Testing the limits of emergent behavior in MAS using learning of cooperative behavior

Jordan Kidney and **Jörg Denzinger**[1]

Abstract. We present a method to test a group of agents for (unwanted) emergent behavior by using techniques from learning of cooperative behavior. The general idea is to mimick users or other systems interacting with the tested agents by a group of tester agents and to evolve the actions of these tester agents. The goal of this evolutionary learning is to get the tested agents to exhibit the (unwanted) emergent behavior. We used our method to test the emergent properties of a team of agents written by students for an assignment in a basic MAS class. Our method produced tester agents that helped the tested agents to perform at their best and another configuration of our system showed how much the tested agents could hold their own against very competitive agents, which revealed breakdowns in the tested agents' cooperation.

1 Introduction

Emergent behavior has become a hot research topic in both the multi-agent systems (MAS) and the artificial life communities. Having a group of agents that –by working together– achieve goals that are beyond what can be expected from them by just adding up their individual abilities is naturally a very desirable goal (especially given that the computer "world" is getting more and more distributed), so that even researchers from areas like software engineering and computer security have become interested in this phenomenon. While there are quite a few examples for the synergetic effects that emergent behavior can achieve (see, for example, [2]), there are still many aspects of emergent behavior that are not well understood. Current research mostly focuses on finding concepts for agent cooperation or interaction that result in some intended emergent behavior.

An aspect of emergent behavior that has not been addressed very much, so far, is the fact that the interactions of a set of agents might not always produce only wanted emergent behavior (which is naturally known, but not much written of) and how to identify situations for a set of agents when unwanted behavior surfaces. Traditionally, finding unwanted behavior of a system is the task of testing the system. But software testing in general struggles to keep up with the growing complexity of todays software systems and there have been very few approaches to deal with testing of multi-agent systems beyond the standard techniques developed decades ago. Following test case scripts, testing of individual system components and visualization of agent interactions help with testing of multi-agent systems, but finding unwanted emergent behavior of these systems requires much more than these techniques. The growing number of cases of users abusing systems by providing unexpected inputs or exploiting bugs in the systems is a clear proof of that (see [13]).

[1] Department of Computer Science, University of Calgary, Canada email: {kidney,denzinge}@cpsc.ucalgary.ca

In this work, we propose to tackle this problem by making use of techniques that helped creating the problem in the first place: *learning of cooperative behavior* of agents. The general idea is to employ a group of cooperating agents that interact with the multi-agent system that we want to test in order to get this multi-agent system to produce an unwanted (emergent) behavior. Learning methods are used by the group of tester agents to adapt their behavior based upon observations made about the tested system. This process is done to get the tested system closer and closer to the unwanted behavior we are looking for. More precisely, we propose to use ideas from evolutionary learning of behavior (see [3]) to develop a set of tester agent strategies that forces the tested system to show the unwanted behavior we are looking for.

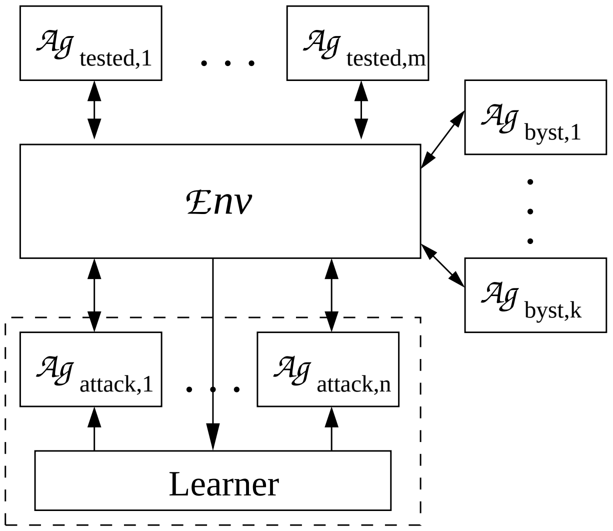
We used our method to test the emergent behavior of a set of agents created by students for their assignment in our multi-agent systems course. The tested agents act within the ARES 2 simulator (see [4]) that is a disaster simulation in which the tested agents have to find and rescue survivors buried under rubble. With our testing method we are able to evaluate the limits of the emergent behavior of the student teams. This testing on one hand works at creating a team of tester agents that tries to help the tested agents as much as possible, boosting the number of survivors rescued by the tested agents. On the other side it works to also create tester agents that keep the score of the tested agents as low as possible, revealing weaknesses in the cooperation strategies employed by the students. Our experimental evaluation shows that our testing method allows us to reveal quite well the (positive and negative) limits of the tested student team to help us in our evaluation (and grading) of the tested team. Additionally, our agents revealed confusion in the behavior of the tested agents in some circumstances, clearly unwanted emergent behavior.

2 Basic definitions

A general definition that can be instantiated to most of the views of agents in literature sees an agent $\mathcal{A}g$ as a quadruple $\mathcal{A}g$ = (Sit,Act,Dat,f_{Ag}). Sit is a set of situations the agent can be in, Act is the set of actions that $\mathcal{A}g$ can perform and Dat is the set of possible values that $\mathcal{A}g$'s internal data areas can have. In order to determine its next action, $\mathcal{A}g$ uses $f_{Ag} : Sit \times Dat \rightarrow Act$ applied to the current situation and the current values of its internal data areas.

In order to interact, a group of agents has to share an environment or at least parts of it. Therefore, a multi-agent system $\mathcal{MAS}$ on a very high level is a pair $\mathcal{MAS}$ = (A, $\mathcal{E}nv$), with A a set of agents and $\mathcal{E}nv$ the environment. $\mathcal{E}nv$ consists of a set of environment states. Based on this view of a multi-agent system, the first 3 components of an agent can be more structured by distinguishing between parts dealing with other agents and parts dealing with the

Figure 1. General setting of our approach



agent in the environment, but we do not need this distinction in this paper. Note that communication between agents is modeled by having on the one side actions and on the other side the communication "channels" provided by the environment. Note also that the set Sit of an agent can be just a view on the environment, i.e. an element of Sit can contain less information than an element of $\mathcal{E}nv$.

3 Testing by evolutionary learning

The method we propose for testing for unwanted emergent behavior of a group of agents A_{tested} uses a set of (learning) tester (or attack[2]) agents A_{attack} that interact with the agents in A_{tested} by playing the role of entities that the agents in A_{tested} usually interact with in their environment. The agents in A_{attack} can play the role of "team members" of the agents in A_{tested} or the role of users or other systems that use/interact with the agents in A_{tested}. Depending on the circumstances, there can be a third set A_{byst} of agents, the bystander agents, which are agents that are not part of the set of tested agents, but interact with them in the environment, and are also not under control of the tester. Figure 1 presents a view of the general setting just described, i.e. all 3 types of agents act in the environment $\mathcal{E}nv$ and by doing that they interact with each other.

Figure 1 also indicates that the agents in A_{attack} are learning, but how learning is organized is indicated rather vague, we only see that the learning gets feedback by observing the environment. There are many different ways how learning of a behavior for a group of agents can be accomplished (see [8] for an overview). We can have one learner that learns off-line and creates a team of attack agents that are not doing any learning while they "work". The other extreme is that each agent $\mathcal{A}g_{attack,i}$ in A_{attack} has as part of it a learning component and the agent is learning on-line. And, as pointed out in [5], there are many other possibilities in between those extremes. We think that different combinations of A_{tested} and $\mathcal{E}nv$ will require different learning methods for A_{attack}.

In this paper, we propose a central off-line learning approach, i.e. the first extreme mentioned above. Naturally, learning is strongly connected with the agent architecture (Sit,Act,Dat,f_{Ag}) of an agent and what architecture to choose depends a lot on the task agents have to perform. From the point of view of testing the limits of emergent behavior of agents in A_{tested}, the task of the agents in A_{attack} is to provoke the agents in A_{tested} into showing a particular behavior (that the human tester would like to know about).

[2] We use the term attack in the following, since "tester" and "tested" are very difficult to distinguish

More precisely, from the point of view of an outside observer, each agent $\mathcal{A}g_{attack,i} \in A_{attack}$ has to perform a sequence of actions $a_{i,1}$,...,a_{i,l_i} ($a_{i,j} \in Act_{attack,i}$) that results in a sequence e_0,e_1,...,e_x of (enhanced) environment states, such that some condition $\mathcal{G}_{emerge}$((e_0,e_1,...,e_x)) is true. By an (enhanced) environment state, we mean a view on $\mathcal{E}nv \times \bigcup_{\mathcal{A}g \in A_{tested}} Dat_{\mathcal{A}g}$, i.e. we allow on the one side that internal data of the tested agents can be made available to our learning agents and on the other side we might not even make everything that is observable in the environment available to our attack agents (and the learner for them). If all agents interact with the environment in a synchronous fashion, then the length of the action sequences of the attack agents is the same and this length is also the number of environment states that we observe after starting from e_0, i.e. $l_i = l_j$ and $l_i = x$ for all i, j.

In the case of the agents interacting asynchronously, we need to record the (enhanced) environmental state every time a change takes place. Then x obviously cannot easily be connected to the l_is. In the asynchronous case, it often can be necessary to add timing information to the action sequence of an $\mathcal{A}g_{attack,i}$, resulting in a sequence ($t_{i,1}$,$a_{i,1}$),...,(t_{i,l_i},a_{i,l_i}) of timed actions, with the $t_{i,j}$ indicating time units and $a_{i,j} \in Act_{attack,i}$ as before. Then $\mathcal{A}g_{attack,i}$ lets $t_{i,j}$ time units pass between performing $a_{i,j-1}$ and $a_{i,j}$.

Based on this action sequence oriented view on the agents in A_{attack}, the learning task for these agents is to come up with an action sequence for each agent that results in a sequence of environment states fulfilling $\mathcal{G}_{emerge}$. Evolutionary learning methods have proven to be quite successful for such settings (see [3], [1]). An evolutionary algorithm works on a set of so-called individuals (the set being called a population). The individuals in a population are evaluated using a fitness measure fit and so-called genetic operators are applied to selected individuals to create new individuals. The selection process usually contains both a fitness-oriented and a random component. The new individuals replace the worst fit individuals in the old population and this process is repeated until an end condition is fulfilled.

We instantiate this very general scheme in the following manner to learn the action sequences for the agents in A_{attack}. An individual consists of an action sequence $a_{i,1}$,...,a_{i,l_i} for each of the agents $\mathcal{A}g_{attack,i} \in A_{attack}$ (or ($t_{i,1}$,$a_{i,1}$),...,(t_{i,l_i},a_{i,l_i}) if we have an asynchronous system). The fitness of an individual (($a_{1,1}$,...,a_{1,l_1}),...,($a_{n,1}$,...,a_{n,l_n})) is based on evaluating the success of the individual when applied as the strategy for the agents in A_{attack} in one or several runs. This means we feed the action sequences to the agents they are intended for, let these agents interact with the other agents, i.e. the agents in A_{tested} and A_{byst}, and the environment and produce the sequence e_0,e_1,...,e_x of (enhanced) environment states that is the consequence of executing the individual. Due to having bystanders, an individual might not produce the same environment states every time the attack agents use this individual (other reasons for this can be random elements in the tested agents or random events occurring in the environment). If this is the case, we perform several runs to evaluate an individual and sum up the fitness of these runs.

To evaluate a run of an individual, we want to measure how near this run came to fulfilling the condition $\mathcal{G}_{emerge}$. Obviously, this depends on $\mathcal{G}_{emerge}$ and on what information is represented in an enhanced environment state. But we found it very useful to base the fitness of a single run $single\_fit$ on an evaluation of the environment state sequence after every new state (see also [1]). Thus we have

$$single\_fit((e_0,...,e_x)) = \begin{cases} \text{j, if } \mathcal{G}_{emerge}((e_0,...,e_j)) = \text{true} \\ \quad \text{and } \mathcal{G}_{emerge}((e_0,...,e_i)) \\ \quad = \text{false for all } i < j \\ \sum_{i=1}^{x} near\_emerge((e_0,...,e_i)), \\ \quad \text{else.} \end{cases}$$

As the name $near\_emerge$ suggests, this function is supposed to measure how near its argument state sequence comes to fulfilling the predicate $\mathcal{G}_{emerge}$ (so, small $near\_emerge$ values indicate good individuals). We will provide an example for how to create such a function for an application in Section 4.2.

There are several schemata for genetic operators that can be used for creating new individuals out of individuals of the form we are using. Here we present four such schemata that we used in our application. Our *standard crossover* needs two individuals $((a_{1,1},...,a_{1,l_1}),...,(a_{n,1},...,a_{n,l_n}))$ and $((b_{1,1},...,b_{1,l_1}),...,(b_{n,1},...,b_{n,l_n}))$. The operator selects randomly a position j in the action sequence of all agents and performs then a standard crossover on each of the strings. The result is then $((a_{1,1},...,a_{1,j},b_{1,j+1},...,b_{1,l_1}),...,(a_{n,1},...,a_{n,j},b_{n,j+1},...,b_{n,l_n}))$.
A *standard mutation* also selects a position j in one individual randomly and changes the action there of every agent to another one, resulting in $((a_{1,1},...,a_{1,j-1},a'_{1,j}, a_{1,j+1},...,a_{1,l_1}),...,(a_{n,1},...,a_{n,j-1},a'_{n,j},a_{n,j+1},...,a_{n,l_n}))$ with $a'_{i,j} \in Act_{attack,i}$ and $a_{i,j} \neq a'_{i,j}$.

We also have so-called targeted genetic operators. Instead of selecting a random position in an action sequence, these operators make use of the fitness of the parent individuals to determine "good" positions. More precisely, we look at the $near\_emerge$ values for all the starting state sequences and identify positions, where taking in the next state into the sequence results in a substantially higher $near\_emerge$ value for this new sequence (indicating that something happened that leads us away from our goal). If we change the actions of our attack agents before this happens, then there is a good chance that we can avoid this. So, for each individual we remember the first index j, such that

$$\begin{aligned} near\_emerge((e_0,...,e_j)) \geq \; & near\_emerge((e_0,...,e_{j-1})) + \\ & too\_much\_lost, \end{aligned}$$

for a parameter too_much_lost. *Targeted crossover* then first identifies the agent indexes with actions in their sequences that are leading to an environment state between $e_{j-cotarg}$ and e_j (with $cotarg$ being another parameter). Targeted crossover then performs a standard crossover, but selects the agent index and the action sequence position only out of the ones identified in the first step. *Targeted mutation* does the same using standard mutations (with a parameter $muttarg$).

4 Experimental evaluation: testing student teams in ARES 2

In this section, we present our experiences in applying our concept from the last section to testing a student team written for the ARES 2 simulator. Our concept allowed us on the one side to test how well this team can make use of agents that try everything to help the team, while we also used the concept to create teams of attack agents that tried to achieve a very bad performance of the tested agent team.

4.1 ARES 2

ARES is a system that simulates a city struck by an earthquake and rescue robots that work together to locate and dig out survivors of the earthquake. In contrast to the RoboCupRescue simulator (see [10]) that provides the same general scenario, the goal in developing ARES was not to be as realistic as possible to allow the development of robot controls that can be used in the real world, the goal was to provide abstractions and simplifications of the real world that allow a team of students in an introductory class on multi-agent systems to enhance their learning experience by creating controls for the robots/agents that rescue many survivors in various problem scenarios within ARES (see [4]). The last version of ARES, ARES 2, allows for the interaction of several student teams within one simulation, thus forcing the students to think about how their agents should interact with other rescue agents, allowing for many possibilities for emergent behavior.

Since ARES was developed as a teaching tool, it contains various parameters, so-called world rules, that produce rather different settings for the students (each semester, a different combination of world rule instantiations can be used to avoid too much "inspiration" from the last semester). For example, the two instantiations for the world rule that determines how agents can replenish their energy can either allow the students to mainly neglect planning around energy (if agents can simply sleep everywhere) or planning around resources becomes very important (if agents can only recharge at certain places in the world). As a consequence, instructors using ARES often do not know what the best concepts for a particular instantiation of the world rules are, which makes evaluating the teams of the students difficult. Especially evaluating how the students deal with other teams is very difficult, since using fixed teams that interact with a student team allows the students to optimize towards these teams instead of developing generally good concepts. Therefore this situation is a perfect test application for our concept for testing emergent behavior.

A simulation run in ARES consists of a sequence of rounds. In each round, ARES allows every rescue agent to submit a non-communication action and, depending on the world rules, either alternatively a communication action or one or several additional communication actions. Each agent is allotted a certain amount of CPU-time each round for making its decisions. ARES then computes out of all action requests the new world state and sends to each agent its new world view. An ARES world (or scenario) consists of a collection of connected grids and each grid contains a so-called stack, where each stack element is either a piece of rubble, requiring a certain number of agents to be removed, or a survivor or survivor group. Associated with each grid field is a move cost indicating the energy an agent looses when moving on the grid. In addition to standard grids, there are fire and instant death grids that "kill" an agent moving on them and there are also recharge grids.

Agents can move from grid to grid, observe the world, remove rubble by digging and rescue survivors. Additionally, agents can send messages to other agents in their team, allowing for planning and cooperation. In order to remove a piece of rubble, this piece has to be on top of the stack of a grid, at least the required number of agents has to perform digging in the same round and, naturally, these agents have to be located on the grid at that time. The agents can belong to different teams. To rescue survivors, they have to be on top of the stack and at least one agent has to perform the rescue survivor action. A world rule allows for different ways of scoring for rescuing survivors if several agents perform the rescue at the same time.

In addition to updating the world and doing the scoring, ARES also keeps track of the energy levels of survivors and agents, letting them die if the level gets to zero or below. ARES also delivers the messages agents send to each other in the round after the round in which the message was sent.

4.2 Instantiating our method

The interaction of agents within ARES is organized in rounds, which means that we are dealing with the synchronous case of Section 3. For each scenario in ARES, there is a given number of rounds l in the simulation, which is then also the length of the action sequences of the agents in A_{attack} and the sequence of environment states. Obviously, ARES represents the environment $\mathcal{E}nv$ and the agents of the students are the agents in A_{tested}. In our experiments, the set A_{byst} was empty, but if we used a second student team, these agents could be the set A_{byst}.

While the agents in A_{tested} naturally used all the actions available to agents in ARES, our attack agents do not need to use any communication actions since their coordination is achieved by the learning process. An environment state is a state of ARES, i.e. the state of the world (all grids and their stacks, positions of all agents, energy levels of survivors and agents) and the scores of all teams. Naturally, we did not want to change the agents of the students to get internal information out of them. Therefore we did not require to enhance the environment states.

To evaluate the agent team of the students, we would like to know how good this team is if the other team, i.e. the agents in A_{attack}, tries to provide help and if this other team tries to actively work against the student team. This means that there is not exactly a particular behavior we are looking for that is shown or not. Instead of having a predicate $\mathcal{G}_{emerge}$, we have a function f_{emerge} that on the one side we want to maximize and on the other side to minimize and this function just counts the survivors the student team got points for. So, within our general scheme of Section 3, $\mathcal{G}_{emerge}$ is always false and f_{emerge} will contribute to our function $near\_emerge$.

Our $near\_emerge$ function that aims at producing agents in A_{attack} that boost the performance of the agents in A_{tested} is based on the ideas of having the agents in A_{attack} stay near the agents of A_{tested}, so that they can help dig, and of trying to get a high f_{emerge} value for the student team. More precisely, we have

$$\begin{aligned} near\_emerge_{pos}((e_0,...,e_j)) = & \sum_{i=1}^{n} dist(\mathcal{A}g_{attack,i}, e_j) \\ & -2 * f_{emerge}(e_j) \end{aligned}$$

where $dist(\mathcal{A}g_{attack,i},e_j)$ is the distance between $\mathcal{A}g_{attack,i}$ and the closest agent from A_{tested} in state e_j.

For producing attack teams that show off the possible negative behavior of the student team, we just use the f_{emerge} value, i.e.

$$near\_emerge_{neg}((e_0,...,e_j)) = f_{emerge}(e_j)$$

In our experiments, this simple function was already sufficient. Note that our algorithm tries to minimize the fitness, so that large $near\_emerge$ values make a strategy for A_{attack} bad.

We would like to point out that this instantiation of our general concept is independent of all the world rules in ARES, so that it can be used for all settings of these rules.

4.3 Experiments

For our experiments, we followed the advice in [4] and created two types of scenarios for ARES, the student team and our system: special scenarios that aim at evaluating special aspects of the students' team and random scenarios that come nearer to what a real-live scenario usually might be. In each scenario/world we have 2 agents from the student team and 2 agents in A_{attack}. The time given to the teams is always 50 rounds and each team participating in a save gets the points for this save. Each agent is given 1 second per round to do its "thinking", which means that a single simulation run takes roughly 200 seconds.

The settings for the evolutionary learner were as follows: We used a very small population size of 7 and had 10 generations only, to account for the rather time consuming evaluation of each individual. Nevertheless, this still means that a complete run of our system (i.e. computing the entry for pos or neg in one entry of Table 1) takes around 5 hours. The selection of the parents for the genetic operators was done using a 3-tournament selection, i.e. each time we need a parent, we randomly select 3 individuals from the population and then select out of these 3 the one with the best (lowest) fitness value. too_much_lost was chosen so that a scored point for the tested team would create a targeted position (for the neg case). Then the other parameters for the targeted genetic operators were set so that the position in the sequence used by the operator was directly before the drop in $near\_emerge$ occurred.

Table 1. Points scored by A_{tested}

World	1×st	2×st	pos	neg
Spec1	6	5	7	2
Spec2	4	8	8	4
Spec3	8	6	9	1
Rand1	11	8	12	6
Rand2	12	9	13	8
Rand3	14	11	16	10

In the 5×5 world Spec1 we have 7 survivors that are buried under rubble, the ones at (1,3), (5,3) and (3,5) (with (1,1) being the upper left corner of the world) requiring 2 agents to dig them out, the ones at (1,1) and (5,1) needing 3 agents and the ones at (1,5) and (5,5) just one. Additionally, at (3,3) there is an unburied survivor. The 2 agents of the student team are at (3,1) and the attack agents flank the tested agents at (1,2) and (5,2). The world Spec2 is a 7×7 world with a survivor in each corner that needs 2 agents to be dug out, 6 survivors placed into the corners of the 3×3 square in the center of the world with 2 survivors in each of the left corners and 1 survivor in each of the right corners, a survivor requiring 3 agents in the center of the world, a survivor requiring 4 agents 2 grid fields below this center and one survivor requiring 3 agents two fields above the center. One tester and one attack agent start at (1,4) and the other two agents start at (7,4). Spec3 is another 5×5 world with buried survivors requiring 2 agents at (2,2), (3,2), (4,2), (2,4), (3,4) and (4,4), a survivor requiring 3 agents in the center of the world and a survivor in the open at (1,3) and (5,3). One agent from each team starts at (3,1) and (3,5).

The worlds Rand1, Rand2, and Rand3 were generated using ARES 2's functionality for creating random worlds. All 3 worlds are 7×7 and require at a maximum 3 agents to rescue a survivor (and naturally not all of the time so many). In contrast to the special worlds, different grid fields can have different move costs. The start positions for the agents were random, but we required that always one tested agent and one attack agent started together at the same grid. All random worlds had 99 survivors in them, with random amounts of energy to start with.

In Table 1, the column 1 × st reports on the success of the students' team when working alone in a world, while column 2 × st gives the results of two incarnations of the students' team. Note that we report in column 2 × st the better score of the two teams. The columns pos, respectively neg report on the scores of the tested team when working with the best teams our learning system came up with to help the team (pos) and to hinder the team (neg). With the exception of Spec2, we see for all worlds that the learner could produce teams that increased, often substantially, the score of the students' team and also with teams that made it really tough for the tested team. Especially for Spec1 and Spec3 the results of the tested team are much less than in all other test series. The difference for the random worlds is not so big, because in these worlds there are many possibilities for the tested agents, too many to block them all.

While from an instructor's perspective, the information in Table 1 is already very useful for grading different students' teams, because it shows the spectrum a particular team can achieve, a closer look at the runs of the evolved attack teams producing the highlights in this table will show the usefulness of our approach for finding unwanted emergent behavior. Looking at the ARES 2 runs of the final attack teams for column pos, we can not report any surprising things. The attack agents stayed within close range of the tested agents, although they did not always occupy the same grid field. The attack agents learned to dig whenever the tested agents expected them to do this and as a result we got the good scores for the tested agents.

Of more interest are the ARES 2 runs that the attack agents for the neg case for Spec1 and Spec3 produce. Naturally, the attack agents did not do anything that helped the tested agents to get points, even if this meant that they also would not get points. While for the other scenarios the students' strategy for their team was able to realize that they do not get cooperation and from there on the tested agents planned only relying on themselves, the attack teams for Spec1 and Spec3 managed to get the tested agents into different kinds of weird behavior where the tested agents did not even try to dig out survivors (that they could dig out alone) when an attack agent came near and where the tested agents in fact stopped doing anything after a while although that had sufficient energy and time to rescue some more survivors. Essentially, our approach was able to produce evidence for problems in the tested agent design, evidence for clearly unwanted emergent behavior.

5 Related work

The current practice for testing for unwanted behavior in software testing is the use of random interaction sequences (see [6]), followed by human analysis and additional constructed sequences. Obviously, random sequences are just the starting point for our method. The efforts to accommodate the needs of multi-agent systems in testing, so far, either focus on developing graphical visualization tools that allow human testers to better observe the behaviors of the tested agents (see, for example, [7]) or develop special protocols and languages that provide a standard way for a developer to inspect the interactions of agents using them (see, for example, [9]).

The use of evolutionary methods to help with software testing has been suggested by several authors. For example, in [11] known (previous) errors of a system were slightly modified (and combined) by an evolutionary algorithm to create additional test cases around known problematic ones. In [12], evolutionary methods were used to develop tests covering the possible paths through a program. But, to our knowledge, there have been no works that deal with using any artificial intelligence concepts to detect unwanted emergent behavior.

6 Conclusion and future work

We presented a method to test the limits of emergent behavior in multi-agent systems that makes use of evolutionary methods to learn the behavior of agents that are interacting with the agents to be tested. The goal of these tester/attack agents is to get the tested agents to exhibit a behavior that we are interested in, for example because we do not want the tested agents to show this behavior. We tested our method by testing a team of rescue agents developed by students to act in the ARES 2 system. For a collection of different rescue scenarios, we were able to evolve specific teams that showed off the ability of the tested student team to make use of very cooperative other agents and the inability of the student team to deal well with agents that work against their efforts in some rescue scenarios.

It should be noted that the evolved agents use a very primitive agent architecture, action sequences, and use implicit cooperation due to the learning process, so that these agents are clearly less complex than the agents that they are testing. This is due to the fact that the tester/attack agents can be much more focused and specialized than the agents they test, since they only have to produce a specific behavior and do not have to cover a lot of use scenarios, which naturally the tested agents have to. Given that the complexity of the systems to test is one of the major problems in todays testing efforts, this feature of our method is very promising!

Naturally, there is a lot of future research that we are interested in. In addition to additional applications, we see a lot of possibilities to improve the learning of the cooperative behavior of the tester/attack agents, especially with regard to building in additional knowledge about the application. This ranges from usage of more pre-defined structure in the action sequences, over more application specific fitness functions to additional genetic operators (for example to deal with the timing information). Also, we expect that more complex systems to test might require a little bit more sophisticated agent architectures.

REFERENCES

[1] B. Chan, J. Denzinger, D. Gates, K. Loose and J. Buchanan, 'Evolutionary behavior testing of commercial computer games', Proc. CEC 2004, Portland, 2004, pp. 125–132.

[2] J. Denzinger, 'Knowledge-Based Distributed Search Using Teamwork', Proc. ICMAS-95, San Francisco, 1995, pp. 81–88.

[3] J. Denzinger and M. Fuchs, 'Experiments in Learning Prototypical Situations for Variants of the Pursuit Game', Proc. ICMAS-96, Kyoto, 1996, pp. 48–55.

[4] J. Denzinger and J. Kidney, *Teaching Multi-Agent Systems using the ARES Simulator*, Italics e-journal 4(3), 2005.

[5] J. Denzinger and M. Kordt, 'Evolutionary On-line Learning of Cooperative Behavior with Situation-Action-Pairs', Proc. ICMAS-2000, Boston, 2000, pp. 103–110.

[6] R.G. Hamlet, 'Predicting dependability by testing', Proc. Intern. Symp. on Software Testing and Analysis, 1996, pp. 84–91.

[7] L. Lee, D. Ndumu, H. Nwana and J. Collins, 'Visualizing and debugging distributed multi-agent systems', Proc. 3rd AA, 1999, pp. 326–333.

[8] L. Panait and S. Luke, 'Collaborative Multi-Agent Learning: A Survey', Technical Report GMU-CS-TR-2003-01, Dept. of Comp. Sci. George Mason University, 2003.

[9] D. Poutakidis, L. Padgham and M. Winikoff, 'Debugging multi-agent systems using design artifacts: The case of interaction protocols', Proc. AAMAS-2002, 2002, pp. 960–967.

[10] RoboCup Rescue: http://jelly.cs.kobe-u.ac.jp/robocup-rescue/. (as viewed on January 3, 2006).

[11] A.C. Schultz, J.J. Grefenstette and K.A. De Jong. 'Adaptive Testing of Controllers for Autonomous Vehicles', Proc. Symposium on Autonomous Underwater Vehicle Technology, IEEE, 1992, pp. 158–164.

[12] J. Wegener, 'Evolutionary testing of embedded systems', In *Evolutionary Algorithms for Embedded Systems Design*, Kluwer, 2003, pp. 1–34.

[13] S. Young and D. Aitel, 'The Hackers Handbook: The strategy behind breaking into and defending networks', Auerbach Publications, 2004.