Exploratory testing for unwanted behavior using evolutionary learning techniques

Jörg Denzinger
Department of Computer Science
University of Calgary, Calgary, Canada
denzinge@cpsc.ucalgary.ca

Technical Report 2007-868-20

June 8, 2007

Abstract

We present a method that allows to create tools that help human testers to do exploratory testing for unwanted behavior of software systems and their components. The general idea is to use evolutionary learning to find interaction schemes with the tested system that result in the tested system showing the unwanted behavior or coming near to it. We present two case studies that instantiate our method and that resulted in detecting not previously known unwanted behavior in both tested systems.

1 Introduction

Testing systems for unwanted emergent behavior, or emergent misbehavior (see [Mo05]) is among the most difficult problems that software testing, in fact system testing in general, faces. At the core of this difficulty is that the starting point for testing for emergent misbehavior is just a vague idea what we do not want to see to happen and then the task is to essentially guess a system configuration of the system to test and interactions with this system that produce the result that the tester hoped not to see.

There are many documented cases of emergent misbehavior in all kinds of distributed systems. For example, [FJ93] reports that in a network with "many apparently-independent periodic processes ... these processes can inadvertently become synchronized" despite the efforts of routers that periodically exchange routing protocol messages. This synchronization effect is abrupt and therefore hard to anticipate. [Mo05] presents several additional cases of emergent misbehavior in and outside of Computer Science. It should be pointed out that currently there are several initiatives in Computer Science that will lead to an increased potential for emergent misbehavior, because their core ideas of self-modification and emergence of system properties aim for achieving positive emergent behavior. Examples for such initiatives are autonomic computing (see [KC03]) and service-oriented computing (see [SH05]) that will require testing methodologies that target the avoidance of certain conditions in a system instead of just testing for compliance of given interaction sequences with the system requirements.

The current "solution" for testing for unwanted system behavior in literature and in industry is to use *exploratory testing* (see [KBP01]). Exploratory testing has as key idea "simultaneous learning, test design and test execution" (see [Bach]) by the tester. Naturally, with such a heavy reliance on the human tester, we have several potential drawbacks of exploratory testing that center around the human element. The ability of a human being to learn something is difficult to predict. It is highly dependent on the area that the learning takes place in and also on the situation of the individual (humans can have bad days that result in quite different learning results compared to a "normal day"). If we additionally want to have human intuition to work for us, then predicting outcomes is even worse. There are guidelines and case studies available that help a human tester in being more predictable when doing exploratory testing, but it is still treated more like an art than as something that everyone with a certain qualification will do the same way (and with the same results). Now, exploratory testing is not the first concept in software engineering that has these problems and there is one way that -while not totally solving the problem- allows for more predictability of results: the development and use of tools that help with the whole process in question.

So far, tool support for exploratory testing has not gone beyond simple bookkeeping that collects the interactions a tester has with the tested system (see, for example, [TeEx]). In this paper, we present a general concept for producing tool support for exploratory testing that concentrates on helping the human tester with finding unwanted behavior of the system under test. Our concept makes use of so-called evolutionary algorithms (see, for example, [Go89]) that use principles from biology to produce systems with the capability to learn that also exhibit behaviors that have some similarities to what human intuition achieves.

More precisely, our approach starts by creating a set of randomly generated interaction sequences with the system that is tested (similar as suggested in, for example, [Ha96]) and then evaluates how near these sequences come to bringing the system to show the particular unwanted behavior we are looking for. Then we create out of the best sequences new sequences that replace the bad old ones, using so-called genetic operators, evaluate them and continue this cycle until we either find the unwanted behavior or run out of the resource limit for this test. Usually, this results in behaviors that come closer and closer to forcing the tested system into showing the unwanted behavior (if the system can show this behavior at all) and even if our approach does not lead to getting the system fully where we want it to go, observing the best sequences that the system comes up with allows the tester to get a good idea what is possible and to come up with additional different tests. While the use of evolutionary algorithms has been suggested for other kinds of testing, our approach targets especially exploratory testing, see Section 5.

We have used our general concept in two case studies, namely testing a commercial computer game in the FIFA series by Electronic Arts and testing multi-agent systems produced by students as assignment in a basic multi-agent systems class. For FIFA, our tool was able to detect many interactions with the game (by a potential user) that resulted in "stupid" behavior of the game, more precisely unnecessary fouls leading to penalty kicks. When testing the multi-agent systems of students, our tool found behaviors for a group of agents interacting with the students' agents that made the students' agents freeze in place, a behavior that we did not observe in our usual tests for these systems. This shows that tools based on our concept are able to produce some of the intended results of exploratory testing on their own, thus greatly helping with testing of complex systems.

This paper is organized as follows: After this introduction, in Section 2 we present our basic view of how a testing environment for unwanted behavior of a system should look like and we introduce basic notations that are used to represent the behaviors of components of a complex system and their interactions. In Section 3, we present our evolutionary learning method that we use to evolve interactions with a tested system that force this system into showing an unwanted behavior. Section 4 presents the two case studies and our results. After presenting related work in Section 5, in Section 6 we conclude with a discussion of our results and possible future work.

2 Our basic view of a testing environment for unwanted behavior

As stated in the last section, we aim at providing automated support for a human tester in finding some unwanted behavior of a system, a system component or a group of system components. And naturally, the behavior of these system components takes place in some kind of environment. Formally, this means that we look at some system to be tested $\mathcal{S}ys_{tested}$ = ($Comp_{tested}$, $\mathcal{E}nv$), where $\mathcal{E}nv$ is the environment $\mathcal{S}ys_{tested}$ is acting in and $Comp_{tested}$ = $\{C_{tested,1}$,...,$C_{tested,m}\}$ is the set of system components that we want to show the unwanted behavior. The behavior of a system or (set of) component(s) depends on the system (components) itself

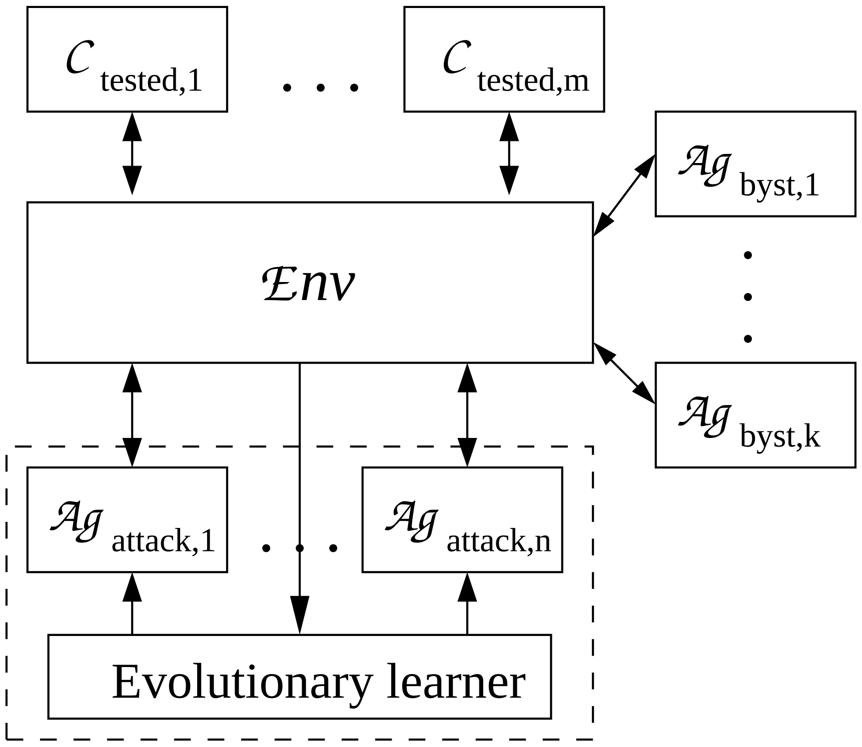

Figure 1: General setting of our approach

and the interactions that the system has within the environment with other systems, components or human users. In this paper, we will use the term *agent* to refer to these three types of entities that interact with $\mathcal{S}ys_{tested}$.

While in theory a tester would like to play the role of all of the agents $\mathcal{S}ys_{tested}$ is interacting with, in reality this might not always be feasible due to the complexity of a particular agent and the available test budget. Therefore we distinguish the agents a system $\mathcal{S}ys_{tested}$ interacts with into a set A_{attack} = $\{\mathcal{A}g_{attack,1}$,..., $\mathcal{A}g_{attack,n}\}$ of agents that the tester can use to attack the system to show the unwanted behavior and a set A_{byst} = $\{\mathcal{A}g_{byst,1}$,...,$\mathcal{A}g_{byst,k}\}$ of bystander agents that interact with $\mathcal{S}ys_{tested}$ and A_{attack} but that are not under control of the tester (and are not part of $\mathcal{S}ys_{tested}$).

The possible (inter)actions of an agent $\mathcal{A}g_{attack,i}$ in A_{attack} with the environment, the other agents and $\mathcal{S}ys_{tested}$ form the set $Act_{attack,i}$. Then a particular test run for $\mathcal{S}ys_{tested}$ can be abstractly described as a sequence of timed actions for each of the the agents $\mathcal{A}g_{attack,i}$ in A_{attack} with $\mathcal{S}ys_{tested}$, $(t_{i,1}$,$a_{i,1})$,...,$(t_{i,l_i}$,$a_{i,l_i})$

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,...,e_x (a *trace*) of what we call enhanced environment states. An enhanced environment state is essentially a view on the state of the environment together with a selection of information from the states of all $C_{tested,i}$ in $Comp_{tested}$. We are allowing for only a view of the complete states of the components and the environment, because this makes evaluating a sequence of enhanced environment states quicker and can require less work in getting state information out of the components. It also allows for adjusting a tool based on our method from Section 3 to see what it can come up with if only certain information is available (for example, if we treat the components as black boxes), which can be important for security testing of a system.

Note that for systems that interact with agents in a synchronous manner (as will be the case in our case studies), we can get rid of the $t_{i,j}$ if we introduce a no-op action (the agents simply interact with $\mathcal{S}ys_{tested}$ every time step). We then also have that all the sequences are of the same length, i.e. $l_i = l_j$ for all i, j and $x = l_i$. We should also point out that doing two test runs using the same action sequences for all agents in A_{attack} might not always result in the same traces. There can be several reasons for this: there might be random steps in $\mathcal{S}ys_{tested}$ or the bystander agents might be in different states at the start of the runs or there might be random events happening in the environment.

3 Our evolutionary learning method

In order to partially automate what an exploratory tester would do when testing $\mathcal{S}ys_{tested}$, we have to look a little bit closer at what it is that an exploratory tester does. Obviously, the tester will start with a particular test goal and will have a certain budget for this particular goal. Then he or she will interact with $\mathcal{S}ys_{tested}$ playing the role of some or all agents in A_{attack} and will look at the traces e_0,e_1,...,e_x that are generated. If no problems are detected, then the tester will use his/her knowledge to analyze the traces and to see if there are things that hint at some unwanted behavior and if there is something then (s)he will again use knowledge or expertise to come up with other interactions with $\mathcal{S}ys_{tested}$ that provide more hints. And this is repeated until either the tester found an interaction sequence with $\mathcal{S}ys_{tested}$ producing the unwanted behavior or the test budget is spent.

Evolutionary algorithms work very similar to what we just sketched in the last paragraph. They are created to fulfill a particular goal, usually are given a certain amount of computing time to fulfill the goal or come as near as possible, and they start by creating some random solution candidates for the given goal. These solution candidates are evaluated with regard to the goal and those candidates that hint at being nearer to solving the goal are used to create new solution candidates, using knowledge about the particular application (and some additional randomness). And this evaluation/creation of new candidates is repeated until either a solution is found or the time is over. Essentially, an evolutionary algorithm kind of tries to learn what the solution is based on experiences and using something that resembles human intuition. The general setting for our approach, using the basic view from the last section, is depicted in Figure 1.

To be more precise, the first step in using our approach is for the human tester to define a test goal that can be evaluated by looking at a trace e_0,e_1,...,e_x as defined in the last section. As a first stage, this goal should be a predicate, let's call it $\mathcal{G}_{test}$, that is true if a trace meets the goal, i.e. shows the particular unwanted behavior the tester wants to test for, and false otherwise. Evolutionary algorithms work on potential solutions for the goal, which means that we have to define how such a solution, which usually is called an *individual*, should look like. In order to evaluate traces, we have to produce them, which means that a potential solution should include the interactions of the attack agents with $\mathcal{S}ys_{tested}$. Additionally, we might want to choose a particular initialization of $\mathcal{S}ys_{tested}$, if this system allows for such initializations, which essentially means providing $\mathcal{S}ys_{tested}$ with values for all the parameters that allow different initializations. So, in general, using our notation from Section 2, an individual has the form

(*init*(p_1,...,p_q),(($t_{1,1}$,$a_{1,1}$),...,(t_{1,l_1},a_{1,l_1})),...,(($t_{n,1}$,$a_{n,1}$),...,(t_{n,l_n},a_{n,l_n}))),

where *init* is the initialization action for $\mathcal{S}ys_{tested}$ and p_1,...,p_q are values for the parameters of *init*.

As stated before, the first step of the evolutionary learner is to create several random individuals, which is called the *initial population* or *first generation*. In order to do this, an $a_{i,j}$ is randomly selected out of $Act_{attack,i}$ and a p_i or a $t_{i,j}$ is randomly created out of the possible parameter values for the particular parameter or the set of reasonable time units which is pre-defined by the tester.

The next step of an evolutionary algorithm is to evaluate all the individuals in the population using a so-called *fitness function*. Naturally, such a fitness function is application dependent, but for our method we have a general idea how to create a fitness function for a particular test goal (although we still have a test goal dependent part in it). As already stated, our fitness function evaluates traces e_0,e_1,...,e_x created by an individual. In fact, due to the fact that the same individual can create different traces each time it is run, we will do a certain number of trial runs and sum up the fitness measure $single\_fit$ of each produced trace and use this sum as our overall fitness for the individual.

The fitness of an individual should reflect how near a trace for this individual comes to fulfilling our test predicate $\mathcal{G}_{test}$ and if $\mathcal{G}_{test}$ is fulfilled, then we want it to be fulfilled as early in the trace as possible (to make it easier to see the results of any changes to $\mathcal{S}ys_{tested}$ that will be made to fix the problem later). This leads to the following general scheme for $single\_fit$:

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

In this scheme, $near\_goal$ is intended to measure how near its argument trace comes to fulfilling $\mathcal{G}_{test}$ and the smaller its value is, the nearer the trace came to the goal. We will provide two examples for $near\_goal$ functions for rather different applications in Section 4.

The next step of an evolutionary algorithm is to create new individuals out of the individuals of the last generation. To do this, we need so-called *genetic operators* that build a new individual out of old ones (the *parent individuals*) and we need to select parent individuals for each application of a genetic operator. For both tasks, there are many different methods described in the evolutionary algorithms literature and how well such a method works is usually dependent on the particular application. In the following, we will describe the methods we used in our case studies.

Given the general setting we work in, there are 3 levels on which genetic operators can work: the team level, the individual agent level and the parameter level. On the team level, the idea of operators is to combine the action sequences of agents from the parents or to add a totally new action sequence for an agent. A *team level crossover* takes two individuals

(*init*$(p_1,...,p_q)$,$((t_{1,1},a_{1,1}),...,(t_{1,l_1},a_{1,l_1})),...,((t_{n,1},a_{n,1}),...,(t_{n,l_n},a_{n,l_n})))$
and
(*init*$(p'_1,...,p'_q)$,$((t'_{1,1},a'_{1,1}),...,(t'_{1,l_1},a'_{1,l_1})),...,((t'_{n,1},a'_{n,1}),...,(t'_{n,l_n},a'_{n,l_n})))$
and creates an individual
(*init*$(p"_1,...,p"_q)$,$((t"_{1,1},a"_{1,1}),...,(t"_{1,l_1},a"_{1,l_1})),...,((t"_{n,1},a"_{n,1}),...,$
$(t"_{n,l_n},a"_{n,l_n})))$,
where $((t"_{i,1},a"_{i,1}),..., (t"_{i,l_i},a"_{i,l_i}))$ is either $((t_{i,1},a_{i,1}),..., (t_{i,l_i},a_{i,l_i}))$ or $((t'_{i,1},a'_{i,1}),$ $...,(t'_{i,l_i},a'_{i,l_i}))$ and $(p"_1,...,p"_q)$ is either $(p_1,...,p_q)$ or $(p'_1,...,p'_q)$ and which one is decided randomly. A *team level mutation* takes one individual
(*init*$(p_1,...,p_q)$, $((t_{1,1},a_{1,1}),...,(t_{1,l_1},a_{1,l_1})),...,((t_{n,1},a_{n,1}),...,(t_{n,l_n},a_{n,l_n})))$
and creates the new individual
(*init*$(p_1,...,p_q)$,$((t_{1,1},a_{1,1}), ...,(t_{1,l_1},a_{1,l_1})),...,((t'_{i,1},a'_{i,1}),...,(t'_{i,l_i},a'_{i,l_i})),...,$
$((t_{n,1},a_{n,1}),...,(t_{n,l_n},a_{n,l_n})))$
where $((t'_{i,1},a'_{i,1}),...,(t'_{i,l_i},a'_{i,l_i}))$ is a randomly created action sequence for agent $\mathcal{A}g_{attack,i}$.

On the agent level, we have two variants of crossover and mutation that differ in how they select the place where the operation works. The idea of a crossover on the agent level is to cut two individuals (resp. the action sequences for all agents in the 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, we 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

$(\textit{init}(p_1,...,p_q),((t_{1,1},a_{1,1}),...,(t_{1,l_1},a_{1,l_1})),...,((t_{n,1},a_{n,1}),...,(t_{n,l_n},a_{n,l_n})))$

and

$(\textit{init}(p'_1,...,p'_q),((t'_{1,1},a'_{1,1}),...,(t'_{1,l_1},a'_{1,l_1})),...,((t'_{n,1},a'_{n,1}),...,(t'_{n,l_n},a'_{n,l_n})))$,

it selects a sequence position j randomly and then creates as new individual

$(\textit{init}(p_1,...,p_q),((t_{1,1},a_{1,1}),...,(t_{1,j},a_{1,j}),(t'_{1,j+1},a'_{1,j+1}),...,(t'_{1,l_1},a'_{1,l_1})),...,$
$((t_{n,1},a_{n,1}),...,(t_{n,j},a_{n,j}),(t'_{n,j+1},a'_{n,j+1}),...,(t'_{n,l_n},a'_{n,l_n})))$.

Standard mutation takes an individual

$(\textit{init}(p_1,...,p_q),((t_{1,1},a_{1,1}),...,(t_{1,l_1},a_{1,l_1})),...,((t_{n,1},a_{n,1}),...,(t_{n,l_n},a_{n,l_n})))$,

selects a sequence position j randomly and creates the new individual

$(\textit{init}(p_1,...,p_q),((t_{1,1},a_{1,1}),...,(t_{1,j-1},a_{1,j-1}),(t'_{1,j},a'_{1,j}),(t_{1,j+1},a_{1,j+1}),...,(t_{1,l_1},a_{1,l_1})),$
$...,((t_{n,1},a_{n,1}),...,(t_{n,j-1},a_{n,j-1}),(t'_{n,j},a'_{n,j}),(t_{n,j+1},a_{n,j+1}),...,t_{n,l_n},a_{n,l_n})))$,

where $a'_{i,j} \in Act_{attack,i}$.

In [Ch+04], we introduced a variant of these standard operators that tries to not use random positions in the sequences but to target positions at which the traces produced by an individual show that $\mathcal{S}ys_{tested}$ is getting away from fulfilling $\mathcal{G}_{test}$. The way we measure the fitness of an individual this means that we see a big increase of the $near\_goal$ value of a trace after we take into account the state produced by an action by one agent. More formally, we select the position from the last paragraph by using the smallest position j in all traces of an individual such that

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

where too_much_lost is a parameter chosen by the human tester. We call the resulting operators *targeted crossover* and *targeted mutation*.

On the parameter level, we concern ourselves with the parameter list of the *init*-action. *Parameter crossover* takes two individuals

$(\textit{init}(p_1,...,p_q),\ ((t_{1,1},a_{1,1}),...,(t_{1,l_1},a_{1,l_1})),...,((t_{n,1},a_{n,1}),...,(t_{n,l_n},a_{n,l_n})))$

and

$(\textit{init}(p'_1,...,p'_q),((t'_{1,1},a'_{1,1}),...,(t'_{1,l_1},a'_{1,l_1})),...,((t'_{n,1},a'_{n,1}),...,(t'_{n,l_n},a'_{n,l_n})))$,

selects a parameter position j randomly and creates the new individual

$(\textit{init}(p_1,...,p_j,p'_{j+1},...,p'_q),((t_{1,1},a_{1,1}),...,(t_{1,l_1},a_{1,l_1})),...,((t_{n,1},a_{n,1}),...,$
$(t_{n,l_n},a_{n,l_n})))$.

And, similar to the other mutation operators, *parameter mutation* takes one individual

$(\textit{init}(p_1,...,p_q),((t_{1,1},a_{1,1}),...,(t_{1,l_1},a_{1,l_1})),...,((t_{n,1},a_{n,1}),...,(t_{n,l_n},a_{n,l_n})))$, a random position j and creates

$$(\mathit{init}(p_1,...,p_{j-1},p'_j,p_{j+1},...,p_q),((t_{1,1},a_{1,1}),...,(t_{1,l_1},a_{1,l_1})),...,((t_{n,1},a_{n,1}),...,(t_{n,l_n},a_{n,l_n}))),$$

where p'_j is a new value out of the possible values for the j-th parameter.

Many of the operators from above are not symmetrical, i.e. it matters which individual is the first parent and which the second. The selection of parents has to take this into account, which means that every time the first parent for an operator application is selected, we need to make sure that the selection method we use provides the preferences that we want it to have. A key point for this step in evolutionary algorithms is to find the right balance between exploiting knowledge we have about the application, which usually is represented by the fitness values of individuals, and exploring possibilities by introducing randomness into the selection. A selection scheme that leans a little bit more towards using fitness, but provides sufficient randomness (at least for the applications we are interested in) is the so-called roulette wheel selection. Essentially, this method selects a parent at random, but the probability for selecting a particular individual is proportional to its fitness (to visualize a way how this can be done, imagine a roulette wheel where the numbers represent the individuals and the sections for the numbers are of different size proportional to how much better on individual is than another). Since in our case a good individual has a small fitness value, we use for determining the probability for its selection the inverse of the fitness value.

The fitness value of individuals also plays a key role in deciding how the next generation, i.e. the starting point for the next iteration of the algorithm, looks like. While all newly created individuals are part of this next generation, we also want some individuals from the current generation to survive. And these parent individuals are the individuals with the best (in our case this is the lowest) fitness values.

4 Case studies

In order to evaluate our approach from the last section, we have performed two case studies that look at rather different systems to test. Both case studies have in common that the interactions between the tested system and its user(s) are synchronous (in the case of ARES obviously so, in the case of FIFA-99 due to the set-up of the communication) which makes individuals a little bit less complex than the general form from the last section. Both case studies show that our approach is able to produce behaviors for the attack teams that reveal unwanted behavior in the tested system.

4.1 Case study 1: FIFA-99

In this first case study, we use our method to test the commercial computer game FIFA'99 by Electronic Arts, a soccer game, for unwanted behavior of the AI component responsible for directing the defensive behavior of the computer player. To do this, we evolve a single attack agent when initializing the game for a cornerkick, having this agent playing the team in the offense (see also [De+05]). Both the AI component and the game environment use some randomness, so that this case study shows that our approach can deal with randomness.

4.1.1 FIFA-99

The FIFA series of games by Electronic Arts is a typical example for the genre of team sports games that are produced by many companies and that have a rather broad fan base. The FIFA series is about soccer and FIFA-99 was enhanced by Electronic Arts with an AI programming interface allowing outside programs to take over the role of human game players. This enhanced version was made available to several universities.

While soccer is about two teams of eleven soccer players that square off against each other trying to kick a ball into the opponent team's goal when on the offensive and trying to avoid such a goal when on the defensive, in the computer game, a single human player is responsible not just for one player but a whole team. This is accomplished by having this human player controlling only one of the players at any point in time while the rest of the players are following behaviors created by the game developers. But among the possible actions a human player can perform is taking over the control of another of the players on its team. The other actions that the human player can have its controlled soccer player do are moving up, down, left and right (given the view of the player of the field), turning the players orientation up, down, left and right, passing the ball (to the next player given the current orientation of the player), shooting the ball on goal and doing nothing (the no-op operation mentioned earlier). And when on the defensive, a player can also foul the player with the ball (if the player is near enough).

There are also special situations in the game in which special actions have to be performed, like a cornerkick, a free kick, a penalty kick or a throw in. Essentially, these situations follow interruptions of the game and therefore define a new setting for a test. And the special action requires the human player to enter some parameters or parameter combinations to create a situation after which only the standard actions mentioned above can be used.

Testing of games like FIFA-99 relies heavily on using human testers with different gaming experience. While a lot of the tests follow interaction scripts that the human gamer employs and where the expected game behavior is clearly described, the testing for unwanted behavior is essentially having experienced gamers play the game (without a particular script), which has to be considered a form of exploratory testing. Since there are very tight deadlines for commercial computer games, there are also tight constraints on the amount of testing that can be done. And one problem that can limit the success of a game (resp. the profit out of this game) are so-called *sweet spots* of a game. A sweet spot is a way to win the game that is easier than intended by the game designers and that the game designers would call an unwanted behavior of the game. Knowing a sweet spot makes the game less interesting for the human player and too many sweet spots will seriously hamper sales of the game (since information about such sweet spots gets around very quickly). While games have sweet spots that the users will not detect as sweet spots (since the way a user wins the game seems to be difficult enough and within the expectations of the user, so that only the designers know that this was not so intended), the worst sweet spots are those that obviously hint at bad game design.

For a game like FIFA-99, sweet spots that are due to unrealistic game behavior have to be considered such bad sweet spots. Our particular testing goal for FIFA-99 was to find ways to consistently score against a team controlled by the game AI out of a cornerkick situation, where we also counted being awarded a penalty kick as scoring. Since FIFA-99 makes very frequent use of a random number generator to determine the outcome of actions taken by the user (and the game AI), scoring consistently with the same sequence of user actions is already rather unrealistic (and obviously also something that does not happen in the real world, since humans might make the same mistake twice but not 10 times). But additionally it is up to the human tester (together with the game designers) to take the behaviors that our tool found and to determine if the game should be changed or if the particular action sequence should be considered from now on a wanted behavior (remember that in computer games it is not the goal to make the game impossible to win).

4.1.2 Instantiating our method

Our tool to test for unrealistic goal scoring evolves an action sequence for a single attack agent $\mathcal{A}g_{attack,1}$. The component $C_{tested,1}$ we test is the so-called AI component of the game that controls the defending team and the environment $\mathcal{E}nv$ is the part of the game that represents the playing field and the simulation of the soccer game (essentially, $\mathcal{E}nv$ is the game without $C_{tested,1}$). The actions in $Act_{attack,1}$ are the actions given in the last subsection. The *init* action of our approach for the test goal we report on, namely unrealistic goal scoring after a cornerkick, has 3 parameters X, Z, and $angle$ that define how a cornerkick is performed (X ranges between 0 and the width of the field, Z between 0 and the length of the field and

$angle$ is the angle to the field plane that the kick is aimed between 0 and 90 degrees). As already stated, $\mathcal{A}g_{attack,1}$ performs its actions in fixed time intervals, so that we do not need the $t_{i,j}$s in an individual.

Enhanced environment states consist of the positions of all players and the ball in the playing field. For evaluating the fitness of an individual, we perform 10 runs with it. The condition $\mathcal{G}_{test}$ is true for a trace e_0,...,e_j, if the ball is in the opponent's goal in state e_i, $1 \leq i \leq j$ or if the attacking team is awarded a penalty kick in one of these states. The key decision for a human tester (or a group of testers) using our approach is how to define the function $near\_goal$. We have chosen to compute $near\_goal$ the following way. We divided the playing field into four zones:

Zone 1 : from the opponent goal to the penalty box

Zone 2 : 1/3 of the field length from the opponent goal

Zone 3 : the half of the field with the opponent goal in it

Zone 4 : the whole field

and associate with each zone a penalty value (pen$_1$ to pen$_4$, the smaller the zone, the smaller the penalty). We then take the distance $dist(e_i)$ of the player with the ball and multiply it with the zone penalty of the zone the player is in. This produces

$$near\_goal((e_0,...,e_i)) = \begin{cases} dist(s_i) \times \text{penalty}, \\ \quad \text{if the own players had the ball in} \\ \quad e_{i-1} \text{ or } e_i \\ max\_penalty \\ \quad \text{else.} \end{cases}$$

The parameter $max\_penalty$ is chosen to be greater than the maximal possible distance of a player with the ball from the opponent's goal multiplied by the maximal zone penalty, so that losing the ball to the opponent results in large $near\_goal$-values and a very bad fitness.

Obviously, there is no team level needed, so that we employed the two pairs of operators of the agent level and the pair on the parameter level. In fact, since a bad parameter combination leads immediately to a loss of the ball, we added an operator that combined a successful *init*-parameter combination with all of the agent action sequences of the current generation (see [De+05]). We also favored targeted operators over the standard version 5 to 1. For the targeted operators, we set too_much_lost to $max\_penalty$, which results in targeting positions in the action sequence that lead to loosing the ball.

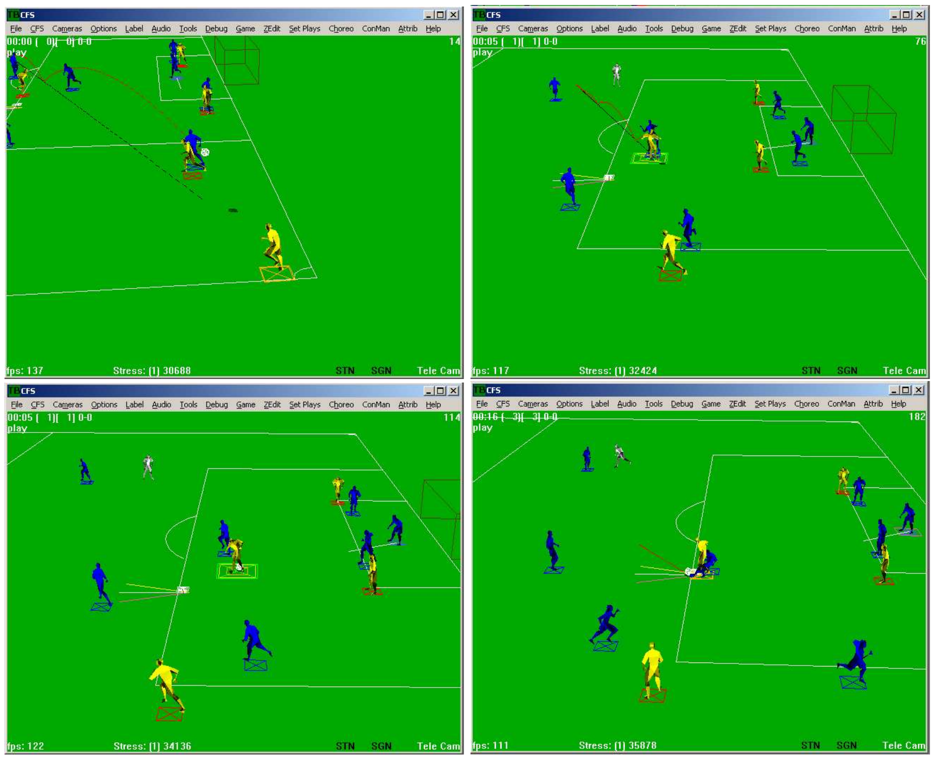

Figure 2: An unwanted behavior in FIFA-99

4.1.3 Some results

Scoring from a cornerkick was not the only test goal we evaluated (see [Ch+04] for results for scoring starting with a kickoff), but it provides an example with an *init*-action and created interesting and easy to describe unwanted behaviors. For each individual, we performed 100 evaluation runs and each generation consisted of 200 individuals (with 20 different *init*-parameter settings among them). The length of an action sequence was 20.

In our experiments, every run of our tool quickly produced one or several action sequences that resulted in scoring goals in most of the evaluation runs (we decided to report only on individuals that scored in 80 of the 100 evaluation runs). In fact, we usually had at least one individual that indeed scored a goal and several individuals that produced penalty kicks for the attacking team. While many of these individuals have as the only problem that they are repeatable so often but do not represent an un-soccer-like behavior, we also found quite a few individuals that showed off a rather “dumb” AI component. Figure 2 represents some screenshots of such a “dumb” behavior. The full individual used to create this behavior is

- *init*(-1954,-775,28.1), NOOP, DOWN, DOWN, RIGHT, SWITCH, MOVE-UPLEFT, MOVE DOWNRIGHT, LEFT, LEFT, SWITCH

The picture in the upper left corner of Figure 2 shows the ball immediately after the *init*-action is transmitted to FIFA-99. In the upper right corner we see the attacker trying to get the ball under control. The attacker is struggling through several actions to get control of the ball which leads to the situation in the picture in the lower left corner of Figure 2. The struggle for control goes on until the attacker nearly has left the penalty box, but this is the moment when the AI decides to foul the attacker, as can be seen in the lower right picture. This then results in a penalty shot and represents really a very stupid behavior of the game AI, especially since it is repeatable.

If we look at the time requirements of our tool (see also [De+05]), then we usually find the first scoring individuals after between 3 to 5 minutes. Since our general scheme for fitness functions favors shorter traces showing the unwanted behavior, it usually is a good idea to continue a run of the tool a little bit more, since it often is then able to find shorter sequences that lead to the unwanted behavior.

4.2 Case study 2: Student teams in ARES

In this second case study, we use our method to evaluate a multi-agent system that acts within ARES, an abstract simulation of a city after being struck by an earthquake. The tested system was written by students of a basic multi-agent systems class and the students’ agents have to locate and rescue survivors of the earthquake. We evolve a team of attack agents that also act in ARES, together with the students’ team, and we want to test how well the students’ team can deal with a team that tries to let them look bad (see also [DK06]).

4.2.1 ARES

ARES and ARES II (Agent Rescue Emergency Simulator, see [DK05]) are testbeds for evaluating multi-agent systems, with ARES II being the newest version of this system. While these testbeds can also be used to evaluate new multi-agent concepts, their main purpose is to provide an environment for students of a basic multi-agent systems class to create multi-agent systems for –as the main assignment of this class. The general setting of ARES is a city that has been struck by an earthquake and now has survivors scattered over the area and buried under rubble. The task for the students is to develop a team of control agents for (very primitive) robots that will search the area and can dig out the survivors (often having to co-operate to achieve this). The simulation within ARES is rather abstracted, seeing the area as a grid where each grid field is essentially a stack of layers composed of rubble pieces and survivors. Each rubble piece requires a certain number of agents performing the action remove_rubble at the same time while standing on the particular grid field. The rescue agents can move around on the grid, communicate with the members of their team, rescue survivors that are on the top layer of a grid and then have different means to replenish their energy. Most actions of the agents require energy and the survivors also spend their energy (without being able to replenish it), so that there is a limited amount of time available for the rescue. The simulation is organized in rounds and in each round each agent sends to ARES the action(s) it wants to perform.

A core feature of ARES and ARES II are so-called world rules that can be instantiated differently and thus change the requirements on the strategies that the agents have to employ. Among others, there are world rules determining how many agents might be maximally needed to remove a piece of rubble, how agents can recharge, how costly communication is, how much information gleaned by observing the world is distorted and how scoring of rescued survivors is done in case that several different agent teams are acting within a particular ARES scenario. This allows to make the task for the students different in each new semester, essentially eliminating the copying of systems from previous semesters.

By looking at the number of rescued survivors and doing this for different scenarios it is easily possible to compare multi-agent systems written by different student teams and this adds a competitive component to the assignment. In fact, each semester there is a winning team that, in addition to an A, gets a prize and therefore there is quite some pressure on the instructor and the TA to quickly evaluate the different teams, starting with finding errors in the agent programs, but more importantly trying to decide how robust the teams are and, with ARES II, trying to see how the students are coping with the existence of other teams in a world scenario.

While it is naturally possible to pit the teams written by the different student teams against each other to evaluate the teams, this leads to inconsistent evaluations between semesters, since a just acceptable team in one semester with a lot of good student teams might look exceptional in another semester with only mediocre competition. And since the world rule settings differ from semester to semester, it is not possible to use the teams from previous semesters as measuring rod. The change in world rule settings also makes it difficult for the instructor to come up with good teams to test the students' teams against and in general it has to be assumed that no team would really evaluate well the weaknesses of all students' teams, which is where our evolutionary testing approach comes in. One obviously very interesting test goal is to see the worst case behavior of a students' team against a nearly clairvoyant team, which is what the instantiation of our general method in the next subsection tests.

4.2.2 Instantiating our method

Our tool for helping us to test the students' teams acting in ARES has to evolve action sequences for each of the rescue agents of the team that we pit against a students' team. If we included a third team into a scenario (for example, a second students' team that we do not test at the moment), this third team would be an example for bystander agents. The tested components are the agents in the students' team and the environment is ARES II. The actions used by our attack agents are all the actions mentioned in the last subsection, except for the communication action, since cooperation between the attack agents is achieved by the learner, so that communication is not necessary. We do not use the *init*-action in this case study, we have the human tester select the specifics of the scenario in which all the agents will act. Due to having the simulation organized in rounds, we do not need the $t_{i,j}$s, again.

Enhanced environment states are just the ARES states (obviously we want to avoid having to "look" into the agents of the students), which includes the positions of the agents, the actual state of the world fields (and layers, including the energy of the survivors and of the agents), and the scores of the teams. While all of this information is of interest for different test goals (see [DK06]), for the goal of seeing how bad it can get for a students' team, we really needed only the score $score(e_i)$ of the students' team in the current environment state. Ideally, the success of the students' team should be a score of 0, which means that $\mathcal{G}_{test}$ is true for a trace e_0,...,e_j, if $score(e_i)$ = 0 for all $1 \leq i \leq j$. Then we get as function $near\_goal$:

$$near\_goal((e_0,...,e_j)) = score(e_j)$$

With regard to the operators, we used all the operators presented in Section 3 and they got equal share in creating the next generation. Given our rather simple $near\_goal$ function, we wanted to target the positions in an individual that led to any change in the score, so that we choose too_much_lost = 1.

4.2.3 Some results

As in case of FIFA-99, we have evaluated our testing tool for students' teams for various settings, not only using different world scenarios of ARES but also other test goals (that were aimed towards not only seeing the worst case behavior of the students' team but also what it can do if faced with an absolutely selfless team of

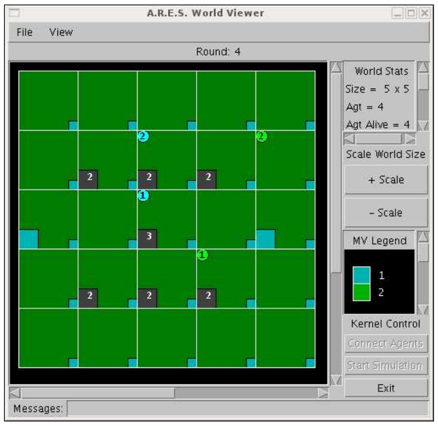

Figure 3: Round 4

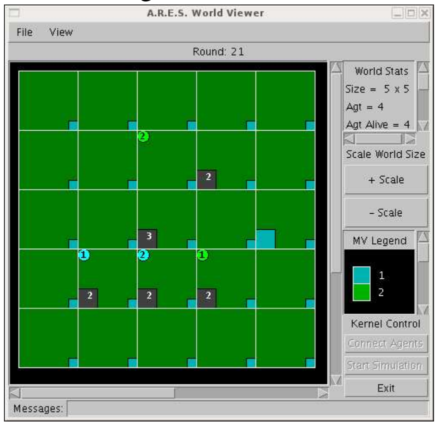

Figure 4: Round 21

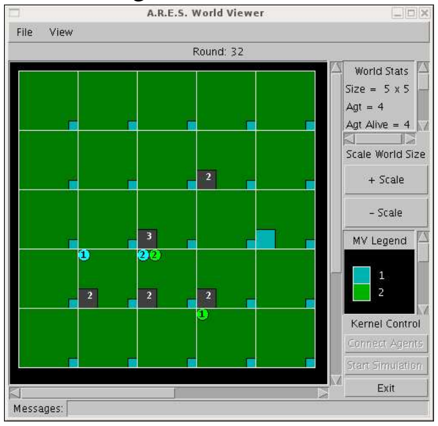

Figure 5: Round 32

attack agents that try to help the tested team to get as good a score as possible, see [DK06]). Again, due to lack of space, we will present here only one of the experiments, one that revealed an error in the tested team. Since the evolutionary learner uses random decisions, we repeated this experiment 10 times and each time the learner was able to center in on the error, although the generation, in which an individual showing it occurred, varied.

ARES allows each agent in a simulation a full second of CPU time to make its decision what actions to perform in the current round. The scenario that we set the students' team in 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. The simulation allows for 50 rounds, so that one evaluation run for an individual takes 200 seconds. Since there are no random effects in ARES or the students' team, we do only one evaluation run per individual, but due to the high time requirements a generation consists of only 8 individuals and we gave the system 10 generations.

The graphical interface of ARES that we use to observe runs is rather simple, as indicated by Figures 3 to 5. In Figure 3 we see the situation in round 4 of the run of the attack team found by our tool that reveals the problem in the students' team. Agents are indicated by a circle with their number in the team in the upper right corner of the grid field they are in. Different teams have different colors and the lighter agents are the tested team, while the agents with darker color are the attack team. The field in the lower left corner indicates the top layer of the stack of the grid field. A light box indicates a survivor, a dark box with a number a piece of rubble that needs the indicated number of agents to be removed. The box in the lower right corner indicates the move cost of a grid in a color code (which is the same for each field in this example).

As Figure 3 indicates, in round 4 no survivors have been rescued yet and no rubble removed, but the students' agents have started to move towards each other (the start position has one agent of each team in the middle of the upper row of fields and one of each in the middle of the bottom row). Figure 4 shows the situation in round 21. The students' agents have rescued one of the survivors that was not buried and two of the survivors that were buried and each of the agents is now sitting on a different field with a buried survivor. In this round, the problem in the students' team surfaces, since in all of the following rounds the students' agents will not move and stay at the positions that they now are in (Figure 5 shows a snapshot for round 32). Clearly, this is not what the students intended, since one of their agents just needs to move to the other and they could rescue another survivor (which, by the way, the students' team is doing perfectly when alone in this particular scenario: the team rescues all survivors that it can, leaving just the one in the middle for which they do not have enough agents).

The students' team did not show the problematical behavior when tested with other students' teams playing the role of the attack agents or with another incarnation of this students' team playing this role. We observed that these other teams usually either helped rescuing one of the survivors on the grid fields the tested agents are frozen on or they rescued one of these survivors on their own. This resulted in the tested agents becoming "unfrozen" so that a quick observation of such a run did not show any problems. But our learner was able to zoom in on this weekness and exploit it. Our learner was also able, using the other goal and fitness function reported in [DK06], to evolve attack agents that enabled the tested agents in this scenario to rescue all of the survivors.

5 Related work

As already stated, so far, tool support for exploratory testing focused on bookkeeping tasks around the tests performed by the human tester (see, for example, Test-Explorer [TeEx]). The use of evolutionary methods in software testing has been suggested relatively early. In [SGD92], a genetic algorithm was used to combine known error behaviors to create additional tests. Like [SGD92], the other known approaches to using evolutionary methods in software testing, which often are also called search-based methods in Software Engineering, are so-called model-based methods, which means that they are using models of the software to test to produce test cases for the software. Our approach, while able to also be applied to models, nevertheless is aimed at working directly on the software system to test, as exploratory testing does. We also do not see testing as just a search, it is a learning process (and there are rather different learning methods available to learn a behavior and not all of them are using search). The additional knowledge provided by a model can boost learning, since this knowledge does not have to be learned, but it also provides a bias and there is also the gap between model and real system (and the real system environment and the bystander agents) to consider.

[BLS06] uses as model that the evolutionary algorithm uses to create fitness measures a scheduling system and tries to evolve the start times for events that a real-time system has to react to. The goal is to come up with start times for a given group of events that fulfill certain constraints and additionally might have a chance to stress the real-time system so that it might fail to complete a task on time. All events need to be known beforehand and there is no possibility to consider different initializations of the tested system. The model also does not include different users of the system, all events are centrally initiated in the model. Like all other known approaches using search-based methods for testing, the fitness function that guides the search is not accumulative over interactions, as our approach is (we introduced accumulative fitness functions for learning of cooperative behavior in [DF96]). Consequently, targeted genetic operators can not be defined for these approaches.

[We03] uses as model flow charts of embedded systems and individuals consist of input parameters for the system that try to push the system into particular branches of the flow chart. The testing goal was to find input parameter settings creating very short and very long execution times for the embedded system, so that the fitness evaluation was done using the real tested system. Input sequences are not considered at all. Finding input parameters that send a tested system into a particular branch of its corresponding flow chart was also the goal of [JSE96], but additionally these parameters should also be near in their value to values that send the system into another path of the chart. [PHP99] also focused on creating input parameters for systems that cover the branches of the tested programs. They called their approach goal-oriented, but their goals are testing a particular branch by getting the program into it which should not be confused with our goal that is to produce a particular behavior or state of the tested system (without using flow charts or other models). [PHP99] also provided an example for the fact that evolutionary algorithms (with complex fitness functions) can be easily parallelized. This is naturally also true for our approach.

6 Discussion and future work

We presented an approach that can be used by tools to support exploratory testing. The general idea of the approach is to use evolutionary learning techniques to mimic in part what a human tester is doing when performing exploratory testing for unwanted behavior of a system or group of system components. By creating interaction sequences with the tested system both based on sequences that come nearer to getting the system to show the unwanted behavior than other sequences and based on random decisions, tools based on our approach can present to the human tester either interaction sequences that reveal that the tested system has an unwanted behavior or sequences that come very near to such a behavior and therefore can serve to provide the human tester with a better understanding of what is possible within the tested system. In our two case studies, tools based on our approach were able to reveal behaviors of the tested systems that were indeed unwanted.

We would like to stress the point that while clearly providing better tool support for exploratory testing and testing for unwanted behavior in general than what is the state-of-the-art, our test tools are not able to replace human testers. They can speed up the testing of a system (especially if we parallelize the evolutionary learning) by providing the tester with good test traces and they can make expertise of one tester available to other testers, but they still require the human testers to provide expertise in form of fitness function and parameters of the tool. And, as with all testing for unwanted behavior, there is no guarantee that a system is not able to show an unwanted behavior, even if our tools, respectively the combination of tool and human testers, do not find a way to produce this unwanted behavior. And obviously our approach does not help with testing the expected behavior which requires other kinds of testing and other tools.

By thinking of our method as *learning* of test cases, we establish a connection to Artificial Intelligence (AI) and the methods in this field. And, as stated in the introduction, methods from AI find more and more their way into mainstream Computer Science, especially methods for self-adaptation and cooperation between systems with the goal of emergent behavior. Using ideas from AI for systems including AI components on the one hand side makes a lot of sense ("battle fire with fire"), but on the other side has to ask the question "What about errors in the testing AI?". While we naturally can give no guarantees regarding tools using our approach with this regard, we would like to point out that our attack agents are very primitive, they execute chains of instructions without conditions or loops, and they are independent from the learner. If we find an unwanted behavior then a tester can just run the individual that created it and can see for him/herself what happens. Additionally, the learning task for our attack agents is usually easier than the tasks that any tested agent has, since we want to learn/construct one particular behavior (and our learner needs only to learn this behavior, the attack agents need only to implement this behavior) while tested agents in most cases have to realize many different behaviors realizing all requirements and tasks given to them.

Naturally, there are quite some possibilities for improvement of our general approach towards usability for very big and complex systems. In fact, we consider our results so far just a proof of concept. At the center of these improvements is the idea of integrating more testing knowledge into the basic method. Having already certain action sequences that are known to stress a system and having them represented as an action "macro" would allow for action sequences consisting of thousands of actions which some test problems will require. These macros could be extended to cover the actions of several attack agents, thus providing the evolutionary learner with useful coordinated pieces to be used. The fitness function is also an obvious place for integration of more knowledge. But the challenge will be to keep the balance between exploiting this knowledge and the ability to explore the possibilities. Using too much knowledge can result in an evolutionary tool not being able to achieve sufficient exploration. Therefore future work will include more evaluations of our approach to explore the advantages and the limitations of our approach more.

References

[Bach] J. Bach: Exploratory Testing Explained, http://www.satisfice.com/articles/et-article.pdf, as seen on Aug 30, 2006.

[BLS06] L. Briand, Y. Labiche, 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.

[Ch+04] 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.

[De+05] J. Denzinger, K. Loose, D. Gates, J. Buchanan: Dealing with parameterized actions in behavior testing of commercial computer games, Proc. CIG-05, Colchester, 2005, pp. 51–58.

[DF96] J. Denzinger, M. Fuchs: Experiments in Learning Prototypical Situations for Variants of the Pursuit Game, Proc. ICMAS-96, Kyoto, 1996, pp. 48–55.

[DK05] J. Denzinger, J. Kidney: Teaching Multi-Agent Systems using the ARES Simulator, Italics e-journal 4(3), 2005.

[DK06] 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.

[FJ93] S. Floyd, V. Jacobson: The synchronization of periodic routing messages, Proc. SIGCOMM'93, 1993, pp. 22–44.

[Go89] D.E. Goldberg: Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley, 1989.

[Ha96] R.G. Hamlet: Predicting dependability by testing, Proc. Intern. Symp. on Software Testing and Analysis, 1996, pp. 84–91.

[JSE96] B.F. Jones, H.-H. Sthamer, D.E. Eyres: Automatic structural testing using genetic algorithms, Software Engineering Journal 11(5), 1996, pp. 299–306.

[KBP01] C. Kaner, J. Back, B. Pettichord: Lessons Learned in Software Testing, Wiley Computer Publishing, 2001.

[KC03] J. Kephart, D. Chess: The Vision of Autonomic Computing, IEEE Computer 36(1), 2003, pp. 41–50.

[Mo05] J.C. Mogul: Emergent (Mis)behavior vs. Complex Software Systems, Internal Report HPL-2006-2, HP Laboratories Palo Alto, 2005.

[PHP99] R.P. Pargas, M.J. Harrold, R.R. Peck: Test-Data Generation Using Genetic Algorithms, Journal of Software Testing, Verification and Reliability 9(4), 1999, pp. 263–282.

[SGD92] A.C. Schultz, J.J. Grefenstette, K.A. De Jong: Adaptive Testing of Controllers for Autonomous Vehicles, Proc. Symp. on Autonomous Underwater Vehicle Technology, IEEE, 1992, pp. 158–164.

[SH05] M.P. Singh, M.N. Huhns: Service-Oriented Computing - Semantics, Processes, Agents, John Wiley & Sons, 2005.

[TeEx] TestExplorer and Exploratory Testing, http://www.sirius-sqa.com/exploratory_testing.html, as seen on Aug 30, 2006.

[We03] J. Wegener: Evolutionary testing of embedded systems, In *Evolutionary Algorithms for Embedded Systems Design*, Kluwer, 2003, pp. 1–34.