A comparison study of twelve paradigms for developing embodied agents

Ladislau Bölöni, Linus J. Luotsinen, Joakim N. Ekblad, T. Ryan Fitz-Gibbon, Charles Houchin, Justin Key, Majid Ali Khan, Jin Lyu, Johann Nguyen, Rex Oleson, Gary Stein, Scott Vander Weide, and Viet Trinh

Corresponding author:

Ladislau Bölöni

School of Electrical Engineering and Computer Science

P.O. Box 162450

Orlando FL 32816-2450

tel: 407-823-2320

fax: 407-823-5835

e-mail: lboloni@eecs.ucf.edu

Abstract

We report on a study in which twelve different paradigms were used to implement agents acting in an environment which borrows elements from artificial life and multi-player strategy games. In choosing the paradigms we strived to maintain a balance between high level, logic based approaches and low level, physics oriented models; between imperative programming, declarative approaches and "learning from basics"; between anthropomorphic or biologically inspired models on one hand and pragmatic, performance oriented approaches on the other.

We have found that the choice of the paradigm determines the software development process and requires a different set of skills from the developers. In terms of raw performance, we found that the best performing paradigms were those which (a) allowed the knowledge of human experts to be explicitly transferred to the agent and (b) allowed the integration of well-known, high performance algorithms. We have found that maintaining a commitment to the chosen paradigm can be difficult; there is a strong temptation to offer shallow fixes to perceived performance problems through a "flight into heuristics". Our experience is that a development process without the discipline enforced by a central paradigm leads to agents which are a random collection of heuristics whose interactions are not clearly understood.

Although far from providing a definitive verdict on the benefits of the different paradigms, our study provided a good insight into what kind of conceptual, technical or organizational problems would a development team face depending on their choice of agent paradigm.

I. Introduction

Researchers have designed a bewildering variety of paradigms for the control of agents. Virtually every paradigm of artificial intelligence, software engineering or control theory was deployed, with more or less success. However, wide ranging comparisons of agent paradigms are rare. When new methods and paradigms are introduced, they are compared with only several, closely related approaches which are considered direct competitors of the proposed paradigm. Making or revisiting comparisons between paradigms is a controversial, difficult and hard-to-sell work [7]. One might argue that a researcher might better spend his or her time designing new paradigms or improving existing ones instead of comparing, say, swarm algorithms with affective computing. There might be people offended by the results, with reasonable claims that the methodology was incorrect, the implementation of the paradigm substandard, or simply, the measured quantity is not relevant to the given paradigm.

The fundamental question, of course, is whether any of these comparisons makes sense. We argue that **if both paradigms A and B can be used in the implementation of the same requirements, then these two paradigms can (and indeed, should be) compared**. That is not to say that the comparison is easy or that it can be reduced to a single numerical "score". Different paradigms have different strengths and weaknesses, and the goal of a comparison study is to shed light on the particulars of the different approaches. Although we do not expect definite answers to questions like "which paradigm can create an agent which passes the Turing test", we can provide insight into lesser but still important questions such as:

- Can a rigorous software engineering process be applied to the development?
- Can the performance be predicted?
- Can human expertise in the problem domain be transferred to the agent?
- Can the implementation provide adequate performance?
- How much development effort and what level of competence is required for an adequate implementation of the paradigm?
- Will the resulting agent be predictable in its actions?

The remainder of this paper is organized as follows. In Section II we present the Feed-Fight-Multiply game, our control problem. We succinctly describe the twelve paradigms and the implemented agents in Section III and we show the software engineering metrics of the implementation efforts in Section IV. Numerical performance results are detailed in Section V, while qualitative findings are presented in Section VI. We conclude in Section VII.

II. The Feed-Fight-Multiply world

To study the benefits and drawbacks of various agent paradigms, we decided to place them in a virtual environment which reflects many of the challenges encountered by agents in the real world.

The types of systems which are the primary focus of this paper are *embodied agents*, that is, agents whose environment is either the physical world, or a simulation of it. Examples of the first are unmanned aerial or ground vehicles, or mobile robots. Examples where the environment is a simulation of the real world are avatars in virtual worlds, game characters and so on. An interesting example of agents whose environments include both the real and the virtual world is the one described in [27], where embodied agents (animated humanoid or animal characters) can move inside their virtual environment, but they can also jump between devices in the real world.

We decided against using one of the existing competitive environments (such as the RoboCup Simulation League) because the existence of previous, optimized implementations would have skewed the result of the comparison. Furthermore, we required that the environment provide *multiple paths to success*. We expected that agents implemented in various paradigms will have a different external behavior as well. Measuring success as the conformance to a predefined behavior would have favored some paradigms and disadvantaged others. In addition, having multiple paths to success is a quality of most natural environments and many artificial ones.

Upon these considerations, we implemented the Feed-Fight-Multiply game, which borrows elements from turn-based multi-player strategy games and artificial life. Agents are moving in a two-dimensional environment having accessible zones and obstacles. The agents can sense the environment within the range of their sensors. Food resources appear at random points in the environment; consuming food increases the energy level of the agents. Agents can attack each other, by destroying an agent, the attacker gains access to its resources. Finally, agents can multiply by (non-sexual) reproduction.

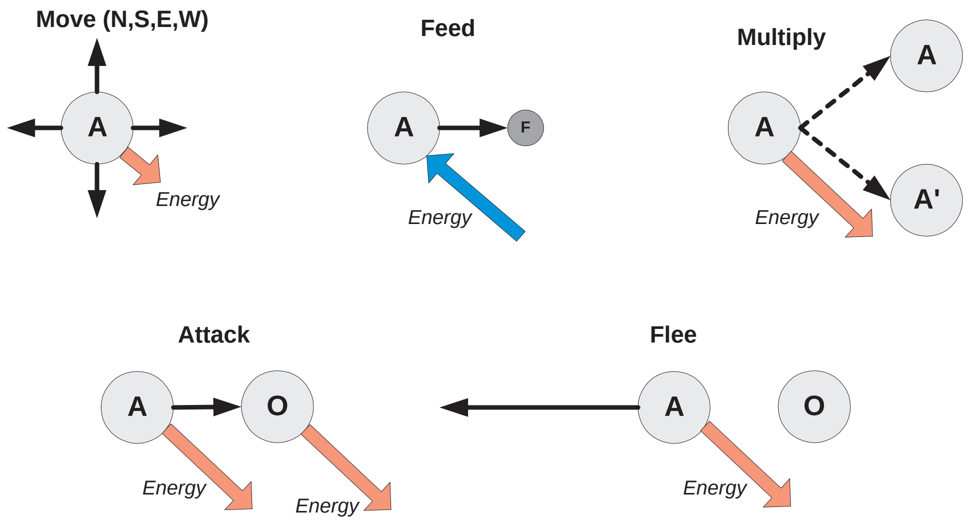

Fig. 1. The possible actions of the agents in the Feed-Fight-Multiply game, as well as the relative energy consumption or gain of the different actions. The exact values of the energy consumptions are parameterized.

The environment was implemented in the Java based YAES simulation environment [3]. To simplify the implementation of the agents, we decided that the most commonly used functionality will be implemented in the environment, and provided as a service to the agents. These services included the scanning of the sensor range for agents and food, tracking of moving agents and identifying the types of the agents in the sensor range. Our goal was to compare the paradigms as implementations of high level agent functions. These low level functions would have been implemented in any agent using imperative programming; providing standard implementations for them allowed a more fair comparison of the paradigms.

Finally, instead of keeping a single score, we decided to record multiple parameters of the agent behavior. This meant that not only there were multiple paths to success, but the final goals of the agents could be different as well. Of course, all agents were required to work towards their survival, but besides that, the criteria for success could be maximum amount of resources gathered, survival on minimum amount of resources, largest number of agents killed, number of individual agents of the same type at the end of the game, or others. Figure 2 shows a typical FFM game in progress.

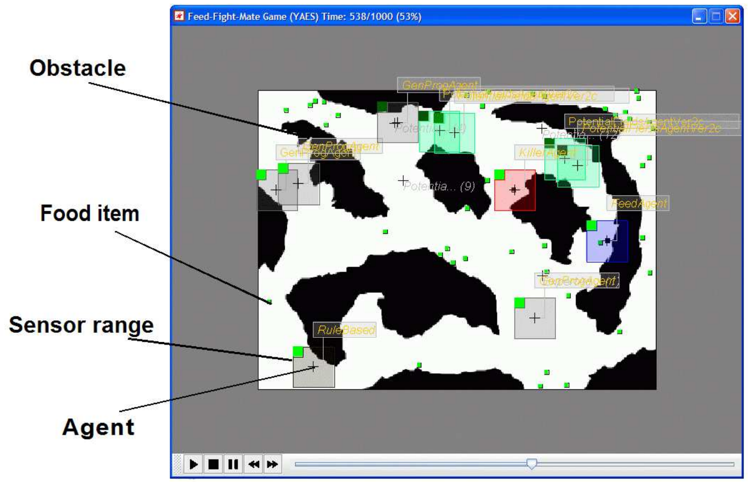

Fig. 2. Screenshot of the Feed-Fight-Multiply environment. The evolution of the game can be followed either in real time, or replayed from the game logs.

We need to discuss our choice of the actions in the FFM game, in particular the lack of the explicit communication actions (such as message send, receive and broadcast). The FFM agents communicate only implicitly through movement and actions. For instance, collaboration can happen by respecting each others feeding range, attacking common enemies and so on.

We did not include explicit communication actions, because they would have made the comparison of the paradigms more difficult. If we would have allowed communication, for instance, through FIPA ACL (or other speech act based communication models), the balance of the comparison would have clearly shifted towards the formally programmed approaches. Many of the low level, paradigm-pure approaches would not have been possible. To work around this, all the agents would have been composed of a high level part, controlling the communication of the agent, and another, low or high level part, implementing the chosen paradigm. This would have made the comparisons less relevant, as the chosen paradigm would have played a smaller role both in the performance and the implementation effort of the agents.

Naturally, this does not mean that such a comparison would not be relevant and interesting. It is of considerable interest whether a certain paradigm can be easily interfaced with a messaging system, and what role can a paradigm play in such a system. For instance, the paradigm can be the decision maker, low level movement control, or simply an internal input which influences of the priorities of various goals. One way to do such a comparison would be to standardize the high level messaging component of the agents. Such a study, however, is outside the scope of this paper.

III. TWELVE AGENTS, TWELVE PARADIGMS

We have developed twelve agents, implemented in twelve different paradigms of agency. In choosing the paradigms we strived to maintain a balance between high level, logic based approaches and low level, physics oriented models; between imperative programming, declarative approaches and "learning from basics"; between anthropomorphic or biologically inspired models on one hand and pragmatic, performance oriented approaches on the other. The implemented agents are concisely described in Table I. The developers were instructed to develop paradigm-pure implementations and to design the agents such that the "spirit" of the paradigm is best expressed. When the paradigm could not cover the required functionality, the developers were allowed to complement it with simple heuristics.

The goal of the developers was to create agents which perform well in a wide variety of game scenarios. Two major types of scenarios were used. In the *non-competitive scenarios* the agents are acting on a map populated with a small number of agents of the same type. In *competitive scenarios*, the map contains two equal size teams of opponent agents. It was left to the choice of the developer to decide whether the agent will implement teamwork behavior. Some agent paradigms such as crowd modeling, presume team interactions, while others, such as social potential fields, strongly facilitate it. In general all the agents were using different social behavior against agents of their own type as opposed to agents of different type.

In the following subsections, we describe the implementation of every agent according to the following framework. First, we succinctly summarize the paradigm, its origins, typical use and the papers or applications which served as the source for our implementation. In the "Implementation" subsection we describe the way in which the agent was implemented in paradigm, while in the "Heuristics" subsection, we describe the heuristics added to the implementation and the reasons which made this addition necessary. The "Results" subsection summarizes the qualitative results or the implementation, in the perception of the developer. The focus of this subsection is how successfully the paradigm allowed the implementation of the agent behavior envisioned by the developer.

Finally in the "Development effort" subsection, we summarize the ways in which the time of the developers was spent in the development process. Considering the particularities of the FFM agent development, we categorize the time spent by the developers in four classes.

Paradigm implementation. The amount of time spent by the developer to implement or adapt a standard implementation of the paradigm. This part of the effort can be, of course, greatly reduced if there is a high quality library. Developers might also reuse their own previous implementations for new agents. In many cases, this effort requires the translation of the pseudocode provided by the foundational papers (or standard textbooks) of the paradigm into the development environment and language.

Agent implementation. This part of the development effort covers the expression of the problem domain in the terms of the paradigm. It includes the explicit design and development work. For many explicitly programmable paradigms, this was the main part of the development effort. For instance, for a rule based agent, agent implementation involves the writing of the rules which determine the agent behavior. For learning-based paradigms, this part of the development effort is usually much smaller. For instance, for a genetic programming model, the agent implementation part covers only the design and implementation of the fitness function.

Heuristics. This part of the development effort covers the development and testing of the heuristics added to the agent to either complement the paradigm, or correct certain behaviors which can not be conveniently expressed in the paradigm. Some paradigms cover only some parts of the behavior of the agent. For instance, social potential fields cover only the movement of the agent, while game theory only the encounters between two agents. In these cases significant development effort goes to the heuristics complementing these paradigms.

Learning and tuning. We will use the term "learning" for the automatic acquisition of knowledge through techniques such as neural networks, reinforcement learning or genetic programming. We will use the term "tuning" for the manual adjustment of the parameters of the agent based on the observation of the behavior by the developer. Note that, in practice, the learning process can rarely be fully automatized. The developer needs to create appropriate scenarios in which the agent can have meaningful learning experiences, it needs to adjust the learning parameters, in many cases, it needs to restart the learning process from scratch.

A. AffectiveAgent: anthropomorphic and affective model

Affective computing [16] proposes the consideration of human-like emotions in the implementation of computing artifacts. The use of affective agents is immediately justified in agents interacting directly with human operators. Recognizing and responding to the human operators' emotions, as well as expressing

TABLE I

CONCISE DESCRIPTION OF THE TWELVE IMPLEMENTED AGENTS

Name	**Paradigm**	**Paradigm coverage**	**Team-work**	**Offline learning**
AffectiveAgent	Affective model, anthropomorphic lifecycle	Limited	No	No
GenProgAgent	Genetic programming	Full	Yes	Yes
Reinforcer	Reinforcement learning	Full	Yes	Yes
CBRAgent	Case based reasoning	Full	No	Yes
RuleBasedAgent	Forward reasoning	Full	Yes	No
NaïveAgent	Naïve programming (scripting)	Full	Yes	No
GamerAgent	Game theory	Limited	Yes	No
CrowdAgent	Particle based crowd modeling	Limited	Yes	No
NeuralLearner	Backpropagation neural networks	Full	No	Yes
SPFAgent	Social potential fields	Limited	Yes	No
CxBRAgent	Context based reasoning	Full	No	No
KillerAgent	Minimal heuristic	Full	No	No

emotions on its own can significantly improve the communication between the agent and user. The use of affective agents in settings without human participants is more controversial; one might question whether we can even speak of emotions in this context at all [22]. What is certain, is that we can design our agents in such a way that the agent possess an emotional frame of reference to weight its decisions. The hope is that the decisions made in this frame of reference would be just as good (or better) than the ones obtained through other techniques.

In broad lines, our implementation is an adaptation of the agents from [23]. We have decided to implement not only emotional states, but also an anthropomorphic life cycle. Agents have emotions such as anger, contentment or fear. In addition, the agents mimic the stages of human life: they have a childhood, maturity and old age, with their corresponding goals and priorities. The affective model plays two roles in the behavior of the agent: *action selection* (e.g., what to do next based on the current emotional state) and *adaptation* (e.g., short or long-term changes in behavior due to the emotional states). As the affective programming paradigm does not cover low level actions such as avoiding obstacles, reaching food and so on, these features were implemented through heuristics.

The short term variables which control the behavior of the agent are the *action tendency* and the *conflict tendency*. The dynamic action tendency is the probability whether an agent will fight or flee in a given situation. To adapt the action tendency to the outcome of the agent's interactions, the action tendency is updated by the *adaptation rule* depending whether the agent is experiencing loss or success. The conflict tendency determines whether an agent seeks conflicts or avoids them. Emotional states such as anger or fear are determined in terms of ranges of the action and conflict tendency.

Implementation: Let us describe the strategy of the affective agent in anthropomorphic terms. The goal of the agent is to multiply and overcome other agents by direct fight and by starving them of resources. The agents will pass through a period of childhood, during which they will choose safe ways to accumulate resources, by feeding, and will refrain from fighting and multiplying. A mature agent will seek opportunities to multiply (if the emotional state is content or fearful) or to attack (if the emotional state is angry). Affective agents will not attack agents of the same type.

We found that agents tend to get stuck into certain emotional states. To avoid this "emotional quagmire", we have implemented "mood swings" by periodically resetting the action tendency of the agent to a random value.

Heuristics: As the affective paradigm does not cover concepts such as motion planning or identifying food resources, this functionality was implemented using heuristics.

Results: The behavior of the affective agents matched the expectations. The visual observation of the affective agents had shown their cautious approach during childhood, followed by a more aggressive stand and high reproductive rate during adulthood. The high reproductive rate created a large number of affective agents, which in some cases successfully starved the adversary. However, affective agents frequently multiplied beyond their resources, which led to a high mortality rate. The agents were less successful in combat, because the adaptation rule did not change the mood of the agent sufficiently quickly to make correct decisions during fight.

Development effort:

Paradigm implementation (10%) The paradigm implementation was a minor part of the development effort. There are no standard data structures and algorithms associated with the affective model, thus this part of the effort covered only the generic part of the representation of the emotional states and stages of maturity.

Agent implementation (20%) This part of the effort involved choosing the emotional states considered, selecting and implementing the appropriate behaviors for each emotional state and choosing the ways in which the emotional states evolve during the game.

Heuristics (20%) As the affective model did not cover all possible behaviors of the agent, this part of the development supplanted the affective behavior.

Learning and tuning (50%) A large part of the development effort was spent in the manual tuning of the parameters of the agent, in particular the transitions between the various emotional states and the behaviors associated with them.

B. GenProgAgent: genetic programming

Genetic programming (GP) [10] is an evolution of the genetic algorithm [9], where the units of evolution, the chromosomes, represent programs. Each chromosome encodes a syntax tree which has functions or statements (conditional, loop, and so on) for its internal nodes, and constants or variables (terminals) as its leaves. The fitness function is a measure of how well the program encoded by a specific chromosome can solve the given problem.

The control parameters of genetic programming are the population size, crossover probability, mutation probability, selection method and the number of generations for which the algorithm is run. As a result of the crossover and mutation operators, the individual syntax trees can grow in size. To avoid chromosomes of excessive size, an additional parameter determines the maximum size of the tree, limiting the complexity of the algorithms which can be evolved.

Implementation: We have used a two stage approach to evolve the genetic programming agent. In the first stage, we evolve *tactical behaviors* for specific situations. In the second stage, we combine the tactical behaviors into *game strategies*.

The first stage aims to evolve the tactical behaviors of the agent in specific situations. Such behaviors could be to locate food resources, to eat them, to avoid or engage in hostile encounters with opponent agents. Note that these tactical behaviors alone can not be used to successfully play the game.

In the second stage the tactical behaviors are used to evolve game strategies. The game strategy describes the tactical behavior which the agent needs to follow in a specific situation of the game. For instance, if the fitness function favors strategies engaging in conflicts or attacking other characters in the game then the GP will prefer individuals with high attack measures and eventually it will converge to form this game strategy.

Evolving tactical behaviors

As the tactical behaviors are tied to specific circumstances in the game, we need to create a set of *situational models*, tailored to the tactical behaviors we are trying to evolve. A situational model is described by a game map, an initial configuration of food and agents and a specialized fitness function. We created models for four types of tactical behaviors: eat-food, attack, flee and wander.

TABLE II

THE LIST OF TERMINAL AND FUNCTION NODES FOR THE EAT-FOOD SITUATIONAL MODEL. TERMINAL NODES MARKED WITH (*) DEPICT NODES THAT REQUIRE SIGNIFICANT EXTERNAL PROCESSING IN THE FORM OF HELPER FUNCTIONS.

Name	Type	Description
*FoodDirection	Terminal	Direction to the closest food.
*AgentDirection	Terminal	Direction to the closest opponent agent.
*UnexploredDirection	Terminal	Direction to the closest unexplored region.
RandomDirection	Terminal	Random direction
CurrentDirection	Terminal	Current direction of movement
ObstacleNorth	Function	Returns `true` if there is an obstacle at north
ObstacleSouth	Function	Returns `true` if there is an obstacle at south
ObstacleEast	Function	Returns `true` if there is an obstacle at east
ObstacleWest	Function	Returns `true` if there is an obstacle at west
Not	Function	Logical NOT operator
If	Function	Conditional statement
Equals	Function	Operator used in conditional statements
NotEquals	Function	Operator used in conditional statements

The function set and terminals for these tactical behaviors are described in Table II. The terminal nodes marked with (*) incorporate significant external processing. For instance, the food direction terminal points to the closest food resource available to the GenProgAgent while the unexplored direction terminal is based on an external data structure which maintains a list of visited waypoints. The obstacle functions, one for each direction, return a boolean value showing whether the GenProgAgent can move in the given direction or not. The remaining functions are used as control structures, with the goal of incorporating decisions in the tree.

The situational model for the eat-food tactical behavior consisted of a 200x200 pixel game map. The map contained obstacles covering approximately 20% of the map and 50 randomly distributed food units. The goal is to evolve a behavior for the agent which does not collide with obstacles and eats all the available food resources.

The fitness function F is represented on a scale between 0 (for highest fitness) and 100 (lowest fitness) according to the following formula:

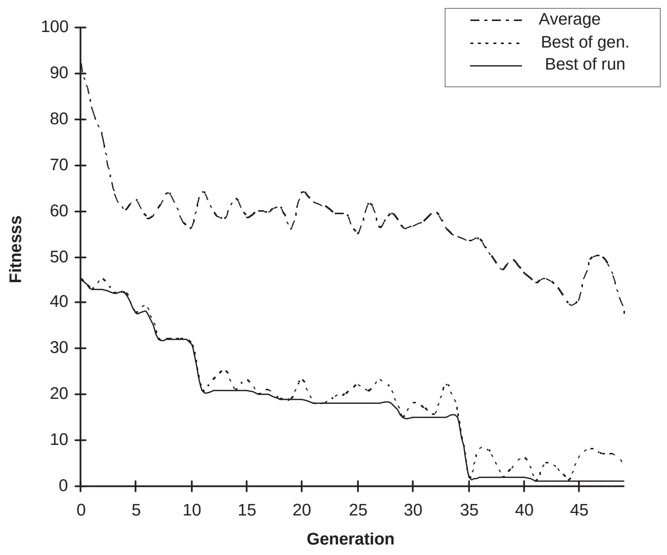

Fig. 3. The evolution of the average fitness, best individual fitness of generation and best individual fitness of run for an experiment evolving the eat-food tactical behavior.

$$F = 50\left(1 - \frac{R_c}{R_a} + \frac{1}{250000}\sum_{s\in S_{\text{failed}}} (length(s))^2\right) \tag{1}$$

where R_a is the number of available food resources, R_c the consumed food resources, and S_{failed} the set of failed move sequences.

Figure 3 shows a fitness graph over 50 generations. The best individual over the experiment was found in generation 41 and it had a fitness value of 1 (very close to the optimum). Average fitness at the last generation was 37.59 and best individual of the last generation had a fitness of 5. The randomly generated individuals in the initial population perform very poorly, with an average fitness of 92.3.

We could have repeated the same process for the other behaviors. However, the other tactical behaviors can also be generated by reusing portions of the already evolved eat-food behavior. If we replace the FoodDirection terminals with AgentDirection terminals in the chromosome we obtain an attack behavior. Notice that the dynamic nature of the terminal also changes the behavior of the agent; while in the eat-food behavior the agent is navigating towards static food items, in the attack behavior it is pursuing mobile agents. Similarly, the wander behavior can be obtained by replacing the FoodDirection terminal with the UnexploredDirection terminal, and the flee behavior by replacing FoodDirection with the reverted AgentDirection terminal.

TABLE III

THE LIST OF TERMINAL AND FUNCTION NODES FOR EVOLVING GAME STRATEGIES.

Name	Type	Description
StateWander	Terminal	Constant representing the wander behavior
StateAttack	Terminal	Constant representing the attack behavior
StateEat	Terminal	Constant representing the eat behavior
StateFlee	Terminal	Constant representing the flee behavior
StateRandom	Terminal	Constant representing any behavior
Add	Function	Addition operator
Mul	Function	Multiplication operator
Sub	Function	Subtraction operator
If	Function	Conditional statement
Equal	Function	Operand used in conditional statements
LesserEqual	Function	Operand used in conditional statements
GreaterEqual	Function	Operand used in conditional statements
Random	Function	A random value
NumberOfAgents	Function	Number of agents in sensor range
NumberOfFood	Function	Number of food resources in sensor range
EnergyLevel	Function	Energy level of the GenProgAgent

Evolving game strategies

A game strategy describes the tactical behavior which the agent needs to follow in a specific situation in the game. Game strategies can be created by combining tactical behaviors in finite state machines. With the tactical behaviors already created, the challenge is to find the appropriate *transition rules*. We used GP to evolve the transition rules for FSM-like structures. Different strategies, corresponding to agents with different goals, can be evolved by adjusting the fitness function of the GP. The GP configuration used to demonstrate this feature is shown in Table III.

The terminal nodes can be StateWander, StateAttack, StateEat and StateFlee, which represent tactical behaviors evolved in the previous step. An additional terminal node, StateRandom represents a randomly selected state and its purpose is to maintain population diversity. The goal of the inside nodes of the chromosomes is to encode the decision process, through which, based on the sensor data, the agent decides which tactical behavior will be applied. The sensor data is described by the NumberOfFood, NumberOfAgents and EnergyLevel functions. We created two types of strategies:

- **Balanced**: seeks to create an agent that doesn't specialize on any type of behaviors

- **Aggressive**: specialize on attacking

The game strategies were evolved on a 400x400 pixel game map containing obstacles and 100 food resources. In addition, six opponent agents of three different types were added to the game during training. The first agent type simply wanders around in the environment. The second type of agent can attack and it is also able to eat food resources. The third agent type can only flee if it is attacked. The purpose of the opponent agents is to generate realistic situations of conflict and competition. Strategies are formed based on weights that depict the importance of each specialized behavior.

The fitness function is calculated using the following formula:

$$F = w_{\text{Food}} \cdot S_{\text{Food}} + w_{\text{Attack}} \cdot S_{\text{Attack}} + w_{\text{Flee}} \cdot S_{\text{Flee}} + w_{\text{Wander}} \cdot S_{\text{W}ander} \tag{2}$$

The food score S_{Food} is a measure of what percentage of the available food was consumed by the agent. The attack score S_{Attack} measures the number of successful attacks, while the flee score S_{Flee} the number of successful flee actions. Finally, S_{Wander} measures the portion of the game map covered by the agent. Each individual in the population represent a valid game strategy and they are evaluated by executing them in the FFM game for 1000 simulation cycles. The balanced game strategy was evolved using equal weights, 0.25 for each weight type. The aggressive strategy on the other hand was configured with higher weight values for attack and flee, 0.40, with the remaining weights set to 0.1.

Heuristics: As we were not successful evolving tactical behaviors which reliably avoid being stuck on obstacles, the helper functions used in stage 1 were extended to make use of the A* path-finding algorithm, to avoid getting stuck on obstacles when traversing the environment.

Results: The genetic programming agent evolved at the end of the development process performed in a satisfactory manner compared with other learning based paradigms. However, our development process also exposed all the problems associated with developing an agent using learning from scratch. Like in the case of other learning based paradigms, the developer has spent significant effort in creating learning environments for learning relatively trivial behaviors. In addition, the genetic programming effort required a large amount of computation power, most of it spent in the calculation of the fitness function. The calculation of the fitness function was expensive because it required the playing of a complete game. We also found it necessary to limit the length of the chromosomes to keep the execution time at a manageable level. This is its turn limited the type of behaviors which can be evolved.

Development effort:

Paradigm implementation (20%) For this agent, the paradigm implementation required us to implement the genetic selection mechanisms, and the mutation and crossover operators. We also had to develop an interpreter for the resulting genetic programs and integrate it with the FFM framework.

Agent implementation (30%) To express the problem in the terms of the paradigm, we developed the two step learning process for tactical behaviors and game strategies. These steps involved choosing the states, implementing the actions associated with the states, as well as implementing the fitness functions.

Heuristics (10%) The heuristics of the genetic programming agents were integrated in the actions of the tactical behaviors, and concerned hard-to-evolve spatial orientation skills.

Learning and tuning (40%) The genetic learning process took a significant processor time. As both the fitness function and some of the terminal actions involved many tunable parameters, the evolutionary learning had to be repeated several times.

C. Reinforcer: reinforcement learning

Reinforcement learning [14] is a technique in which an agent explores an unknown domain, and adjusts its behavior through positive and negative reinforcements. Thus, learning happens in real time, without the need of an explicit off-line learning stage. A specific implementation of reinforcement learning is described by the states of the environment, the actions the agent can take, and the reinforcement applied as a result of taking an action. In every state, the agent can choose from a list of possible actions, which cause a transition to a new state or a self-transition. Every action which the agent can take in a given state is labeled with a reinforcement value; the agent will choose the action with the highest reinforcement value. The set of reinforcement values for all states is called a policy. The goal of reinforcement learning is to find a global optimum policy, which leads to an agent behavior where the actions taken by the agent in all states maximally contribute to the goals of the agent. The difficulty is that the positive or negative effects of actions might not become measurable immediately, only later in time. The calculation of this "delayed reward" is the central problem of reinforcement learning.

Our implementation uses temporal difference (TD) learning [25]. We estimate the reinforcement function $Q(s, a)$ representing a combination of local and global reward for every state-action pair. The value of $Q(s, a)$ is maintained for all the combinations of state and possible actions. The value of $Q(s, a)$ is updated as follows:

$$Q(s_t, a_t) = r_t + \gamma \cdot max(Q(s_{t+1}, a_{t+1})) \tag{3}$$

where $Q(s_t, a_t)$ is the value for the state-action pair at time t, r_t is the the (immediate) reward for that state action pair and $\gamma \in [0, 1]$ represents the balance between local and global reward. Notice that the values at time t can be rewarded only after the actions at time $t+1$ are known. The formula can be expanded to a longer chain of actions as follows:

$$Q(s_t, a_t) = r_t + \gamma \cdot r_{t+1} + \ldots + \gamma^{n-1} \cdot r_{t+n-1} + \gamma^n \cdot max(Q(s_{t+n}, a_{t+n})) \tag{4}$$

If an action only leads to a positive result several stages later but the action itself has no reward, that result will immediately propagate back, based on the gamma value, to the initial action and therefore favor it. It can be shown, however, that the TD formula converges to the same Q function regardless of the choice of n, the choice of n having an impact only on the speed of convergence.

If the system in which the TD-learning takes place is nondeterministic, where the successor state of an action is known only probabilistically, the formula needs to be modified as follows:

$$Q(s_t, a_t) = (1-\alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot max(Q(s_{s+t}, a_{t+1}))) \tag{5}$$

where $\alpha = 1/(1 + visits(s, a))$ is related to the number of times that the state action pair has been visited.

Implementation: The agent was implemented using a TD-learning algorithm, extended to a chain of 5 actions. We chose to leave the discovery of game intelligence completely to the learning algorithm. The task of the programmer was only (a) to define the actions, (b) define the states and (c) define the way in which the reinforcements are applied.

The behavior of the agent was modeled with 24 actions. 20 actions represented the movement of the agent in the four directions with the five possible speeds. Four actions modeled the ability to eat, attack, flee and multiply.

The state of the agent was represented by a triplet composed of the Energy, ActionPossibility and Sensing sub-states as enumerated in Table IV. The Energy and ActionPossibility sub-states are self-explanatory. For the Sensing sub-state, we have divided the sensing area in four quadrants, and the observations in each of these quadrants were assigned to one of the four categories: Obstacle, Food, Agent or Nothing. To reduce the number of states, we have deployed tie-breaking algorithms, and the quadrants were characterized with the most relevant sensed information. Combining the Energy, ActionPossibility and Sensing sub-states, we obtained a system with 5120 possible states and 117760 possible action-state pairs.

The main component of the reward or penalty of the agent was the energy difference between turns. In addition to this, the agent was penalized for failed actions such as hitting an obstacle, attacking without

TABLE IV

POSSIBLE VALUES FOR THE SUBSTATES OF THE REINFORCEMENT LEARNING AGENT

Name	**Count**	**Values**
Energy	5	0-5000, 5000-10000, 10000-15000, 15000-20000, >20000
ActionPossibility	4	CanFlee, CanEat, CanAttack, NoAction
Sensing	256	e.g. (North:Food, West:Obstact, East:Agent, South:Nothing)

the presence of an enemy agent and so on. To discourage infinite loops, a penalty was given to repetitive movements.

For each of the 117600 state-action pairs we stored the Q value in a floating point variable and the number of visits as an integer value. The size of the database was 1.5 MB, stored in persistent storage, and loaded by the reinforcement agent at the beginning of the game. When operating in the learning mode, the table was updated by the agent during runtime and saved at the end of each run. The training runs were performed by letting several ReinforcementAgents compete against each other in a series of games.

The reinforcement was applied using the action-chain TD update rule, with the length of the chain being 5. Let us provide some intuitive justification behind the choice of this number: (a) the rules of the movement of the game allow a series of 4 moves which return to the same position, a combination we wanted to penalize, (b) we found that any food item in the sensor range can be eaten with a combination of at most 4 moves and 1 eat action, a combination we wanted to explicitly reward.

Heuristics: We decided to use a radical paradigm-pure approach. No heuristics were implemented and the the actions learned by the agent directly match the low level actions provided by the FFM framework. In hindsight, this led to a performance disadvantage compared to other implementations which were more liberal in adding heuristics and domain specific knowledge.

There are several, relatively natural ways to embed heuristics in a reinforcement learning agent. One way, for instance, is to introduce high level actions, which encapsulate well known algorithms or action sequences which empirically have proven useful. One example of such an action would be “move to (x,y) while avoiding obstacles”, “pursue and repeatedly attack agent A until it is destroyed”, or “flee from agent B until it is out of sensor range”. Another approach would be to introduce high level states, which are results of evaluation functions such as “being close to an opponent agent with a superior energy level”. Using such high-level states would shift the work from the learning algorithm to the programmer;

in the current implementation the choice of actions and states is very straightforward and it requires only a knowledge of the rules of the game, not of its tactic and strategy.

Another possibility would be to explicitly remove forbidden actions from consideration, rather than learning to avoid them through negative reinforcement. This would have no influence over the performance of the agent, but it would accelerate the initial phase of learning.

Results: We found it instructive to observe the behavior of the Reinforcer agents during their learning process. As the Q function was initialized to zero at the beginning, the agent started by performing random actions, most of them being invalid. As the learning was progressing, the agent was gradually learning to ignore invalid actions and the behavior became subjectively more “purposeful”.

After several hundred iterations, the agent managed to independently discover several complex behaviors. One example is finding food in its sensor range, adequately navigate to it and eat it. The navigation created a relatively complex problem, as the agent had only four directions of movement at various speeds. For food situated at an angle, the agent needed to learn a combination of movements on two directions. If the agent approached the food with a speed too large, it could overshot it, and find it only after several back and forth movements. The agent learned to keep away from obstacles but could not discover more advanced algorithms for avoiding them.

The training of the agent was done in an environment which contained several (very aggressive) KillerAgents and several simple Feeder agents which were explicitly developed as training partners for the ReinforcementAgent. The inter-agent behavior learned by the agent became to avoid or flee from all the other agents in the environment. We speculate that this behavior was motivated by the fact that any kind of fight led to an immediate negative reinforcement. The learning algorithm had apparently not managed to discover the larger positive reinforcement stemming from successfully destroying another agents. The relatively short chain of reward might have played a role in this.

To summarize, the system learned to find the food, avoid obstacles, avoid other agents and survive if it is either alone on the map or in the company of relatively non-aggressive agents.

Development effort:

Paradigm implementation (10%) The reinforcement learning paradigm has a well specified, generic implementation, which was easily translated from pseudocode to the implementation environment.

Agent implementation (10%) This part of the development effort involved designing the states and the associated actions, and implementing the behavior associated with the actions. As the actions of the reinforcement learning paradigm in our design correspond directly to the low level actions of the FFM environment, the implementation effort was small.

If we would have chosen an approach based on higher level actions and/or higher level observations (based on the preprocessing of immediate sensor readings), the development effort in this step would have been significantly higher.

Heuristics (0%) There were no heuristics in the implementation of the reinforcement agent.

Learning and tuning (80%) Most of the development effort went into the learning process. With its relatively large state-action space, the agent required very extensive training sessions as naturally, every point of the state-action space needs to be reached several times for effective learning to take place.

As the training depends on the environment and the behavior of the other agents as well, the learning process was repeated several times, with different environment scenarios and “training partners”.

D. CBRAgent: case-based reasoning

The majority of human problem solving skills rely on heuristic knowledge or rules of thumb. Case-based reasoning (CBR) is a paradigm that reuses heuristic knowledge by maintaining a knowledge base of cases that have been used to successfully solve problems in the past. The concept of CBR was first introduced in [21], where the activity of learning is modeled as a dynamic memory. The dynamic memory model has been formalized for the field of artificial intelligence in [1] to form the paradigm that is currently known as CBR.

The basic steps of CBR are *retrieve*, *reuse*, *revise* and *retain*. First, CBR retrieves the most relevant case to the current problem at hand. The retrieved case is reused and revised to incorporate minor variations in the solutions. This adaptation step gives CBR the capability to form more precise and accurate solutions to select problems. Finally the revised solution is retained for future use. [11] has shown the capability of CBR as an intelligent search method for controlling the navigation behavior of the autonomous robot. Our implementation is an adaptation of this model.

Implementation: Our implementation of the CBR model contains six main components: feature identification module, case library, case matching, case selection, case adaptation, and case retain. Feature identification relies on the sensor data available for the agent. The FFM problem was represented with the sensor parameters in Table V.

The case library of the CBRAgent contains 19 historic cases were selected based on the performance of agent in the training simulations. Agent behavior can be refined by adding more specific cases to the library.

The implementation of case matching is based on a distance matrix measuring the similarity between cases. The distance is represented as the weighted sum of squared differences of sensor data. The weights

TABLE V

PROBLEM REPRESENTATION INCLUDING SENSORY DATA.

Name	**Weight**	**Possible Values**
Food density	5	0...N
Distance to the closest food	1	0...40
Enemy density	10	0...N
Distance to the closest enemy	1	0...40
Distance to the closest obstacle	1	0...40
Ratio of agent and enemy health	10	0...N
Can attack	10	0,1
Can flee	10	0,1
Can eat	10	0,1
Can multiply	10	0,1
Action	N/A	eat closest food item, attack closest agent, multiply, flee from closest agent, avoid, wander, move

allow the agent to differentiate between the more and less important sensor readings. We used two different case selection methods. The first approach utilizes elitism, with only the most similar case selected for execution. The second approach uses the random selection of cases that reside within a predefined distance Δd of the minimum distance. The first approach is adopted when there are not enough historic cases to efficiently use the second approach.

Heuristics: The CBRAgent follows very simple heuristics. When a high level action is specified by the CBR module, the heuristic module adapts the action to suit the current environment. The CBR module can be thought as the conscious mind which strategizes over high level actions, while the heuristic module resembles an "instinct" or a learned skill.

The first heuristic determines whether the specified action is feasible. For example, eating requires food to be within the sensory range. If there is no food in the sensor range, the heuristic module marks the specified action as failed and proceeds to the next cycle. Such a failure will penalize the case that produced the unfeasible action by reducing the reliability of the case. The reliability value is used as a weight during the calculation of the distance matrix, so that cases with the lower reliability will have greater distance in the subsequent cycles, and are less likely to be chosen for execution.

The second heuristic controls the direction and the speed of the agent movement. Actions require different directions and speeds depending on the goal of the movement. For instance, eating requires a movement towards the food and a speed appropriate to the goal of stopping at the food location. Wandering actions strive to explore more areas in the world and therefore select the direction and the speed in a randomized manner.

Results: Our implementation of the CBRAgent yielded acceptable performance, having a good survival rate and relatively high levels of energy at the end of the game. The main problem in creating an efficient CBR based agent is the selection of the correct set of cases. In our implementation, this was done by hand, requiring a careful analysis of the game by the developer. Alternatively, we could have tried to learn the cases, for instance, through a clustering algorithm such as K-means. This way, the bulk of the development effort would have been shifted from game analysis to the guiding of the learning process. Our initial experiments, however, had shown that the FFM game, despite its apparent simplicity, is very difficult to learn through random exploration. Thus, the developers felt that a set of cases created by clustering would almost certainly have yielded a lower performance than a hand-crafted one.

Development effort:

Paradigm implementation (20%) The implementation of the generic framework of the case-based reasoning paradigm is a straightforward task, as the part of the paradigm which can be expressed in a domain independent way is relatively minor. As the complexity of the agent increases, the implementation of the CBR paradigm will become an increasingly smaller portion of the overall development effort.

Agent implementation (40%) This part of the development effort contained the identification of the cases, the implementation of the corresponding behaviors in the case library. Agent implementation also covered the choice and implementation of the case matching techniques. This technically could have been implemented as part of the paradigm implementation, but the particularities of the FFM game made it difficult to express it in a domain independent way.

Heuristics (20%) The heuristics deployed in the CBRAgent were (a) testing the feasibility of the action proposed by the CBR engine and (b) control the direction and speed of agent movement.

Learning and tuning (20%) In our implementation, the CBRAgent was using online learning to adjust the reliability value of its cases. For implementations which build their own case libraries (as opposed to handcrafting it), this part of the development effort would also include this unsupervised learning step.

E. RuleBasedAgent: forward reasoning

RuleBasedAgent is implemented using the forward-reasoning rule based paradigm, with hand-crafted rules. The agent is composed of an inference engine and a knowledge base. The knowledge base is partitioned into a fact base and a rule base. The fact base is the repository of all the truths sensed or understood by the agent. An important part of the fact base is the current set of sensor readings, updated at every reasoning cycle. The fact base can also contain data created by the rules interpreted by the inference engine.

The rules have an IF antecedent THEN consequent structure. The inference engine determines which are the rules whose antecedents are satisfied by the current fact base, then resolves the potential conflicts based on the *salience* of the rule. The result of the inference process determines the next action by the agent and updates the fact base with the changed status and short term objectives of the agent.

We found that the decisions with the most impact on the success of the rule based system are the choices of representation for the actions and game situations. The FFM game is performed on a two dimensional map containing obstacles, food, friendly and opponent agents. In addition, the game has a temporal dimension as well: for instance, the agents can move with different speeds. A low granularity, complete representation of the FFM world would require a very complex model, including, between many others, a complete temporal logic, a logic of the 2-dimensional spatial relations, reasoning about the objectives of opposing agents, and so on. Besides inflating the rule base, this representation requires an inference engine which can match antecedents based on spatial and temporal relationships. A system of such complexity was clearly out of the scope of the present project.

Our solution was to design the antecedents and consequents of the rule with game specific high level terms. For instance, a rule could say: IF an opponent agent is in the sensor range AND its energy level < my energy level THEN approach and attack it. This solution has the disadvantage that it needs procedural calculation for the verification of the antecedents, and its consequents need to be translated into the spatial domain. The advantage, however, is that relatively small number of rules can capture the strategy of the game (our implementation used 15 rules). These rules are also easier to read, debug and adapt to small changes in the rules of the game. Being similar in language to a common-sense description, they allow for easy transcription of human expertise. However, the rules are specific to the game, and they cannot form the basis of a general theory.

Implementation: The implementation of the rule based agent relies on the analysis of the game by the developer and the encoding of the proposed strategy as rules. The agent is successful in the FFM game if it can gather a large quantity of food, collaborate with agents of the same kind, and prevent the success of the opponent agents by either direct attack or by starving them of resources.

Some behaviors of the agent can be described by immediate, one step actions, such as eating a food item which is in the sensor range. Other behaviors represent longer term decisions, represented by special facts introduced in the fact base as a consequence of rules. Examples of these facts are AGGRESSIVE, PASSIVE, FLOCK or GREEDY.

Let us consider, for instance, the relationship of the agent to opponent agents and other RuleBasedAgents. An agent has two choices when encountering the opponent. The first choice would be to avoid the agent and get beyond the other agent's sensor range. This avoids a possible fight that would decrease the agent's energy. Alternatively, the agent may find it beneficial to pursue an agent with the intention of attacking and destroying the opponent agent.

The choice between the passive and aggressive behaviors depend on the existence of the PASSIVE or AGGRESSIVE facts in the agent's fact base. These facts are added or removed in function of the ratio of energies between the agent and the opponent. We found that a threshold of 1.2 works well in practice, that is, the agent behaves aggressively if it has at least 20% more energy than the opponent. By adjusting this threshold, the developer can create agents with various levels of aggressiveness.

The rules describing these behaviors can be summarized as follows:

```
AVOIDENEMY
    antecedents
        sensing opponent agent(s)
        PASSIVE fact in KB
    consequent
        move away from closest opponent agent

TOENEMY-1
    antecedents
        sensing opponent agent(s)
        PASSIVE fact not in KB
        AGGRESSIVE fact in KB
    consequent
        move towards the closest opponent agent

TOENEMY-2
    antecedents
        sensing opponent agent(s)
        PASSIVE fact not in KB
        FLOCK fact in KB
    consequent
        move towards the closest opponent agent
```

The rule based agents implement a model of cooperative behavior loosely based on the concept of flocking [19]. Whenever two rule-based agents come within each other's sensor range, the agent with the lower id acts as a leader and is followed by the agents with higher ids. When food is in the range of the flock, the agents act depending on the presence of the GREEDY fact. When the fact is present, the agent attempts to move and acquire the food without additional considerations. When the GREEDY fact is not present in the fact base, only the agent which is the closest to the resource is attempting to acquire it, thereby promoting a more efficient feeding process. As we have seen above, being part of a flock also changes the aggressiveness of the agents. When a flock encounters an opposing agent all the agents attack the opponent agent, irrespectively of the energy ratio.

Obviously, these are just some examples of possible behaviors which can be achieved. We note the remarkable ease with which strategies developed by observing the game can be transferred to the agent. This ability to explicitly express human knowledge or preference in the game strategies led to a linear and predictable implementation process.

By playing against other agents, the rule based agent was continuously updated to offer better performance. However, this process always passed through a human observer, who, through the study of the game logs, devised new strategies to be encoded as rules. The rule-based agent does not implement direct learning from experience.

Heuristics: The rule based paradigm is intrinsically heuristic; the rules can be seen as constructs capturing the experience of a human expert. In addition to this, our specific implementation contains further heuristics in the choice of the high level antecedents and consequents.

Results: Overall, the rule based agent provided a good performance, and it was remarkably easy to create an agent which displayed the behavior envisioned by the developers. The number of rules being relatively low, there was little conflict between the various rules, which was resolved manually using simple salience values. The performance of the agent basically mirrors the ability of the developer to analyze the game and express its knowledge in form of rules.

Development effort:

Paradigm implementation (20%) We have implemented a simple forward chaining rule based system. The reason for this was the ability to do custom, procedural matching of the antecedents of the rules against the knowledge base.

There are many, freely and commercially available rule based systems, using complex, high performance algorithms, such as RETE. In our case, we considered that the ability to perform arbitrarily complex matching of the antecedents is more important than overall performance (due to the small size of the knowledge base and the small number of rules).

In a rule based agent implementation, the development effort for this step can range from zero, if an existing rule based system is used, to very large, if a complex, high performance engine is implemented from scratch. However, the choice of a rule based system which does not allow to conveniently express the concepts of the problem domain can make the agent implementation require significantly more development effort.

Agent implementation (70%) The design and writing of the rules represented the largest part of the implementation of the rule based agent.

Heuristics (0%) There were no heuristics in the implementation of the rule based agent, the human knowledge being expressed directly in the rules.

Learning and tuning (10%) The rule based agent does not perform automatic learning. The agent was tuned by observing its behavior in test runs. The perceived incorrect decisions were corrected by adding or modifying rules.

F. NaïveAgent: naïve programming, scripting

Naïve programming is a style of coding that allows the developer to hand-optimize the code for a particular task. While other agents in this study are centered around a "paradigm", the goal of NaïveAgent is to investigate what performance can be achieved without allegiance to any paradigm. Naïve programming is frequently the method of choice in character development for first person shooter and role playing games, where the agents are made to play according to relatively restricting scripts. A scripted agent does not implement an autonomous entity in the sense of the Belief Desire Intention model. The agent does not take actions to further its clearly represented goals; rather, the sensors of the agent are used to determine which predefined scenario does an agent need to follow. We note, however, that the psychological theories of transactional analysis [2], [24] also conjecture that humans are enacting certain, relatively inflexible scenarios in their behavior. Therefore, scripting-based naïve programming might not be as "paradigm-free" as it appears.

1) Implementation: The development of the NaïveAgent consisted on identifying the possible encounters and for each possible encounter, a script was written specifying how the agent should react.

The main objective of the NaïveAgent was to maximize the number of NaïveAgents surviving at the end of the game. To achieve this, the agent implements an aggressive mating strategy, multiplying at every possible opportunity when this is technically feasible. In addition, the NaïveAgent shows altruistic behavior towards other NaïveAgents in food gathering. If a NaïveAgent senses a food item which is in the sensing range of another NaïveAgents, the agent with the higher energy level will refrain from taking the food in favor of its peer. This behavior gives an advantage in cases when the number of NaïveAgents is large, allowing more agents to survive on the same map.

The movement of the NaïveAgent is governed by simple, hand-crafted heuristics. Whenever it encounters an obstacle, the NaïveAgent makes an evasive movement towards the right. To avoid repeated movements, a `failcount` variable is incremented. After more than five right turns, the agent tries to move in random directions.

The encounters between the NaïveAgent and opponent agents are governed by the following heuristics. If the energy level of the opponent is more than 120% of the NaïveAgent, the NaïveAgent flees. If the energy level of the opponent agent is less than 80% of the NaïveAgent's, the NaïveAgent attacks. If there is more than one opponent agent in the sensor range which meats the condition of attack, it will attack the weaker one.

Heuristics: For the NaïveAgent, the heuristic *is* the paradigm, thus we consider the scripts described above to be part of the heuristics of the agent. No additional heuristics were implemented.

Results: As expected, the naïve agent was successful in implementing the behavior envisioned by the developer. The performance of the agent was, therefore dependent on the ability of the developer to analyze the game, and express its knowledge in form of scripts applicable to certain situations. In this respect, it is similar to other explicitly programmable agents, such as the rule based agent. We found, however, that the scripting approach led to a more rigid, less parameterizable behavior. Overall the code organization did not favor tuning, the behavior of the agent being deeply encoded in code.

Development effort:

Paradigm implementation (0%) No paradigm framework was developed, the scripts were encoded directly into the Java code. Other implementations might develop a special scripting language and interpreter (as many game development systems are doing for controlling avatar behavior).

Agent implementation (0%) As we consider that the use of heuristics was the paradigm for this implementation, we consider the full implementation under the heuristics header.

Heuristics (80%) Most of the development effort went into the creation of the scripts which encode the behavior of the agent in various situations.

Learning and tuning (20%) The agent was tuned by modifying the parameters of scripts by observing the behavior of the agent in competitive scenarios.

G. GamerAgent: game theory

Game theory [15] is a mathematical formulation of cooperative or competitive interaction between multiple entities. The assumption behind the game theoretic agent is that every encounter between two opponent agents is modeled as a separate zero-sum game. At every encounter, the agent has the choice of attacking or fleeing. An agent might attack with the intention of engaging into a major fight continuing until one of the agents is destroyed; but it can also attack in order to force the other agent to flee (and thus, waste energy and leave the agent's zone of influence). Therefore the decision to attack or flee is dependent not only on the energy balance of the two agents but also on the history of previous encounters.

The decision process of the agents, modeled as a two-choice, two-player game, is described by the game matrix. The utility function of each agent depends on the λ and μ parameters. λ is the ratio of difference in energy level of the GamerAgent E_g and the energy of the opponent E_a and is defined as $\lambda = (E_g - E_o)/max(E_g, E_o)$. μ is the likelihood of attack by opponent agent (based on previous interaction). Naturally, $1 - \mu$ shows the likelihood of fleeing by opponent agent. The utility functions are defined as follows.

Utility for attacking when opponent agent is expected to attack:

$$U_{A,A} = (1 - \mu) \times 100 + \lambda \times 200 \tag{6}$$

It is evident that the agent gets maximum utility by attacking when it has a positive high energy difference ratio (i.e. when λ is closer to 1) and the likelihood of attack is minimal (i.e. when μ is closer to 0). On the other hand, the lower energy difference ratio (i.e. when λ is closer to -1) and high likelihood of attack (i.e when μ is closer to 1) tends to lower the utility to attack. It should be noted that change in utility is more drastic with energy difference ratio than with the likelihood of attack. The reason is that the agent would like to avoid a conflict if it has less energy, even if the opponent is not likely to attack.

Utility for attacking when opponent agent is expected to flee:

$$U_{A,F} = \mu \times 100 + \lambda \times 200 \tag{7}$$

This utility function behaves the same way as above except that the highest utility is achieved with highest value of λ and highest likelihood of attack μ. The reason for that is that opponent is expected to flee and our agent should get highest utility by attacking the opponent who has been frequently attacking before. However, if the opponent is less likely to attack, the utility of attacking is slightly less. Note that this does not mean that in case of large discrepancy of energy level, our agent will not attack (only that it will get lesser utility by doing that).

Utility for fleeing when opponent agent is expected to attack:

$$U_{F,A} = \mu \times 100 - \lambda \times 200 \tag{8}$$

In this case, the agent gets highest utility in fleeing if the opponent is very strong (i.e when λ is closer to -1) and is highly likely to attack. Slightly lower utility is obtained by fleeing if the agent is not likely to attack. The least utility in fleeing is acquired when the gamer agent is very strong (i.e when λ is closer to 1) and the other agent is not likely to attack (i.e μ is closer to 0).

Utility for fleeing when opponent agent is expected to flee:

$$U_{F,F} = (1 - \mu) \times 100 - \lambda \times 200 \tag{9}$$

In this case, the most utility is obtained by fleeing if the opponent agent is very strong and is not likely to attack. Since the opponent agent is expected to flee, the GamerAgent is also better off fleeing. The least utility in fleeing is obtained when the gamer agent is very strong and the opponent agent has been frequently attacking.

Based on these utility functions, the game matrix is shown in Figure 4.

		Opponent	
		Attack	Flee
GamerAgent	Attack	$U_{A,A}$	$U_{A,F}$
	Flee	$U_{F,A}$	$U_{F,F}$

Fig. 4. Game matrix for the game of the encounter between two opponent agents.

Given this matrix, the optimal strategy is selected based on opponent behavior. Figure 5 shows the utility for attacking or fleeing when opponent attacks. It is evident from this figure that GamerAgent will attack when the ratio of energy difference is a large positive value, (i.e. when λ gets closer to 1), and

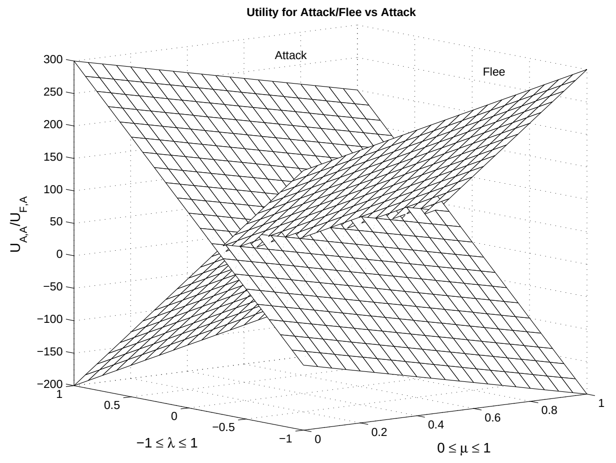

Fig. 5. Utility for attacking or fleeing when opponent attacks

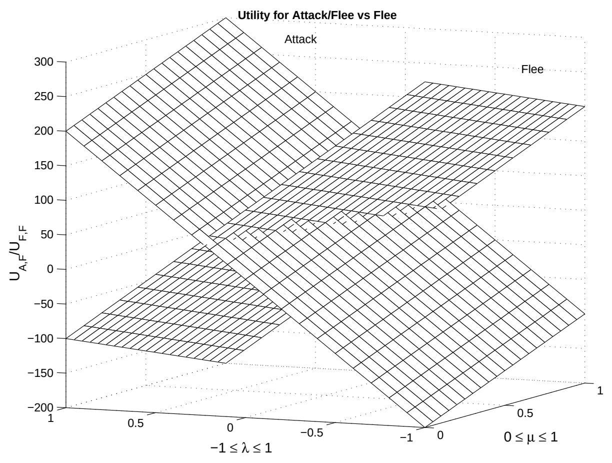

Fig. 6. Utility for attacking or fleeing when opponent flees

will flee if λ is negative. However, when the value of λ is closer to 0, the gamer agent's behavior is determined by previous interactions. In this case, gamer agent will flee if the opponent had previously shown aggressive behavior, and it will attack if the opponent has been previously defensive.

Figure 6 shows the utility for attacking or fleeing when opponent flees. In this case, if the value of λ is closer to zero, then the gamer agent will attack if the opponent has been attacking frequently and flee if the opponent has not been attacking frequently.

Heuristics: As the game theory model describes the behavior of the agents only in the situations of the encounter between two opponent agents, the remainder of the behavior needs to be controlled through heuristics. The heuristics deployed were described in terms of IF condition THEN consequent rules. However, only a single rule was allowed to fire per turn, and the action of the agent had to be immediately determined by the consequent of the rule. Thus, the heuristic part of the GamerAgent does not qualify as a full featured rule based system. Note, however, that the game theoretic model can easily be integrated with other models such as rule based, context based reasoning or scripting.

Rather than provide a listing of the rules deployed by the agent we shall provide a description of the policies they interpret. If food is visible in its sensor range and no opponent agent is present, the GamerAgent navigates to the closest food item and eats it. In absence of any opponent or food item in the sensor range, the agent exhibits an exploration behavior. A map of the previously explored locations is maintained and the agent tries to explore the less recently visited locations. If there is an opponent agent in sight the agent resorts to its main, game theoretic paradigm to determine its behavior; if there is more than one opponent agent in the sensor range, the agent always chooses to flee. Finally, the agent uses the multiply action whenever its energy level reaches a threshold (currently set to 35000) and no opponent is present.

Results: The GamerAgent performed well in the game. One of the conclusions of the development was that the game situations where the game theoretic model can be applied are relatively rare. Our game theoretic model is applied only at the encounter between two agents. If the other agents on the map are avoiding the encounter - for instance, by moving away from the GamerAgent, the agent can only enforce these encounters by maneuvers pursuing other agents. Similarly, some game situations, such as an encounter where the GamerAgent interacts with an agent which has a lower energy value can occur only if the GamerAgent was better (or more lucky) in energy gathering than the other agent.

In conclusion, at least in the embodied agent domain, game theory can be used as a basis for agents only in combination with heuristics or other paradigms, as certain tasks, such as resource gathering, or path finding can not be conveniently expressed in forms of games.

Development effort:

Paradigm implementation (10%) This part covered the implementation of the generic part of a two player game, a relatively minor effort.

Agent implementation (20%) For the game theoretic agent, implementation included the specification of the games and the design of the utility functions. The agent also included code for remembering previous encounters, and based on that, to decide whether it will expect the opponent to attack or flee in a given situation.

Heuristics (60%) The agent's behavior between agent-to-agent encounters was controlled by the heuristics. In our implementation, this included the policy implementation, the implementation of the map of the explored parts and the exploration strategies.

Naturally, the effort for this part of the implementation depends on the decisions made by the developer. However, as game theory only relates to agent-to-agent encounters, which are a small part of the game, heuristics will always be a large part of the implementation of the game theoretic agent.

Learning and tuning (10%) This part of the development effort was spent in the observation of the behavior of the agent in game runs, and tuning the utility functions and certain heuristics parameters for improving the behavior.

H. CrowdAgent: crowd model

The goal of implementing the CrowdAgent was to create an agent which accurately models the behavior of humans in crowds. Although the primary goal was anthropomorphism, we also hoped that with the addition of some simple heuristics of feeding, fighting and obstacle avoidance, the crowd model will emerge teams of agents which can act successfully in the game.

Crowd simulation frequently relies on models of fluid systems or particle systems. These techniques have their common roots in the fluid models based on Euler or Lagrange forms. Fluid systems set an attractive or repulsive force and then model the frictional forces acting on the individual agents. Particle system use frictional forces as well, but place motion decision with the individual agents [4]. Crowd phenomena emerges naturally when a particle system is initialized with the appropriate parameters [13], [26], [29].

Implementation: We decided to implement a system in which CrowdAgents are forming groups based on their aggressiveness level. The aggressiveness level of an agent a at time t is $A_a(t) \in [0, 10]$. An agent will start with an initial aggression rating $A_a(0) = A_a^{init}$, and then migrate towards the aggression rate of the surrounding agents:

$$A(t+\Delta t) = A(t) + \frac{(A(0) - A(t)) \cdot \Delta t}{am} + \sum_{b=0}^{n} \frac{A_b(t) - A(t)}{cosh\left((A_b(t) - A(t))^2\right)} \cdot \frac{\Delta t}{\max\left(dist^2(a,b), 4\right)} \tag{10}$$

This formula achieves two goals: (a) if the agent is not surrounded by other agents, it will gradually return to its initial aggression level and (b) if the agent is surrounded by other agents, its aggression level is pulled towards the aggression level of the neighboring agents, with agents which are close exercising a greater influence.

The motion of an agent is related to the position of all the other agents in the sensor range, through the following equations:

$$\begin{cases} X(t+\Delta t) &= X(t) + V_x \cdot \Delta t \cdot \sum_{b=0}^{n} \frac{pF_b(X(t) - X_b(t))}{\max(dist^3(a,b),1)} \\ Y(t+\Delta t) &= Y(t) + V_y \cdot \Delta t \cdot \sum_{b=0}^{n} \frac{pF_b(Y(t) - Y_b(t))}{\max(dist^3(a,b),1)} \end{cases} \tag{11}$$

The pF attribute is based on the aggression of the agent of interest and the aggression of the agent in the sensor range. This is an attractive/repulsive attribute which is defined by a piecewise function as follows:

$$pF = \begin{cases} -\frac{pfA}{pfB^2} \cdot |A - A_b|^2 + pfA & \text{if } |A - A_b| <= pfB \\ 4 \cdot pfC \cdot \left(|A - A_b| - \frac{pfB + (10 - pfB)}{2}\right)^2 - pfC & \text{if } |A - A_b| > pfB \end{cases} \tag{12}$$

The pF factor will give an attractive influence for distances between 0 and pfB and a repulsive influence for distances larger than pfB. The summation of these forces generate dynamic groups of agents. As long as the attractive forces are not too large, the individuals have the ability to separate from a group, and join another group.

Heuristics: Crowd modeling deals only with the coordinated movement of the agents. While the visual inspection of a set of CrowdAgents acting in the game did provide the desired anthropomorphic impression of a crowd, this was not sufficient for the agents to be actually successful in the game. Thus, a relatively high number of heuristics had to be added to complete the agents behavior in the situations when the crowd modeling paradigm did not provide an answer.

We found that even the move command required some heuristic additions. One such heuristic was movement towards food. When an agent spotted a food item in its sensor range, the heuristic moved the agent towards it on the shortest path. In these cases, the heuristic took over the control of the movement of the agent for a limited time. When the heuristic finished, the agent was reverted to movement according to the crowd modeling equations.

Another situation where the movement of the agents was controlled by a heuristic was fleeing. Although the paradigm does provide an answer to this problem, by applying a repulsive influence away from the agents of differing allegiance, it was found that this is insufficient for an individual agent to get out from an unwanted fight. Therefore we added a heuristic which explicitly calls the flee action in these situations. One concern in this case is that through fleeing an agent might loose sensor contact with the group to which it was belonging. If this does not happen, however, the flee action can actually help to move the whole group away from the threat. As the agent always attempts to flee from opponent agents, its behavior is highly defensive. The CrowdAgent can get next to an opponent only as a result of a pursuit by the opponent. In these cases, however, the CrowdAgent will always choose to attack.

Another concern was the behavior of an agent outside a group. According to the equations of crowd modeling, a solitary agent would stand still, which is not a good strategy in the context of the FFM game. Therefore, we have implemented a random wandering strategy, which controls the movement of the agent when there are no agents in its sensor range. Obstacle avoidance was implemented by repeatedly trying out various directions of movement when an obstacle is present. Finally, the CrowdAgent is multiplying with a specific probability whenever a sufficiently high energy threshold is reached.

Results: The CrowdAgent was successful in exhibiting anthropomorphic, crowd-like behavior. We found that about 6 agents are needed for the influence of the crowd model to be visible. The CrowdAgent was also relatively successful in surviving in the game; however, its performance was almost completely determined by the heuristics. The crowd modeling approach covering a relatively small part of the behavior of the agent, a major implementation challenge was the integration of the main paradigm with the heuristics. For instance, in the case of fleeing, there was a clear conflict between the goal of the paradigm of maintaining the integrity of the groups, and the goal of the heuristic to get the agent as far away from the opponent as possible. The reconciliation of these strategies is difficult, as the essentially rule based heuristics and the physics based crowd model has no ways to communicate with each other. The naive solution, used in our implementation, is to pass the control of the agent abruptly between the heuristics and the paradigm. This is clearly suboptimal.

Development effort:

Paradigm implementation (20%) This part of the effort covered the adaptation of the particle crowd model to our game environment, the implementation of the movement rules, and mapping them to the actions of the agent.

Agent implementation (20%) Agent implementation covered the establishment of the aggressiveness based affective model, translating it to the particle model, and mapping it to the events and actions in the FFM game.

Heuristics (40%) As the paradigm covered only the movement of the agent, the remainder of the actions were implemented by heuristics. Heuristics also controlled the movement of the agent in the achievement of specific short term goals such as moving towards food.

Learning and tuning (20%) The parameters of the particle model were tuned by observing the agents' behavior and manually modifying the parameters to obtain behaviors closer to the one envisioned by the developer. This tuning process was complicated by the fact that the behavior of a CrowdAgent is a result of a complex interaction between several agents, with the interaction between the model parameters and the emerging behavior being complex and non-linear.

I. NeuralLearner: neural networks

Artificial neural networks are composed of (a) *neurons*, processing units which perform a simple step or sigmoid function over the summation of their inputs, and (b) *synapses*, interconnections of varying strength, which connect the output of a neuron to one or more other neurons. The behavior of the neural network is determined by the number of neurons, their interconnections and the connection weights. Although in some cases the connection weights are hand-crafted, they are normally learned through supervised or unsupervised learning. The most frequently used neural network is the multilayer perceptron (MLP), usually trained through a variant of the backpropagation algorithm [20]. Backpropagation is a supervised learning algorithm: a network is presented with a set of inputs and desired outputs; the errors are propagated back to the weights of the network and an adjustment is made in the direction of the steepest descent towards a correct output. Today, the most successful applications of neural networks are those related to pattern classification. Relatively few studies have applied them as the control paradigm of the agents. The main problem is the difficulty in acquiring good quality training data, as a study comparing neural networks and genetic algorithms in evolving artificial life movement problems has shown [31]. The Polyworld artificial life simulation [30] uses a multi-paradigm approach which encompasses both evolutionary algorithms and very general neural networks using Hebbian learning. The structure of the neural network itself is evolved through a genetic algorithm [32]. A different artificial life study [8] suggested that the combination of learning and evolution has a strong advantage over either element taken on its own, but found the evolution of neural networks a very slow process.

Implementation: NeuralLearner is implemented with a single multi-layer perceptron which is used to make all decisions regarding the behavior of the agent. The inputs to this network consist of the agent's current energy level and the presence and direction of food items, opponent agents and obstacles. Also included in the input is whether the agent can currently eat, multiply, attack, or flee. The output of the network is an action selection (move, eat, attack, flee, multiply), a direction (north, south, east, or west), and the value of speed.

With this architecture, the neural network should, in principle, be able to represent and learn the actions necessary to be successful in the environment. It was found however, that the creation of the training data presented a major difficulty in the implementation of the NeuralLearner agent. Although our findings mirror those found in several other studies concerning the implementation of agents based on neural networks, let us review the various choices in the context of the FFM game.

Handcrafting the training data based on domain knowledge. In this case, we assume that we know the mapping that the neural network should approximate, then create the training data on which the network can learn the mapping.

We found that the best way to capture human experience about the game is through rules of thumb. Although humans can readily generate a hypothetical game scenario to illustrate a rule, the handcrafting of thousands of examples is not feasible. One way to create the necessary training data is to encode the rules of thumb into an actual program, and use it to generate training data. Note, that this practically amounts to the implementation of a rule based agent to serve as a target of imitation for NeuralLearner.

Unsupervised learning based on game feedback. The idea of this approach is that the neural network agent will generate its own training data by effectively playing the game. The successful moves made by the agent will be later fed back as a training data, while the unsuccessful moves discarded. Viewed as a black box, the overall system would appear to be doing unsupervised learning, while the selection of the successful moves will be analogous to reinforcement learning. The implementation of this approach requires an initially random agent which is exploring the environment, generates training data and goes through periodic learning experiences.

We found that using an initially random agent is a very inefficient way to collect learning experiences. Most actions in the FFM game carry a small punishment (the energy expense associated with moving). Actions where rewards can be acquired, such as eating or killing an opponent, are relatively rare and they require a set of preparatory steps. For instance, the eating action needs to be prepared by actions to approach the food, which on their own carry no reward and are too complex to happen accidentally.

Thus we found it necessary to replace the random agent with an "explorer agent", created by biasing the random agent towards behaviors which would yield interesting training experiences. The agent was moving towards food, other agents and obstacles with higher probability and it was favoring the more interesting actions (such as eat, fight, flee and multiply) to movement actions. Notice that, again, we reached a situation where we essentially handcrafted a naïve agent implementation. However, this agent successfully generated some usable training data.

Results: The final agent was trained using the training data collected from game runs of the explorer agent. The neural network was a three-layer perceptron with 7 inputs and 13 outputs. All values were normalized to the range [-0.5, 0.5]. The output was coded as either 0's or 1's indicating the choice of action, direction, and speed.

The performance of the NeuralLearner was in general disappointing. As a partial success, the agent has managed to learn a set of useful behaviors. It learned to move towards the closest item of food in its sensor range and eat it if it was in its near proximity. It learned to move away from other agents.

One interesting observation is that, as implemented, the agent was unable to learn the apparently trivial random wandering behavior. This behavior was the default for most other agent implementations, as a way to discover available food resources on the map. However, due to the fact that a multilayer perceptron is deterministic, the agent has always learned a fixed behavior for the case when no food opponent agents were in the range. For instance, the final version of the neural agent has learned to always go north whenever there was no food, or opponent agent in its range. One way to solve this problem would be to provide a source of randomness to the agent - such as by adding a random variable to one of its inputs. This, however, would likely lower its already low learning rate.

Development effort:

Paradigm implementation (10%) Our agent used the Java-based Joone neural network library. Thus, the effort of paradigm implementation included only the effort of integration of the library with the FFM code.

Agent implementation (10%) This step included the design of the neural network, the encoding of the sensor readings and actions, as well as connecting the neural network's outputs to agent actions.

Heuristics (0%) As the neural network agent is paradigm-pure, no heuristics were implemented.

Learning and tuning (80%) The majority of the development effort for the NeuralLearner agent was spent in the training of the neural network, and the creation and tuning of the training environments in which the agent can learn useful behaviors.

J. SPFAgent: social potential fields

Social potential fields [17] are a way of controlling autonomous agents using inverse-power force laws to exhibit organized, group behavior. In physics, a potential field describes the force per unit experienced by an object at a given point. They are used to describe the effects of gravity on a mass or the force per unit charge of a charged particle.

The first investigations into social potential fields involved solving obstacle avoidance problems, motion planning, and collision avoidance of single robots. It was found that SPF allows the definition of behaviors for the interaction of different types of agents, whether it is a predator-prey, parasitic, or symbiotic relationship, and offers ways to implement obstacle avoidance. One of the problems of the approach is the agent becoming stuck in local minima of the potential field. Thus the agent cannot operate exclusively under the control of the potential fields.

Implementing agents based on social potential fields involves determining of the parameters of the inverse power force laws, such that the desired behavior emerges from the combination of attractive and repulsive forces. Let us consider the case of a leader-follower relationship. The follower agent will be strongly attracted to the leader, so that it follows closely the lead's movements. The leader has a weaker attraction to the follower, such that its path is not affected significantly by the follower agent's presence. While this example seems deceptively simple, it has been found that for a dynamic system determining the force laws for a given behavior is polynomial-space hard [18]. In practice, determining the coefficients to enact a given behavior is an iterative process based on experience and experimentation.

Implementation: The movement of the social potential fields agent are governed by forces of attraction and repulsion towards the other entities of the game. To simplify the behavior of the agent, and allow for an easier setting of the parameters, we decided that the forces will be composed of only one attractive and one repulsive component:

$$\overline{F}(\overline{p_a}, \overline{p_o}) = \left(\frac{c_a}{r^{\sigma_a}} + \frac{c_r}{r^{\sigma_r}}\right) \cdot \frac{\overline{p_a} - \overline{p_o}}{r} \tag{13}$$

TABLE VI

THE PARAMETERS OF THE SOCIAL POTENTIAL FIELDS FOR THE FOUR CLASSES OF ENTITIES CONSIDERED

Object of interest	c_r	c_a	σ_r	σ_a
Obstacle	5.0	0.0	5.0	N/A
Food	0.0	20.0	N/A	0.8
Opponent (n)	45.0	0.0	1.0	N/A
Friendly agent	45.0	20.0	1.0	0.7

with $c_a, c_r \geq 0$ and $\sigma_a > \sigma_r > 0$. $\overline{p_a}$ is the position vector of the agent, $\overline{p_o}$ is the position vector of the object, and $r = ||\overline{p_a} - \overline{p_o}||$ is the Euclidian distance between the agent and the object. As a further simplification, we assume that the agent can not recognize individual entities only classes of them. The four classes of entities recognized by the agent are (a) friendly agents (other SPFAgents), (b) opponent agents, (c) food items and (d) obstacles. The coefficients c_a, c_r are the magnitude of the attraction and repulsion force at unit distance, while the inverse power constants σ_a and σ_r determine the speed of decay of the force with the distance. All these constants depend on the nature of the entity.

Programming the movement of the SPFAgent is done by determining the values for these constants. Reasoning and experimentation plays an equally important role in this setting. The values used in the final version of the SPFAgent are shown in Table VI. One of the problems we encounter is that while the selection of values for pair-to-pair situations is relatively straightforward, the emergent behavior of the agent when a large number of forces act over it is frequently unpredictable.

Let us illustrate the process through which an informal description of the desired behavior can be transformed into force parameters, and the difficulties encountered in this process. It is desirable that the agent avoids obstacles, therefore the agent should be repulsed by obstacles. On the other hand, an obstacle should have an influence over the agent only when it is in its immediate proximity. We can achieve this by making the exponent of the denominator relatively large, $\sigma_r = 5$, and the scaling constant relatively small, $c_r = 5$. This creates a force which is relatively strong at short distance, but decreases rapidly with distance. There is no situation under which the obstacle should attract the object, therefore we set $c_a = 0$, and the value of σ_a is not relevant in this case.

Let us now consider the interaction between agent and food. The agents are unconditionally attracted to food items, and never repulsed by them. At the same time, we want this attraction to work from relatively large distances. If we would set $\sigma_a = 0$ the agent would be equally attracted to all food resources in its sensor range. However, we want to have a stronger attraction to the closer food items. Therefore we pick the exponent as $\sigma_a = 0.8$ which means that the force is decaying somewhat slower than the increase in distance. We pick the constant of the attraction force as $c_a = 20$. The constant of the repulsion force c_r is set to zero.

The relationship between the agent and opponent agents needs to be first decided at a policy level, followed by the transcription of this policy in the terms of attraction and repulsion forces. We decided to create an agent which always tries to avoid confrontations. Therefore we set up a strong repulsive force with a constant $c_r = 45$ and $\sigma_r = 1$, with the constant of attractive force $c_a = 0$. Note that the constant of repulsion from opponent agents is larger than the constant of attraction to food items.

Finally, the interaction of the SPFAgent with friendly agents should result in a flocking behavior. The SPFAgents should move in configurations where they keep a more or less constant distance between each other, such that they can efficiently forage for food items, but not compete for the same food. We can achieve this by setting up the force fields in such a way that a force of repulsion is acting if the agents are too close to each other, but a force of attraction if they are too far. In general, the force with the larger σ value will be dominant at short distance, while the force with the smaller σ will dominate at longer distances. The agents will tend to position themselves at the equilibrium point, where the two forces cancel each other out. This happens at the distance:

$$r_{\text{equilibrium}} = \left(\frac{c_a}{c_r} \right)^{\frac{1}{\sigma_a - \sigma_r}} \tag{14}$$

With the values chosen in Table VI the equilibrium distance is 14.92 units.

Heuristics: The social potential field model only controls the movement of the agent, and it does not account for other actions such as feeding, fighting and multiplying. However, as we have found, even the movement of the agent required certain actions which are difficult to express with the social potential field model. Therefore our agent deployed two types of heuristics: (a) the ones used to supplement the movement rules as described by the social potential field and (b) the ones which control the actions of the agent which can not be described as a movement.

The forces of the social potential field model have laws similar to the gravitational or electromagnetic forces. As a result, an immediate mapping of these forces into movement yields results such as an agent orbiting or oscillating around a food item; agents stuck in locations where two opposing forces cancel each other out (the equivalent of the Lagrange points of the planetary systems) and so on. The absence of a force of friction or viscous resistance is the explanation for this behavior.

To correct the agents behavior regarding feeding, we added a heuristics which limits the influence of the food items to the closest one and forces the agent to use a constant speed in the approach to food (as opposed to the acceleration which would be implied by a direct application of the social potential field equations, and which would make the agent overshoot the food location). An additional heuristic detects periodic movements in the behavior of the agent (oscillation or orbit) and gradually disconnects forces until the agent escapes the periodic movement.

The attack and multiply behavior of the agent was modeled with hand-scripted heuristics.

Results: The results of the SPFAgent were disappointing. We found it difficult to express the requirements of the game in terms of attraction and repulsion forces. Some of the basic settings are intuitive, such as being attracted to food and repulsed from enemies, being repulsed from obstacles. However, there are many situations when these simple approaches fail. For instance, if an agent is between two food items, the two forces will cancel (or at least weaken) each other. Repulsion from obstacles makes it impossible for an agent to eat a food item situated right next to the obstacle and so on. One approach to solve these problems would be heuristics which turn on and off selected forces at appropriate moments.

We also found it difficult to use the SPFAgent's teamwork facilities to its advantage. The objective was for the SPFAgents to cover the area in a relatively uniform grid which partitions the resource area to avoid competing for resources. We found that the inter-agent forces need to be relatively strong to allow this grid formation. However, these strong forces were interfering with the other activities of the agents.

Development effort:

Paradigm implementation (30%) This part of the effort included the implementation of the calculation of the forces of the social potential fields and the movement calculation based on these forces.

Agent implementation (10%) Once the force and movement calculation is in place, the implementation of the agent is reduced to the choice of the c_a, c_r, σ_a and σ_r parameters of the forces.

Heuristics (20%) As the social potential field model covers only the movement of the agent, the remainder of the actions were implemented using heuristics. One of the conclusions of the development of the SPFAgent is that the number of situations which can be handled by the SPF model alone is lower than we anticipated. The experience of the implementation and the low performance of the agent led to the conclusion that a competitive implementation needs to use more heuristics and/or an additional paradigm.

Learning and tuning (40%) While there were some useful rules of thumb in the choice of the c_a, c_r, σ_a and σ_r values, the exact values had to be determined by experimentation, and the observation of the agents in specific situations. The existence of a large number of interacting forces makes this task difficult. We found that for a competitive implementation we would need to develop debugging and visualization tools, which would display the forces acting on the agent in real time.

K. CxBRAgent: context based reasoning

Context-based Reasoning (CxBR) [5], [6] is a behavior representation paradigm utilizing situational contexts as a reasoning method similar to how humans make tactical decisions. CxBR models have shown promise in experiments within simulated and physical environments [28] such as simulations involving tactical missions undertaken by submarines, tanks, and other military vehicles, as well as automobiles driving on the road. CxBR is based on the idea that:

- Any recognized situation calls for a set of actions and procedures that properly address the current situation.
- As a mission evolves, a transition to another set of actions and procedures may be required to address a new situation.
- What is likely to happen under the current situation is limited by the current situation itself.

Let us now outline the implementation of an agent in the CxBR paradigm. The main goals of the agent are contained in the *mission context*, which also specifies the initial context and the default context of an agent. At any given time, the behavior of the agent is determined by a single active major context. The mission begins with the *initial context* being active. Whenever there are no other valid contexts, the agent is controlled by the *default context*. The mission of the agent contains the list of *major contexts* and the *universal sentinel rules*. A major context completely determines the behavior of the agent at a given time; it encapsulates all the functions, rules and compatible context transitions, as well as a list of applicable *sub-contexts*. The active main context changes not only as a response to external events and circumstances, but also as a result of actions taken by the agent. Sub-contexts are low level procedures that are auxiliary in nature to major contexts. A sub-context can belong to more than one major context, encouraging code reuse.

The transitions between major contexts are controlled by sentinel rules, which constantly monitor the agent and its environment. When a context is no longer valid, a sentinel rule fires and a context switch is initiated by a transition sentinel rule. Universal sentinel rules are defined in the mission and they can initiate transitions from an arbitrary context, even if the current context is still valid. Universal sentinel rules take precedence over transition sentinel rules.

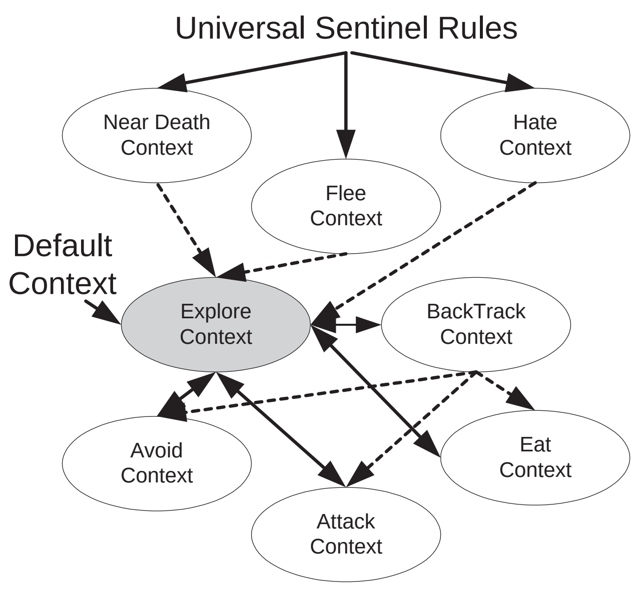

Fig. 7. The contexts implemented by the CxBRAgent and the transitions allowed by the universal sentinel rules (continuous line) and transition sentinel rules (dotted line)

Implementation: The CxBR agent was implemented using eight contexts (see Figure 7).

ExploreContext is the default and initial context of the mission. The agent keeps track of its position on an internal map, and it moves towards directions which were not previously explored. When it reaches a dead end in exploration, it will transition to BackTrackContext.

BackTrackContext is called from ExploreContext when there is nothing new to explore in the map at the current location of the agent. The agent will then retrace its movement and search for new places to explore. When it reaches a location which was not yet explored, it transitions back to the ExploreContext.

AttackContext is called from either the ExploreContext or BackTrackContext when there is an opponent agent in the sensor range and the agent can attack the other agent based on a pre-established attack criteria (e.g. the CxBR agent has 2000 more energy units than the other agent). The agent will move towards the other agent and attack when the opportunity arises. The attack will continue until the other agent is destroyed and the current agent goes back to ExploreContext or until a universal sentinel rule brings the agent into the NearDeathContext.

AvoidContext is called from either the ExploreContext or BackTrackContext when there is an opponent agent in the sensor range and the agent cannot attack the other agent. The agent will move away from the hostile agent, trying to avoid being chased or attacked by the other agent. After the agent has successfully avoided the other agent by establishing a safe distance, it will return to ExploreContext.

EatContext is called from either the ExploreContext or BackTrackContext when there is food in the sensor range. The agent will move towards the food and invoke the eat command on the food source. After the agent eats the food, it will return to ExploreContext.

FleeContext is called by the universal sentinel rules when the agent has been attacked. This context will stay active until the `fleecycle` reaches zero. Because the flee command has a high cost, it will only be invoked for a limited number of cycles, followed by the regular move command at maximum speed to continue running away.

MultiplyContext is called through the universal sentinel rules when the agent's multiplying cycle has reached zero and the agent has enough energy to multiply. The transition to this context happens with a pre-determined multiplying probability. Once in the MultiplyContext, the agent will spawn a child agent.

NearDeathContext is called through the universal sentinel rules when the energy level of the agent is below a certain threshold. This allows the agent to change its priorities of action towards self-preservation.

All the agent specific variables such as multiply cycles, attack proportion, or near death threshold can be adjusted according to desired behavior. The CxBRAgent does not perform any kind of online or offline learning. The complete functionality of the agent is hard coded in the contexts.

Heuristics: As the CxBR paradigm allows for a direct encoding of the heuristics, no heuristics outside the paradigm was needed. The tuning of parameters such as the preference between the avoidance and attack behaviors, the mating interval, a level of energy where the NearDeathContext is activated and so on was done on an empirical basis. The CxBRAgent had also implemented an incrementally built internal map of the environment, which allowed for a better exploration behavior.

Results: The CxBR paradigm provided a simple way to analyze and encode the behavior of the agents. As it leaves considerable freedom to the programmer in the implementation of the contexts, it allowed the developer to implement the desired behavior directly. In these respects, the implementation experience resembled the one in the naïve programming agent - however, the CxBR paradigm enforced a more structured work methodology and a greater implementation discipline. This allowed the developer to implement relatively sophisticated behaviors and transfer human expertise to the agent.

Development effort:

Paradigm implementation (20%) This part of the development effort covered the implementation of the generic part of the CxBR model, the implementation of the context engine, the contexts and the subcontext model, as well as the generic framework for the sentinel rules.

Agent implementation (60%) The main part of the development was spent on the implementation of the contexts and the sentinel rules.

Heuristics (0%) As the human knowledge about the game was explicitly encoded in the contexts, there was no separate effort for developing heuristics.

Learning and tuning (20%) The CxBR agent was tuned by observing its behavior in test runs and adjusting the context parameters and the sentinel rules accordingly.

L. KillerAgent: simple heuristics

The goal of implementing KillerAgent was to achieve maximum game performance with minimal amount of code.

Implementation: The KillerAgent's implementation does not follow any paradigm, relying only on very simple heuristics.

Heuristics: KillerAgent uses a set of simple heuristics to implement a highly aggressive behavior. The agent keeps track of direction, failed moves and successful moves. If the number of consecutive failed moves exceeds a predefined threshold, the agent changes direction. When the agent detects food in its sensor range it will immediately navigate to it and eat the food. If the agent senses other agents in its sensor range, it will attack them; attacking takes priority over any other action. If the agent itself is attacked it will flee. KillerAgent does not utilize the multiply feature or any teamwork strategies.

Results: The KillerAgent's behavior corresponded to the intentions of the developers. Interestingly, even with its very simple heuristics and unbalanced behavior, the performance of the KillerAgent puts it roughly in the middle of the field, outperforming all learning based paradigms, but showing a lower performance than most explicitly programmable paradigms.

Development effort:

Paradigm implementation (0%)

There is no paradigm for the KillerAgent.

Agent implementation (0%)

There is no paradigm-compliant part of the implementation.

Heuristics (100%)

The full development effort of KillerAgent went into the implementation of the heuristics. We should note that this implementation effort was much smaller than the one for the other agents in this study. The total development time was several hours, rather than about weeks.

Learning and tuning (0%)

No learning or tuning was performed. The implementation of the heuristics were done by the initial assessment of the game by the developer. The heuristics were not tuned or modified based on observation of the agent in the game.

IV. Implementation effort

An important consideration in the choice of an agent paradigm is the effort and complexity of the implementation. Everything else being equal, a simpler implementation is frequently preferred, as it leads to a more maintainable code, with usually smaller number of defects.

The software engineering metrics for each implementation are shown in Table VII. The total lines of code (LOC) in conjunction with total lines of code inside all method bodies (MLOC) are an indicator of the development effort for each paradigm. In addition, we use cyclomatic complexity [12], to measure the complexity of the conditional flow within each implementation. We show values for the maximum cyclomatic complexity (MCC) found in a single method as well as the sum of cyclomatic complexity (SCC) over the complete implementation of the agent. All these measurements were applied strictly to the specific agent code and did not include the services provided by the environment neither external libraries. The only paradigm using an external library was NeuralLearner.

Table VII shows that the NeuralLearner required more developer effort than any other paradigm. Also, the NeuralLearner was implemented using an external neural network library not included in the software metric calculations. The behavioral models such as social potential fields and crowd modeling have a relatively low count of lines, while agents with explicit programming models, such as CBR, CxBR or RuleBasedAgent have a relatively large number of code. As expected, KillerAgent is the most trivial agent, implemented with only 96 lines of code where 75 lines of code reside in method bodies.

We use the SCC to MLOC ratio for comparison of cyclomatic complexity between the paradigms. AffectiveAgent, NaiveAgent and CxBRAgent all have ratios greater than 0.4. This means that for every line of code (inside method bodies) there is on average 0.4 conditional flow statements. These values indicate high complexity in learning and decision making within the paradigms. In contrast, Reinforcer, CrowdAgent and KillerAgent have low SCC to MLOC rations ranging from 0.22 to 0.27.

Table VIII summarizes the partition of the development effort. While at the presentation of the individual agents we discussing this partition for each agent in part, let us now compare these efforts across the various agents in the study.

We need to treat separately the two “paradigm-less” implementations NaïveAgent and KillerAgent. For these agents, there is no effort spent on the implementation of the paradigm, all the efforts going into the heuristics and the tuning of the agent. In the following comparisons we will ignore these two agents (except where explicitly referred to).

TABLE VII

CANONICAL SOFTWARE METRICS FOR EACH PARADIGM. LOC = LINES OF CODE, MLOC = METHOD LINES OF CODE, MCC = MAX CYCLOMATIC COMPLEXITY, SCC = SUM OF CYCLOMATIC COMPLEXITY. PARADIGMS MARKED WITH (*) HAVE EXTERNAL LIBRARIES THAT ARE NOT INCLUDED IN THE METRIC CALCULATIONS.

Name	**LOC**	**MLOC**	**MCC**	**SCC**	**SCC/MLOC**
AffectiveAgent	223	117	31	48	0.41
GenProgAgent	647	477	25	143	0.30
Reinforcer	313	236	13	52	0.22
CBRAgent	1357	1060	31	320	0.30
RuleBasedAgent	706	536	18	176	0.33
NaïveAgent	327	289	81	118	0.40
GamerAgent	1259	944	24	343	0.36
CrowdAgent	425	345	13	87	0.25
NeuralLearner*	1454	1119	36	297	0.27
SPFAgent	229	179	48	55	0.30
CxBRAgent	1135	689	21	298	0.43
KillerAgent	92	75	17	20	0.27

First, we note that the paradigm implementation was a small to moderate (10%-30%) part of the development effort for each agent. None of the developers have reported the paradigm implementation to be challenging or problematic. The reason for this is that for most paradigms there are well specified implementation instructions, pseudocode or even existing software libraries.

The agent implementation part, which, according to our definition, is the part of the implementation process in the terms of the paradigm, ranges from 10% to 70%. It is largest for the explicitly programmable paradigms, such as the rule based, context based reasoning, and, to some level, case based reasoning, while it is low for the agents relying on learning.

The percentage of the development effort allocated to heuristics depends on whether the paradigm can, on its own, cover the range of actions and situations encountered by the agent. For instance, the game theoretic paradigm covers only the cases of encounters between two agents, while the social potential fields and crowd models cover only the movement of the agents. In many cases, the choice of using more or less heuristics is frequently a developer decision. In the course of the development process, we have found several instances where heuristics were introduced because of the developer frustration with the limitations of the paradigm.

TABLE VIII

SUMMARY OF THE PARTITION OF THE DEVELOPMENT EFFORT (PERCENTAGES)

Name	**Paradigm implementation**	**Agent implementation**	**Heuristics**	**Learning and tuning**
AffectiveAgent	10%	20%	20%	50%
GenProgAgent	20%	30%	10%	40%
Reinforcer	10%	10%	0%	80%
CBRAgent	20%	40%	20%	20%
RuleBasedAgent	20%	70%	0%	10%
NaïveAgent	0%	0%	80%	20%
GamerAgent	10%	20%	60%	10%
CrowdAgent	20%	20%	40%	20%
NeuralLearner	10%	10%	0%	80%
SPFAgent	30%	10%	20%	40%
CxBRAgent	20%	60%	0%	20%
KillerAgent	0%	0%	100%	0%

The learning and tuning component covers the automatic learning and the manual adjustment of the parameters of the agent. The boundaries between learning and tuning are fuzzy, as the learning process frequently requires manual intervention in the choice of learning scenarios, adjustment of learning parameters and so on. Naturally, for the learning based paradigms this component of the development effort is larger: 80% for the reinforcement learning agent and the neural networks agent and 40% for the genetic programming agent.

A certain amount of tuning is necessary for every agent. However, we found that for some agents developers spent a very long time tuning the agent (such as the case of social potential fields and the affective agent). This is usually a sign of the fact that the developer is having difficulties expressing the derived behavior in the constraints of the paradigm. In many cases this happens if the tuning of the behavior of the agent happens through indirect parameters with complex interactions. Better debugging and visualization tools might help improve this process. However, if the tuning of parameters becomes a very large part of the development process, it is usually a sign of a major problem, and the developers need to re-assess the implementation paradigm.

TABLE IX

FFM GAME CONFIGURATION USED IN ALL PERFORMANCE EVALUATION MEASUREMENTS.

Configuration	**Value**	**Description**
Map size	300x200	Width and height
Simulation cycles	1000	Game loop iterations
Initial energy unit population	50	Energy resources at start of game
Maximum energy unit population	60	Maximum number of concurrently available energy resources
Energy unit growth	0.1	Energy growth per simulation cycle
Maximum energy unit level	800	Maximum amount of energy available in each energy resource
Initial agent energy	8000	Agent energy at start of game
Agent processing cost	1	Processing cost applied at each game loop iteration
Agent maximum flee speed	15	Speed at which an agent can flee
Agent weight	0.1	Determines energy cost for movement
Agent maximum speed	10	Maximum normal speed which an agent can move
Agent energy intake	200	Maximum amount of energy that can be gathered per iteration
Sensor area	1600	Sensor area for each agent
Multiply cost	4000	Energy cost for invoking multiply command
Attack cost	100	Energy cost for invoking attack command
Agent attacked cost	1000	Energy removed from agent when being attacked

V. Experiments

In the following we present the results of a series of test runs which quantitatively measure the performance of the implementations under various scenarios. We repeat our disclaimer: the results do not reflect the potential performance of the paradigms themselves, only the performance of a specific agent, implemented by reasonably proficient developers, in a limited amount of time, following the paradigm. We are publishing this information in the hope that it provide a useful input to software developers in similar situations.

We performed two types of experiments. In the *non-competitive experiments* the agents are acting on a map populated with a small number of agents of the same type. In *competitive experiments*, the map contains two equal size teams of opponent agents. All experiments were executed using the FFM game configuration presented in Table IX. The map we used is shown in Figure 2.

TABLE X

INDIVIDUAL AGENT PARADIGM EVALUATION RUNS. DATA IS COLLECTED OVER 50 GAMES.

Name	Agents	Survivors	Energy	Exploration	Move	Attack	Flee	Multiply	Eat
AffectiveAgent (A1)	12.46	8.02	2784.14	0.10	421.80	0.83	0.57	0.89	11.94
GenProgAgent (A2)	1.00	1.00	17144.27	0.32	805.06	0.00	0.00	0.00	93.94
Reinforcer (A3)	1.00	1.00	7330.79	0.11	356.46	0.00	0.00	0.00	0.40
CBRAgent (A4)	1.00	1.00	13388.70	0.26	770.08	0.00	0.00	0.00	53.82
RuleBasedAgent (A5)	2.36	2.36	19582.18	0.33	659.67	0.00	0.00	0.32	83.11
NaiveAgent (A6)	29.24	7.24	1894.32	0.52	349.31	0.00	0.00	0.97	19.97
GamerAgent (A7)	2.16	2.16	21105.46	0.39	588.96	0.00	0.00	0.37	107.80
CrowdAgent (A8)	2.62	2.62	6629.97	0.10	432.65	0.00	0.00	0.59	17.41
NeuralLearner (A9)	1.00	1.00	9294.27	0.10	957.52	0.00	0.00	0.00	15.68
SPFAgent (A10)	11.18	6.48	2225.94	0.08	557.78	0.63	0.00	0.82	7.32
CxBRAgent (A11)	2.80	2.56	27705.05	0.61	564.42	0.00	0.00	0.39	179.25
KillerAgent (A12)	1.00	1.00	19152.07	0.41	780.30	0.00	0.00	0.00	117.28

A. Results: Non-competitive

Non-competitive experiments are performed by running games where the agents are acting in an environment populated with friendly agents of the same type. The goal of the agents is to survive and maximize the sum of their energy levels. There is still a level of competition for the finite amount of resources, although the agents might choose to implement an equalitarian sharing of the resources. From the implementation point of view, agents recognize as friendly the agents with the same type string. We can still create opponent agents from the same code by labeling their type differently, e.g. CBRAgent-1, CBRAgent-2 and so on.

The statistics in Table X are the averages of 50 game runs for each paradigm and show the average values for total number of agents created during the game; the number of survivors at the end of the game; the average agent energy at the end of the game; the map exploration percentage (ranging from 0-1) and the average number of move, attack, flee, multiply and eat commands invoked. Naturally, the attack and flee statistics will be low in these experiments since most agents were designed to never attack friendly agents. The box-plots in Figure 8 and Figure 9 show the median, inner quartile, outer quartile and outliers of the collected data for each agent paradigm.

Through these diagrams we can interpret the general strategies of the paradigms, as well as their relative success in following their strategy. The number of agents created during the game shows that only about half of the implementations choose to take advantage of the multiply feature of the game. In general, the paradigms which use teamwork had chosen to multiply more often, such that they can better deploy their teamwork strategies. SPFAgent, AffectiveAgent and CrowdAgent exhibit this behavior. The strategy of the NaiveAgent is to create a large number of agents, however, most of these agents failed to survive. In the experimental setup we can see that the cost for multiplying is high. Therefore, NaïveAgent's overall performance in terms of maintaining a healthy population is poor. NaïveAgent's main weakness is that it can not adapt to changes in game setup without re-scripting parts of the agent.

CxBRAgent, GamerAgent and RuleBasedAgent are the most efficient paradigms at gathering energy resources and exploring the map. Notice, that this performance depends mostly on the heuristics. For instance, in the absence of opponent agents on the map, the main paradigm of the game theoretic agent was never invoked in these runs. There is a strong correlation between average agent energy and exploration. CxBRAgent, GamerAgent and RuleBasedAgent are also able to maintain a healthy population.

B. Results: Competitive

The results for the competitive games were obtained through a pair-wise scrimmage of the paradigms. Every game involved agents from two paradigms. The initial number of agents from each paradigm was 3. During the game, the number of agents can change through the multiply action, or by agents being destroyed or starved. The type of agent whose sum of energy was the largest was declared the winner. Table XI shows the percentage of wins over 64 games for each potential pair of paradigms.

The results are an average of 64 games. The final results are presented in the win-loose matrix in Table XI. The results show the percentage of wins and losses over the repeated games in terms of the total amount of energy of the paradigms agents at the end of the game.

As expected, the explicitly programmable paradigms provided the best results: CxBRAgent, GamerAgent, RuleBasedAgent and NaïveAgent are the most successful paradigms in the competitive experiment.

NaïveAgent presents an interesting case: while most paradigms fare better if they are in a non-competitive environment, NaïveAgent, which has relatively sophisticated scripts for encounters with opponent agents, but no self-regulating model, fares better in a competitive environment. We find this observation illustrative for the strengths and limitations of the scripting based approaches.

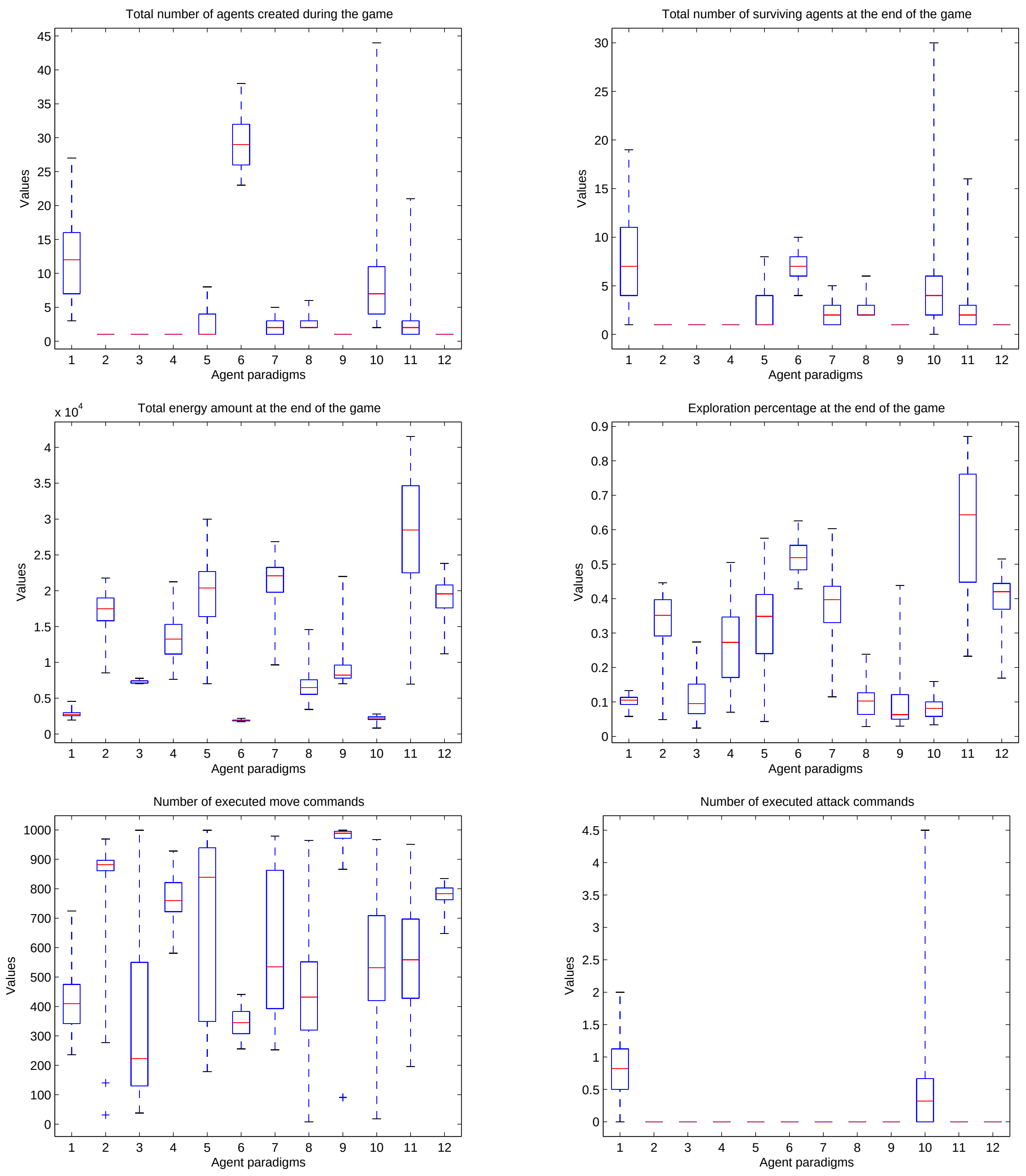

Fig. 8. Results of the non competitive experiments. The box plots show lower quartile, median, upper quartile and outliers of 50 repeated experiments.

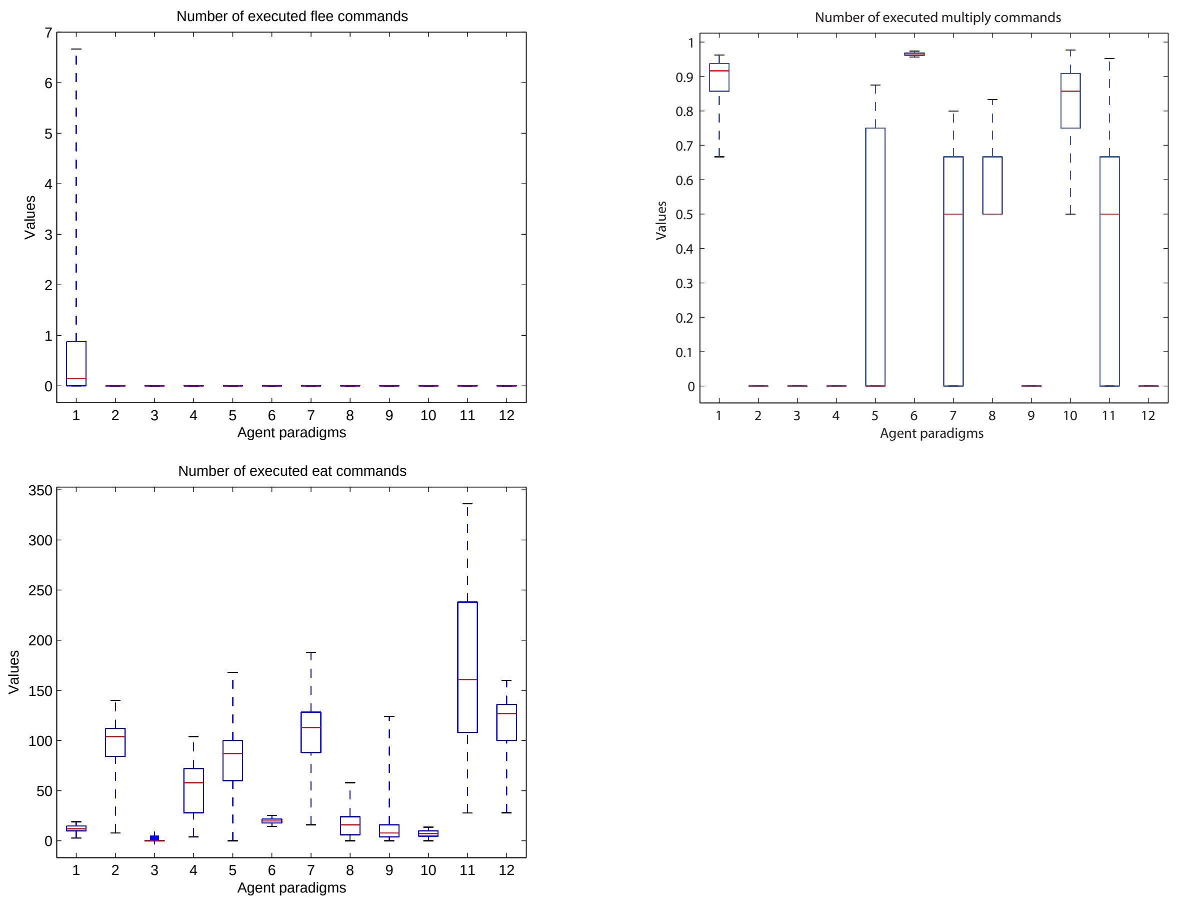

Fig. 9. Results of the non competitive experiments (cont'd). The box plots show lower quartile, median, upper quartile and outliers of 50 repeated experiments.

VI. QUALITATIVE FINDINGS

A. Development process

The process of developing the twelve agents for this comparative study was organized in two stages. In the first, closed phase, the developers were working on the agents in isolation. Only a very simple random agent and the initial version of the KillerAgent was provided as an illustration of the API. The testing in this phase was performed mostly in non-competitive settings. Some developers have tested their agents in competitive scenarios, by running their agents under different type strings. In some learning based paradigms the developers created "bootstrap" agents as training partners or ways to collect training information.

In the second, open phase of the development process, regular competitive runs were organized,

TABLE XI

WIN-LOOSE MATRIX FOR PAIR-WISE AGENT COMPETITIONS. DATA IS COLLECTED OVER 64 GAMES. THE STATISTICS SHOW PERCENTAGE OF WINS AND LOSSES WITH RESPECT TO FINAL GROUPED ENERGY IN ROWS AND COLUMNS RESPECTIVELY.

	A1	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**A10**	**A11**	**A12**
A1	-	50.00	84.38	56.25	15.63	0.00	23.44	76.56	71.88	59.38	1.56	42.19
A2	50.00	-	98.44	64.06	21.88	28.13	12.50	78.13	75.00	81.25	3.13	31.25
A3	15.63	1.56	-	1.56	1.56	1.56	1.56	17.19	1.56	51.56	0.00	0.00
A4	43.75	35.94	98.44	-	25.00	18.75	17.19	62.50	60.94	73.44	1.56	21.88
A5	84.38	78.13	98.44	75.00	-	31.25	51.56	92.19	85.94	92.19	32.81	78.13
A6	100.00	71.88	98.44	81.25	68.75	-	40.63	76.56	84.38	75.00	31.25	65.63
A7	76.56	87.50	98.44	82.81	48.44	59.38	-	92.19	93.75	78.13	18.75	78.13
A8	23.44	21.88	82.81	37.50	7.81	23.44	7.81	-	43.75	57.81	1.56	10.94
A9	28.13	25.00	98.44	39.06	14.06	15.63	6.25	56.25	-	65.63	3.13	9.38
A10	40.63	18.75	48.44	26.56	7.81	25.00	21.88	42.19	34.38	-	6.25	28.13
A11	98.44	96.88	100.00	98.44	67.19	68.75	81.25	98.44	96.88	93.75	-	95.31
A12	57.81	68.75	100.00	78.13	21.88	34.38	21.88	89.06	90.63	71.88	4.69	-

which allowed the developers to observe the behavior of their own and of competing agents. However, the developers could not use opponent agents in scripted training scenarios. The developers had the opportunity to further develop or fine tune their agents to improve their performance in these competitive runs. We encouraged the developers to share their heuristics with each other, and emphasized that the goal is not to obtain the agent with the highest performance, but to best express the "spirit" of the paradigm. Nevertheless, a certain level of competitive pressure did develop between the developers.

By allowing encounters between the agents during the development process, we tried to guarantee that the agent can not win the game by a "surprise tactic" which might have been overlooked by the other developers. An example of this occurred when one of the developers discovered a hole in the game API, through which the agent could change the type string during the game and thus prevent attacks by masquerading as a friendly agent. This security hole was patched.

The comparison study allowed us to observe the software engineering approach through which the developers analyzed the problem, approached the implementation, debugged or refined the agents. We were also interested in the ways whether the developers changed their software engineering methodology in response to the initial performance results. We found that the choice of the paradigm was critical in determining the flow of the development process. We encouraged the developers to use disciplined software engineering approaches, in particular the iterative waterfall model, following clear cycles of requirements analysis, design, implementation and testing. We found that this model was followed very well in paradigms which involved explicit programming.

For paradigms where the behavior of the agent was determined by learning or encoded in variables such as force fields or affective or grouping parameters, software engineering techniques could only be applied to the development of the framework of the implementation. The main problem was that the results of a certain chunk of development effort was difficult to predict. Re-starting the learning process with a new set of parameters had frequently led to an agent with a lower performance. Adjusting the parameters of the force field to achieve a new behavior frequently invalidated previously achieved behaviors. All this made it difficult to apply quality assurance techniques.

B. *Explicit vs. implicit programming*

We found that the explicit programming paradigms (scripting, rule based, CxBR and, to a certain level CBR) were easier to program, yielded a steady improvement in their performance during the development process and did not require costly rewriting (although, occasional Java refactorings were performed). The human knowledge integrated in these agents allowed them to outperform their implicit cousins in test runs.

All paradigms which relied on learning (neural networks, genetic programming and reinforcement learning) have been successful in creating agents which can survive in the environment in the absence of predators. For these paradigms the developers spent significant time designing learning scenarios in which the learning process can be steered in the right direction. A major difficulty was that these scenarios had to be populated with agents. The most problematic turned out to be the neural network agent, whose supervised learning algorithm required an existing agent to perform the scenario to generate "correct" input and output pairs. Admittedly, this issue would be irrelevant in applications where such an imitation target exists and can be used at will.

Another challenge was the acquiring of meaningful learning experiences. Many developers expressed frustration that the Feed-Fight-Multiply game is "not a good target for learning". Indeed, most of the life of an agent is spent in random wandering. Thus an overwhelming majority of the actions of the agent carry no reward: in fact, they have a small penalty in the form of the energy expenditure for movement. Fighting carries a heavy initial penalty, thus all learning algorithms developed an overly defensive behavior. The energy reward for destroying an opponent agent is easily quantifiable but almost impossible to accidentally discover. The indirect benefits, such as the removal of a competitor for resources are very difficult to quantify. Nevertheless, we find that the FFM game resembles many real environments in that meaningful learning experiences are rare, and the chance of accidental discovery of a correct solution negligible.

Still, it would be a mistake to conclude that learning paradigms are not useful for developing embodied agents. Rather, before the agent paradigm, its granularity level and its coverage is chosen, the developers need to perform a careful analysis of what can, and what (most likely) can not be learned in the given environment. We found that in the context of the FFM game there are three clearly distinguishable subjects of learning.

Parameter learning is the search for the optimal value of certain numerical parameters with continuous values which determine the behavior of the agent. An example is an agent which learns the best value of the speed to be used in exploring the environment for food, such that the explored territory is maximized, while the energy expenditure minimized relative to the possible reward.

Policy learning is the learning of a model which can decide which behavior (chosen from a discrete, limited set) has to be applied in various circumstances. An example is an agent which learns to decide whether to flee or fight an opponent agent.

Algorithm learning is the learning of a multi-step algorithm whose application benefits the agent. An example would be an agent which discovers an algorithm for visiting food locations in the optimal order.

All learning agents were successful in parameter and policy learning and performed better than human intuition for these tasks. Algorithm learning however, poses significant theoretical and practical problems. From the studied paradigms only genetic programming is able to explicitly represent a general purpose algorithm, but algorithms can be encoded indirectly in the neural network weights or the reinforcement function. However, the size of the data structures associated with the current implementations significantly limit the complexity of the algorithms which can be represented.

The closest thing to algorithm learning were the approaches developed by the reinforcement learning, case-based reasoning and neural network agents for collision avoidance, or approaching the food items. None of the models were successful in developing path planning algorithms (such as an approach for visiting food locations). The genetic programming approach was the only learning model which would represent (and, theoretically evolve) such a model. However, an examination of the evolved genetic programs showed that they are more policy learning than algorithm learning.

The inability of the learning paradigms to discover algorithms inevitably put them at a disadvantage against explicitly programmed agents which can deploy advanced algorithms such as A* path planning and incrementally built internal maps. The practical conclusion is that developers can obtain the best performance by limiting learning to parameter and policy learning, and supply the agents with explicitly programmed algorithms.

C. A rose by another name

Many paradigms led to surprisingly similar implementations, while giving very different interpretations to the variables involved. For instance, the developers of affective models AffectiveAgent and CrowdAgent used the variables describing the emotional states as just another state variable and applied regular programming techniques on them, on hindsight labeling them with emotional significance. For instance, the write-up for affective agents contained terms such as "emotional quagmire" for being stuck in a local minima. On the other hand, many developers assigned anthropomorphic significance to their state variables ("the agent gets angry"), even if their paradigm did not require it.

A similar phenomena was observed related to contexts. Context based reasoning, as implemented in the CxBRAgent requires the developer to actively identify the context of the agents operation and describes ways to handle it. However, the idea of a context was used in similar ways in other approaches. RuleBasedAgent represented context with generated facts inserted in the knowledge base. The CBRAgent utilized its "cases" in similar ways to contexts. The tactical behaviors evolved in the first step of the evolution of the GenProgAgent are essentially behaviors appropriate for certain contexts, while the second step of the evolution determines the context change rules.

The SPFAgent and the CrowdAgent ended up deploying very similar attraction and repulsion forces, starting from different physical models and very different high level interpretations.

D. The flight into heuristics

Although the game was not easy (humans playing it at first time did not perform better than agents), human users could easily come up with rules of thumb which can be used to make decisions in the game. **The ease of representing these heuristics in the agents was a determining factor in the performance**. Paradigms which can do this only in a very convoluted way (such as learning based models, and, in lesser degree, the potential field, crowd and affective models), had scored the worst in direct comparisons. What is more, certain types of mistakes committed by the agents were obvious even for casual observers, and led to significant developer frustration.

Interestingly, developer frustration with certain types of suboptimal agent behaviors were not always in direct proportion to its impact on game performance. For instance, due to the game physics, the agents could easily overshot the food location if they approached it with a too fast speed. In this case, many agents had to perform several back-and-forth movements to position themselves next to the food.

However, although damaging to the agents' "perceived performance", this action actually consumed very little energy, and had almost no influence over the performance of the agent. On the other hand, incorrect decisions regarding the attack or flee behavior, or a bad choice in the multiplying behavior was not detected by onlookers.

As a result, many developers have spent significant effort to fix the visually obvious mistakes. In most cases, this involved a "flight into heuristics". This tendency of abandoning the paradigm for ad hoc fixes for perceived performance problems was especially visible in the second, open phase of development.

E. Paradigm-specific goals and generic paradigms

This study considered a large number of paradigms, bound together only by their ability to implement an agent which can function in the Feed-Fight-Multiply environment. Most of these paradigms were not originally alternatives to each other. Some paradigms were originally proposed as a generic programming model (for instance, rule based systems), while others were developed as solutions for specific problems (such as crowd modeling). The historical evolution of the paradigm and computer science in general, had frequently changed the positioning of the paradigm. Some paradigms were found to be applicable to a larger number of goals than originally designed. For instance, in this study we have found that context based reasoning, originally proposed for the simulation of anthropomorphic agents, was found to be a good methodology for the design of performance oriented agents.

For other paradigms it was found that they outperform other approaches on a more limited set of problems than originally proposed. For instance, neural networks remain a vigorous research area with strong applications, but they are not viewed any more as a general purpose solution to the goals of artificial intelligence. Finally, some paradigms have shifted their focus from the domains they were initially proposed.

The open nature, and the multiple paths to success of the Feed-Fight-Multiply game allowed us to evaluate the strengths and weaknesses of the various paradigms against different stated goals. Raw performance (in terms of maximum energy or highest number of agents) is not the only criteria against which to evaluate an agent. If an agent is used in a simulation study, for instance, the *anthropomorphism* of an agent - its ability to simulate the behavior of a human under similar conditions is the main goal. A related concept is the *believability* of the agent. In contrast to anthropomorphism, which sets hard standards for simulating human behavior, believability strives to create an *illusion* of natural behavior. This is important in entertainment applications, such as games but also in training and interactive tutoring systems. Agents used as game opponents might need to be designed to provide a predetermined level of performance in a consistent manner, such that they act as enjoyable game partners.

Different paradigms might be better choices depending on the stated goal of the agent. The goal of the crowd modeling is not to design an agent which would provide a high performance, but to provide an agent which models the behavior of humans in a similar situation. Anthropomorphism and/or believability is also the central goal of the affective computing paradigm. Several studies have indicated that the use of affective model can also provide agents with higher performance [23]. In our study we did not find that the affective model would provide an immediate performance advantage, however affective concepts were used successfully as design principles in many agents which obtained good performance results. It is difficult to say if these performance advantages were obtained because of the use of affective concepts, or whether the developers simply gave an affective interpretation to the terms *after* the design was completed.

Table XII concisely summarizes the application domain generalizations, specifications or shifts we have experienced during the design and implementation of our agents. Note that the "initially proposed application domain" column is specifying the "common wisdom" associated with these paradigms as perceived by the authors, while the "perceived best use" is based on the implementation experience of the Feed-Fight-Multiply agents.

F. "Paradigm-pure models considered harmful" or "Let us now praise the paradigm"?

Finally, let us consider a very general question: are paradigms useful at all in agent development?

To answer this, we need to first clarify the difference between an agent development paradigm and an algorithm. A paradigm is a guiding principle around which the agent development is organized. Some of the paradigms compared in this paper are also general purpose algorithms, which can be used in limited parts of the agent, without requiring an overall commitment from the developers.

The question is whether there is any advantage of organizing the development of an agent around a paradigm. This study required paradigm-pure implementations from the developers, that is, the developers were not allowed to borrow elements from other paradigms. In general, academic research projects would more likely be insisting on paradigm purity, as opposed to product development projects in the industry. This is partially justified by the different deliverables of an academic research group vs. a development team in the software or hardware industry. If our object is the study of a certain paradigm, paradigm purity is a natural choice. But what about the case when the objective is to optimize some type of performance measure?

In our study, the developers' subjective opinion was strongly against the paradigm purity requirement. As we have shown in the "flight into heuristics" section, many developers felt a significant peer pressure

TABLE XII

INITIALLY PROPOSED APPLICATION DOMAIN OF THE PARADIGMS VS. PERCEIVED BEST USE IN THE CONTEXT OF THIS STUDY

Paradigm	**Initial proposition**	**Perceived best use**
Affective programming	Anthropomorphism, believability, performance	Anthropomorphism, believability, if loosely interpreted, general purpose design methodology.
Genetic programming	General purpose learning	Parameter and policy learning
Reinforcement learning	General purpose learning	Parameter and policy learning
Case based reasoning	Algorithm learning	Policy learning
Rule based, forward reasoning	General purpose AI	General purpose
Naïve programming (scripting)	Believability (in games)	General purpose, performance
Game theory	Decision making	Decision making
Crowd modeling	Anthropomorphism, credibility	Anthropomorphism, credibility
Neural networks	General purpose AI	Parameter learning
Social potential fields	General purpose movement control	Specialized movement control
Context based reasoning	Anthropomorphism	General purpose design paradigm.

to add additional heuristics, at the expense of the paradigm, to correct perceived performance and behavior problems.

Certainly, some paradigms made it exceedingly difficult to transfer human knowledge and rules of thumb to the agents, leading to performance problems. But, the other side of the coin is that an unchecked free-form development leads to a random collection of heuristics whose interactions are poorly understood. A good example is the NaïveAgent whose heuristics provided good performance in competitive scenarios, but which starved itself of resources by exponential multiplication when it was alone on the map. Of course, this could have been corrected with another heuristic, but it still leaves the basic problem unsolved.

In general our study supports the choice of a paradigm which can provide a coherent narrative to the development process, but it is not so restrictive that it would hinder the transfer of human knowledge to the agent. In particular, the ability to incorporate pre-existing algorithms is critical to the development of high performance agents.

VII. Conclusions

In this paper we reported on a study comparing 12 paradigms for implementing autonomous agents in an environment inspired from artificial life and turn based strategy games. The paradigms cover a wide variety of approaches, from high level, logic based approaches to behavior based, physical models. Although apparently we compare "apples to oranges" these approaches are united by the fact that all of them can serve as the central paradigm of an agent implementation. When a paradigm did not cover all the aspects of the agent implementation, we complemented them with simple heuristics.

We have found that the choice of the paradigm determines the software development process and requires a different set of skills from the developers. In terms of raw performance, we found that the best performing paradigms were those which (a) allowed the knowledge of human experts to be explicitly transferred to the agent and (b) allowed the integration of well-known, high performance algorithms. We have found that maintaining a commitment to the chosen paradigm can be difficult; there is a strong temptation to offer shallow fixes to perceived performance problems through a "flight into heuristics". Our experience is that a development process without the discipline enforced by a central paradigm leads to agents which are a random collection of heuristics whose interactions are not clearly understood. We have also found that many, apparently different paradigms, are different labels of fundamentally similar principles, such as the notion of modeling movement through forces, the notion of context, and so on.

This study is part of our ongoing effort to investigate and compare methodologies for building embodied agents. The source code and playable simulation runs is available from the website `http://netmoc.cpe.ucf.edu/Yaes/index.html`.

This research was partially supported by a fellowship from the Interdisciplinary Information Science and Technology Laboratory (I^{2}Lab) at the University of Central Florida.

Research was sponsored in part by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-06-2-0041. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation heron.

References

[1] A. Aamodt and E. Plaza. Case-based reasoning: foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59, 1994.

[2] E. Berne. *Games People Play : The basic handbook of transactional analysis*. Ballantine Books, 1996.

[3] L. Bölöni and D. Turgut. YAES - a modular simulator for mobile networks. In *Proceedings of the 8-th ACM/IEEE International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWIM 2005)*, pages 169–173, October 2005.

[4] E. Bouvier, E. Cohen, and L. Najman. From crowd simulation to airbag deployment: Particle systems, a new paradigm of simulation. *J. Electronic Imaging*, 6(1):94–107, 1997.

[5] A. J. Gonzalez and R. H. Ahlers. A novel paradigm for representing tactical knowledge in intelligent simulated opponents. In *Proceedings of the International Conference of Industrial Engineering Applications and A.I. and Expert Systems*, pages 515–523, 1994.

[6] A. J. Gonzalez and R. H. Ahlers. Context-based representation of intelligent behavior in simulated opponents. In *Proceedings of the Computer Generated Forces and Behavior Representation Conference*, 1996.

[7] S. Hanks, M. E. Pollack, and P. R. Cohen. Benchmarks, test beds, controlled experimentation, and the design of agent architectures. *AI Magazine*, 14(4):17–42, 1993.

[8] Hodjat and Shahrzad. Introducing a dynamic problem solving scheme based on a learning algorithm in artificial life environemtns. In *IEEE International Conference on Neural Networks. IEEE World Congress on Computational Intelligence.*, pages 2333–2338, 1994.

[9] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[10] J. R. Koza. Genetically breeding populations of computer programs to solve problems in artificial intelligence. In *Proceedings of the Second International Conference on Tools for AI*, pages 819–827, November 1990.

[11] M. Likhachev, M. Kaess, and R. C. Arkin. Learning behavioral parameterization using spatio-temporal case-based reasoning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 1282–1289. IEEE, 2002.

[12] McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[13] C. McPhail and R. T. Wohlstein. Individual and collective behaviors within gatherings, demonstrations, and riots. *Annual Review of Sociology 9*, pages 579–600, 1983.

[14] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[15] J. Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[16] R. W. Picard. *Affective Computing*. MIT Press, 2000.

[17] J. Reif and H. Wang. Social potential fields: A distributed behavioral control for autonomous robots. In *Proceedings of the International Workshop on Algorithmic Foundations of Robotics (WAFR)*, pages 431–459, 1995.

[18] J. H. Reif and S. R. Tate. The complexity of n-body simulation. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP-1993)*, pages 162–176, 1993.

[19] C. W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, 1987.

[20] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel distributed processing: explorations in the microstructure of cognition, vol. 1: Foundations*, pages 318–362. MIT Press, Cambridge, MA, USA, 1986.

[21] R. C. Schank. *Dynamic Memory: A Theory of Reminding and Learning in Computers and People*. Cambridge University Press, New York, NY, USA, 1983.

[22] M. Scheutz. Agents with or without emotions?. In *Proceedings of the Fifteenth International Florida Artificial Intelligence Research Society Conference*, pages 89–93, 2002.

[23] M. Scheutz. Useful roles of emotions in artificial agents: A case study from artificial life. In *The Nineteenth National Conference on Artificial Intelligence - AAAI-04*, pages 42–48, 2004.

[24] C. M. Steiner. *Scripts People Live: Transactional Analysis of Life Scripts*. Grove/Atlantic, 1990.

[25] R. S. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3(1):9–44, 1988.

[26] J. Thollot, L. Heigeas, A. Luciani, and N. Castagne. A physically based particle model of emergent crowd behaviors. In *Proceedings of the International Conference on Computer Graphics and Vision (GraphiCon-2003)*, 2003.

[27] B. Tomlinson, M. L. Yau, and E. Baumer. Embodied mobile agents. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 969–976, New York, NY, USA, 2006. ACM Press.

[28] V. C. Trinh, B. S. Stensrud, and A. J. Gonzalez. Implementation of a prototype context-based reasoning model on a physical platform. In *Swedish American Workshop on Modeling and Simulation*, 2004.

[29] T. Vicsek, D. Helbing, and I. Farkas. Simulating dynamical features of escape panic. *Nature*, pages 487–490, 2000.

[30] L. Yaeger. Computational genetics, physiology, metabolism, neural systems, learning, vision and behavior or PolyWorld: Life in a new context. In C. G. Langton, editor, *Artificial Life III, Proceedings Volume XVII*, pages 263–298. Addison-Wesley, 1994.

[31] G. N. Yannakakis, J. Levine, J. Hallam, and M. Papageorgiou. Performance, robustness and effort cost comparison of machine learning mechanisms in FlatLand. *IEEE Proceedings of the 11th Mediterranean Conference on Control and Automation*, June 2003.

[32] X. Yao. Evolutionary artificial neural networks. In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 33, pages 137–170. Marcel Dekker Inc., 1995.