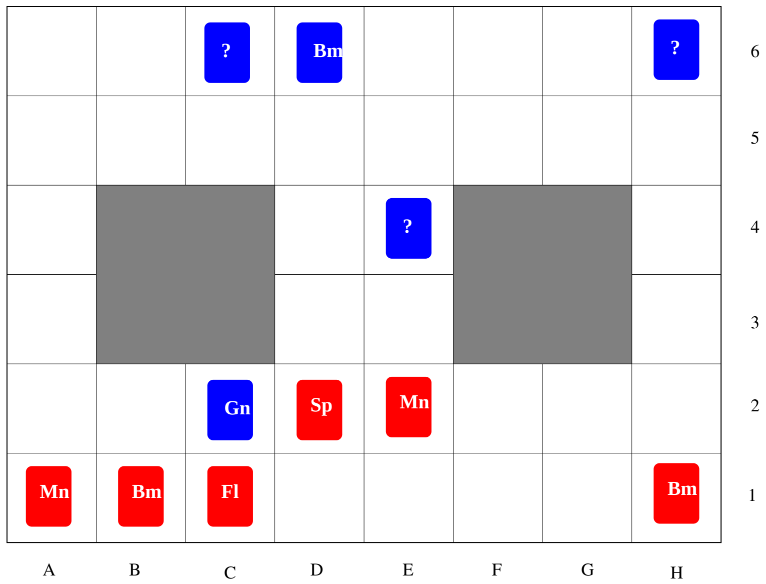


Figure 7.8: Move 54

Move	**Stratesys**	**Human**	**Comment**
47	e1-e2	c5-b5	
48	e2-d2	b5-a5	
49	a4-a3	a5-a4	
50	a2-a1	a4xa3	Human general beats sergeant
51	d2-d3	a3-a2	
52	e3-e2	a2-b2	
53	d3-d2	b2-c2	
54	d2xc2	e4-d4	Stratesys spy beats general

```
f-51 (i-have-moved)
f-52 (enemy-general west)
f-53 (update-scores 0 0 10000 0 0)
For a total of 54 facts.
Score for staying:  0
Score for moving forward:  0
Score for moving left:  10000
Score for moving backward:  0
Score for moving right:  0
```

The spy is about to move, here is decides to attack the general. This is the result of a very high score (see Figure 7.8).

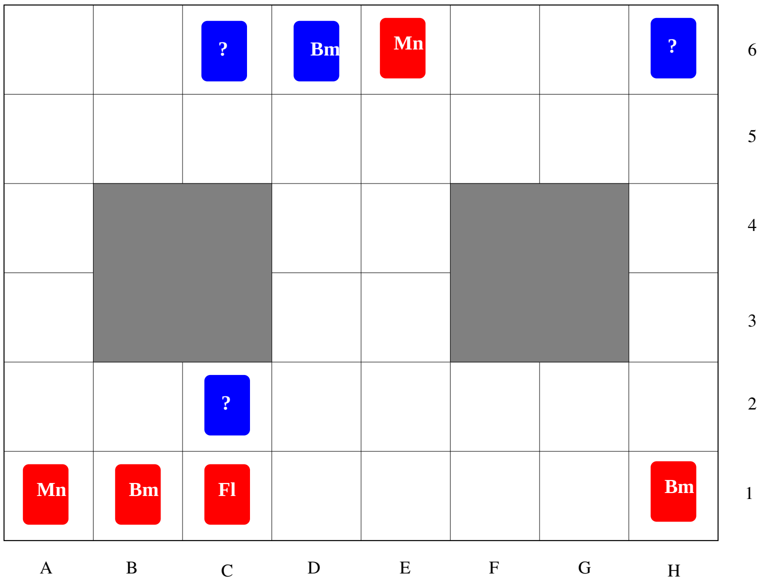


Figure 7.9: Move 58

Move	**Stratesys**	**Human**	**Comment**
55	e3-e4	d4-d3	
56	e4-e5	d3-d2	
57	e5-e6	d2xc2	Human miner beats Stratesys spy
58	e6xd6	c2xc1	Stratesys miner beats bomb and human miner beats Stratesys flag!

```
f-51 (i-have-moved)
f-52 (enemy-bomb west)
f-53 (attack-bomb west)
f-54 (update-scores 0 0 1000 0 0)
f-55 (wander)
f-56 (update-scores 0 50 25 25 10)
For a total of 57 facts.
Score for staying:  0
Score for moving forward:  50
Score for moving left:  1025
Score for moving backward:  25
Score for moving right:  10
```

The miner at e6 is about to move, since it is a miner and a bomb is next to it, an attack is evident. Unfortunately the miner of the human player guesses the position of the Stratesys flag: Stratesys looses (see Figure 7.9).

Chapter 8

Conclusion

In this investigation we have examined a multi-agent approach for playing the game Stratego. This approach involves playing the Stratego game with multiple agents that each represent a piece in the Stratego army. The approach was based on the hypothesis that for some complex problems distributed techniques for solving them can result in more intuitive solutions.

We assumed that the Stratego game could serve as an excellent playground for testing the hypothesis. Typically, the game brings about a high complexity that makes an analysis of it rather difficult. This is due to the fact that both players have incomplete information about the board status. Furthermore, the game's characteristics lend their selves well for the distributed techniques that we proposed to use. Upon modeling the agents we can make use of the fact that the Stratego army consists of military men that each have their own dedicated role within the army. In other words, up to some level the agents can act as individuals where each of them has their own set of characteristics.

8.1 Evaluation

At the start of the investigation a literature survey has been done about the agent and multi-agent concepts. Firstly, this has led to insights about the level of rationality, autonomy and intelligence of agents. Additionally, studying the literature about the field of multi-agent systems gave us an understanding of different types of multi-agent systems and their uses, which allows us to fit in our approach with the field.

Unfortunately there have not been any studies or analyses about the Stratego game. We have written down some general considerations about the game's characteristics, but a thorough analysis is not a purpose of the investigation. In an attempt to acknowledge the complexity of game playing in general, we have studied some papers on game playing theories. These theories apply to perfect information games. Theories about games with information information however are sparse, and where available they are very complicated. This leads us to believe that an intuitive solution for finding a way to play Stratego is to be found using multiple agents, where we avoid reasoning about board positions seen from a central point of view.

One of the things that the literature survey has resulted in, is an inspiration for the agent's model using the subsumption architecture by Brooks. We have modeled the agent as a system that can exhibit behavior patterns which result from separate internal behaviors. These internal behaviors may inhibit each other, harmonize or even conflict with each other.

This way of modeling the agent has some important advantages. First of all, this way of modeling systems comes closer to explaining human intelligence than the traditional way of modeling intelligence. Secondly, a pragmatic advantage of this type of architecture is that the agent's behavior can be extended easily. By adding new internal behaviors, the agent expresses a new overall behavior.

We have chosen to use rule-based approach for specifying the agent's behavior. It turns out that for a Stratego agent we can specify a lot of rules which result from the characteristics of the game. We can easily define a number of goals for the agents such as staying alive, capture enemy pieces, etc.

Our design reflects the desire to build a generic tool for the investigation. The basic functionality of the design has been implemented successfully. The prototype (Stratesys) currently allows for playing Stratego games locally or via a local area network. The most important part of Stratesys is the implementation of a framework for the agents in which they have:

- Perception.
  - → Visual perception in the form of a diamond of five squares wide.
  - → Sonic perception in the form of a listener for messages from fellow agents.
- Evaluation of its environment, using the visual percepts and a memory. The memory takes the form of known facts about its own status and about the board situation.
- Effectors which allow them to send move requests, move to another square if allowed or send messages to fellow agents.

Note that every agent has its own characteristics. For each of the ranks we have implemented specific rule-sets. The behavior of the agent results from the specific rule-set that it uses according to its rank, its perception and its own memory.

Stratego is a game with many open problems. As stated before there has not been any analysis of the game. Therefore we advocate to use a corpus-based approach to build up a library of games which can be used for studies and experiments about Stratego. The Client-Server model that we have implemented is a framework with which several experiments can be run regarding agents and Stratego. An interesting property of the framework is that the Server can be run in the background. This way we build up the library by asking human players to log on to Stratesys and play Stratego against another human player or against the agents.

We have tested Stratesys by letting the agents play against a human player. The experiments have resulted in some valuable ideas about our multi-agent approach. First of all, the results of our experiments support our hypothesis, which stated that for some complex problems distributed techniques have a more intuitive solution. We have shown that playing the game with multiple agents is an excellent approach to break down complexity of the game.

8.2 Future work

The future work that needs to be done to continue the investigation can be divided into several categories. First of all, several improvements to the framework can be made. As stated before, we have implemented the basic functionality of the design. An important part that still has to be done is adding extra functionality to the agents by specifying a communication scheme with which they can exchange information. Additionally, the communication should be a means for cooperation between the agents.

One of the aspects of the agents that can be extended are the rule-sets. First of all we want to have more functionality in the rule-sets. Extra rules are needed for communication and cooperation. Furthermore, to make the framework suitable for modeling agents as individuals, the framework should be made more flexible by adding functionality regarding parameter settings for the agents. Examples of such parameters are the behavior scores. Manipulating the scores will make interesting experiments possible.

Using the rule-sets that we have implemented the agents do not know how to play the end game. This requires specific knowledge about how to play it. The rule-set that we have implemented is not enough for the end-game, therefore extra rules have to be used. Filling in the message-passing mechanism will be an important demand for the extra rules. Hereby the agents can also communicate information across the army which means that they are not only dependent on their visual perception. Rule-sets could also be made temporal. We can make specific rules for the opening, mid game and end game.

Additionally, the Server should have extended functionality, to make it more suitable for the corpus-based approach to build up Stratego libraries. Currently it can only handle two players at the time. We would prefer to allow for many more Stratego players, logging on and off to the system. Also the Server does not have the ability to record games yet. These additions to the Server will be valuable assets to its functionality, and will make building Stratego-libraries possible.

Appendix A

Preference rules

In this chapter all preference rules will be given in tables. The tables contain the following abbreviations and symbols:

- A bullet •, which means that the corresponing condition for the rule has to be met.
- A hyphen -, which means that the corresponding condition for the rule is **not** to be met.
- An empty field signifies a 'dont't care', the corresponding condition may or may not be met.
- The abbreviations of the ranks:

  Sp The spy, (rank 2).

  Sc The scout, (rank 3).

  Mn The miner, (rank 4).

  Sg The sergeant, (rank 5).

  Lt The lieutenant (rank 6).

  Cp The captain (rank 7).

  Mj The major (rank 8).

  Gn The general (rank 9).

Table A.1: Preference rules of the general

general						
rule #	enemy spy captured	I have moved	my rank revealed	enemy distance 1	enemy distance 2	preference
1	•		•	U		**attack**
2	•		•		U	**attack**
5	•		•	Gn		**attack**
6	•		•		Gn	**attack**
7	•		•	L		**attack**
8	•		•		L	**attack**
9	•		-	Gn		**stay**
10	•		-		Gn	**attack**
11	-			U		**attack**
12	-				U	**stay**
13				Bm		**avoid**
14					Bm	**avoid**
15						**wander**

Table A.2: Preference rules of the major

major						
rule #	enemy general captured	I have moved	my rank revealed	enemy distance 1	enemy distance 2	preference
1	•	•	•	U		**attack**
2	•	•	•		U	**approach**
5	•	•	•	Bm		**avoid**
6	•	•	•		Bm	**avoid**
7	•	•	•	Mj		**stay**
8	•	•	•		Mj	**stay**
9	•	•	•	L		**attack**
10	•	•	•		L	**approach**
11	-	•	•	U		**attack**
12	-	•	•		U	**approach**
13	-	•	•	H		**flee**
14	-	•	•		H	**flee**
15	-	•	•	Bm		**avoid**
16	-	•	•		Bm	**avoid**
17	-	•	•	Mj		**stay**
18	-	•	•		Mj	**stay**
19	-	•	•	L		**attack**
20	-	•	•		L	**approach**
31	-	•	-	U		**attack**
32	-	•	-		U	**approach**
33	-	•	-	H		**stay**
34	-	•	-		H	**approach**
35	-	•	-	Bm		**avoid**
36	-	•	-		Bm	**avoid**
35	-	•	-	Mj		**stay**
36	-	•	-		Mj	**approach**
39	-	•	-	L		**attack**
40	-	•	-		L	**approach**
41	-	-	-	U		**attack**
42	-	-	-		U	**approach**
43	-	-	-	H		**stay**
44	-	-	-		H	**approach**
45	-	-	-	Mj		**stay**
46	-	-	-		Mj	**stay**
47	-	-	-	L		**attack**
48	-	-	-		L	**approach**
49						**wander**

Table A.3: Preference rules of the captain

captain					
rule #	I have moved	my rank revealed	enemy distance 1	enemy distance 2	action
1	•	•	U		**stay**
2	•	•		U	**stay**
3	•	•	H		**flee**
4	•	•		H	**flee**
5	•	•	Bm		**avoid**
6	•	•		Bm	**avoid**
7	•	•	Cp		**stay**
8	•	•		Cp	**stay**
9	•	•	L		**attack**
10	•	•		L	**approach**
11	•	-	U		**attack**
12	•	-		U	**approach**
13	•	-	H		**stay**
14	•	-		H	**approach**
15	•	-	Bm		**avoid**
16	•	-		Bm	**avoid**
15	•	-	Cp		**stay**
16	•	-		Cp	**approach**
19	•	-	L		**attack**
20	•	-		L	**approach**
21	-	-	U		**stay**
22	-	-		U	**stay**
23	-	-	H		**stay**
24	-	-		H	**approach**
25	-	-	Cp		**stay**
26	-	-		Cp	**stay**
27	-	-	L		**attack**
28	-	-		L	**approach**
29					**wander**

Table A.4: Preference rules of the lieutenant

lieutenant					
rule #	I have moved	my rank revealed	enemy distance 1	enemy distance 2	preference
1	•	•	U		**attack**
2	•	•		U	**approach**
3	•	•	H		**flee**
4	•	•		H	**flee**
5	•	•	Bm		**avoid**
6	•	•		Bm	**avoid**
7	•	•	Lt		**stay**
8	•	•		Lt	**stay**
9	•	•	L		**attack**
10	•	•		L	**approach**
11	•	-	U		**attack**
12	•	-		U	**approach**
13	•	-	H		**stay**
14	•	-		H	**approach**
15	•	-	Bm		**avoid**
16	•	-		Bm	**avoid**
15	•	-	Lt		**stay**
16	•	-		Lt	**approach**
19	•	-	L		**attack**
20	•	-		L	**approach**
21	-	-	U		**attack**
22	-	-		U	**approach**
23	-	-	H		**stay**
24	-	-		H	**approach**
25	-	-	Lt		**stay**
26	-	-		Lt	**stay**
27	-	-	L		**attack**
28	-	-		L	**approach**

Table A.5: Preference rules of the sergeant

sergeant					
rule #	I have moved	my rank revealed	enemy distance 1	enemy distance 2	preference
1	•	•	U		**attack**
2	•	•		U	**approach**
3	•	•	H		**flee**
4	•	•		H	**stay**
5	•	•	Sg		**stay**
6	•	•		Sg	**stay**
7	•	•	B		**avoid**
8	•	•		B	**avoid**
9	•	•	L		**attack**
10	•	•		L	**approach**
11	•	-	U		**attack**
12	•	-		U	**approach**
13	•	-	H		**stay**
14	•	-		H	**stay**
15	•	-	Sg		**stay**
16	•	-		Sg	**approach**
17	•	-	B		**avoid**
18	•	-		B	**avoid**
19	•	-	L		**attack**
20	•	-		L	**approach**
21	•	-	U		**attack**
22	•	-		U	**approach**
23	•	-	H		**stay**
24	•	-		H	**stay**
25	•	-	Sg		**stay**
26	•	-		Sg	**approach**
27	•	-	B		**avoid**
28	•	-		B	**avoid**
29	•	-	L		**attack**
30	•	-		L	**approach**
31	-	-	U		**attack**
32	-	-		U	**approach**
33	-	-	H		**stay**
34	-	-		H	**stay**
35	-	-	Sg		**stay**
36	-	-		Sg	**stay**
39	-	-	L		**attack**
40	-	-		L	**approach**
41					**wander**

Table A.6: Preference rules of the miner

miner						
rule	enemy bombs captured	I have moved	my rank revealed	enemy distance 1	enemy distance 2	preference
1	•	•	•	U		**attack**
2	•	•	•		U	**stay**
3	•	•	•	H		**flee**
4	•	•	•		H	**flee**
5	•	•	-	U		**attack**
6	•	•	-		U	**attack**
7	•	•	-	H		**stay**
8	•	•	-		H	**stay**
9	•	-	-	U		**stay**
10	•	-	-		U	**stay**
11	•	-	-	H		**stay**
12	•	-	-		H	**stay**
13	-	•	•	U		**flee**
14	-	•	•		U	**stay**
15	-	•	•	H		**flee**
16	-	•	•		H	**flee**
17	-	•	-	U		**flee**
18	-	•	-		U	**stay**
19	-	-	-	U		**stay**
20	-	-	-		U	**stay**
21	-	-	-	H		**stay**
22	-	-	-		H	**stay**
23			-	L		**stay**
24			-		L	**stay**
25			•	L		**attack**
26			•		L	**attack**
27				Bm!		**attack**
28					Bm!	**attack**
29						**wander**

Table A.7: Preference rules of the scout

scout					
rule	I have moved	my rank revealed	enemy distance 1	enemy distance 2	preference
1		•	U		**attack**
2		•		U	**approach**
3		•	H		**flee**
4		•		H	**flee**
5	•	-	U		**attack**
6	•	-		U	**approach**
7	•	-	H		**flee**
8	•	-		H	**stay**
9	-	-	U		**attack**
10	-	-		U	**approach**
11	-	-	H		**stay**
12	-	-		H	**stay**
13					**find enemy**

Table A.8: Preference rules of the spy

spy						
rule	enemy general captured	I have moved	my rank revealed	enemy distance 1	enemy distance 2	preference
1	•	•		U		**attack**
2	•	•			U	**approach**
3	•	•		Sp		**attack**
4	•	•			Sp	**approach**
5	•	-		U		**attack**
6	•	-			U	**stay**
9	-	•		Gn		**attack!**
10	-	•			Gn	**stay**
11	-	-		Gn		**attack!**
12	-	-			Gn	**stay**
13	-	•		U		**flee**
14	-	•			U	**flee**
15	-	•		Bm		**avoid**
16	-	•			Bm	**avoid**
17	-	-		U		**stay**
18	-	-			U	**stay**

Appendix B

The agents' source code

```
; agent.clp
; the general rule-engine of an agent
;
; This file contains several general definitions
; that apply to all ranks. The code contains a
; defclass that is an external address, which
; makes exchange of data between Jess and Java
; possible.
;
; The actions that can be used are:
;   flee
;   attack
;   attack-general
;   attack-bomb
;   stay
;   avoid
;   wander
;
; Upon asserting an action, the defclass (a Java-bean)
; will be updated by asserting the corresponding
; score.
(defclass scores agents.Scores)

; In the java source, the bean is passed to Jess as follows:
;
; r.store("SCORES", new Scores());
; r.executeCommand("(bind ?s (fetch SCORES))");
;
; Following we can use the following jess commands:
;       (call ?s getCenter)
```

```
;       (call ?s setCenter value)  and
; as getters and setters for the bean, where in this case
; the Scores bean has a property 'center'


; This will make sure that the globals will keep their
; values upon issuing a 'reset'
(set-reset-globals nil)

(deffacts distances
   (distance north 1)
   (distance west 1)
   (distance south 1)
   (distance east 1)
   (distance north-north 2)
   (distance north-west 2)
   (distance west-west 2)
   (distance south-west 2)
   (distance south-south 2)
   (distance south-east 2)
   (distance east-east 2)
   (distance north-east 2)
)

(deffacts flee-scores
   (flee-score north -200 50 200 50)
   (flee-score west  50 -200 50 200)
   (flee-score south 200 50 -200 50)
   (flee-score east  50 200 50 -200)
   (flee-score north-north -100 50 100 50)
   (flee-score north-west  -75 -75 75 75)
   (flee-score west-west   50 -100 50 100)
   (flee-score south-west  75 -75 -75 75)
   (flee-score south-south 100 50 -100 50)
   (flee-score south-east  75 75 -75 -75)
   (flee-score east-east   50 100 50 -100)
   (flee-score north-east  -75 75 75 -75)
)

(deffacts attack-scores
   (attack-score north 200 -50 -200 -50)
   (attack-score west  -50 200 -50 -200)
   (attack-score south -200 -50 200 -50)
   (attack-score east  -50 -200 -50 200)
```

```
   (attack-score north-north 100 -50 -100 -50)
   (attack-score north-west  75 75 -75 -75)
   (attack-score west-west   -50 100 -50 -100)
   (attack-score south-west  -75 75 75 -75)
   (attack-score south-south -100 -50 100 -50)
   (attack-score south-east  -75 -75 75 75)
   (attack-score east-east   -50 -100 -50 100)
   (attack-score north-east  75 -75 -75 75)
)

(deffacts avoid-scores
   (avoid-score north -1000 0 0 0)
   (avoid-score west  0 -1000 0 0)
   (avoid-score south 0 0 -1000 0)
   (avoid-score east  0 0 0 -1000)
)

(deffacts misc-scores
   (stay-score 250)
   (wander-score 50 25 10 25)
)

(deffacts attack-bomb-scores
   (attack-bomb-score north 1000 0 0 0)
   (attack-bomb-score west  0 1000 0 0)
   (attack-bomb-score south 0 0 1000 0)
   (attack-bomb-score east  0 0 0 1000)
)

(deffacts attack-general-scores
   (attack-general-score north 10000 0 0 0)
   (attack-general-score west  0 10000 0 0)
   (attack-general-score south 0 0 10000 0)
   (attack-general-score east  0 0 0 10000)
)

(defrule action-attack
   (attack ?enemy)
   (attack-score ?enemy ?forward ?left ?backward ?right)
   =>
   (assert (update-scores 0 ?forward ?left ?backward ?right))
)

;for the miners
```

```
(defrule action-attack-bomb
   (attack-bomb ?enemy)
   (attack-bomb-score ?enemy ?forward ?left ?backward ?right)
   =>
   (assert (update-scores 0 ?forward ?left ?backward ?right))
)

;for the spy
(defrule action-attack-general
   (attack-general ?enemy)
   (attack-general-score ?enemy ?forward ?left ?backward ?right)
   =>
   (assert (update-scores 0 ?forward ?left ?backward ?right))
)

(defrule action-flee
   (flee ?enemy)
   (flee-score ?enemy ?forward ?left ?backward ?right)
   =>
   (assert (update-scores 0 ?forward ?left ?backward ?right))
)

(defrule action-stay
   (stay)
   (stay-score ?center)
   =>
   (assert (update-scores ?center 0 0 0 0))
)

(defrule action-avoid
   (avoid ?enemy)
   (avoid-score ?forward ?left ?backward ?right)
   =>
   (assert (update-scores 0 ?forward ?left ?backward ?right))
)

(defrule action-wander
   (wander)
   (wander-score ?forward ?left ?backward ?right)
   =>
   (assert (update-scores 0 ?forward ?left ?backward ?right))
)
```

```
(defrule update-scores
   (update-scores ?c ?f ?l ?b ?r)
   =>
   (call ?s setCenter (+ (call ?s getCenter) ?c))
   (call ?s setForward (+ (call ?s getForward) ?f))
   (call ?s setLeft (+ (call ?s getLeft) ?l))
   (call ?s setBackward (+ (call ?s getBackward) ?b))
   (call ?s setRight (+ (call ?s getRight) ?r))
)
```

```
; general.clp
; the rule-engine of the general
;
; makes use of the action available in agent.clp
; available actions are:
;   flee
;   attack
;   attack-general
;   attack-bomb
;   stay
;   avoid
;   wander
;
(batch agents/agent.clp)

;these are the specific globals for the general
;upon change, call the assert-globals function
;to assert the new corresponding facts
(defglobal ?*i-have-moved* = false)
(defglobal ?*my-rank-revealed* = false)
(defglobal ?*enemy-spy-captured* = false)

(deffunction assert-globals()
    (if (eq ?*i-have-moved* true) then
        (assert (i-have-moved))
    )
    (if (eq ?*my-rank-revealed* true) then
        (assert (my-rank-revealed))
    )
    (if (eq ?*enemy-spy-captured* true) then
        (assert (my-rank-revealed))
    )
)

;these are the specific rules for the general

;preference rules 1 and 2
(defrule enemy-unknown-1
   (enemy-spy-captured)
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)
```

```
;preference rule 11
(defrule enemy-unknown-2
   (not (enemy-spy-captured))
   (enemy-unknown ?enemy)
   (distance ?enemy 1)
   =>
   (assert (attack ?enemy))
)

;preference rule 12-> beware of the spy
(defrule enemy-unknown-3
   (not (enemy-spy-captured))
   (enemy-unknown ?enemy)
   (distance ?enemy 2)
   =>
   (assert (stay))
)

;preference rules 5 and 6
(defrule enemy-general-1
   (enemy-general ?enemy)
   (my-rank-revealed)
   =>
   (assert (attack ?enemy))
)

;preference rule 9
(defrule enemy-general-2
   (enemy-general ?enemy)
   (not (my-rank-revealed))
   (distance ?enemy 1)
   =>
   (assert (stay))
)

;preference rule 10
(defrule enemy-general-2
   (enemy-general ?enemy)
   (not (my-rank-revealed))
   (distance ?enemy 2)
   =>
   (assert (attack ?enemy))
)
```

```
;preference rules 7 and 8
(defrule enemy-lower-rank
   (enemy-lower-rank)
   =>
   (assert (attack ?enemy))
)

;preference rules 13 and 14: avoid the bomb
(defrule enemy-bomb
   (enemy-bomb ?enemy)
   =>
   (assert (avoid ?enemy))
)

;preference rule 15: wander behavior
(defrule wander
   (initial-fact)
   =>
   (assert (wander))
)
```

```
; major.clp
; the rule-engine of the major
;
; makes use of the action available in agent.clp
; available actions are:
;   flee
;   attack
;   attack-general
;   attack-bomb
;   stay
;   avoid
;   wander
;
(batch agents/agent.clp)

;these are the specific globals for the major
;upon change, call the assert-globals function
;to assert the new corresponding facts
(defglobal ?*i-have-moved* = false)
(defglobal ?*my-rank-revealed* = false)

(deffunction assert-globals()
    (if (eq ?*i-have-moved* true) then
        (assert (i-have-moved))
    )
    (if (eq ?*my-rank-revealed* true) then
        (assert (my-rank-revealed))
    )
)

;these are the specific rules for the major

;preference rules 1 and 2
(defrule enemy-unknown-1
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)

;preference rules 7 and 8
(defrule enemy-higher-rank-1
   (enemy-higher-rank ?enemy)
   =>
   (assert (flee ?enemy))
```

```
)

;preference rule 9
(defrule enemy-higher-rank-2
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   (distance ?enemy 1)
   =>
   (assert (stay))
)

;preference rule 10
(defrule enemy-higher-rank-3
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   (distance ?enemy 2)
   =>
   (assert (attack ?enemy))
)


;preference rules 3 and 4
(defrule enemy-major
   (my-rank-revealed)
   (enemy-major ?enemy)
   =>
   (assert (stay))
)

;preference rule 11
(defrule enemy-major
   (not (my-rank-revealed))
   (enemy-major ?enemy)
   (distance ?enemy 1)
   =>
   (assert (stay))
)

;preference rule 12
(defrule enemy-major
   (not (my-rank-revealed))
   (enemy-major ?enemy)
   (distance ?enemy 2)
   =>
```

```
   (assert (attack ?enemy))
)

;preference rules 5 and 6
(defrule enemy-lower-rank
   (enemy-lower-rank ?enemy)
   =>
   (assert (attack ?enemy))
)

;preference rules 13 and 14: avoid the bomb
(defrule enemy-bomb
   (enemy-bomb ?enemy)
   =>
   (assert (avoid ?enemy))
)

;preference rule 17: wander behavior
(defrule wander
   (initial-fact)
   =>
   (assert (wander))
)
```

```
; captain.clp
; the rule-engine of the captain
;
; makes use of the action available in agent.clp
; available actions are:
;   flee
;   attack
;   attack-general
;   attack-bomb
;   stay
;   avoid
;   wander
;
(batch agents/agent.clp)

;these are the specific globals for the captain
;upon change, call the assert-globals function
;to assert the new corresponding facts
(defglobal ?*i-have-moved* = false)
(defglobal ?*my-rank-revealed* = false)

(deffunction assert-globals()
    (if (eq ?*i-have-moved* true) then
        (assert (i-have-moved))
    )
    (if (eq ?*my-rank-revealed* true) then
        (assert (my-rank-revealed))
    )
)

;these are the specific rules for the captain

;preference rules 1 and 2
(defrule enemy-unknown-1
   (i-have-moved)
   (my-rank-revealed)
   (enemy-unknown ?enemy)
   =>
   (assert (stay))
)

;preference rules 9 and 10
(defrule enemy-unknown-2
   (i-have-moved)
```

```
   (not (my-rank-revealed))
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)

;preference rules 13 and 14
(defrule enemy-unknown-3
   (not (i-have-moved))
   (not (my-rank-revealed))
   (enemy-unknown ?enemy)
   =>
   (assert (stay))
)



;preference rules 3 and 4
(defrule enemy-higher-rank-1
   (my-rank-revealed)
   (enemy-higher-rank ?enemy)
   =>
   (assert (flee ?enemy))
)

;preference rule 11
(defrule enemy-higher-rank-2
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   (distance ?enemy 1)
   =>
   (assert (stay))
)

;preference rule 11
(defrule enemy-higher-rank-3
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   (distance ?enemy 2)
   =>
   (assert (attack ?enemy))
)
```

```
;preference rules 5 and 6
(defrule enemy-captain
   (enemy-captain ?enemy)
   =>
   (assert (stay))
)

;preference rules 7 and 8
(defrule enemy-lower-rank-3
   (not (my-rank-revealed))
   (enemy-lower-rank ?enemy)
   =>
   (assert (attack ?enemy))
)


;preference rules 15 and 16: avoid the bomb
(defrule enemy-bomb
   (enemy-bomb ?enemy)
   =>
   (assert (avoid ?enemy))
)

;preference rule 17: wander behavior
(defrule wander
   (initial-fact)
   =>
   (assert (wander))
)
```

```
; lieutenant.clp
; the rule-engine of the lieutenant
;
; makes use of the action available in agent.clp
; available actions are:
;   flee
;   attack
;   attack-general
;   attack-bomb
;   stay
;   avoid
;   wander
;
(batch agents/agent.clp)

;these are the specific globals for the lieutenant
;upon change, call the assert-globals function
;to assert the new corresponding facts
(defglobal ?*i-have-moved* = false)
(defglobal ?*my-rank-revealed* = false)

(deffunction assert-globals()
    (if (eq ?*i-have-moved* true) then
        (assert (i-have-moved))
    )
    (if (eq ?*my-rank-revealed* true) then
        (assert (my-rank-revealed))
    )
)

;these are the specific rules for the lieutenant

;preference rules 1 and 2
(defrule enemy-unknown
   (i-have-moved)
   (my-rank-revealed)
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)


;preference rules 3 and 4
(defrule enemy-higher-rank-1
```

```
   (my-rank-revealed)
   (enemy-higher-rank ?enemy)
   =>
   (assert (flee ?enemy))
)

;preference rule 9
(defrule enemy-higher-rank-2
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   (distance ?enemy 1)
   =>
   (assert (stay))
)

;preference rule 10
(defrule enemy-higher-rank-3
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   (distance ?enemy 2)
   =>
   (assert (attack ?enemy))
)

;preference rules 5 and 6
(defrule enemy-lieutenant-1
   (my-rank-revealed)
   (enemy-lieutenant ?enemy)
   =>
   (assert (stay))
)

;preference rule 15
(defrule enemy-lieutenant-2
   (not (my-rank-revealed))
   (enemy-lieutenant ?enemy)
   (distance ?enemy 1)
   =>
   (assert (stay))
)

;preference rule 16
(defrule enemy-lieutenant-3
   (not (my-rank-revealed))
```

```
   (enemy-lieutenant ?enemy)
   (distance ?enemy 2)
   =>
   (assert (attack))
)

;preference rules 13 and 14: avoid the bomb
(defrule enemy-bomb-1
   (enemy-bomb ?enemy)
   =>
   (assert (avoid ?enemy))
)

;preference rule 15: wander behavior
(defrule wander
   (initial-fact)
   =>
   (assert (wander))
)
```

```
; sergeant.clp
; the rule-engine of the sergeant
;
; makes use of the action available in agent.clp
; available actions are:
;   flee
;   attack
;   attack-general
;   attack-bomb
;   stay
;   avoid
;   wander
;
(batch agents/agent.clp)

;these are the specific globals for the sergeant
;upon change, call the assert-globals function
;to assert the new corresponding facts
(defglobal ?*i-have-moved* = false)
(defglobal ?*my-rank-revealed* = false)
(defglobal ?*enemy-bombs-captured* = false)

(deffunction assert-globals()
    (if (eq ?*i-have-moved* true) then
        (assert (i-have-moved))
    )
    (if (eq ?*my-rank-revealed* true) then
        (assert (my-rank-revealed))
    )
)

;these are the specific rules for the sergeant

;preference rule 1 and 2
(defrule enemy-unknown-1
   (i-have-moved)
   (my-rank-revealed)
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)

;preference rule 9 and 10
(defrule enemy-unknown-2
```

```
   (i-have-moved)
   (not (my-rank-revealed))
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)

;preference rule 23 and 24
(defrule enemy-unknown-3
   (not (i-have-moved))
   (not (my-rank-revealed))
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)

;preference rule 3
(defrule enemy-higher-rank-1
   (i-have-moved)
   (my-rank-revealed)
   (enemy-higher-rank ?enemy)
   (distance ?enemy 1)
   =>
   (assert (flee ?enemy))
)

;preference rule 4
(defrule enemy-higher-rank-2
   (i-have-moved)
   (my-rank-revealed)
   (enemy-higher-rank ?enemy)
   (distance ?enemy 2)
   =>
   (assert (stay))
)

;preference rules 9 and 10
(defrule enemy-higher-rank-3
   (i-have-moved)
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   =>
   (assert (stay))
)
```

```
;preference rules 15 and 16
(defrule enemy-higher-rank-4
   (not (i-have-moved))
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   =>
   (assert (stay))
)

;preference rules 5 and 6
(defrule enemy-sergeant-1
   (i-have-moved)
   (my-rank-revealed)
   (enemy-sergeant ?enemy)
   =>
   (assert (attack ?enemy))
)

;preference rule 11 and 17
(defrule enemy-sergeant-2
   (not (my-rank-revealed))
   (enemy-sergeant ?enemy)
   (distance ?enemy 1)
   =>
   (assert (stay))
)

;preference rule 12
(defrule enemy-sergeant-3
   (my-rank-revealed)
   (not (my-rank-revealed))
   (enemy-sergeant ?enemy)
   (distance ?enemy 2)
   =>
   (assert (attack ?enemy))
)

;preference rule 18
(defrule enemy-sergeant-3
   (not (i-have-moved))
   (not (my-rank-revealed))
   (enemy-sergeant ?enemy)
   (distance ?enemy 2)
```

```
   =>
   (assert (stay))
)

;preference rule 7 and 8
(defrule enemy-lower-rank
   (enemy-lower-rank ?enemy)
   =>
   (assert (attack ?enemy))
)

;preference rules 19 and 20: avoid the bomb
(defrule enemy-bomb-1
   (enemy-bomb ?enemy)
   =>
   (assert (avoid ?enemy))
)

;preference rule 21: wander behavior
(defrule wander
   (initial-fact)
   =>
   (assert (wander))
)
```

```
; miner.clp
; the rule-engine of the miner
;
; makes use of the action available in agent.clp
; available actions are:
;   flee
;   attack
;   attack-general
;   attack-bomb
;   stay
;   avoid
;   wander
;
(batch agents/agent.clp)

;these are the specific globals for the miner
;upon change, call the assert-globals function
;to assert the new corresponding facts
(defglobal ?*i-have-moved* = false)
(defglobal ?*my-rank-revealed* = false)
(defglobal ?*enemy-bombs-captured* = false)

(deffunction assert-globals()
    (if (eq ?*i-have-moved* true) then
        (assert (i-have-moved))
    )
    (if (eq ?*my-rank-revealed* true) then
        (assert (my-rank-revealed))
    )
    (if (eq ?*enemy-bombs-captured* true) then
        (assert (enemy-bombs-captured))
    )
)

;these are the specific rules for the miner

;preference rule 1
(defrule enemy-unknown-1
   (enemy-bombs-captured)
   (i-have-moved)
   (my-rank-revealed)
   (enemy-unknown ?enemy)
   (distance ?enemy 1)
   =>
```

```
   (assert (flee ?enemy))
)

;preference rule 2
(defrule enemy-unknown-2
   (enemy-bombs-captured)
   (i-have-moved)
   (my-rank-revealed)
   (enemy-unknown ?enemy)
   (distance ?enemy 2)
   =>
   (assert (stay))
)

;preference rules 5 and 6
(defrule enemy-unknown-3
   (enemy-bombs-captured)
   (i-have-moved)
   (not (my-rank-revealed))
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)

;preference rules 9 and 10
(defrule enemy-unknown-4
   (enemy-bombs-captured)
   (not (i-have-moved))
   (not (my-rank-revealed))
   (enemy-unknown ?enemy)
   =>
   (assert (stay))
)

;preference rule 13
(defrule enemy-unknown-5
   (i-have-moved)
   (my-rank-revealed)
   (not (enemy-bombs-captured))
   (enemy-unknown ?enemy)
   (distance ?enemy 1)
   =>
   (assert (flee ?enemy))
)
```

```
;preference rule 14
(defrule enemy-unknown-6
   (i-have-moved)
   (my-rank-revealed)
   (not (enemy-bombs-captured))
   (enemy-unknown ?enemy)
   (distance ?enemy 2)
   =>
   (assert (stay))
)

;preference rule 17
(defrule enemy-unknown-7
   (i-have-moved)
   (not (my-rank-revealed))
   (not (enemy-bombs-captured))
   (enemy-unknown ?enemy)
   (distance ?enemy 1)
   =>
   (assert (flee ?enemy))
)

;preference rule 18
(defrule enemy-unknown-8
   (i-have-moved)
   (not (my-rank-revealed))
   (not (enemy-bombs-captured))
   (enemy-unknown ?enemy)
   (distance ?enemy 2)
   =>
   (assert (stay))
)

;preference rules 19 and 20
(defrule enemy-unknown-9
   (enemy-unknown ?enemy)
   (not (i-have-moved))
   (not (my-rank-revealed))
   (not (enemy-bombs-captured))
   =>
   (assert (stay))
)
```

```
;preference rules 3 and 4
(defrule enemy-higher-rank-1
   (enemy-bombs-captured)
   (i-have-moved)
   (my-rank-revealed)
   (enemy-higher-rank ?enemy)
   =>
   (assert (flee ?enemy))
)

;preference rules 7 and 8
(defrule enemy-higher-rank-2
   (enemy-bombs-captured)
   (i-have-moved)
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   =>
   (assert (stay))
)

;preference rules 11 and 12
(defrule enemy-higher-rank-3
   (enemy-bombs-captured)
   (not (i-have-moved))
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   =>
   (assert (stay))
)

;preference rules 15 and 16
(defrule enemy-higher-rank-4
   (i-have-moved)
   (my-rank-revealed)
   (not (enemy-bombs-captured))
   (enemy-higher-rank ?enemy)
   =>
   (assert (flee ?enemy))
)

;preference rules 21 and 22
(defrule enemy-higher-rank-5
   (not (i-have-moved))
   (not (my-rank-revealed))
```

```
   (not (enemy-bombs-captured))
   (enemy-higher-rank ?enemy)
   =>
   (assert (stay))
)

;preference rule 23 and 24
(defrule enemy-lower-rank-1
   (enemy-lower-rank ?enemy)
   (not (my-rank-revealed))
   =>
   (assert (stay))
)

;preference rule 25 and 26
(defrule enemy-lower-rank-2
   (enemy-lower-rank ?enemy)
   (my-rank-revealed)
   =>
   (assert (attack ?enemy))
)

;preference rules 27: hey a bomb at distance 1!
(defrule enemy-bomb-1
   (enemy-bomb ?enemy)
   (distance ?enemy 1)
   =>
   (assert (attack-bomb ?enemy))
)

;preference rule 28: hey a bomb at distance 2!
(defrule enemy-bomb-2
   (enemy-bomb ?enemy)
   (distance ?enemy 2)
   =>
   (assert (attack ?enemy))
)

;preference rule 29: wander behavior
(defrule wander
   (initial-fact)
   =>
   (assert (wander))
)
```

```
; scout.clp
; the rule-engine of the scout
;
; makes use of the action available in agent.clp
; available actions are:
;   flee
;   attack
;   attack-general
;   attack-bomb
;   stay
;   avoid
;   wander
;
(batch agents/agent.clp)

;these are the specific globals for the spy
;upon change, call the assert-globals function
;to assert the new corresponding facts
(defglobal ?*i-have-moved* = false)
(defglobal ?*my-rank-revealed* = false)

(deffunction assert-globals()
    (if (eq ?*i-have-moved* true) then
        (assert (i-have-moved))
    )
    (if (eq ?*my-rank-revealed* true) then
        (assert (my-rank-revealed))
    )
)

;these are the specific rules for the scout

;preference rules 1 and 2
(defrule enemy-unknown-1
   (my-rank-revealed)
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)

;preference rules 5 and 6
(defrule enemy-unknown-2
   (i-have-moved)
   (not (my-rank-revealed))
```

```
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)

;preference rules 9 and 10
(defrule enemy-unknown-3
   (not (my-rank-revealed))
   (not (i-have-moved))
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)

;preference rules 3 and 4
(defrule enemy-higher-rank-1
   (my-rank-revealed)
   (enemy-higher-rank ?enemy)
   =>
   (assert (flee ?enemy))
)

;preference rule 7
(defrule enemy-higher-rank-2
   (not (my-rank-revealed))
   (i-have-moved)
   (enemy-higher-rank ?enemy)
   (distance ?enemy 1)
   =>
   (assert (flee ?enemy))
)

;preference rule 8
(defrule enemy-higher-rank-3
   (not (my-rank-revealed))
   (i-have-moved)
   (enemy-higher-rank ?enemy)
   (distance ?enemy 2)
   =>
   (assert (stay))
)

;preference rule 9 and 10
(defrule enemy-higher-rank-3
```

```
   (not (my-rank-revealed))
   (not (i-have-moved))
   (enemy-higher-rank ?enemy)
   =>
   (assert (stay))
)

;preference rule 13:always watch out for the bomb!
(defrule enemy-bomb
   (enemy-bomb ?enemy)
   (distance ?enemy 1)
   =>
   (assert (avoid ?enemy))
)

;preference rule 14: wander behavior
(defrule wander
   (initial-fact)
   =>
   (assert (wander))
)
```

```
; spy.clp
; the rule-engine of the spy
;
; makes use of the action available in agent.clp
; available actions are:
;   flee
;   attack
;   attack-general
;   attack-bomb
;   stay
;   avoid
;   wander
;
(batch agents/agent.clp)

;these are the specific globals for the spy
;upon change, call the assert-globals function
;to assert the new corresponding facts
(defglobal ?*i-have-moved* = false)
(defglobal ?*enemy-general-captured* = false)

(deffunction assert-globals()
    (if (eq ?*i-have-moved* true) then
        (assert (i-have-moved))
    )
    (if (eq ?*enemy-general-captured* true) then
        (assert (enemy-general-captured))
    )
)

;these are the specific rules for the spy
(defrule enemy-unknown-1
   (enemy-unknown ?enemy)
   (i-have-moved)
   =>
   (assert (flee))
)

; preference rule 6 (stay)
(defrule enemy-unknown-2
   (enemy-unknown ?enemy)
   (not (i-have-moved))
   =>
   (assert (stay)
```

```
   )
)

;hey the general is at distance 1-->attack him!
(defrule enemy-general-1
   (enemy-general ?enemy)
   (distance ?enemy 1)
   =>
   (assert (attack ?enemy))
)

;we haven't moved yet and the enemy general is at distance 2,
;the spy definately wants to stay!
(defrule enemy-general-2
   (not (i-have-moved))
   (enemy-general ?enemy)
   (distance ?enemy 2)
   =>
   (assert (stay))
)

;the general is at distance 2, the spy may want to stay foot
(defrule enemy-general-2
   (i-have-moved)
   (enemy-general ?enemy)
   (distance ?enemy 2)
   =>
   (assert (stay))
)

;always watch out for the bomb!
(defrule enemy-bomb
   (enemy-bomb ?enemy)
   (distance ?enemy 1)
   =>
   (assert (avoid ?enemy))
)
```

Bibliography

Brooks, R. A. (1986). A robust layered control system for a mobile robot, *IEEE Journal of Robotics and Automation* **RA-2**: 14–23.

Cañamero, D. (1997). Modeling motivations and emotions as a basis for intelligent behavior, *Proceedings of the First International Symposium on Autonomous Agents, AA'97*, The ACM Press.

Ferber, J. (1999). *Multi-Agent Systems, An Introduction to Distributed Artificial Intelligence*, Addison Wesley, England. (translated from the French "Les Systèmes Multi-Agents: Vers une Intelligence Collective", 1995).

Finin, T., Labrou, Y. & Mayfield, J. (1997). Kqml as an agent communication language, *Software Agents*, MIT Press, Cambridge. (to appear).

Franklin, S. & Graesser, A. (1996). Is it an agent, or just a program?: a taxonomy for autonomous agents, *Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages*, Springer-Verlag.

Freeman, E. et al. (1999). *JavaSpaces™Principles, Patterns and Practice*, Addison-Wesley.

Friedman-Hill, E. J. (2000). Jess, the java expert system shell, *Sand98-8206 (revised)*, Sandia National Laboratories, Livermore, CA.

Gillogly, J. (1972). The technology chess program, *Artificial Intelligence* **3(1–4)**: 145–163.

Jarmulak, J. (1999). *Case-Based Reasoning for NDT Data Interpretation*, PhD thesis, Delft University of Technology.

Jeon, H., Petrie, C. & Cutkosky, M. R. (2000). Jat*Lite*: A java agent infrastructure with message routing, *IEEE Internet Computing* **Mar/Apr**.

Keijzer, F. A. (1999). Intelligentie zit niet in je hoofd, *De Psycholoog, Wetenschap* .

Lange, D. B. & Oshima, M. (1998). *Programming and Deploying Mobile Agents with Java*, Addison-Wesley, Reading, MA.

Maes, P. (1994). Agents that reduce work and information overload, *Communications of the ACM* **37, 7**.

Maes, P. (1995). Artificial life meets entertainment: Life like autonomous agents, *Communications of the ACM* **38, 11**: 108–114.

Marsland, T. (1983). Relative efficiency of alpha-beta implementations, *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pp. 763–766.

Marsland, T. (1986). A review of game tree pruning, *ICCA Journal* **9(1)**: 3–9.

McCauley, T. L. & Franklin, S. (1998). An architecture for emotion, *AAAI 1998 Fall Symposium "Emotional and Intelligent: The Tangled Knot of Cognition"*, AAAI Press.

McFarland, D. (1994). Towards robot cooperation, *From Animals to Animats 3, Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, MIT Press, pp. 440–444.

Minsky, M. (1985). *The Society of Mind*, NY: Simon & Schuster.

Pearl, J. (1980). Asymptotic properties of minimax tress and game searching procedures, *Aritificial Intelligence* **14(2)**: 113–138.

Russel, S. & Norvig, P. (1995). *Artificial Intelligence – A Modern Approach*, Englewood Cliffs, NJ: Prentice Hall.

Samuel, A. (1959). Some studies in machine learning using the game of checkers, *IBM Journal of Research & Development* **3**: 210–229.

Schaeffer, J. (1983). The history heuristic, *ICCA Journal* **6(3)**: 16–19.

Steels, L. (1995). When are robots intelligent autonomous agents?, *Robotics and Autonomous Systems* **15**: 3–9.

Steels, L. (1997). A selectionist mechanism for autonomous behavior acquisition, *Robotics and Autonomous Systems* **20**: 117–131.