

# Testing harbour patrol and interception policies using particle-swarm-based learning of cooperative behavior

Tom Flanagan, Chris Thornton, and Jörg Denzinger

**Abstract**—We present a general scheme for testing multi-agent systems, respectively policies used by them, for unwanted emergent behavior using learning of cooperative behavior via particle swarm systems. By using particle swarm systems in this setting, we are able to create agents interacting/attacking the tested agents that can use parameterised high-level actions. We also can evaluate the quality of an attack using several measures that can be prioritised and used in a multi-objective manner in the search. This solves some general problems of other testing approaches using learning.

We instantiate this general scheme to test harbour patrol and interception policies for two Canadian harbours, showing that our approach is able to find problems in these policies.

## I. INTRODUCTION

The backbone of any security operation are the policies that guide for everyone involved in the operation their actions and behaviors. While it is very difficult to have policies that eliminate every risk, the better the policy the less risk is to be expected, assuming that everyone follows their roles in the policy. Therefore, testing policies to identify attacks that are not covered by them is an important task for policy developers. Finding problems in policies, like finding problems in many systems, is a very challenging task that usually requires domain experts that do not fall prey to making assumptions that later turn out to be wrong. What problems such assumptions can cause was demonstrated by Lenat’s systems (see [1]) quite some time ago. But not only wrong assumptions cause problems with the kind of testing required for security policies: the possible events and their timing cover so large possibility spaces that it is impossible to systematically test each possibility. Then the intuition of the testers influences the test outcome substantially. And human intuition is not always working on command.

Computational Intelligence, more precisely evolutionary learning of cooperative behavior, can assist in the testing of security policies. In [2] and [3], the concept of evolutionary learning of cooperative behavior to test complex systems and especially multi-agent systems was introduced. It allows for the use of learning and cooperating agents to take over the role of the human tester when provided with some kind of measure for situations that indicates how near to an unwanted behavior the tested system has come in the measured situation. While the two cited examples showed the big potential of this approach, we had to observe several problems when trying to use the instantiation presented in [3] for the problem of testing harbour patrol and interception

policies. In particular, testing complex policies requires interaction sequences that are too long to make them feasible to be learned. There are subproblems involved that are difficult to solve via learning but for which good other solution approaches exist. Descriptions of unwanted behavior often include several conditions that need to be fulfilled for the behavior to be called unwanted. And the fitness measure suggested in [3] requires some weighting of these conditions that is difficult to come up with.

In this paper, we describe an extension, resp. alternative, to the approach in [3] that solves the problems mentioned above well enough to allow us to test harbour patrol and interception policies in simulations by evolving coordinated “attacks” by so-called attack agents that reveal weaknesses (unwanted emergent behavior) in these policies, resp. in a multi-agent system employing these policies. The general ideas of our approach are to use a multi-objective particle swarm system to evolve waypoints for the attack agents, realize the navigation from waypoint to waypoint using standard path planning approaches and compare the quality of two waypoint sequences using the lexicographic combination of several sets of multiple objectives. The later allows to make use of the different conditions on what is considered a successful attack against a policy.

The experimental evaluation of our approach with several patrol and interception policies we created for two harbours with military installations showed that our method is able to find various types of weaknesses in the policies, thus -as in case of [3]- fighting “fire with fire” by using the emergent behavior of a learning multi-agent system to identify emergent misbehavior of another multi-agent system.

## II. BASIC CONCEPTS

In this section, we explain the general setting for using learning of behavior to test systems and introduce basic notations around this setting. In the second subsection, we present Particle Swarm Systems (PSS) and how they can be used as a method for multi-objective optimisation.

### A. Learning of cooperative behavior for testing

A multi-agent system (MAS) usually consists of a set of agents  $A$  and an environment  $Env$  in which the agents of  $A$  interact with each other. Testing such a MAS for unwanted behavior usually means that there are either additional agents outside of  $A$  acting in  $Env$  or that  $A$  is split into a set of agents to be tested and a set of agents that are under control of the tester. In both cases, from a testing point of view, we have two sets of agents, namely the set  $A_{tested}$

T. Flanagan, C. Thornton, and J. Denzinger are with the Department of Computer Science, University of Calgary, Calgary, Canada (email: {thomas.m.flanagan,chris.thornton,denzinge}@ucalgary.ca).

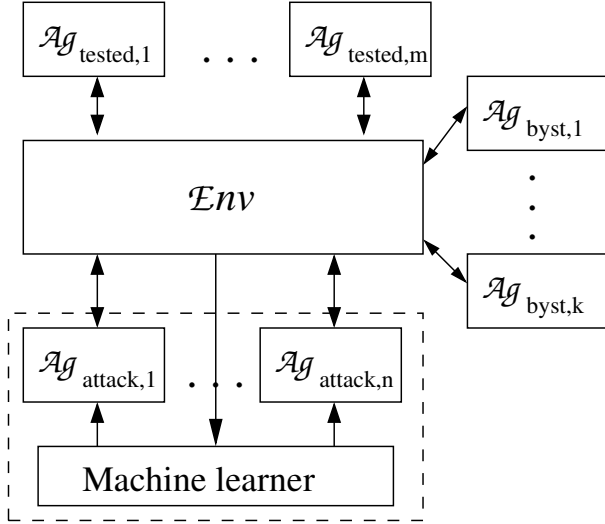


Fig. 1. General unwanted behavior test setting

$= \{Ag_{tested,1}, \dots, Ag_{tested,m}\}$  of agents that are tested<sup>1</sup> and the set  $A_{attack} = \{Ag_{attack,1}, \dots, Ag_{attack,n}\}$  of agents that aim to attack the statement that  $A_{tested}$  does not show a particular unwanted behavior. In [3], a third set of agents, so-called bystander agents,  $A_{byst} = \{Ag_{byst,1}, \dots, Ag_{byst,k}\}$ , was also introduced to model the fact that often there will be also agents in  $Env$  that are not part of  $A_{tested}$  and also not under the control of the (human) tester.

Instead of having a human tester guiding the agents in  $A_{attack}$ , trying to find interactions that result in the agents in  $A_{tested}$  showing the unwanted behavior, a machine learner directs the agents in  $A_{attack}$ , creating for each  $Ag_{attack,i}$  a strategy, having the agents in  $A_{attack}$  with the created strategies interact with the agents in  $A_{tested}$  and  $A_{byst}$  in  $Env$  (or a simulation of it), analysing the resulting behavior of the tested agents (and what happened in  $Env$ ) leading to new strategies for the attack agents, and repeating this cycle until the unwanted behavior emerges or a resource limit is reached. Figure 1 presents the general structure of this kind of testing system.

If we describe the set of actions of an agent  $Ag_{attack,i}$  by the set  $Act_{attack,i}$ , then a particular test run using a particular strategy for each attack agent can be described as a sequence of timed actions for each agent in  $A_{attack}$ :  $(t_{i,1}, a_{i,1}), \dots, (t_{i,l_i}, a_{i,l_i})$ , with  $a_{i,j} \in Act_{attack,i}$  and  $t_{i,j}$  a number of time units, that creates a sequence  $e_0, e_1, \dots, e_x$  (a *trace*) of environmental states. Naturally, environmental state changes are not only due to actions of attack agents, they also are the result of the actions of the other agents or even due to environmental events.

## B. Particle Swarm Systems

Particle Swarm Systems (PSS, see [4]) are among the many set-based search models used for optimisation and inspired by nature. As the name suggests, PSS are inspired by physics and biology, enhancing the idea of a moving particle

with the attraction behavior of members of a swarm. The search state in a PSS is represented by a set of  $l$  particles  $p_i$  each of which is characterised by its current position  $pos_i$ , its current velocity  $v_i$ , and its best position  $best_i$ . The position of a particle represents a possible solution to the search problem the PSS is supposed to solve, and usually such a solution is a vector of continuous variables, although by rounding to the nearest allowed value it is possible to also deal with discrete variables. The quality of a position/solution is with respect to a function  $f$  that the PSS is supposed to optimise.

The search in the basic variant is performed by updating each particle in the state according to the following equations:

$$v_i^{new} = Wv_i + C_1r_1(best_i - pos_i) + C_2r_2(Best - pos_i), \quad (1)$$

$$pos_i^{new} = pos_i + v_i^{new}, \quad (2)$$

where  $W$  is a weight parameter controlling the influence of the previous velocity,  $C_1$  is the so-called *cognitive learning factor*,  $C_2$  the so-called *social learning factor* and  $r_1, r_2 \in [0, 1]$  are random values chosen by the search control.  $Best$  is the best position the whole swarm has found so far. Naturally, if a particle reaches a new best position, i.e.  $f(pos_i^{new})$  is better than  $f(best_i)$ , then  $best_i$  is updated, i.e.  $best_i^{new} = pos_i^{new}$ , else it stays, i.e.  $best_i^{new} = best_i$ . This update of each particle is repeated either for a given number of update rounds or until a given amount of time has elapsed or the best position fulfills certain conditions.

Particle Swarm Systems can also be modified for multi-objective optimisation. In multi-objective optimisation, we are not just interested in finding an optimal solution of variable values for one quality function  $f$ , but for a whole vector  $\vec{f} = (f_1, \dots, f_q)$ . Usually, there is not one position that is optimal for all quality functions, instead positions that are very good for one  $f_i$  often are not so good for an  $f_j$ . A key concept of multi-objective optimisation is the *domination* of one solution  $x_1$  over a solution  $x_2$ , denoted by  $x_1 \succ_{\vec{f}} x_2$  which is defined by  $f_i(x_1) \geq f_i(x_2)$  for all  $i$  (if our goal is to maximise all functions in  $\vec{f}$ ). The subset  $POS$  of all possible solutions  $sol$  to a multi-objective optimisation problem where for each  $x_1 \in POS$  we have that there is no  $x_2 \in Sol$ ,  $x_1 \neq x_2$ , such that  $x_2$  dominates  $x_1$  is the so-called Pareto-optimal set of the particular instance of the problem.

Since the domination relation gives us only a partial ordering on positions in a PSS, a PSS for multi-objective optimisation is more complex than the basic version. In fact, there are many different variants of PSS for multi-objective optimisation (see [5] for an overview). For our application, the following rather primitive variant was sufficient. We extend the definition of a particle to a triple  $p_i = (pos_i, v_i, Ownbest_i)$ , where the set  $Ownbest_i$  records all previous positions of  $p_i$  that are not dominated by any of the other previous positions of  $p_i$ . Instead of just one solution  $Best$  for the whole particle swarm, we select the position  $Best$  in Equation (1) out of the sets  $Ownbest_{(i-1) \bmod l}$  and  $Ownbest_{(i+1) \bmod l}$  of non-dominated solutions of the “neighbours” of particle  $p_i$ .

<sup>1</sup>These tested agents will be called patrol and interception agents in our application in Section IV-A.

The selection is done randomly every time Equation (1) is applied, as is the selection of an element from  $Ownbest_i$  to play the role of  $best_i$  in (1). After the new position of  $p_i$  is created, it is checked if it is dominated by an element of  $Ownbest_i$ . If it is not, it is added to  $Ownbest_i$  and all elements in  $Ownbest_i$  that are dominated by  $pos_i^{new}$  are removed from it. Again, the search is finished if a given time limit or number of updates is reached or the union of all  $Ownbest_i$ s fulfills certain conditions.

### III. PSS-BASED LEARNING OF ATTACK BEHAVIOR

As already stated, our goal is to modify the general testing approach by learning of cooperative behavior from Section II-A to allow for actions with parameters that can be expressed as real numbers and to allow for several measures how near to an unwanted behavior an attack team has come, without having to come up with one combined measure. As a side effect, being able to have actions with parameters allows then also to use “high-level” actions in the attack agents that trigger complex computations generating whole sequences of “low-level” or primitive actions of an agent. All such high-level actions usually require parameters.

Our solution is to use a multi-objective particle swarm system to perform the learning of the behavior of the attack agents. This allows us to express the strategy of an individual attack agent  $Ag_{attack,i}$  as a sequence of timed *parameterised* actions:

$(t_{i,1}, a_{i,1}(p_{i,1,1}, \dots, p_{i,1,pa_{max}})), \dots, (t_{i,l_i}, a_{i,l_i}(p_{i,l_i,1}, \dots, p_{i,l_i,pa_{max}}))$ , where  $t_{i,j}$  is again a number of time units,  $a_{i,j}$  the (numerical) identifier for a parameterised action and  $p_{i,j,o}$  (numerical) parameter values for actions. A position  $pos$  for our PSS then consists of a strategy for each attack agent, i.e.

$$pos = (((t_{1,1}, a_{1,1}(p_{1,1,1}, \dots, p_{1,1,pa_{max}})), \dots, (t_{1,l_1}, a_{1,l_1}(p_{1,l_1,1}, \dots, p_{1,l_1,pa_{max}}))), \dots, ((t_{n,1}, a_{n,1}(p_{n,1,1}, \dots, p_{n,1,pa_{max}}))), \dots, (t_{n,l_n}, a_{n,l_n}(p_{n,l_n,1}, \dots, p_{n,l_n,pa_{max}})))).$$

Since all elements in this complex vector are numerical values, real numbers in fact, the vector is suitable for the operations a PSS performs during search. But before we go into more detail regarding the learning process using PSS, we first present how an attack agent  $Ag_{attack,i}$  creates out of its strategy sequence its real behavior in  $Env$ .

Given a strategy sequence,  $Ag_{attack,i}$  rounds each  $t_{i,j}$  to the next integer and does the same with each  $a_{i,j}$ . It waits (rounded)  $t_{i,1}$  time units before performing the action that it associates with the number indicated by the rounded value of  $a_{i,1}$  for the situation it is currently in (which, at the beginning of a test run, is  $e_0$ ). Since not all actions are possible in all situations, we order the possible actions for each situation in a deterministic manner, so that we can assign to a natural number a parameterised action (according to its position in the order of actions). If we have a number that is larger than the number of possible parameterised actions, we determine

the action by taking the rounded number from the sequence modulo the number of possible actions (as suggested in [2]).

The action associated with  $a_{i,1}$  will require a number  $pa_{a_{i,1}}$  of parameters, which is smaller or equal to the maximal number of arguments for any actions denoted by  $pa_{max}$ . And these parameter values are provided by the first  $pa_{a_{i,1}}$  elements of  $(p_{i,1,1}, \dots, p_{i,1,pa_{max}})$  (converted to the appropriate types out of the real numbers). The remaining parameter values will be unused (but are necessary, since we can not know how the value representing the action might change during the PSS-based search and how the previous actions might change, which naturally can result in being in different situations). The indicated parameterised action with the appropriate number of arguments is then interpreted and if it is a high-level action it will be transformed into the appropriate sequence of low-level actions. If performing the parameterised action (high- or low-level) took less than  $t_{i,2}$  time units then  $Ag_{attack,i}$  will wait the remaining time units in  $t_{i,2}$  until repeating the described steps for the next parameterised action in the sequence (and so on, until all actions are performed).

As already stated, the just defined position representing strategies for all attack agents allows to directly apply the update operations for a particle defined in the last section. What remains to define to create our complete learning method is how to compare the positions of particles and, since we will be using a multi-objective particle swarm system, how to determine dominance of one position over another. Naturally, the objectives measuring how near the strategies for the attack agents come to the unwanted behavior we are searching for are dependent on the application, which means the set  $A_{tested}$ ,  $Env$ , and the particular unwanted behavior. While [3] presented one generic measure for its approach, we found that for some problems there are several measures that should be taken into account when deciding what strategy is better than another. Even more, some measures only make sense (or even only can be computed) after other measures have reached or went beyond certain values, while for other groups of measures it is not clear at the beginning of a learning run which of the individual measures is more important than others (in finding a strategy that creates the unwanted behavior). See Section IV-A for example measures. As a consequence, we propose the following general structure for comparing two position vectors  $pos_1$  and  $pos_2$ : Our ordering structure has the form

$(\{f_{11}, \dots, f_{1q_1}\}, \dots, \{f_{u1}, \dots, f_{uq_u}\})$ , (or  $(f_1, \dots, f_u)$  for short), where  $f_{ij}$ ,  $1 \leq i \leq u$ ,  $1 \leq j \leq q_i$ , is a quality function assigning an integer to a trace  $e_0, e_1, \dots, e_x$  of environmental states produced by the strategy for the attack agents represented by a position when applied in  $Env$  interacting with the other agents<sup>2</sup>. If  $\triangleright$  denotes the ordering that is created by this ordering structure, then we have

$pos_1 \triangleright pos_2$ , if

<sup>2</sup>We added a second index to an  $f$  to reflect that we now have a vector of sets of objectives, instead of just a vector as has been usual in multi-objective optimization, so far.

$pos_1 \succ_{\vec{f}_1} pos_2$ , or  
 $pos_1 =_{\vec{f}_1} pos_2$  and  $pos_1 \succ_{\vec{f}_2} pos_2$  or  
 ... or  
 $pos_1 =_{\vec{f}_1, \dots, \vec{f}_{u-1}} pos_2$  and  $pos_1 \succ_{\vec{f}_u} pos_2$ .  
 $pos_1 =_{\vec{f}_i} pos_2$  in this context means that  $pos_1$  and  $pos_2$  have an identical quality value in each of the measures  $f_{ij}$  in  $\vec{f}_i$  (and  $=_{\vec{f}_1, \dots, \vec{f}_i}$  is short for  $=_{\vec{f}_1}$  and  $=_{\vec{f}_2}$  and ... and  $=_{\vec{f}_i}$ ). This essentially represents a lexicographical combination of multi-objective domination orderings, which -due to the partiality of the domination orderings- is itself a partial ordering, so that two positions might not be comparable. This is an extension of the lexicographical combination for orderings for PSS for single-objective optimisation presented in [6].

The lexicographical combination allows us to express that certain quality measures need to be achieved before improving other quality measures makes sense (in the search), by putting the certain quality measures before the other measures in our combined ordering. Having each element in the lexicographical ordering as a set of measures, for which the ordering is realized using the domination criterion, is very useful for our purposes because often it makes no difference which agent achieves a certain effect in the interaction with environment and other agents and we can then have an individual measure for such an effect for each agent in the set. Since  $\triangleright$  is a partial ordering (and has domination ordering components) we have to use a PSS for multi-objective optimisation (with  $\triangleright$  being the ordering used to determine domination between positions) and we use the one described in Section II-B for evolving the behavior of our attack agents. The end criterion contains both a resource limit and having achieved the unwanted behavior within this resource limit.

#### IV. TESTING HARBOUR PATROL AND INTERCEPTION POLICIES

Harbour security is becoming more and more a serious issue, especially for harbours that serve both civilian and military vessels. With traffic above water and the possibility of attackers also below water, a total coverage of the whole harbour with the necessary sensors is nearly impossible to achieve (simply due to cost), so that mobile sensor platforms are needed that either alarm interceptor platforms or also serve as interceptor platforms themselves. While detecting potential attackers is the first important task of any harbour security policy, neutralising attackers before they can complete their attack is as important.

##### A. Instantiating our approach

With regard to the general setting described in Section II-A and Figure 1, the multi-agent system we want to test are a fleet of harbour patrol and interception vessels following a patrol and interception policy for a particular harbour. So, each  $Ag_{tested,i}$  represents one vessel, with all its sensor capabilities, and its implementation of the general policy. The elements of  $A_{attack}$  are a group of attack (or intruder) vessels, which in our experiments did not include any below water vessels or assets. The bystander agents of  $A_{byst}$  would

be all other vessels that could navigate a harbour, from ferries, over container ships to pleasure and rowing boats. The environment  $\mathcal{Env}$  that we should be interested in is the real harbour, but naturally it is not possible to evaluate various policies with real vessels in the real harbour setting. Instead, we are using a GIS-based harbour simulation. The GIS (Geographic Information System) provides all the geographic information about the harbour and it also stores the current locations of all agents in the simulation (and their history). The movement of all vessels (agents) is computed in frames of 1/10ths of a second using Euler integration on forces acting on the vessel. These forces are boat drag, throttle and the rudder positions as provided by the vessel.

To instantiate our approach from Section III, we need to define the components of a particle position, i.e. the parameterised actions, how they are transformed into low-level actions (i.e. rudder positions and throttle values) and how we define the ordering  $\triangleright$  on particle positions. For navigating in a harbour, our attack agents need only one parameterised action, namely  $GoToWaypoint(x,y, speed)$ . The values of  $x$  and  $y$  are any real numbers and  $speed$  is between 0.1 and 1 indicating the throttle position. To realize  $GoToWaypoint$  for an agent, we use simple path planning, as first described in [7], with the goal of minimising the travelled distance. This planning results in additional waypoints so that there are no obstacles between a pair of subsequent waypoints allowing to appropriately set the rudder position. If a waypoint is not located over water, then the path planning gets as near to it as possible and then continues planning from there to the next waypoint.

So, since there is only one parameterised action and the timing is already controlled by one of its parameters, we can simplify the general scheme for a position to

$$((x_{1,1}, y_{1,1}, speed_{1,1}), \dots, (x_{1,l_1}, y_{1,l_1}, speed_{1,l_1})), \dots, \\
 ((x_{n,1}, y_{n,1}, speed_{n,1}), \dots, (x_{n,l_n}, y_{n,l_n}, speed_{n,l_n})).$$

To create an initial position vector for a particle we limit the possible values for the coordinates for the waypoints, requiring that each following waypoint in the sequence for an agent is at most 600 meters away from the previous waypoint.

A simulation run for evaluating a position has all vessels starting from their initial positions, with the attack agents starting outside of sensor range of the whole harbour. Every created frame is available as environment state to be evaluated by quality functions. The number of frames depends on the duration of the simulation run. We end a run, if either the attack objective is fulfilled (i.e. the unwanted behavior of the tested agents has occurred), all attack vessels have reached the end of their action sequence, or all attack vessels have been intercepted by the tested agents. In order to intercept an attacker, naturally they first have to be detected and classified as having to be intercepted. For this, the tested agents use sensors. In our proof-of-concept system, each tested agent perceives the environment as a circle around it, with the diameter being a system parameter (and naturally it is possible to hide behind obstacles for a particular sensor). This is a rather crude realization of perception by the tested

vessels and we plan to include better sensor simulations in the future, but for showing the usefulness of our testing approach it is sufficient.

The unwanted behavior of the patrol and interception agents, resp. of the policy that guides them, is the ability of one of the agents in  $A_{attack}$  to reach a spot in the harbour that is supposed to be secure without being intercepted. Such a spot could be the docking slip of a particular ship or a position from which a certain harbour facility could be destroyed. While the following quality functions are aimed at finding attack strategies for reaching one single spot (i.e. a particular coordinate  $(x_{goal}, y_{goal})$ ), they can easily be extended to whole areas or collections of spots. If  $e_0, \dots, e_x$  is the trace produced by the simulator run for a particle position  $pos$ , then

$$f_{intercept}((e_0, \dots, e_x), pos) = \begin{cases} 0, & \text{if there is an } j, \text{ such} \\ & \text{that all } Ag_{attack,i} \text{ are} \\ & \text{intercepted in } e_j \\ 1, & \text{else} \end{cases}$$

with  $pos_1 \succ_{intercept} pos_2$ , if

$$f_{intercept}((\dots), pos_1) > f_{intercept}((\dots), pos_2).$$

$$f_{success}((e_0, \dots, e_x), pos) = \begin{cases} 1, & \text{if there are } j, i, \text{ such} \\ & \text{that } Ag_{attack,i} \text{ reached} \\ & \text{the target spot in } e_j \\ 0, & \text{else} \end{cases}$$

with  $pos_1 \succ_{success} pos_2$ , if

$$f_{success}((\dots), pos_1) > f_{success}((\dots), pos_2).$$

$$f_{dist,i}((e_0, \dots, e_x), pos) = \sum_{j=1}^{\lfloor x/100 \rfloor} dist(e_{100j}, Ag_{attack,i}) + dist(e_x, Ag_{attack,i})$$

where  $dist(e, Ag_{attack,i})$  is the length of the shortest path created from the position of  $Ag_{attack,i}$  in  $e$  to the target spot (again computed using path finding). We define  $pos_1 \succ_{dist,i} pos_2$ , if  $f_{dist,i}((\dots), pos_1) < f_{dist,i}((\dots), pos_2)$ . Then the ordering structure for  $\triangleright$  is

$$\{\{f_{intercept}\}, \{f_{dist,1}, \dots, f_{dist,n}\}, \{f_{success}\}.$$

This means that an attack strategy that has all attackers intercepted is always worse than a strategy that has some attacker “alive” at the end of the simulation (due to the first component in the ordering structure). The second component in the ordering structure on the one hand side makes sure that the search favours positions that have the attack agents move towards the target spot, but since it represents this objective for each individual agent, positions that have attack vessels drawing away the tested agents are not considered bad as long as one attack agent gets towards the target spot. The third component of the ordering structure then favours successful attacks over unsuccessful ones, giving our system the possibility to “optimise” successful attacks more (naturally with regard to the previous components, which means here getting to the target faster).

We also found it useful to introduce an additional possibility to update a particle based on the idea of targeted operators

from [2]. But instead of trying to avoid a bad action as in [2], our targeted update tries to force a good action by an agent if the evaluation of a position shows a particular behavior. More precisely, if the strategies from a particle position lead to a point in the simulation where all but one attack agent are intercepted, then we can update the particle so that the next waypoint for the agent after the waypoint when all other agents are intercepted is changed to the target spot. This reflects the hope that now the way is clear and that this hope should be tried out.

## B. Experimental evaluation

For evaluating our system described in the last subsection and with this also our general testing by learning of behavior approach, we created two high-level patrol and interception policies that can be applied to any harbour and any target spot to protect, instantiated these policies for two particular harbours, namely Halifax Harbour in Nova Scotia and Esquimalt Harbour in Victoria, Vancouver Island, and tried to attack the policies using our testing system. While we naturally would have liked to test real patrol and interception policies for these harbours, for security reasons this was not possible. Security also did not allow to use old policies, even ones with known flaws.

The first patrol and interception policy divides the agents in  $A_{tested}$  into patrollers and interceptors. The general idea is that the patrollers identify potential intruders and alert the interceptors that then approach a potential intruder, identify it and, if necessary, take it out. As long as a potential intruder is in range of the sensors of a patroller, this patroller updates the alarmed interceptor about the course of the potential intruder. If a potential intruder comes close enough to a patroller to be identified, then the patroller will disable it. With the exception of this case, patrollers stay to their predetermined route. The interceptors are stationed at predefined positions in the harbour and only become active when called upon by a patroller. When active, an interceptor determines the best position to come near to a potential intruder, based on the information from the patroller. If the intruder is not found at that position (resp. within sensor range) then the interceptor returns to its standard position. We call this policy *pat-int*.

The second policy, *all-pat*, does not distinguish the elements of  $A_{tested}$ . All patrollers follow a circuit around the harbour and the available vessels are evenly spaced on this circuit. When any patroller detects a potential intruder, the closest available patroller (to the intruder) is sent to identify the boat and if this identification has as result the need to intercept, then this interception takes place.

Both policies naturally have weaknesses. If there are more intruders than agents in  $A_{tested}$ , then the “defenders” can be easily overwhelmed. Therefore we limited the number of agents in  $A_{attack}$  to  $|A_{tested}|$  or less. Also, we did not use any bystander agents (although they would clearly make the task for the agents in  $A_{tested}$  harder, but we wanted to see how good our testing approach was under hard conditions, i.e. no distractions for the tested system and policy). In all our scenarios, the perception radius of an agent in  $A_{tested}$

was 300 meters with 20 meters being the maximal distance for being able to identify a boat as threat or not.

We instantiated both  $all - pat$  and  $pat - int$  for the two harbours we used in our tests. The instantiations of the general policies were hand-coded by us and communication between the tested agents was achieved using the GIS. This means that there were no communication failures possible. For  $pat - int$  and Esquimalt ( $pat - int_{Esq}$ ) this meant having two patrollers and two interceptors. One patroller circles the mouth of the harbour, while the second patroller, the “goatender”, does a small circle very close to the target. The two interceptors have their inactive positions near the dock adjacent to the target spot. The target is placed deep inside the harbour behind a pier (see Figure 2 for the position of the target, which is the same for all our experiments with this harbour). Policy  $pat - int$  for Halifax ( $pat - int_{Hal}$ ) also uses two patrollers and two interceptors, with the same idea for the patrollers, i.e. one circling the mouth of the harbour and one (Patroller 2 in Figure 3) doing its patrol route relatively near to the target. Figure 3.1 shows also the positions for the two interceptors when not active, near the target, which is indicated on the right of the picture (again, this target spot is the same for all our experiments for this harbour).

For policy  $all - pat$  and Esquimalt ( $all - pat_{Esq}$ ) we used 4 patrollers circling around inside the harbour (the four shots in Figure 2 cannot show the whole harbour, therefore we do not see Patroller 4 at all and Patroller 3 only comes into view in the second shot). We also used 4 patrollers for the Halifax scenarios ( $all - pat_{Hal}$ ). As for Esquimalt, Figure 3 only shows part of the harbour, so that the patrollers have to do a rather large circuit on their patrols making them vulnerable for good timed attacks. Figures 2 and 3 do not show the full routes of the vessels (since they would fill up the pictures too much), but provide some idea of recent movement by showing where a vessel comes from using the indicated “tail”.

For each of the four combinations of policy and harbour we did several scenarios with different numbers of attackers. For the  $all - pat$  policy for Esquimalt, we used only 5 actions (waypoints) per agent strategy and the learner used  $l=10$  particles with a maximum of 40 updates. The  $pat - int$  policies and the  $all - pat$  policy for Halifax were more difficult to attack, so that in our tests we used 10 waypoints per agent strategy and  $l=20$  particles in the PSS, with a maximum of 50 updates. The PSS parameters were set to  $W = 0.8$ ,  $C_1 = 0.2$ , and  $C_2 = 0.4$ . The whole system ran on a Pentium 4 machine (2.8 GHz) running Windows XP. A single simulation run (for attack agents with 5 waypoints) took on average 3 minutes to complete, so that a complete run of the system could take days. Since the PSS involves random factors, we repeated each system run at least 3 times.

Our experiments revealed weaknesses in both policies (resp. the concrete instantiations of them as represented by the behavior of the agents in  $A_{tested}$ ). These weaknesses fall in two categories: timing of the (cooperative) attack and use of decoys/sacrifices. We tested  $all - pat_{Esq}$  with one and

two intruders ( $all - pat_{Esq,1}$  and  $all - pat_{Esq,2}$ ). In scenario  $all - pat_{Esq,1}$ , in all our runs our test system evolved a strategy for the intruder that timed its approach to the target spot in such a way that it is only detected in the last few moments before reaching the target and due to this very late detection the nearest patroller is not able to intercept the intruder. In  $all - pat_{Esq,2}$  (see Figure 2), our test system evolved attacks where the first intruder takes out the nearest patroller (Patroller 1 in Figure 2), so that Intruder 2 can pass by and then Intruder 2’s approach is timed in such a manner that it is not detected by Patroller 2 that is moving away from the target (relying on Patroller 1 to cover the area, which does not happen). To test  $all - pat_{Hal}$ , we started with two attackers (i.e. with  $all - pat_{Hal,2}$ ) and were successful, again. In fact, the successful attack had one of the intruders avoid all patrollers by appropriately timed waypoints while the other intruder did not come near to either a patroller or the target. So, essentially this attack is also possible for a single attacker (i.e. for  $all - pat_{Hal,1}$ ), which is why we did not run our system for just one attacker.

Policy  $pat - int$  was supposed to fix the conceptual problems of  $all - pat$ , by focusing the resources more on defending the target spot. We hoped especially that there would be no attacks relying solely on good timing of the approach. But this hope was only partially fulfilled. While our system was not able to learn a successful attack strategy within the given resource limits for a single attacker/intruder for Halifax harbour ( $pat - int_{Hal,1}$ ) where Patroller 2’s sensors can see everything that comes near the target spot, the geography around Esquimalt harbour and the resulting patrol route for the goatender still allowed our system to evolve a timed approach to the target where the attacker/intruder was not detected.

The larger search space created by two intruders resulted in unsuccessful runs for  $pat - int_{Hal,2}$  and  $pat - int_{Esq,2}$  within the given resource limits. But both scenarios with three intruders ( $pat - int_{Hal,3}$  and  $pat - int_{Esq,3}$ ) resulted in attack strategies for the intruders that are successful. For Esquimalt harbour, the two interceptors do their job by intercepting two of the attackers, but this takes them out of the game for the third attacker that is detected by the goatender but passes it beyond identification range and therefore gets by to the target. For Halifax harbour, a successful attack found by our system is depicted in Figure 3. The first two intruders let themselves be spotted by Patroller 1, which draws out both interceptors, so that when Intruder 3 is spotted by the goatending Patroller 2, the nearer Interceptor 1 is nevertheless too far away to get to Intruder 3 before it reaches the target. Note that Intruder 3 passes by Patroller 2 at the farthest distance possible, so that Patroller 2 can not identify it (and therefore does not become an interceptor).

For all scenarios where our testing system found a successful attack it did so in all runs we performed (although usually with a different number of particle updates). The found solutions might differ a little bit, but the basic ideas behind the success were the same.



Fig. 2. The Esquimalt attack for  $all - pat_{Esq,2}$ . Attack target in 1: the red circle near Patroller 2.

The presented experiments show that our testing approach is able to find weaknesses in instantiations of harbour patrol and interception policies and when looking at several scenarios for a particular general policy, the found attack strategies reveal general weaknesses of the policy. Timing of the actions was very important to reveal the weaknesses and allowing for this timing is one of the features of our approach that is new compared to the previous testing approach using learning of cooperative behavior. The experiments also highlight a second important new feature, namely the evaluation of attack team strategies using an ordering structure that allows for decoy/sacrifice strategies, i.e. different roles for different attack agents, without having to build such roles explicitly into the learner. And, as mentioned in the introduction, without being able to use high-level actions it would not have been possible to create the complex behavior that our attack agents showed in the successful attacks.

## V. RELATED WORK

Support for testing multi-agent systems for emergent misbehavior is an area that is not very well researched. In addition to [3], which is the application of the basic idea from [2] to a multi-agent system, [8] comes nearest to finding interactions of multiple outside events with a system creating unwanted behavior, namely creating infeasible schedules in

the model of a scheduler (not the real scheduler). Obviously, in our work, we are also working with a model/simulation, since trying out all strategies created by our learner is simply not possible in the real world. But we are learning action sequences for several attack agents, not a global sequence of events as in [8] and we allow for high-level actions and use of different learning mechanism.

There are a lot of applications of evolutionary methods to military and security problems. We are aware of two projects that involve aspects of path planning of relevance to our project. In [9], evolutionary methods are used to produce a configuration of an aircraft for a mission. The mission is then executed using mostly a conventional path planner, which is similar to how we create the low-level waypoints. [10] presents an online learning system based on ant systems that tries to learn how to pass by defenders to reach a mission target. Naturally, online learning has the risk of bad early decisions that can make it impossible for the system to solve the given problem despite the fact that a solution exists. In both papers, the defenders are stationary.

## VI. CONCLUSION AND FUTURE WORK

We presented a new approach for testing multi-agent systems and policies for groups of agents for unwanted behavior based on evolutionary learning of cooperative behavior. Our

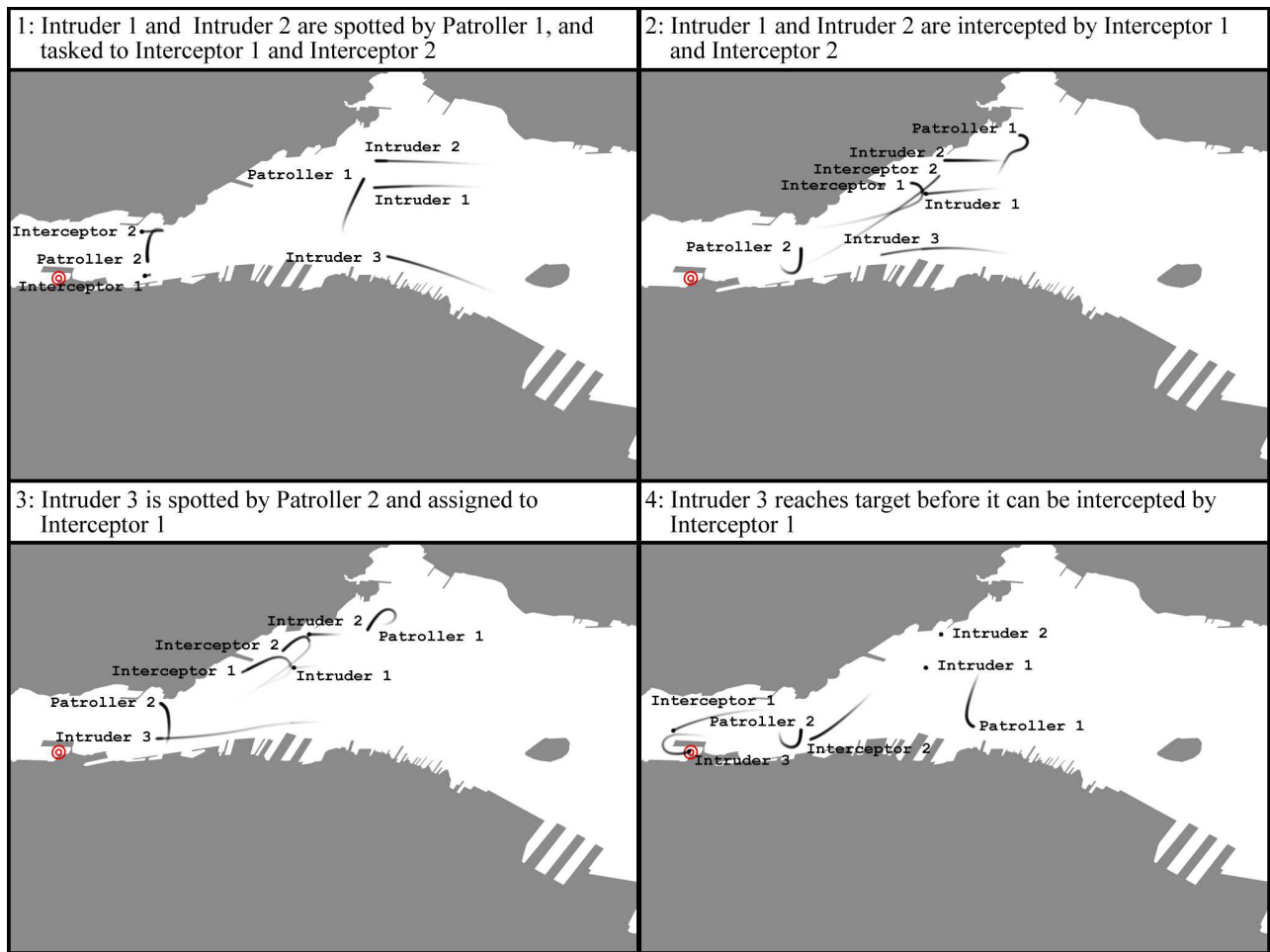


Fig. 3. The Halifax attack for  $pat - int_{Hal,3}$ . Attack target red circle on left.

new approach allows for the use of parameterised actions and combinations of different evaluation functions in all kinds of mixtures of hierarchical and multi-objective fashions, thus giving a test system based on our approach the ability to create well-timed interactions with the tested system, to incorporate existing planning algorithms for creating low-level action sequences for high-level actions, and to use many possible objectives concurrently to drive the search. Our experiments with the instantiation of the general concept for testing harbour patrol and interception policies showed that our approach is indeed able to find unwanted behavior by the policies.

In future work, we plan to compare different ordering structures to explore more the possibilities offered by this concept. Applying the general approach to different testing tasks is also of interest. While for the harbour application the comparison of our approach with the previous approach clearly favours the new approach (since we were not able to achieve anything with the previous approach), for other applications for testing for unwanted behavior the previous approach might be better suited. Developing guidelines when to use which approach therefore becomes necessary.

## REFERENCES

- [1] D.B. Lenat and J.S. Brown: Why AM and Eurisko appear to work, *Artificial Intelligence* 23(3), 1984, pp. 269–294.
- [2] 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.
- [3] J. Kidney and J. Denzinger: Testing the limits of emergent behavior in MAS using learning of cooperative behavior, *Proc. ECAI 2006*, Riva del Garda, 2006, pp. 260–264.
- [4] J. Kennedy and R.C. Eberhart: Particle swarm optimization, *Proc. IEEE ICNN 1995*, Piscataway, 1995, pp. 1942–1948.
- [5] M. Reyes-Sierra and C.A. Coello Coello: Multi-Objective Particle Swarm Optimizers: A Survey of the State-of-the-Art, *Int. Jour. Comp. Int. Res.* 2(3), 2006, pp. 287–308.
- [6] T.E. Mora, A.B. Sesay, J. Denzinger, H. Golshan, G. Poissant, and C. Konecnik: Fuel Optimization using biologically-inspired Computational Models, *Proc. IPC 2008*, Calgary, 2008 (on CD).
- [7] P.E. Hart, N.J. Nilsson, and B. Raphael: A Formal Basis for the Heuristic Determination of Minimum Cost Paths, *IEEE Trans. Systems Science and Cybernetics* 4(2), 1968, pp. 100–107.
- [8] L. Briand, Y. Labiche and M. Shousha: Using Genetic Algorithms for Early Schedulability Analysis and Stress Testing in Real-Time Systems, *Genetic Programming and Evolvable Machines* 7(2), 2006, pp. 145–170.
- [9] C. Miles and S.J. Louis: Case-Injection Improves Response Time for a Real-Time Strategy Game, *Proc CIG-05*, Colchester, 2005, pp. 149–156.
- [10] J.A. Sauter, R. Matthews, H. Van Dyke Parunak, and S. Brueckner: Evolving adaptive pheromone path planning mechanisms, *Proc. AAMAS-02*, Bologna, 2002, pp. 434–440.