ISSN 1392–124X (print), ISSN 2335–884X (online) INFORMATION TECHNOLOGY AND CONTROL, 2014, T. 43, Nr. 1

Neuroevolution Based Multi-Agent System with Ontology Based Template Creation for Micromanagement in Real-Time Strategy Games

Iuhasz Gabriel, Viorel Negru, Daniela Zaharie

West Univeristy of Timisoara,
Timisoara, Romania
e-mail: iuhasz.gabriel@info.uvt.ro, vnegrul@info.uvt.ro, dzaharie@info.uvt.ro

crossref http://dx.doi.org/10.5755/j01.itc.43.1.4600

Abstract. This paper presents a multi-agent system that handles unit micromanagement using online machine learning in real time strategy games. We used rtNEAT algorithm in order to obtain customized neural network topologies, thus avoiding to complex network architecture. We use an ontology based template to create suitable input and outputs for unit agents enabling them to cooperate and form teams for their mutual benefit and eliminating communication overhead. The AI system was implemented using the JADE framework and the BWAPI handled communication between our system and the game. We have chosen Starcraft as a testbed. As a baseline we compared the in game AI as well as several other AI solutions that use adaptive mechanisms.

Keywords: multi-agent systems; machine learning; real time games; neu-ral networks; neuroevolution; ontology.

1. Introduction

Real-time strategy (RTS) games are games where a player has to engage in real time actions. The objective of these games is to achieve either military or territorial superiority over opponent factions. Each player has the capability tobuild structures, collect resources and build or train military units as well as non-combat utilitarian units. By using the utility units to collect resources players can expand their base and create additional units. These units can then be commanded to attack opponent armies or bases.

Video games and in particular RTS games have gained popularity as testing platforms for novel machine learning and AI methods for real world systems [22]. This is largely thanks to the fact that most games are populated by intelli-gent entities that can be considered agents that accomplish a series of finite and clear goals in real time stochastic environments using incomplete information. This makes games an ideal platform for obtaining high quality behaviours from agents as well as groups of coordination behaviours using machine learning techniques [2].

Most games have complex non-linear relationship between environment and the agents that interact with it. This makes neural networks a prime candidate for adaptive game AI systems [24]. However the game industry is reluctant to use machine learning techniques in commercial titles beause of the perceived high cost in both knowledge acquisition and agent coordination. One other major downside is the fact that no guaranties can be made to overall system performance with current learning methods. Some games have been released which do use some form of adaptation mechanism as their core gameplay mechanic. One good example are the games from the Black & White series published by Lion Head studios in 2001 [19]. The goal of the game is to train a creature using reinforcement learning techniques.It is prone to learn less than optimal behaviours.

Modern games use non-adaptive techniques in their AI systems. These techniques have one major disadvantage as once a weakness is discovered it can be exploited repeatedly thus ruining gameplay [29]. A good RTS AI system must be able to adapt during gameplay (online). This adaptation is manly focused on the adaptation to the opponents behaviours in a complex partially observable game environment where there is little time for optimization. To accomplish this goal of real time learning, many techniques have been proposed and one of the most promising is Neuroevolution of Augmenting Topologies (NEAT) proposed by Stanley [26].

NEAT is used to evolve and train NN automatically. Evolution adjusts both the connection weights and the topology of the NN. It can add or remove connections and neurons to the NN topology as well as modify the network connection weights. A real - time version of NEAT called rtNEAT has been

created and used in a proof of concept game called NERO [25]. The goal in NERO is to deploy agents in a training environment where they are trained in some desired tactics and then are pitted against enemy agents to see how well they have been trained.

Game AI development is not an easy task and requires a lot of research and development. Usually the development of the main game engine and that of the AI subsystems is done at the same time. This is usually a good approach however during the latter stages of development any changes to the core gameplay mechanics have a catastrophic effect on AI systems based on non-adaptive techniques. It is also important to note that game AI is not reused. By establishing a MAS that is able to adapt to new situations in real time not only aids gameplay but also reusability. In order to facilitate interoperability and reusability formalizing ontologines are needed. In this paper we focus on a simple unit template ontology that is used to create starting topologies for our NN trained with rtNEAT.

In this paper we developed a multi-agent system (MAS) that is able to adapt and learn on-line using rtNEAT. Each unit will be controlled by a NN which will then adapt themselves automatically to different unit types and tactics. We used the RTS StarCraft developed by Blizzard Entertainment for our experiments. Starcraft is well suited for our task because it is used in many competitions and in some countries such as South Korea it is considered a national sport. Also there are a number of competitions aimed to test AI systems against each other[1]. In StarCraft the single objective given to the player is to destroy all opponents. In order to accomplish this objective numerous tasks must be completed such as gathering resources, creating structures, training units and attacking the enemy. These tasks are the bare minimum of what is necessary to accomplish the main objective. Additional tasks can be performed in order to gain an advantage over the enemy such as scouting and research. In Starcraft there are three distinct races; protoss, Terran, Zerg. In contrast to other RTS games these races have radically different play styles and at the same time they are equally matched.

This paper is an extended version of a paper in which further details are given for the MAS that handles unit micromanagement and is able to adapt/learn during game play. To achieve this, we adapted the rtNEAT algorithm in order to obtain tailor made NN topologies thus eliminating overcomplexification by manual design. Also by defining internal and external inputs for each agent we managed to create independent agents that are able to cooperate and form teams for their mutual benefit and at the same time eliminate unnecessary communication overhead. The MAS was implemented using Java Agent DEvelopment Framework (JADE) and the BWAPI to interface with the game itself. We used an ontology based template to create the initial NN topology. More details are presented in Section 3. We used as a baseline the in game AI and also tested it against other adapting AI systems in order to compare their performance against our system. The experimental setup as well as the results are presented in Section 4, while the conclusions and future work are presented in Sections 5 and 6.

[1] http://eis.ucsc.edu/StarCraftAICompetition

2. Related work

Current video games are ideal platforms to test novel AI techniques. RTS games are particularly interesting because they provide several challenges: adversarial real-time planning, decision making under uncertainty, opponent modeling (learning), spatial temporal reasoning, resource management, collaboration, pathfinding [3]. They also poses an extremely complex environment which according to the classification given by Russell and Norvig [20] is: partially observable, deterministic, sequential, dynamic and continuous. RTS games have a massive state space. When comparing it to chess which has according to [21] a state-space estimated to 10^{43} a RTS game has 10^{11500} according to [1]. Also, the number of actions available to the player is also superior in the case of RTS as it has approximately 1 million possible actions while chess has only 30.

Until recently the main stream game AI system used traditional static techniques such as Finite StateMachines (FSM), Decision Trees etc. Because these techniques lack the ability to adapt they are easily defeated after the player learns each method idiosyncrasies. Researchers are starting to focus more and more on machine learning techniques in order to make game AI more challenging by making it less predictable.

Static techniques have one major drawback: they become predictable after a relatively short amount of time but also they are extremely hard to maintain and debug when they are used to model more complex behaviors. For example, FSMs use transitions to determine when to switch states. These transitions as well as the number of states (s) grow exponentially with the number of events (e): $s = 2^e$, consequently increasing the number of transitions or arcs (a) even faster: $a = s^2$.

Rule-base Systems are brittle and inflexible as when faced with a problem that is out of bounds with their knowledge base, they can not cope as they are unable to rely on past experience to select a similar rule or update their rule base. Similarly to FSMs, expert knowledge is used in their creation which, once in place, can not be modified and requires substantial effort to maintain and debug. Some techniques are ubiquitous such as the A* algorithm commonly used for path-finding and FSM for decision making. These two techniques make up more than 2/3 of current game AI systems [32].

In RTS games, computational intelligence has been applied to tactics using Monte Carlo [4] planning and strategy selection using NEAT [9].

Cognitive architecture is designed for developing mechanisms that highlight human cognition and provides the framework necessary for integrating heterogeneous competences and are also able to reason about multiple goals [11]. It also focuses on performing evaluation at the system level [12] and also uses means-ends analysis when confronted with new problems. However it lacks inherent capabilities for solving some fundamental game AI problems such as low level decision making (micromanagement).

SOAR is one of the best known examples of a cognitive architecture and features multitasking as well as planning capabilities, it performs state abstractions and employs a learning mechanism called chunking. This is, in fact, a caching mechanism and is used to intermix learning and problem solving [12].

Goal-driven autonomy (GDA) model was designed in order to handle unanticipated failures during plan execution in complex, dynamic environments [1]. GDA uses a conceptual model that enables agents to detect, and reason about unanticipated events. One of its more interesting features is the fact that it contains several components and interfaces between these components. It leaves the implementation details unrestricted. This model has been used for RTS games by [15]. Also some ML techniques such as reinforcement learning have been used in [8].

Reactive planning systems have been used to create autonomous software agents for a number of years [14]. Their main design feature is the fact that no apriori planning is done and actions are selected at every instant. This enables these types of systems to handle meta-game concepts such as novel game strategies. Systems that use reactive planning are particularly adept at handling real-time events in dynamic environments such as tasks that require real-time actions. The main strength when it comes to game AI is the ability to enact incomplete plans while pursuing goal-directed tasks [18, 10].

Case-based reasoning is a methodology that enables the creation of systems that learn from prior experience [17]. It has been applied to solve some particular problems in RTS games such as strategic and tactic selection [5] as well as micromanagement [28]. It is important to note that because of the extremely challenging environment and challenge RTS games pose to AI researcher, a number of competitions are held in order to test AI systems against each other. One prime example is the competition held during the AIIDE conference for which the BWAPI was developed.

We should also note that some interesting solutions to RTS learning AI problem have been proposed during the AIIDE conferences StarCraft competition[2]. During this competition AI systems play against each other in a tournament style competition. There are 2 classes of competition. The first is a full AI vs AI competition where the systems play the game from the beginning until one of them loses. The second one focuses on unit micromanagement where the systems only play set piece battles.

The overall winner of the 2010 competition was the Overmind agent created by a team from Berkeley University. It used a shallow planner to enforce resource constraints and tech progression. It also used a potential field whose parameters were tweaked by reinforcement learning[3]. Another noteworthy agent from this competition is the EISBOT developed by Weber, Mateas and Jhala from the University of Santa Cruz. They used Goal-Driven Autonomy conceptual model [30]. However this system doesn't deal explicitly with the problem of micromanagement instead implementing static methods that the Starcraft gaming community identified [31].

Nova[4] was developed by Alberto Uriarte from Technical University Barcelona. It uses a plethora of AI techniques such as: potential fields, FSM, flocking etc. The most important contribution of this system is the micromanagement subsystem which creates micro agents for each game unit which are then coordinated by squad agents. These agent types use FSM for their decision making.

Our system uses the rtNEAT algorithm to evolve the NN that handles each game units micromanagement. In most RTS game such as Starcraft the number of commands issued to units during a battle is directly proportional to the chance of winning [13]. By using an ontology based template for NN starting topology that enable us to evolve tailor made NN for unit micromanagement we show that our MAS is a good starting point to create a fully operational AI system that incorporates online machine learning techniques.

3. MAS architecture

3.1 The RTS micromanagement domain

In RTS games there are three layers of abstraction when it comes to behavior: high, medium and low. There is the strategic layer in which, as its name implies, the overall strategy is being planned. In this layer we decide if we want to focus on defense, offense, economy, research or which enemy to attack first. The second layer is the tactical layer. When the overall strategy has been decided upon, then this layer handles the planning of attacks and defense. This layer handles terrain analysis, army movement etc. The third and last layer is the micromanaging layer.

RTS players issue movement or attack commands in order to increase individual unit effectiveness

[2] AIIDE 2012 StarCraft Competition - http://tinyurl.com/cv2e2af

[3] Overmind winner of AIIDE 2010 StarCraft Competition - http://tinyurl.com/6lhur6b

[4] NOVA STARCRAFT AI - http://nova.wolfwork.com/

during combat. By issuing these commands the player can in effect override the default low-level behavior of a unit. When a military unit receives a command to attack a particular location by default, it will attack the first enemy unit it comes into range. This type of behavior can lead to ineffective unit performance because of target selection lack of inter agent coordination and damage avoidance. A human expert level player will manually select the targets for each unit to engage. This technique enables units to focus fire on specific targets, which reduces the enemy unit effectiveness. We can also prioritize specific types of units or even issue fast movement commands. By using this technique we can force the enemy either to engage new targets or to pursuit. It is easy to see that by micro-managing unit actions we can increase the effectiveness of a squad or individual unit.

Being a highly reactive process, micromanagement requires a lot of actions to be performed in a short amount of time. Expert human player can manage up to 300 actions every minute [31]. A direct correlation has been shown between the number of actions a player is capable of executing and the number of victories[5].

3.2. rtNEAT

NEAT was developed by Stanley and Miikulainen [27]. It evolves both the connection weights and the topology of the NN. This in contrast with other genetic algorithms which are only used to evolve connection weights the topology being designed by human hands. The designing of a NN topology, namely the distribution of hidden neurons into hidden layers, is extremely difficult to do as a NN is considered to be a black box. There are three important contributions of NEAT to other methods based on Topology and Weight Evolving Artificial Neural Networks (TWEANN): crossover between different topologies, protection of structural innovation using speciation and minimizing dimensionality[16].

The real time variant of NEAT (rtNEAT) is largely the same but because it would be too costly to replace the whole population in real time members of the population are constantly evaluated and are replaced in the population by creating a new individual after a certain number of game ticks or if the unit that is represented by a member of the population dies [26].

NEAT and the real time variant rtNEAT has been used in different AI problems with great success and is able to solve highly complex problems [27]. We used this algorithm in our system as decision making mechanism for each game unit. A problem of NN is that as we start to add more neurons to their topology the amount of neural computation also increases. By using rtNEAT to evolve the NN topology we eliminate this problem and are able to use NN topologies tailor made to the given task of unit micromanagement.

[5] StarCraft AI script selection - http://tiny.cc/loeuew

Building an ontology or any knowledge systems entails the creation of an abstraction model of the target domain. Ontologies are defined as an explicit specification of conceptualization [7]. They can be used for knowledge acquisition, knowledge exchange, knowledge-system design, domain-theory development among many others. Description logics (DL) are a set of formal languages that are typically used for formalizing ontologies. One of the main benefits of DLs is that they automatically check the consistency of the ontology and that they classify new concepts and instances. We used the Protégé ontology editor and knowledge acquisition systems in order to create our unit templates [6]. Modern video games have a wide array of game units with different roles in the core gameplay. At first, it may seem that creating templates for each unit type in all game genres is a monumental task but in reality unit types follow well established archetypes. For example, most unit types fall into two categories when it comes to their attack range: melee and ranged. By exploiting this fact we can easily create a game unit taxonomy that can be mapped with relative easy to different game genres. Protégé comes with reasoners that can make inferences that assist in the unit template creation. We used the Pellet reasoner [23].

3.3 MAS agents

We created a multi-agent system (MAS) that uses neu- roevolution to evolve and/or adapt during game play, meaning unit interaction. It is able to use each unit to its full potential thus maximizing battle performance. Each unit uses as its decision making technique a neural network evolved using rtNEAT. In fact, we can consider that the input values of these networks will be used to create a tactical analysis and the output will represent the result of this analysis. Some of rtNEATs characteristics have influenced the design of our system. As this algorithm needs a starting topology to which it complexifies, we generate this starting minimal topology using a unit template based on querying our unit ontology.

Our system has 3 types of agents; Abstractor, Spawn and Unit. The Abstractor agent handles all external environmental percepts and has a role in agent initialization. The Spawn agent handles unit creation and evaluation. The process of selecting the best performing phenotypes from the population and the creation of offspring from them is the main goal of this agent. In order to evaluate the population this agent will receive from the Abstractor evaluation data regarding the performance of each member from the population. Unit agents represent a particular unit from the game environment. They are initialized by the Spawn agent and have to register with the Abstractor agent in order to receive external environment data. We will present each agent type and function in more detail below as well as in Fig. 3.

The tasks a game AI has to accomplish can be split up into three levels of abstraction. These layers have a

hierarchic structure based on the inherent tasks in the video game domain.

The first level deals with high level thinking and planning, in essence it has the highest level of abstraction and deals with strategic decision making. In this level of abstraction most if not all agents can be considered proactive agents being able to run autonomously. It is important to note that the mean planning time for this level is 3 minutes.

The second level of abstraction handles tactical planning and decision making. One example of such planning is group movement and in some cases terrain analysis and even opponent modeling. Tasks and agents inherent to this level can be split into 2 categories: proactive agents which handle longer term planning and reactive agents which receive and efficiently process data and return the results. Because reactive agents run in a stateless manner they are highly scalable.

The third and last level of abstraction is that of microman-agement or reactive control whose main goal is to maximize unit utility. Because of the fine grained nature of the task the mean execution time in this level is under 1 second.

Not all game AIs need all three levels in order to play an effective game, the decision for this being based on the game genre. Racing games need only the micromanagement layer while strategy games need all three. As stated before, we focus on the micromana-gement layer. The abstractor agent has a larger role as it will also be the main source of percepts for the rest of the layer.

Abstractor Agent

Abstractor agent is the central agent in the system. It will handle almost all of environmental percepts. These percepts will be processed using influence maps as well as terrain analysis. This agent will send this processed information to all other agents present in the system which require environmental information. Also it is important to note that not all information handled by this agent is needed for other agents to perform their functionalities. Event handling is needed to filter what information is sent to every agent. In order to facilitate this event handling all agents which want to receive a specific environmental percepts will have to register to the abstractor agent in order to receive the requested percepts information. For instance, if a Unit agent wants to receive information concerning enemy strength in its immediate vicinity, it registers with the abstractor agent which then sends the required information to it.

Agents that form the system also receive initialization information from this agent. When the Spawn agent is first initialized it needs to know how many and what type of agents there are in the environment in order to create the appropriate Unit agent. This agent creates and handles a database of past environmental data. This was used for MAS fine-tuning and will be used in future extension of our system by other agents.

Spawn Agent

The Spawn agent handles unit agent creation, evaluation and evolution. It has a cardinality of 1, and by this we mean that there will be one such agent per system. As its name implies it will create and initialize unit agents. This initialization process consists in creating the initial ANN topology consisting in a fully connected feed forward neural network.

As we can see in Fig. 1 there are 13 input nodes and 3 output nodes. There are 2 types of inputs: internal and external. Internal inputs are $R_1 - R_6$, which represent unit sensors. These are placed on the unit agent as seen in Fig. 2. They are internal because this information is not received from the abstractor agent as this information is extremely critical to unit agent capability to perform its assigned task. This eliminates significant communication overhead between the unit agents and the abstractor agent. With the help of these sensors unit agents will be able to sense enemy units as well as environmental obstacles and hazards. The unit sensors take values in [−1, 0) if an environmental obstacle is detected and (0,1] if an enemy unit is detected. The bigger the value the closer the enemy unit is to that unit sensor.

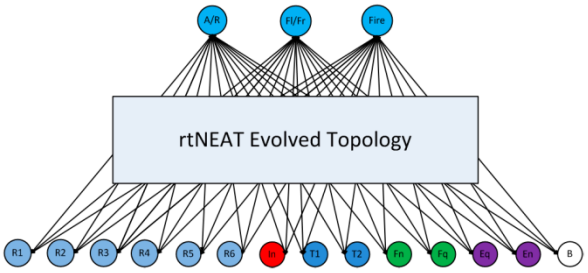

Figure 1. Starting Unit Agent Neural Network Topology

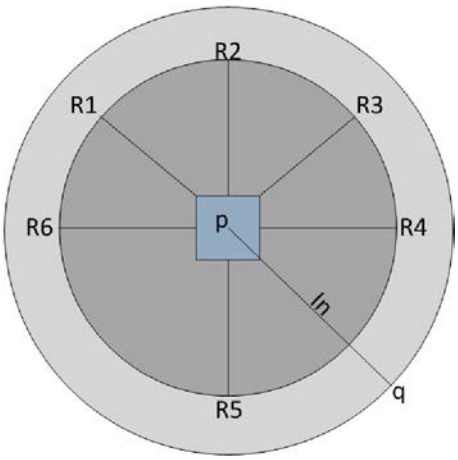

Figure 2. Unit Agent Sensory Field

Also there are two more internal inputs I_n and T_1. I_n signals the range to the closest enemy unit. It can have values in the range [−1,1]. If an enemy is within

firing range, the biggest value is assigned. T_1 represents unit type. There are only two unit types which we considered during micromanagement.

There are: melee and ranged units. Melee units will have the value of −1 and ranged units will have +1. Melee units are close combat units, their firing range being zero as they have to get up close to enemy units in order to attack them. Ranged units, on the other hand, don't have this limitation thus having a tactical advantage that in an ideal scenario can be exploited by our system.

External inputs consist of the following values T_2, F_n, F_q, E_n, E_q. These inputs are received from the abstractor agent and are sent to every registered unit agent. T_2 represents unit type, the only difference being that they represent the enemy units type that is the closest to the current unit agent. F_n represents the current number of friendly units present in the immediate vicinity while E_n represents the number of enemy units.

F_q represents the quality of the enemy units. This refers to the relative strength of these units. For example, in chess some pieces have a higher value, the queen has a higher value then the knight. This is the same in the case of most modern games. Although we should note that this value is relative. By this we mean the intransitive superiority problem which states no clear superiority based only on unit type and ca-pabilities. There is always a unit type or tactic that nullifies the advantages or even the capabilities of a particular unit. E_q input is the quality of enemy units and is similar to the F_q and how it is assigned. For the purpose of this micro-managing MAS layer, we created only a limited ontology in order to identify melee and ranged unit weaknesses.

It also receives abstracted environmental data from the abstractor agent an it will use this information in the evaluation process . This process entails gathering information about the performance of the unit agents. This information will be then incorporated in the evaluation function.

As we mentioned above, we have 3 output neurons which, in fact, will represent the decisions the neural network has made based on its inputs. The first output is A/R which represents whether the agent should advance or retreat. Fl/Fr represents if the agent should move right or left. And the last output F_i represents if the agent should engage the enemy. If the values are in the range of [0,1], the unit will attack and in the [−1,0) range the unit will retreat. It is the same in the case of the left and right action outputs.

Because rtNEAT continuously evaluates agent performance the Spawn agent will receive evaluation information for each agent present in the environment until the game timer runs out or the unit agent dies in battle. The game timer is a governing principle of rtNEAT and is calculated based on equation (2).

The evaluation data the Spawn agent needs are as follows: $N_k, D_D, D_T, SURVIVE.$ N_k represents the number of enemy units killed (from 0 to 24 units), D_D represents the total damage done to the enemy while D_T represents the damage taken from enemy units. These can be quantified quite easily as they are intrinsically represented by integer values. The main problem is that *SURVIVE* is a boolean value as it can be only true or false. We decided to give it a numerical value of 40 for evaluation purposes. The variable Ag is the aggressivity coefficient. In testing, it is set to 18 thus rewarding an aggressive style of play. Equation (1) describes the function used to compute the fitness value:

$$f_e = N_k + Ag(\frac{D_D}{D_T}) + SURVIVE. \quad (1)$$

After the evaluation is done and the best and worst performing unit agents from the population are identified the evolution process starts. We select the best performing unit agents to replace either the worst performing agent from the population or to replace a killed unit. We use elitism to protect the best performing agents to a certain extent as to ensure a stable performance level during game play. This is done by rtNEAT but we also created a Phenotype database which saves unit agents that perform beyond a certain threshold. This means that although the phenotype might be killed of by rtNEAT we still have a copy of it that can be used to bootstrap the population if evolution doesn't deliver any meaningful progress or even suffers from overfit- ting or gets stuck in a local maxima.

As we can see in [16], rtNEAT has a number of parameters that can be adjusted to better perform in its given task. It is important to note that NNs perform extremely well if they have a clearly defined task to perform. Also if a problem can be easily solved using a discrete series of steps and the problem to be solved is static, a NN approach to solving these types of problems is ill advised.

We can see a list of all the relevant parameters in Table 1. We should note that not all parameters are listed, and that the rest of the parameters were left at their default values during the experimental run. We used two sets of values because of the need for fast evolution during the initial phases of the experiment which will be further detailed in section 4. There are a total of 9 relevant parameters for our system. The compatibility threshold (δ_t) is the distance measure between species and is used in speciation and can be used to ensure that the desired number of species is maintained by raising or lowering its value thus making the inclusion into a species less or more likely. The *inter species mating* is the likelyhood that members from different species will recombine. The recurrent connection parameter sets the likelihood of recurrent connections being added during mutation. Parameter P represents the total population, m represents the minimum time alive for any member of the population measured in game ticks. In the case of StarCraft $1tick \simeq 40ms$. While add node and add link

represent the likelihood that a new node or link will be added to the topology.

Table 1. rtNEAT Parameters

Parameter	Value 1	Value 2
δ_t	[3.7-4.3]	[3.7-4.3]
Inter Species Mating	0.004	0.001
Recurrent Connection	0.009	0.007
Number of Species	3	3
P	12	12
I	0.5	0.5
m	400	500
add node	0.05	0.04
add link	0.03	0.02

As was shown in [26, 25], if members of the population are removed too frequently from the population, then they will not evolve to their full potential and conversely if they are removed too frequently, then evolution slows down thus the law of eligibility was formulated:

$$n = \frac{m}{|P|n}. \quad (2)$$

From this law of eligibility, we can formulate how many game ticks (n) should elapse between replacements. In the case of our system $n_1 = 66.7$ and $n_2 = 83.3$. The n_1 value is used during the initial phase of the experiment while n_2 for the later part. Also we divide by 100 the evaluation function f_e in order to ensure that it takes values in the range $[-1,1]$.

Unit Agents

Lastly unit agents represent each unit from the game environment. Each of these is created by the Spawn agent and must register with the abstractor agent. As we mentioned before, each unit agent will represent a particular unit from the game environment. Each unit is also initialized by the Spawn agent. Because in a dynamic system such as modern games there are potentially hundreds even thousands of units in the environment at any given time it initially seems a poor design decision to create units for each of them. However there are two good reasons that we chose to create so many unit agents.

First we only have a maximum of 200 units in Starcraft RTS. Most other genres of games use even less units. First Person Shooters (FPS) use only about 20-30 units at any given time. So even a moderate computing system could handle this amount of agents. Also most units like resource gathering units don't require specialized unit agents as they perform their task automatically and most battles don't include the entire army at the systems disposal so just a fraction of unit agents will be actively evaluated and controlled.

On the other hand, because of the parallel nature of MAS we can deploy some of the more resource consuming agents on different computing nodes or even on a cloud computing platform. Thus we could have an extremely powerful computing backbone to accomplish all the analysis and decision making in our MAS.

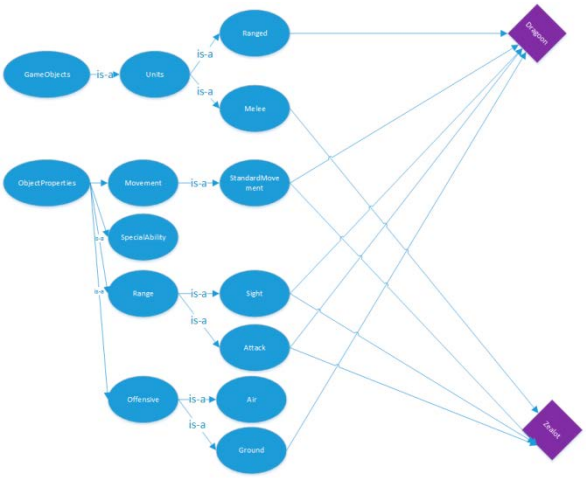

Figure 3. Ontology Outline

Unit Template Usage

Our ontology consists of two main classes: *GameObjects* and *ObjectProperties*. As we are currently only interested in game units the *GameObject* class consists only of melee and ranged unit subclasses. The *ObjectProperties* class is the superclass for all the properties each unit can have. As an example we take two units, a zealot which is a melee unit and a dragoon which is a ranged unit. Fig. 3 represents the basic template ontology. We can see that we have a number of object properties which we use to create our starting topology. In Table 2 we see what each object property defines. By querying the ontology we get that the zealot unit is a melee unit with ground attack and no special ability.

Table 2. Object Properties

Property	Function
hasMovement	Defines the standard movement range of untis
hasRange	Defines range of weapons as well as sight
hasAbility	Defines if the unit has special abilities
hasOffensive	Defines what target the unit can attack
hasDefensive	Defines the overall health of the unit

4. Experiments

Implementation

Because most of the current commercial games are closed source users cannot directly create AI systems for these games, external third party tools are used instead to inject the custom AI into them. In the case of StarCraft there is the BWAPI that enables users to inject their custom AI into the game. It was developed in C++ but extensions that allow users to use it for

other languages have been implemented. We used the ProxyBot extension written for Java. BWAPI splits the game into 5 basic objects: game, player, unit, bullet, force.

The game object holds all information about the current game state. The player object holds information that the player would have during game play. Unit object represent the pieces in the game like units and resources. Bullet objects represents projectiles from ranged units. The force object represents a collection of players attempting a singular objective.

The MAS itself is created using JADE where the abstractor agent is mapped to a GameObject which helps the system to keep track of what it is doing as well as GameUpda- teObject which updates the environment state. Because of the three databases present in our system we use the JADE DataStore ability. In later versions of our system we will create a different database implementation but as our system does not produce a large amount of data this setup has proved adequate.

Also all behaviors from the systems agent, as seen in Fig. 4, are threaded behaviors because the three communication protocols (sdRegData, sdUinitEval, sdUinitEvol) are not synchronized and can operate independently.

As an example protocol we have Fig. 5. representing sdEVALDATA. This protocol handles the external data necessary for Unit agent evaluation. The INFORM EVAL-DATA message loops until the game instance ends or a fatal failure has occurred. CONFIRM EVALDATA is sent only once the first iteration of INFORM EVALDATA is sent. This is done in order to ensure that the message has been received. The messages INFORM CORRUPT and INCOMPLETE are sent when data are incomplete or are in an inconsistent state. Resend REVAL DATA resends the last message before one of the two exception messages has been received.

It should be also noted that the population used by rt-NEAT is stored in the Phenotype Data Base (DB) in XML format. Once a new NN topology has been evolved the XML will be loaded using the TopLoad Java object into the appropriate Unit agent.

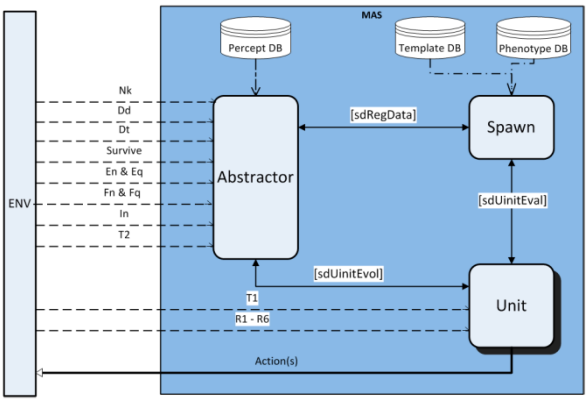

Figure 4. Systems Architecture

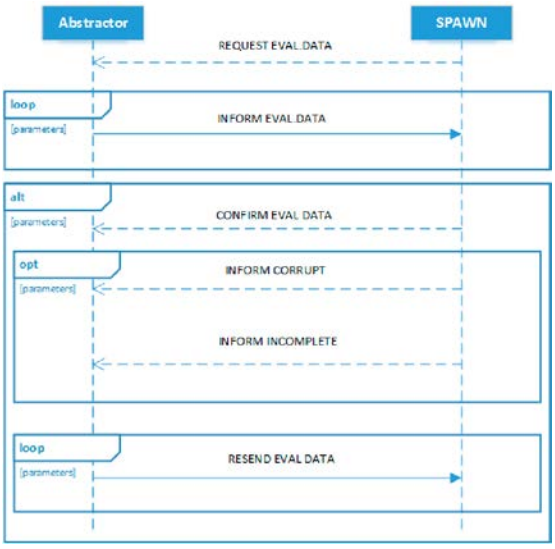

Figure 5. Diagram of sdEVALDATA Protocol

Experimental setup

We used the tier 1 tournament arena maps from the AIIDE StarCraft competition. These maps have a simple square shape to best highlight micro-management abilities. There are 2 phases in our experiment to best test our systems learning capabilities. We chose as our two unit types from the Protoss race: Zealot for the melee unit and Dragoon for the ranged unit.

During the first phase our systems performance was evaluated against the built in AI system of the game. In Star- Craft the AI is represented by a script loaded at the beginning of the game. For each of the 3 races there exist 2 scripts[6]. These scripts are extremely predictable once the player played against them for a limited amount of time. We ran 300 matches of 12 versus 12 ranged units where one group was controlled by our MAS while the other group by the standard (vanilla) AI. The match starts with all 12 units in the environment. In order to win a match our system has to deplete the enemy AIs reinforcement. Each player has 100 units in reserve, once a unit dies a new one is created from the reserves. Other 300 games were run where instead of ranged units melee units were used. This setup was chosen because of the inherent difference in play style of the two unit types will help in the testing of the adaptive capability of our system. The fitness is calculated based on equation (1).

During the first phase the ranged unit versus melee unit experiments where also done. During the first 300 games our MAS played as the ranged units and for the second 300 games it played as the melee units. Because the melee units have a tactical disadvantage against the ranged units with this setup we wish to see if our system can utilize this advantage or in the case of the second type of matches to nullify it by efficient unit micromanagement.

[6] StarCraft AI script selection - http://tiny.cc/loeuew

The second and last phase consists of our MAS against preexisting AI systems from the AIIDE 2010 competition. We used the EISBOT and Overmind and played 300 matches against each one of them. We also used during this phase of the task the best performing members of the population from phase 1 to bootstrap the evolutionary process of phase 2. Also the rtNEAT was set to the second values detailed in Table 1 in order to ensure a more stable rate of evolution.

There are two important facts that need to be mentioned. First because of a memory leak problem it is impossible to run more than 500 games consecutively. The program becomes extremely unstable if run for an extended period without restarting it. In our system we set the testing runs to no more than 300 consecutive games.

The second consideration is a limitation of StarCraft as we need to separate instances of the game to run in order to test our system against other AI systems such as EISBOT. The two game instances communicate with each other through LAN connections. There are ways to run 2 instances of the game on the same computer but we used separate computers for each game instances.

Results

Phase 1

As we mentioned before the testing platform for our MAS is StarCraft. It should be noted that because of the nature of the game, time is measured in game ticks as opposed to seconds. So let us consider that 1 game tick = 40 *ms*. If we consider that a new unit will be created every 500 ticks and we have a maximum number of 100 units that can be created. We have 50000 game ticks for each match which means 33 per game. This is, of course, the worst case scenario as running 300 games that last each of them 33 minutes would be extremely impractical. Thus we sped up the game with a factor of 16. It follows that on average 300 games took 4 hours.

As we can see from Table 3, our system can consistently beat the in game AI during all of the matches. It performs best when it can exploit the inherent tactical superiority of the units that are under its control. For example, it won 89% of the matches played with it controlling ranged units against melee units by the technique known in gaming as kitting. In principle this means that the range units don't sit still while fighting but keep moving thus being able to minimize the damage done by enemy melee units that need to get in close in order to attack. Our system won 65% when controlling melee units against ranged units but considering the built in AI won only 11% of games when controlling the melee our system performed extremely well.

Fig. 6 represents the evolution of the average fitness during the 300 matches. We can see that in the beginning the fitness values fluctuate substantially. This was expected because of the relatively high rtNeat parameter values set as seen in Table 1. After 30 games the fitness values start to stabilize and at game 182 the fitness values hit a local maxima. By this we mean that the values stagnate in both cases presented in Fig. 6. It should be also noted that the highest fitness value achieved was 0.8 out of a maximum value of 1. It is extremely unlikely that even if we run thousands of games that any agent would reach that high mark nor is it necessary. At a fitness value of 0.5 units are sufficiently evolved to pose a significant challenge to players. They are capable of exploiting tactical weaknesses of the enemy in a consistent manner.

Our system is able to rapidly learn and converge to a higher fitness value during a single match. In Fig. 7 we see the evolution during a period of 3000 game ticks from a fitness value of 0.4 to 0.6. This means that in a 2 minute battle our system is able to adapt at a rate of 0.2 fitness points in the ideal case. On average the adaptation rate was 0.02 every 3000 game ticks. During empirical testing it was shown that the learning rate is fast enough to make this a viable real time learning technique.

Table 3. Results for all phase 1 experimental runs - R represents Ranged while M represents Melee

Type Of Game	Win	Average Fitness	Best Fitness
R vs R	77%	0.69	0.76
M vs M	74%	0.65	0.7
M vs R	65%	0.53	0.71
R vs M	89%	0.72	0.8

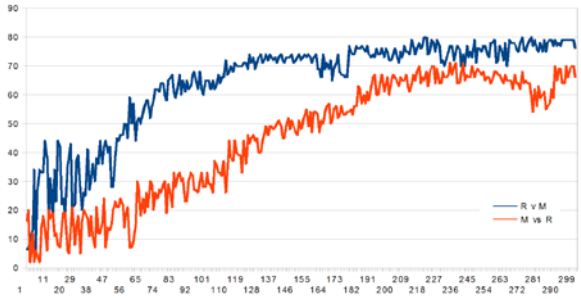

Figure 6. Best and worst experimental run - x axis represents the number of games while the y axis represents the fitness value

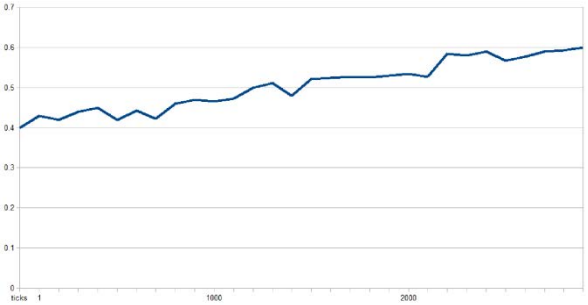

Figure 7. Evolution of fitness during a single game

Phase 2

During the last phase our system played against the standard game AI which utilized standard FSM. In this phase we evaluated our MAS against two more sophisticated AI systems EISBOT and Overmind. As illustrated in the results of the AIIDE 2011 competition results[7], these systems have a significantly higher performance. Because of this, we changed the parameter setting of rtNEAT to the second values listed in Table 1 to ensure a more stable performance from the beginning of the experiment. Also we used the best evolved individuals from phase 1 to bootstrap the evolution process in this phase. This technique has proven to be extremely effective as the limited runs done with a minimal starting topology did not converge to a sufficiently good fitness value in a reasonable time.

We ran the same types of experiments as in the previouse phase for EISBOT and Overmind as can be seen in Tables 4 and 5. Our result mimicked the ones from phase 1 in that our system performed at its best when it controlled ranged units and the worst performance when it commanded melee units against ranged units. However this is to be expected as in the latter scenario there is a significant tactical advantage on the side of the ranged units. There are two immediate observations that we can make. First, our system statistically outperformed EISBOT in unit micromanagement. The best win ratio of 69% was in the scenario in which our system had the tactical advantage. As before, it was able to recognize and utilize this. The second observation is that although we bootstrapped with the phase 1 best performing population, the average fitness has a smaller value. This is because the fitness function is relative to the performance of the enemy against which our system plays.

Table 4. Results for games played against EISBOT

Type of Game	Win	Average Fitness	Best Fitness
R vs R	57%	0.64	0.76
M vs M	52%	0.63	0.73
M vs R	51%	0.60	0.69
R vs M	69%	0.68	0.79

Table 5. Results for games played against OVER-MIND

Type of Game	Win	Average Fitness	Best Fitness
R vs R	51%	0.53	0.64
M vs M	50%	0.52	0.7
M vs R	43%	0.5	0.6
R vs M	61%	0.53	0.67

We also executed experimental runs against the Overmind AI system. This system was the overall winner of the AIIDE 2010 StarCraft competition[8]. Our system was able to statistically beat Overmind in unit micromanaging except for the case when Overmind had the advantage where our system was unable to beat it consistently wining only 43% of the time. It should be noted that if more experimental runs had been made, than we are convinced that our system could consistently defeat Overmind even here. The only major drawback would be the amount of time and computational resources necessary to accomplish this task.

5. Conclusions

We designed a MAS able to handle unit micromanage- ment in RTS games. Also it is able to learn and adapt to new situations and tactics by using rtNEAT. It introduces a MAS designed specifically around neuroevolution creating agents that handle different steps of rtNEAT. We were able to prove that our MAS outperforms the standard AI found in StarCraft in all the experimental scenarios.

It also enjoyed success in phase 2 when it was evaluated against more sophisticated AI implementations being able to statistically beat them with the exception of one scenario where it only won 43% of the time. This however is still much better than the wast majority of AI implementations for StarCraft[9].

We also showed that rtNEAT has a learning rate which makes it suitable for fast adaptation in real time by having an average fitness gain of 0.02 per 3000 game ticks. This rate can be adjusted either by optimizing rtNEAT parameters or the addition of more members to the population being evaluated/evolved. During testing we chose the population size of 12 because of the default squad size selectable in StarCraft. In a typical engagement there are 2 to 3 groups of 12 units each, so the population can be easily expanded thus ensuring more interaction between population members and the enemy that can be used for evaluation.

Our system is capable of identifying and utilizing tactical advantages in battle as well as raising the utility of units that have a tactical inferiority. By the process of bootstrapping we can guarantee a minimum level of performance at the beginning of a new evolution cycle.

One shortcoming of our system that has been observed is that there is a communication bottle neck when more than 150 unit agents are created and registered with the abstractor. This bottleneck is more of a technological limitation and can be easily avoided. We can create Unit agent only when agents

[7] AIIDE 2011 Competition Results - http://tiny.cc/ipeuew

[8] Overmind winner of AIIDE 2010 StarCraft Competition - http://tinyurl.com/6lhur6b

[9] AIIDE 2011 Competition Results - http://tiny.cc/ipeuew

are in combat because there are a limited number of units participating in combat at any one time. For the rest of the time a more traditional AI can handle unit movement and deployment.

The ontology based template creation which was used to create starting NN topologies has proven to be effective. This technique could be augmented by adding support to automated ontology learning which could enable the mapping relations between unit types. This in turn can be used to solve intransitive superiority between said unit types.

Our MAS is a good platform from which we can build upon and expand our system in order to build a complete RTS game.

6. Future work

In this paper we adjusted the parameters of rtNEAT by empirical testing although automatic parameter adjusting techniques can be applied to dynamically alter the algorithms behavior in certain situations. Also we used an extremely limited ontology to create starting NN input and outputs thus enabling new Unit agent types to be created as needed.

Our MAS is focused on only one facet of RTS games namely micromanagement. By applying a bottom up design method we can build additional layers to handle other tasks in the game. For example, a tactical layer can handle terrain analysis and target prioritization, while a strategic layer can handle high level planning such as economic planning.

Our starting topology based on a unit template ontology has proven effective in our experiments. One of the possible expansions of this idea is to create a generalized ontology so that many more information as well as inter object relationships (like the intransitive superiority problem) could be expressed using an ontology an then queried during game-play.

Acknowledgments

This work was partially supported by the strategic grant POSDRU/CPP107/ DMI1.5/S/78421, Project ID 78421 (2010), co-financed by the European Social Fund - Investing in People, within the Sectoral Operational Programme Human Resources Development 2007 - 2013.

References

[1] **D. W. Aha, M. Molineaux, M. Ponsen**. Learning to win: case-based plan selection in a real-time strategy game. In: *Proceedings of the 6th international conference on Case-Based Reasoning Research and Development, ICCBR'05*, Berlin, Heidelberg, 2005, Springer-Verlag, pp. 5-20.

[2] **S. C. Bakkes, P. H. Spronck, H. J. van den Herik.** Opponent modelling for case-based adaptive game {AI}. *Entertainment Computing*, 2009, Vol. 1, No. 1, 27-37.

[3] **M. Buro**. Real-time strategy gaines: a new ai research challenge. In: *Proceedings of the 18th International Joint Conference on Artificial intelligence, IJCAI'03*, San Francisco, CA, USA, 2003, Morgan Kaufmann Publishers Inc., pp. 1534-1535.

[4] **M. Chung, M. Buro, J. Schaeffer.** Monte Carlo planning in RTS games. CIG 2005.

[5] **M. Fagan, P. Cunningham.** Case-based plan recognition in computer games. In: *Proceedings of the Fifth ICCBR*, 2003, Springer, pp. 161-170.

[6] **J. H. Gennari, M. A. Musen, R. W. Fergerson, W. E. Grosso, M. Crubézy, H. Eriksson, N. F. Noy, S. W. Tu.** The evolution of protégé: an environment for knowledge-based systems development. *International Journal of Human-Computer Studies*, 2003, Vol. 58, No. 1, 89-123.

[7] **T. R. Gruber**. Ontolingua: A mechanism to support portable ontologies. Tech. rep. 1992.

[8] **U. Jaidee, H. Muñoz Avila, D. W. Aha.** Integrated learning for goal-driven autonomy. In: *Proceedings of the Twenty-Second international joint conference on Artifcial Intelligence IJCAI'11*, AAAI Press, 2011, Vol. 3, pp. 2450-2455.

[9] **S. H. Jang, J. W. Yoon, S. B. Cho.** Optimal strategy selection of non-player character on real time strategy game using a speciated evolutionary algorithm. In: *Proceedings of the 5th International Conference on Computational Intelligence and Games CIG'09*, Piscataway, NJ, USA, 2009, IEEE Press, pp. 75-79.

[10] **D. P. Josyula.** A unified theory of acting and agency for a universal interfacing agent. PhD thesis. *College Park, MD, USA*. 2005. AAI3202442.

[11] **P. Langley, D. A. Choi.** A unified cognitive architecture for physical agents. In: *Proceedings of the 21st National Conference on Artificial intelligence AAAI'06*, AAAI Press, 2006, Vol. 2, pp. 1469-1474.

[12] **J. F. Lehman, J. Laird, P. A. Rosenbloom.** A gentle introduction to soar, an architecture for human cognition. In: *S. Sternberg & D. Scarborough (Eds), Invitation to Cognitive Science*, MIT Press, 1996.

[13] **J. M. Lewis, P. Trinh, D. Kirsh.** A corpus analysis of strategy video game play in starcraft: Brood war. In: *Proceedings of the 33rd Annual Conference of the Cognitive Science Society*, 2011.

[14] **A. B. Loyall.** Believable agents: building interactive personalities. PhD thesis. *Pittsburgh, PA, USA*. 1997. AAI9813841.

[15] **H. Muñoz Avila, U. Jaidee, D. W. Aha, E. Carter.** Goal-driven autonomy with case-based reasoning. In: *Proceedings of the 18th International Conference on Case-Based Reasoning Research and Development ICCBR'10*, Berlin, Heidelberg, Springer-Verlag, 2010, pp. 228-241.

[16] **J. Olesen, G. Yannakakis, J. Hallam.** Real-time challenge balance in an RTS game using RTneat. In: *IEEE Symposium on Computational Intelligence and Games CIG '08*, 2008, pp. 87-94.

[17] **S. Ontañón, K. Mishra, N. Sugandh, A. Ram.** Learning from demonstration and case-based planning for real-time strategy games. In: *B. Prasad (ed.), Soft Computing Applications in Industry, Vol. 226 of Studies in Fuzziness and Soft Computing*, Springer Berlin Heidelberg, 2008, 293-310.

[18] **L. Pryor, G. Collins.** Planning for contingencies: a decision-based approach. AI Access Foundation, Vol. 4, pp. 287-339.

[19] **S. Rabin.** AI Game Programming Wisdom. *Charles River Media, Inc., USA*. 2002.

[20] **S. J. Russell, P. Norvig.** Artificial Intelligence: A Modern Approach, 2 ed. *Pearson Education*. 2003.

[21] **C. E. Shannon.** Programming a computer for playing chess. In: *Computer chess compendium*. Springer-Verlag New York, Inc., USA, 1988, 2-13.

[22] **A. Shantia, E. Begue, M. Wiering.** Connectionist reinforcement learning for intelligent unit micro management in starcraft. In: *The 2011 International Joint Conference on Neural Networks (IJCNN)*, 2011, pp. 1794-1801.

[23] **E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, Y. Katz.** Pellet: A practical OWL-DL reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web 5*, 2007, Vol. 2, 51-53.

[24] **P. Spronck, S. I. Kuyper, E. Postma.** Difficulty scaling of game AI. In: *Proceedings of the 5th International* Conference on Intelligent Games and Simulation (GAME-ON 2004), 2004, pp. 33-37.

[25] **K. O. Stanley, B. D. Bryant, R. Miikkulainen**. Real-time neuroevolution in the nero video game. *IEEE Trans. Evol. Comp*, 2005, Vol. 9, No. 6, 653-668.

[26] **K. O. Stanley, R. Miikkulainen.** Efficient reinforcement learning through evolving neural network topologies. In: *Proceedings of the Genetic and Evolutionary Computation Conference GECCO '02*, San Francisco, USA, 2002, Morgan Kaufmann Publishers Inc., pp. 569-577.

[27] **K. O. Stanley, R. Miikkulainen**. Competitive coevolution through evolutionary complexification. *J. Artif. Int. Res.*, 2004, Vol. 21, No. 1, 63-100.

[28] **T. Szczepanski, A. Aamodt**. Case-based reasoning for improved micromanagement in real-time strategy games. In: *Proceedings of the Workshop on Case-Based Reasoning for Computer Games, 8th International Conference on Case-Based Reasoning ICCBR*, 2009, 139-148.

[29] **M. van der Heijden, S. Bakkes, P. Spronck.** Dynamic formations in real-time strategy games. *CIG*, 2008, 47-54.

[30] **B. G. Weber, M. Mateas, A. Jhala.** Applying goal-driven autonomy to starcraft. *AIIDE*. 2010.

[31] **B. G. Weber, M. Mateas, A. Jhala.** Building human-level ai for real-time strategy games. In: *Proceedings of the AAAI Fall Symposium on Advances in Cognitive Systems*, San Francisco, 2011, AAAI Press.

[32] **S. Yildirim, S. B. Stene.** A survey on the need and use of AI in game agents. In: *Proceedings of the 2008 Spring simulation multiconference SpringSim '08*, San Diego, USA, 2008, Society for Computer Simulation International, pp. 124-131.

Received June 2013.