Evaluating Different Genetic Operators in the Testing for Unwanted Emergent Behavior using Evolutionary Learning of Behavior

Jörg Denzinger, Jordan Kidney
Department of Computer Science
University of Calgary, Calgary, Canada
{denzinge,kidney}@cpsc.ucalgary.ca

Abstract

We present an experimental comparison of different genetic operators regarding their use in an evolutionary learning method that searches for unwanted emergent behavior in a multi-agent system. The idea of the learning method is to evolve cooperative behavior of a group of so-called attack agents that act in the same environment as the tested agents. The attack agents use action sequences as agent architecture and the quality of a group of such agents is measured by how near their behavior brings the tested agents to show the unwanted behavior. Our experiments within the ARES II rescue simulator with an agent team written by students show that this method is able to find unwanted emergent behavior of the agents. They also show that rather standard genetic operators (on the team level and the agent level) are already sufficient to find this unwanted behavior.

1 Introduction

One of the hot research topics in multi-agent systems (MAS) is achieving emergent behavior of a group of agents. The form of emergent behavior researchers are most interested in is the creation of synergetic effects that have as result that the group of agents achieves goals that are well beyond what can be expected from them if we just add up the individual abilities of these agents. But unfortunately, emergent behavior can also result in a system behavior that is not wanted by the developers of the system.

So far, very few researchers have looked into the problem of unwanted emergent behavior. Detecting unwanted behavior of a system in general is usually the task of the system test phase in the development, but software testing as a discipline also has not looked intensively into this problem. There the best practice is the use of randomly generated interaction sequences to see if something bad is happening (see [4]). The detection of unwanted emergent behavior of a system, which can also be caused by adaptive abilities of a single agent, still is the sole responsibility of human testers, with very limited tool support (see [6], [9]).

In [3], we presented an approach to support testing for unwanted emergent behavior that fights "fire with fire" by using techniques from learning of cooperative behavior to search for a particular unwanted behavior. The general idea of this approach is to have a group of tester (or attack) agents act as users and other systems that a group of agents, the tested agents, interact with. We then use learning techniques for the attack agents to have them produce interaction sequences with the tested agents that come nearer and nearer to showing the particular behavior that is unwanted. In [3], we used as learning method an evolutionary learning method that uses sequences of actions as individuals, one sequence for each attack agent.

[3] presented a proof of concept application that tested multi-agent systems developed by students of a basic MAS class. While fitness functions for two rather different behaviors were presented, other parameters of the evolutionary learning method, like genetic operators or population size, and their influence on the results were not evaluated. There have been many studies on the influence of such parameters on the performance of evolutionary algorithms in general (see, for example, [13] or [7]), but the gist of these studies is that it depends on the particular application what setting of these parameters offers a chance for a good performance, with the usual caveat that there still might be some problem instances for which this setting might not be good.

Since we used a class of genetic operators, namely so-called targeted operators, that were specially developed for action sequences, in this paper we examine and analyze different genetic operators and their combinations with regards to their influence on finding unwanted emergent behavior. Our experiments with simulated rescue teams written by students show that quite a few of the genetic operators allow the system to find a particular unwanted behavior (the students' agents freeze at some point in the simulation). But in-

terestingly, the combination of rather standard operators on the level of the agent team and on the level of a single agent was consistently the best choice in our experiments both with regard to quality of the evolved attack team and the number of generations needed to produce the attack team.

2 Agents and multi-agent systems

Since we provide a rather general framework for testing multi-agent systems we use a rather general definition of agent (although our attack agents will naturally be a very specific instantiation of the following definition). An agent $\mathcal{A}g$ = (Sit,Act,Dat,f_{Ag}). Sit is a set of situations that the agent can find itself 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. When having 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 to make this decision.

Obviously, a multi-agent system requires a set of agents A. But these agents need to share an environment (or at least parts of it) in order to interact with each other, and this environment $\mathcal{E}nv$ has quite some impact on what the agents can do. So, formally a multi-agent system $\mathcal{M}AS$ is a pair $\mathcal{M}AS$ = (A, $\mathcal{E}nv$). Actions of the agents might change the environment (or it can change on its own) and therefore $\mathcal{E}nv$ should consist of a set of environment states. Going back to our agent definition, a situation in an agent's set Sit might contain a view on the current environment state, information about the agent itself that has not found its way into a data area in Dat, and information about the other agents based on the perception of the agent.

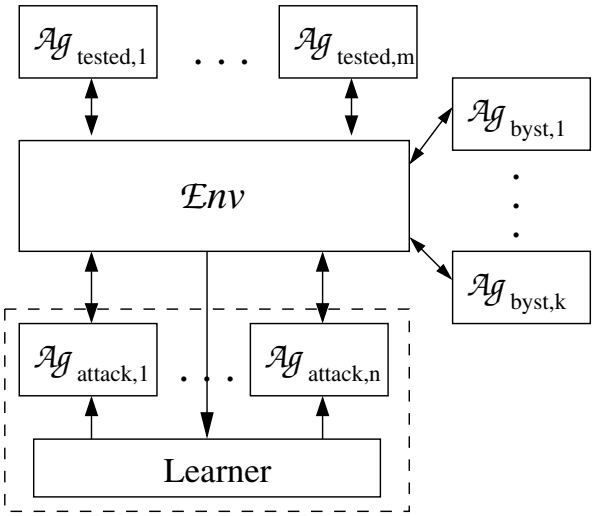
3 Using learning of behavior to test for emergent behavior

Our general idea for testing a multi-agent system $\mathcal{M}AS_{tested}$ = (A_{tested}, $\mathcal{E}nv$) for unwanted emergent behavior is to use a set of so-called attack agents[1] A_{attack} that play the role of entities that the agents in A_{tested} usually interact with in $\mathcal{E}nv$. Such entities can be some of the team mates of the agents in A_{tested}, human users interacting with $\mathcal{M}AS_{tested}$ or other systems that interact with $\mathcal{M}AS_{tested}$. In our general setting, we also allow for the existence of a third set A_{byst} of agents, the bystander agents, which are agents that are neither in A_{tested} nor under control of the tester, but interact with at least some agents in A_{tested} in the environment.

Figure 1 presents a graphical view of the general setting. Key for our general approach is that the behavior of

[1] We use the term attack in the following, since "tester" and "tested" are very difficult to distinguish and what the attack agents are doing can definitely be seen as an attack on the tested agents.

Figure 1. General setting of our approach



the agents in A_{attack} is the product of learning that aims at figuring out what interactions of these agents with the environment, the tested agents and the bystander agents may produce (or at least comes near) a certain behavior by the tested agents. While Figure 1 suggests that this learning is performed by a central learner that feeds behaviors into the attack agents (which is the approach we have taken in Section 4), we can also envision that the learning takes place inside of the agents of A_{attack}, which then obviously requires concepts for coordinating the learning.

More formally, our general approach is as follows: If we look at the setting in Figure 1 as an outside observer, then the interaction of an agent $\mathcal{A}g_{attack,i} \in A_{attack}$ with the other agents can be described as a sequence of actions $a_{i,1}$,...,a_{i,l_i} ($a_{i,j} \in Act_{attack,i}$). The action sequences of all the agents in A_{attack} then result in a sequence e_0,e_1,...,e_x of (enhanced) environment states. An (enhanced) environment state is a view on $\mathcal{E}nv \times \bigcup_{Ag \in A_{tested}} Dat_{Ag}$, which means that we allow the observer (which could be the learner) to see some aspects of the environment (hence our use of the term "view") and the content of some or all the internal data areas of the tested agents. An unwanted emergent behavior of the tested agents can then be characterized by defining a condition $\mathcal{G}_{emerge}$ on a set of (enhanced) environment states, i.e. $\mathcal{G}_{emerge}$((e_0,e_1,...,e_x)) = true.

One of the distinctions we can make regarding multi-agent systems is whether they interact with the environment and each other in a synchronous fashion or in an asynchronous fashion. If interaction takes place synchronously, then the length of the action sequences of the attack agents is always the same and by going through the action sequences simultaneously, we also create a sequence of environment states of this length (formally: $l_i = l_j$ and $l_i = x$ for all i, j). Allowing for asynchronous interac-

tions requires from us to record every change of the environment state. Even more, it might be necessary to add to the action sequences of the attack agents information on the time that passed between performing the actions of the sequence. This leads us to a sequence of *timed actions* for an attack agent $\mathcal{A}g_{attack,i}$: ($t_{i,1}$,$a_{i,1}$),...,(t_{i,l_i},a_{i,l_i}). Every $t_{i,j}$ indicates the number of time units that pass between the previous action and the new action and, as usual, $a_{i,j} \in Act_{attack,i}$.

Coming back to the task of finding an unwanted behavior, the goal of the learning process we use has to be to create instantiations of the attack agents in A_{attack} (or *behaviors*) that produce action sequences for these agents that result in a sequence of environment states that fulfills $\mathcal{G}_{emerge}$. Obviously, the particular learning approach for this will depend on the agent architecture and the possible instantiations of it that we have available, what we (resp. the learner(s)) can observe, what kind of actions the agents have available and last but not least how $\mathcal{G}_{emerge}$ is defined.

4 Evolutionary learning to test for unwanted behavior

There are various methods described in the literature about how behavior for a single agent or a group of agents can be learned ([8] provides an overview of most of these methods). Evolutionary learning methods have a very good track record for being able to come up with looked-for behaviors requiring very little knowledge about the particular setting a group of agents finds itself in (see, for example, [1]). The merging of knowledge-based components and random components that is typical for evolutionary algorithms creates effects similar to what human intuition achieves, which is not only important for learning in general but especially something that is very valuable in testing of systems.

An evolutionary algorithm works on a set of so-called individuals that often represent solution candidates for the problem instance that the algorithm tries to solve. The set of individuals is called the population and every individual in the population is evaluated using a so-called fitness measure fit that tries to represent a lot of the knowledge that the developer of the algorithm has about the problem to solve. The evolutionary search is performed by applying so-called genetic operators to selected individuals, which creates new individuals. These new individuals replace the individuals in the population with the worst fitness. Often, several new individuals are created at once and then the replacement of the bad individuals takes place. The new individuals with the remaining ones are then called the next generation. This process is repeated until a given end condition is fulfilled. The selection of the individuals for the application of the genetic operators usually combines some random component with some fitness orientation, creating the intuition-like effect that we mentioned above.

This general scheme can be easily instantiated to the intended learning process from the last section. We want to learn behaviors (or *strategies*) for the attack agents in A_{attack}. The easiest way to describe such a behavior is by action sequences, as we did in the last section. Therefore, an obvious agent architecture for the agents in A_{attack} is just to execute a given action sequence. And this means that an individual in an evolutionary algorithm aiming at creating particular behaviors best 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). So, an individual has the general form (($a_{1,1}$,...,a_{1,l_1}),...,($a_{n,1}$,...,a_{n,l_n})).

To evaluate the fitness of an individual we follow the general idea used to evaluate programs in evolutionary computation: we apply the attack team represented by the individual within the environment and let the attack agents interact with the tested agents and the bystander agents. As stated before, this will produce a sequence e_0,e_1,...,e_x of (enhanced) environment states and this environment states sequence represents the obvious item to measure the success of the individual on. Note that there are several reasons why repeating the application of the attack team can lead to a different environment sequence. There might be random events in the environment, the bystander agents are not under our control and therefore cannot be set into the same start state all the time, and the tested agents might do some random decisions. Since we are interested in reproducible results, we perform several of these trial runs for each individual and sum up the fitness of each of these runs.

The fitness of an individual should reflect how near the produced environment state sequence comes to fulfilling the condition $\mathcal{G}_{emerge}$ that we are interested in. Obviously, in general this depends on $\mathcal{G}_{emerge}$ and on what information is represented in an enhanced environment state. In [3] and [1], we based the fitness of a trial run $single\_fit$ on evaluating the environment state sequence after every new state:

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

where $near\_emerge$ is measuring how near its argument state sequence comes to fulfilling $\mathcal{G}_{emerge}$. How to define $near\_emerge$ for a particular multi-agent system $\mathcal{MAS}_{tested}$ depends on this multi-agent system. We will describe an example in the next section.

While the structure of an individual is rather straightforward given our view of a multi-agent system from the last section, this structure is already rather complex and there-

fore allows for many ideas regarding how to define operators on individuals for the creation of new individuals. In fact, there are two levels on which such operators can work, namely the *team level* and the single *agent level*. And, as for many evolutionary algorithms, on both levels we can have crossover operators that essentially recombine traits from several individuals to create new ones and mutations that allow for the creation of new traits in individuals. In this paper, we will be looking at 3 pairs of operators, one on the team level (that we call team level operators) and two on the agent level, namely the standard operators and the targeted operators.

The idea of the team level operators is to create new individuals by combining individual agent strategies (at least for the crossover operator). So, given two individuals $((a_{1,1},...,a_{1,l_1}),...,(a_{n,1},...,a_{n,l_n}))$ and $((b_{1,1},...,b_{1,l_1}),...,(b_{n,1},...,b_{n,l_n}))$, *team level crossover* creates the individual $((c_{1,1},...,c_{1,l_1}),...,(c_{n,1},...,c_{n,l_n}))$, with $(c_{i,1},...,c_{i,l_i}) = (a_{i,1},...,a_{i,l_i})$ or $(c_{i,1},...,c_{i,l_i}) = (b_{i,1},...,b_{i,l_i})$. The decision for each agent where to get the strategy from is done randomly. *Team level mutation* takes one individual $((a_{1,1},...,a_{1,l_1}),...,(a_{n,1},...,a_{n,l_n}))$ and creates the new individual $((a_{1,1},...,a_{1,l_1}),...,(a'_{i,1},...,a'_{i,l_i}),...,(a_{n,1},...,a_{n,l_n}))$, with $1 \leq i \leq n$ and $(a'_{i,1},...,a'_{i,l_i})$ created randomly using the actions of $Act_{attack,i}$.

The first pair of agent level operators represents the usual ideas of evolutionary algorithms with regard to sequences: for crossover, cut two individuals at the same position and take the beginning of the sequence from one individual and the end of the sequence from the other individual to create the new individual. And for mutation, select one element of the sequence and change it to another element from the set of possible elements. More formally, this means that *standard crossover* requires as parents two individuals $((a_{1,1},...,a_{1,l_1}),...,(a_{n,1},...,a_{n,l_n}))$ and $((b_{1,1},...,b_{1,l_1}),...,(b_{n,1},...,b_{n,l_n}))$, it selects a sequence position j randomly and then creates as new individual $((a_{1,1},...,a_{1,j},b_{1,j+1},...,b_{1,l_1}),...,(a_{n,1},...,a_{n,j},b_{n,j+1},...,b_{n,l_n}))$. *Standard mutation* takes an individual $(a_{1,1},...,a_{1,l_1})$, $...,(a_{n,1},...,a_{n,l_n}))$, selects again a sequence position j randomly and creates the new individual $(a_{1,1},...,a_{1,j-1},a'_{1,j},$ $a_{1,j+1},...,a_{1,l_1}),...,(a_{n,1},...,a_{n,j-1},a'_{n,j},a_{n,j+1},...,a_{n,l_n}))$ with $a'_{i,j} \in Act_{attack,i}$ and $a_{i,j} \neq a'_{i,j}$.

Our third pair of operators is what we consider an improvement of the standard operators aiming at taking away the randomness of the place within a sequence where crossover, resp. mutation occur and targeting positions that are not good. Here not good means that during the evaluation runs that are done to compute the fitness of an individual, the actions of the agents at this position are not good. And this then means that we see quite a drop in the evaluation of the state that was created by these actions (compared to the previous states). More precisely, we use the $near\_emerge$ values for all the starting state sequences and identify in each run the first position in the produced environmental state sequence, where adding the next state produces a substantially higher $near\_emerge$ value for this new sequence. And then we use as core for selecting the position for a standard crossover or a standard mutation the smallest such position j over the evaluation runs done for the fitness computation. What is substantially higher is defined via a parameter too_much_lost and the inequation

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

Since the reason for getting into a much worse state might not be the last action taken, but for sure an action taken very recently, we have *targeted crossover* select a crossover position between $e_{j-cotarg}$ and e_j. $cotarg$ is another parameter. *Targeted mutation* selects the mutation position between $e_{j-muttarg}$ and e_j (with $muttarg$ the third parameter regarding targeted operators).

5 Experimental evaluation in ARES II

In this section, we will first present a short introduction to the ARES II system, which provides us with the environment in which all our agents are interacting. Then we instantiate our method from last section to testing agent teams, written by students, that act within ARES II. Next, we present details on one particular attack team found by our method that reveals a weakness in one student team that we were not able to find without our method. Finally, we compare the results of the different evolutionary operators presented at the end of the last section (and their combinations) for several different ARES II worlds and searching for attack teams that let the student team look bad.

5.1 The ARES II system

The ARES and ARES II systems simulate a city struck by an earthquake and allow to have rescue robots that are controlled by external programs (agents) act within this simulation (see [2]). While the well-known RoboCupRescue simulator (see [10]) tries to provide a rather realistic simulation of such a disaster scenario, the goal in developing ARES and ARES II was to provide abstractions and simplifications of the scenario that allow a team of students in a basic multi-agent systems class to create agents controlling the rescue robots within the very limited time of this class. ARES II is a new version of ARES that allows several teams of rescue agents to interact within one rescue scenario simulation, which naturally sets up a lot of opportunities for emergent behavior between all agents.

A basic component of the ARES II system are the so-called *world rules*. They allow for setting rather different tasks for the students by choosing their instantiations

differently. Among the world rules are how many agents maximally are needed to remove rubble in the world, how agents can recharge their energy (that is used up by their actions), how communication actions are counted with regard to the action allotment, and how rescuing survivors results in scores in situations where agents from different teams are doing the rescuing. A simulation run in ARES II is performed in rounds or turns. In each turn, each agent can perform one action and has a certain amount of processing time to decide on this action. The action(s) of an agent are send to ARES and ARES then updates its simulated world accordingly. After such a world update, ARES sends to each agent the results of its actions and all communications that were directed to the agent.

A world (or scenario) in ARES consists of a collection of connected grids (see Figure 2). Each grid contains a stack with each layer in the stack either being a piece of rubble or a survivor or survivor group. In Figure 2, the top layer of the stack of a grid field is indicated in the lower left corner of the field. A grey block with a number indicates a rubble piece with the number telling us how many agents are needed to remove this piece. A block without number (in blue) indicates a survivor. Each grid field also has a move cost associated with in. The ARES viewer in Figure 2 uses a color code to indicate this move cost in the lower right block (the smaller block) in a grid field. The picture below the title MV Legend provides the meaning of these colors. The move cost is the amount of energy that an agent looses when moving on this field. A grid field can also indicate that it is a fire or instant death field (which results in killing an agent moving on it), but we did not use those in the world of Figure 2. If agents get new energy by sleeping on charge grids, then we usually also have this type of grids in a world. In the graphical representation of an ARES grid we can also see some circles with numbers, starting in the upper left corner of the field. These are the agents, identified by their numbers, that are occupying the field. Agents in different teams have different colors (grey scales).

To rescue a survivor, agents have to dig them out using the dig command. Only if at least the necessary number of agents to remove the piece are sending the dig command in the same round while standing on the grid ARES will remove this piece of rubble. Other actions of an agent are naturally moving from grid to grid and rescuing survivors. Agents can also observe other grids and communicate with other agents.

When creating a scenario for ARES, the user can either tell ARES to create a random world or the user can totally control the world by defining each grid individually. The user then also defines the energy of each agent and each survivor (survivors loose energy every turn and therefore can die), the start position of the agents and how many turns a simulation will take.

5.2 Instantiation of our method

The most crucial part in instantiating our method from Section 4 is obviously the fitness function, more precisely the function $near\_emerge$. While in [3] we presented two such functions for opposite unwanted behaviors, in this paper we concentrate on the $near\_emerge$-function that tries to learn an attack team that lets the student team look bad. This means that we are interested in very low rescue scores by the team A_{tested} (ideally this would be a score of 0). We called the function that measures the score of a team f_{emerge} and thus have

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

We can then define $\mathcal{G}_{emerge}$ by $\mathcal{G}_{emerge}((e_0,...,e_j))$ = true if $f_{emerge}(e_k) = 0$ for all $1 \leq k \leq j$.

With regard to the length of the action sequences for the agents in A_{attack} we use the number of turns given by the user. The agents in A_{attack} do not need to use communication actions, since their coordination is already achieved by the learner. They also do not observe the world, so that we are left with the movement actions, the digging action and the rescue action. As $near\_emerge$ shows, our view on the environment (which is the ARES II system) is very limited and we do not use any internal information from the agents in A_{tested}.

5.3 An interesting result

Our first experiments in [3] resulted in finding some clearly unwanted behavior in one of the student teams that we tested. The agents of the students sometimes freeze when working in a scenario with another team (although this unfortunately did not happen when we tested this team using teams from other students and the tested team itself as partner team). We were able to create attack agent teams for several ARES II worlds that achieved this effect and in this subsection we will provide some more information on what happens for a world that we called Path3 (for pathological 3).

Path3 is a simplistic scenario where there are a total of nine survivors in the world. Two of the survivors are out in the open and can be easily saved, six more survivors are buried under one layer of rubble each that requires two agents to remove it. Finally there is one survivor that is buried under one piece of rubble that requires three agents to be removed. This piece of rubble was intended to see if the agents from the different teams would cooperate (the world rule settings score a survivor for every team that performs the save action after the rubble is removed and therefore the different student teams should try to cooperate, which can be tested by using a different fitness function, see [3]).

Table 1. Comparison of scores and generation for different operator combinations

Exp.	Te		St		Ta		Te&St		Te&Ta		St&Ta		All	
	sc.	gen.	sc.	gen.	sc.	gen.	sc.	gen.	sc.	gen.	sc.	gen.	sc.	gen.
Path3	3	4	3	4	3	6	3	1	5	2	3	2	3	6
Rand2	3	2	5	1	5	3	3	2	3	5	7	2	3	2
Rand3	5	4	9	4	7	1	4	4	6	3	10	2	8	7

Working alone the student team rescues all 8 possible survivors.

Figures 2 to 4 show the situations after 4, 21 and 32 turns when the student team works together with the attack team that our method found. The attack team is represented by the darker agents. In Figure 3, we see that the student team was able to save one of the survivors in the open and two of the survivors buried under rubble. But this picture shows the student agents in the same positions as they were after turn 20. And these two grids are where the agents will stay for the rest of the run, as Figure 4 indicates. The attack team does not save any survivors (there was no need to do that to get the low score for the student team and the attack teams score is not measured for its fitness), but the agents in A_{tested} simply do not get their act together.

5.4 Experiments with different genetic operators

To evaluate the three pairs of genetic operators presented in Section 4 we have selected 3 examples out of the ones presented in [3]. We already presented information on Path3. Rand2 and Rand3 are randomly created worlds of size 7×7 with 99 survivors in it. For all scenarios, we give the agents 50 turns which results in a time requirement of roughly 200 seconds per simulation run. Note that in Rand3 the agents of the students do not freeze after some time.

Due to the rather long time requirements for a single simulation run, we used a very small population size of 7 and 10 generations, again. The selection of the parents for a genetic operator used 3-tournament selection and too_much_lost was chosen so that a scored point for the tested team would create a targeted position. The crossover to mutation ratio was always set 30:70. The runs using two pairs of operators give 50 percent of the operation application to each of the pairs and then use the 30:70 ratio again within the pair. To make sure that all operators are used in every generation, we had to perform each operator once when using all operators (with an additional targeted crossover for the seventh new individual). In Table 1, Te refers to the team level operators, St to the standard operators and Ta to the targeted operators.

As Table 1 shows, there is a clear winner, namely the combination of the team level operators with the standard operators for strings on the agent level. This combination produces always the best result with regard to the score of the tested agents (columns sc.) and it produces this result always the earliest (as indicated by column gen. that reports the generation the result first occurred). And it seems that the team level operators are very important for this, since using them alone is already very good: for Path3 and Rand2 we get the same score and the score for Rand3 is only one survivor higher than the score by the combination.

To our disappointment, it seems that the targeted operators, that were crucial for the success in [1], are not very useful. In combination with the team level operators they only produce the best score for Rand2, but even for this example they need 3 generations longer. And the combination of all three pairs of operators performs only at the top for Rand2, while needing the most generations to produce the freeze for Path3. For Rand3 it only beats the combination of the agent level operators.

The results suggest that concentrating more on the team level leads to better results, at least for this application and this particular team of tested agents. Given the large time requirements of the simulations and the consequent low number of individuals that we want to produce, the combination of Te and St should be the first choice of a user.

6 Related work

As already stated, there is nearly no work on helping human testers finding unwanted emergent behavior in MAS. Exploratory testing (see [5]) introduces learning from tests as an important part of testing, but puts the task of learning solely with the human tester. In [6], we find some work that helps visualize the behavior of multi-agent systems, which naturally can help finding unwanted behavior. [9] is an example for special protocols and languages that are developed to give a developer or tester a standard way to inspect the interactions of agents.

The use of evolutionary methods to help in software testing is developing in a rather active research area. [11] is an early paper that used an evolutionary algorithm to slightly modify and combine known errors to create new test cases for controllers. The work in [12] helps with code inspections by finding examples that cover the possible paths through a program. While there are other works, whole workshops in fact, these two works come the nearest to our approach but even they are not dealing with finding unwanted behavior of an MAS from scratch.

7 Conclusion

We presented an evolutionary learning method that allows to find unwanted emergent behavior of a multi-agent system. Our method uses a rather primitive agent architecture for the agents that interact with the agents that we want to test, thus testing agents with less complex agents. We evaluated several combinations of genetic operators and found that the usage of operators that work on the team level, i.e. exchange whole agents when creating new attack behaviors, is clearly recommended. This team level operators were newly introduced for the method in this paper. Future work will be looking at more experiments with student teams developed for ARES II for other world rule instantiations.

References

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

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

[3] J. Denzinger, J. Kidney: Testing the limits of emergent behavior in MAS using learning of cooperative behavior, Proc. ECAI-06, Riva del Garda, 2006, pp. 260–264.

[4] R.G. Hamlet: Predicting dependability by testing, Proc. Intern. Symp. on SW Testing and Analysis, 1996, pp. 84–91.

[5] C. Kaner, J. Back, B. Pettichord: Lessons Learned in Software Testing, Wiley Computer Publishing, 2001.

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

[7] S. Luke, L. Spector: A comparison of crossover and mutation in genetic programming, Proc. GP-97, Stanford, 1997, pp. 240–248.

[8] L. Panait, 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, 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, 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] X. Yao: An empirical study of genetic operators in genetic algorithms, Microprocessing and Microprogramming 38, 1993, pp. 707–711.

Figure 2. Round 4: A_{tested} + A_{attack}

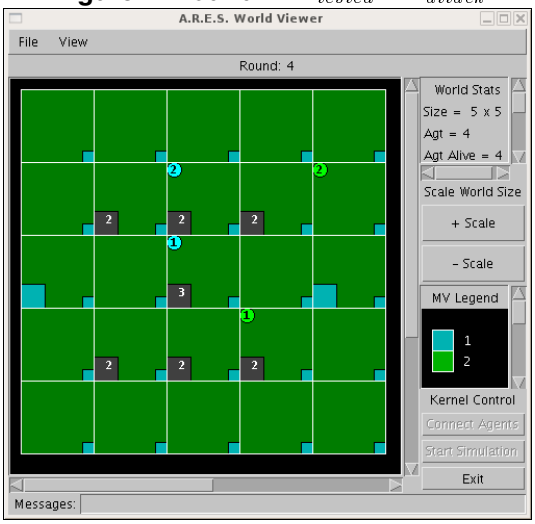


Figure 3. Round 21: A_{tested} + A_{attack}

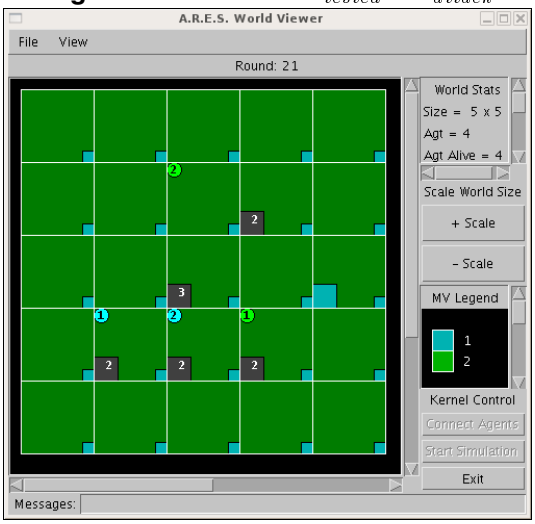


Figure 4. Round 32: A_{tested} + A_{attack}