Dealing with parameterized actions in behavior testing of commercial computer games

Jörg Denzinger, Kevin Loose
Department of Computer Science
University of Calgary
Calgary, Canada
{denzinge, kjl}@cpsc.ucalgary.ca

Darryl Gates, John Buchanan
Electronic Arts Inc.
4330 Sanderson Way
Burnaby, Canada
{dgates, juancho}@ea.com

Abstract- We present a method that enhances evolutionary behavior testing of commercial computer games, as introduced in [CD+04], to deal with parameterized actions. The basic idea is to use a layered approach. On one layer, we evolve good parameter value combinations for the parameters of a parameterized action. On the higher layer, we evolve at the same time good action sequences that make use of the value combinations and that try to bring the game in a wanted (or unwanted) state. We used this approach to test cornerkicks in the FIFA-99 game. We were able to evolve many parameter-value-action sequence combinations that scored a goal or resulted in a penalty shot, some of which are very short or represent a rather unintelligent behavior of the players guided by the computer opponent.

1 Introduction

In recent years, many AI researchers have turned to using commercial computer games as "AI friendly" testbeds for AI concepts, like improved path finding algorithms and concepts for learning. Often, the guiding idea is to make the used game smarter to beat the human players, which naturally shows off the capabilities of a particular AI concept and scores points in AI's constant struggle to develop intelligent programs. But we see games incorporating such AI concepts rather seldom, in fact the current commercial computer games have to be seen as rather behind what AI researchers have accomplished with available source code of older games (see, for example, [vR03]). The reason for this lies in the priorities game companies have: their games are supposed to keep a human player or a group of human players entertained, which requires them to walk a very fine line between making it too easy and too difficult to win. Therefore, techniques that make games less predictable (for the game designer) have to be seen as a risk, especially given the current state of testing methods for games. Unfortunately, the aim of many AI methods is exactly that: to make games less predictable and more resembling what humans would do.

But we think that AI itself can come to the rescue: by using AI methods to improve the testing of commercial computer games we can not only improve the current situation with regard to testing games within the short time spans the industry can afford, we can also open the way for incorporating more AI methods into games, allowing for more complex game behavior that can even include aspects of learning and adaptation to the human player. We see as a key method for improving game testing the use of concepts from learning of (cooperative) behavior for single agents or agent groups, like [DF96], to find game player behavior that brings the game into unwanted game states or results in unwanted game behavior in general. Learning of behavior can also be used to find various game player action sequences that bring the game into states that match conditions a game designer has for certain special game behaviors, which also helps improving the testing of such ideas of the designers.

In [CD+04], we presented an evolutionary learning approach that allowed us to evolve action sequences, which lead with more than a certain high probability, to scoring a goal in the game FIFA-99 when executed. Evolutionary learning with its ability to achieve something similar to human intuition due to the mixture of randomness with exploitation of knowledge about individuals is very well suited to imitate game player behavior and the methods some players use to find *sweet spots* in games. In contrast to a search using more conventional methods like and-or-trees or -graphs, an evolutionary search can deal with the indeterminism that many commercial computer games rely on to offer different game playing experiences every time the game is played in a natural manner.

Our approach of [CD+04] used a fixed set of actions on which the action sequences to be evolved were built. Most sport games have a rather limited number of actions that are "activated" by pressing buttons and that are interpreted by the game in a manner appropriate to the situation in which the action is activated. But in many games there are at least some actions that require the game player to provide some parameter-values for the action (again, using buttons in a special manner). Examples of such actions are a pitch in a baseball game or a cornerkick in a soccer game. [CD+04] did not cover such parameterized actions.

In this paper we present an extension of the method of [CD+04] that allows to deal with special parameterized actions occurring at specific points in an action sequence. The basic idea (inspired by [St00]'s idea of layered learning) is to use a two-layered evolutionary algorithm, one layer evolving action sequences and one layer evolving parameter combinations for the parameterized action. Naturally, the fitness of a parameter layer individual has to depend on the fitness that this individual produces when applied in an individual of the action sequence layer, in fact it depends on the fitness of many individuals on the action sequence layer and vice versa. We were also inspired by concepts from

co-evolving cooperative agents, see [PJ00] and [WS03], although we make additionally use of the fact that the fitness on the action sequence layer is based on an evaluation of the game state after each action and therefore we can emphasize also the evaluation of the outcome of the special action.

We evaluated our extension to special actions within the FIFA-99 game, again. The special action we targeted was the cornerkick and our experiments show that the old soccer saying *a corner is half a goal*, while not true anymore in todays professional soccer, is definitely true within FIFA-99. Many of the behaviors we found (with regard to scoring) are rather acceptable, but we also found some very short sequences that always scored and some rather "stupid" opponent behavior consistently leading to penalty shots awarded to the attackers.

This paper is organized as follows: After this introduction, we re-introduce the basic ideas of [CD+04], namely our abstract view on the interaction of human players with a game in Section 2 and the evolutionary learning of action sequences for game testing in Section 3. In Section 4, we introduce our extension of the learning approach to allow for special parameterized actions. In Section 5, we present our case study within FIFA-99 and in Section 6 we conclude with some remarks on future work.

2 An interaction behavior based view on commercial computer games

On an abstract level, the interaction of a human game player with a commercial computer game can be seen as a stream of inputs by the player to the game. These inputs have some influence on the state of the game, although often there are other influences from inside and sometimes outside of the game. If we have several human players, then the other players are such an outside influence, since the game combines all the input streams and thus produces a sequence of game states that describe how a particular game run was played.

More precisely, we see the inputs by a game player as a sequence of action commands out of a set Act of possible actions. Often it is possible to identify subsets of Act that describe the same general action inside the game, only called by the game with different parameter values (like a pitch in baseball, where the parameters determine where the player wants to aim the ball at, or a penalty shot in soccer). We call such a subset a parameterized action (described by the general action and the parameter values, obviously) and all interactions of the player with the game that are needed to provide general action and parameter values are replaced by the parameterized action in our representation of the interaction as action sequence from now on. By $PAct$ we refer to the set of parameterized actions.

The game is always in a game state out of a set S of possible states. Then playing the game means producing an action sequence

$$a_1, ..., a_m;\ a_i \in Act \cup PAct$$

and this action sequence produces a game state sequence

$$s_1, ..., s_m;\ s_i \in S.$$

Naturally, the game has to be in a state s_0 already, before the action sequence starts. This start state can be the general start state of the game, but many games allow to start from a saved state, also.

What state the game is in has some influence on the actions that a player can perform. For a situation s, by $Act(s) \subseteq Act \cup PAct$ we define the set of actions that are possible to be performed in s (are recognized by the game in s). Especially the parameterized actions in most games are only possible to be performed in a limited number of "special" states, since providing the parameter values for such an action is often rather awkward (what usually is *not* done is to allow a user to enter a numerical parameter value directly; instead some graphical interface has to be used that often requires quite some skill to handle). The "special" states recognizing parameterized actions often allow only for one particular parameterized action.

For the learning of action sequences that bring the game into a state fulfilling some wanted (or, for testing purposes, unwanted) condition –we call this also a wanted (unwanted) behavior– the fact that not every action can be executed in every state poses a certain problem. But the fact that $Act(s)$ is finite for each s (if we count a parameterized action, regardless of the used parameter values, as just one action) allows us to order the elements in $Act(s)$ and to refer to an action by its index in the ordering (which is a natural number). And if we use the index number of an action to indicate it in an action sequence, we can guarantee that in every game state there will be an action associated with any index number. If such a number ind is smaller than $|Act(s)|$ in situation s, then obviously it refers to a legal action (although to different actions in different states). If $ind > |Act(s)|$, then we simply execute the action indicated by ind modulo $|Act(s)|$ which is a legal action, again.

If we look at action sequences and the game state sequences they produce, then in most games the same action sequence does not result in the same state sequence all the time (especially if we use indexes to indicate actions, as described in the last paragraph). This is due to the fact that the successor state s' of a game state s is not solely based on the action chosen by the game player and s. In addition to other players, random effects are incorporated by game designers in order to keep games interesting and these random effects influence what s' will be. For example, wind is often realized as a random effect that influences a pitch or shot in a sports game or the aim of an archer in a fantasy role playing game.

It has to be noted that everything we have observed so far with regard to keeping a human player entertained makes games less predictable and hence more difficult to test. We have to run action sequences several times to see if they have an intended effect, for example. But fortunately there are also "shortcomings" of human players that help with the testing effort. For example, producing an action sequence that totally solves or wins a game, i.e. starting from the game start and going on until the winning condition is fulfilled, is very difficult, since we are talking about a sequence that might have more than tens of thousands of actions. But

fortunately human players also cannot produce such a sequence in one try, respectively without periods in which they can relax. Therefore commercial computer games allow the user to save game states, as already mentioned, and structure the game into subgoals, like minor quests in role playing games or one offensive in a soccer or hockey game. This means that for testing purposes smaller action sequences can be considered, although it might be necessary to look at several ways how these sequences are enabled.

3 Evolutionary behavior testing without parameterized actions

In [CD+04], we used the interaction behavior based view on computer games from the last section to develop an approach to testing of computer games that aims at evolving action sequences that bring the game into a state fulfilling a given condition. This condition might either be a condition that the game designer does not want to occur at all, a condition that should occur only in certain circumstances and the designer wants to make sure that there is no action sequence that brings us there without producing the circumstances, or a condition that describes a game state from which the game designer or tester wants to launch particular targeted tests. The approach in [CD+04] did not deal with parameterized actions. In the following, we will briefly describe this original approach, on which we will base the extension that we describe in the next section.

The basic idea of [CD+04] was to use a Genetic Algorithm working on action sequences as individuals to produce a game behavior fulfilling a condition $\mathcal{G}_{unwanted}$. Due to the indeterminism incorporated in most games, this condition has to be fulfilled in more than a predefined percentage of evaluation runs of an action sequence. The crucial part of this idea is the definition of a fitness function that guides the evolutionary process towards action sequences that fulfill $\mathcal{G}_{unwanted}$. Following [DF96], the fitness of an individual is based on evaluating each game state produced during an evaluation run of the action sequence (as input to the target game), summing these evaluations up and performing several evaluation runs (starting from the same start state) and summing up the evaluations of all these runs.

The basic idea of an evolutionary algorithm is to work on a set of solution candidates (called individuals) and use so-called Genetic Operators to create out of the current individuals new individuals. After several new individuals are created, the original set and the new individuals are combined and the worst individuals are deleted to get a new set (a new *generation*) the size of the old set. This is repeated until a solution is found. The initial generation is created randomly. What individuals are selected to act as "parents" for generating the new individuals is based on evaluating the *fitness* of the individuals (with some random factors also involved). The idea is that fitter individuals should have a higher chance to act as parents than not so fit ones. The fitness is also used to determine what are the worst individuals.

This general concept was instantiated in [CD+04] as follows. The set of possible individuals $\mathcal{F}$ is the set of sequences of indexes for actions, i.e.

$$\mathcal{F} = \{(a_1,...,a_m) \mid a_i \in \{1,..., \max\{|\mathcal{A}ct(s_i)|\}\}.$$

The fitness of an individual is based on k evaluation runs with this individual serving as input to the game from a given start game state s_0 (k is a parameter chosen by the tester of the game). A low fitness-value means a good individual, while a high fitness-value is considered bad.

A single run of an individual $(a_1,...,a_m)$ produces a state sequence $(s_0,...,s_m)$ and we define the single run fitness $single\_fit$ of $(s_0,...,s_m)$ as follows:

$$single\_fit((s_0,...,s_m)) = \begin{cases} j, \\ \quad \text{if } \mathcal{G}_{unwanted}((s_0,...,s_j)) = \text{true} \\ \quad \text{and } \mathcal{G}_{unwanted}((s_0,...,s_i)) \\ \quad = \text{false for all } i < j \\ \sum_{i=1}^{m} near\_goal((s_0,...,s_i)), \\ \quad \text{else.} \end{cases}$$

Remember that $\mathcal{G}_{unwanted}$ is the condition on state sequences that we are looking for in the test. Note that we might not always require $\mathcal{G}_{unwanted}$ to take the whole sequence into account. If we only look for a property of a single state, then we have to test s_j only. By using as fitness of a run fulfilling $\mathcal{G}_{unwanted}$ the number of actions needed to get to the first game state fulfilling $\mathcal{G}_{unwanted}$, we try to evolve short sequences revealing the unwanted behavior to make analyzing the reasons for the behavior easier.

Within the function $near\_goal$ we have to represent the key knowledge of the game designer about the state or state sequence he/she is interested in. $near\_goal$ has to evaluate state sequences that do not fulfill $\mathcal{G}_{unwanted}$ and we need it to measure how near these sequences come to fulfilling $\mathcal{G}_{unwanted}$. $near\_goal$ depends on the particular game to test and the particular condition we want to fulfill.

$single\_fit$ sums up the evaluations by $near\_goal$ for all subsequences of $s_0,...,s_m$ starting with the subsequence s_0,s_1. On the one hand, this assures in most cases that a state sequence not fulfilling $\mathcal{G}_{unwanted}$ has a $single\_fit$-value much higher (usually magnitudes higher) than a sequence that does fulfill $\mathcal{G}_{unwanted}$. On the other hand, we award sequences that come near to fulfilling $\mathcal{G}_{unwanted}$ and stay near to fulfilling it. And finally, we are able to identify actions (resp. indexes in the individual) that move us away from the behavior or result we are looking for (after coming near to it).

To define the fitness fit of an individual $(a_1,...,a_m)$, let $(s_0,s_1^1,...,s_m^1)$,..., $(s_0,s_1^k,...,s_m^k)$ be the game state sequences produced by the k evaluation runs. Then

$$fit((a_1,...,a_m)) = \sum_{i=1}^{k} single\_fit((s_0,s_1^i,...,s_m^i))$$

As Genetic Operators, the standard operators Crossover and Mutation on strings can be employed. Given two action sequences $(a_1,...,a_m)$ and $(b_1,...,b_m)$, Crossover selects randomly a number i between 1 and $m-1$ and produces $(a_1,...,a_i,b_{i+1},...,b_m)$ as new individual. Mutation also selects randomly an i and a random action a out of $Act - \{a_i\}$ to produce $(a_1,...,a_{i-1},a,a_{i+1},...,a_m)$ as new individual.

[CD+04] also introduced so-called *targeted* variants of Crossover and Mutation that make use of the fact that

with $near\_goal$ we can judge the consequences of individual actions within sequences and can identify actions within a sequence that lead away from game states fulfilling $\mathcal{G}_{unwanted}$. The precise definition of Targeted Crossover and Targeted Mutation require that when evaluating an evaluation run of an action sequence (a_1,...,a_m), we remember the first position s_j such that

$$near\_goal((s_0,...,s_j)) \geq near\_goal((s_0,...,s_{j-1}))+ lose\_goal,$$

for a parameter $lose\_goal$. *Targeted Crossover* between (a_1,...,a_m) and (b_1,...,b_m) first selects one of the reported positions in (a_1,...,a_m), say a_j, then selects a position in (a_1,...,a_m) between $a_{j-COtarg}$ and a_{j-1} and performs a Crossover at this selected position. The same modification leads us to a *Targeted Mutation* using $Mtarg$ to define the interval to choose from.

4 Layered evolutionary behavior testing for parameterized actions

There are several possible ways by which handling parameterized actions could be integrated into the method defined in the last section. One rather obvious way is to treat them like ordinary actions, start them off with random parameter values and add a Genetic Operator that mutates the parameters of parameterized actions. While this allows for having parameterized actions everywhere in an action sequence, it is not easy to balance the mutation of the parameters with the other Genetic Operators, the search spaces for action sequences and parameter value combinations multiply each other and there is a big danger that targeted Genetic Operators target primarily parameterized actions in early stages of the search, since the parameter value combinations are not evolved enough. Additionally, the general quality of a parameter value combination cannot be expressed, since we have only a fitness for a whole action sequence.

Since in many games parameterized actions only occur under special circumstances, we decided to use a different approach that separates the evolution of action sequences and parameter value combinations a little bit more and allows for evolving action sequences that "work" with several value combinations and value combinations that are producing good results with several action sequences. Additionally, we made the assumption that parameterized actions are a good point to break action sequences into parts, i.e. being able to perform a particular parameterized action is often a good subgoal in a game. This then means that the parameterized action is the first action of an action sequence after the subgoal is achieved (like a cornerkick or penalty kick in soccer or a pitch in baseball). And naturally reaching the subgoal should be used as good point to save the game, so that we are provided with a clearly defined start state for the sequence starting with the parameterized action.

Under these assumptions, integrating parameterized actions can be achieved by having a second evolutionary learning process that tries to evolve good parameter value combinations, in a kind of lower layer of the whole process (similar to the layered learning idea of [St00] for reinforcement learning). The crucial problem that has to be solved is how the fitness of such a parameter value combination is calculated, once again. In fact, also the question of how to evaluate the fitness of an action sequence has to be raised, again. Our solution is to evaluate the fitness of a particular parameter value combination by employing it in several action sequences as the parameter values of the parameterized action the sequence starts with. Part of the fitness is then the combination of the fitnesses of the evaluation runs of the sequences with the value combination. But we also incorporate the immediate result of the parameterized action by measuring the resulting game state after performing it. As a consequence, we also modified the fitness of an action sequence by not only using the evaluation run(s) for one value combination, but the runs of all value combinations with the action sequence.

More formally, if a_{param} is the parameterized action with which we start an action sequence and if it requires p parameters out of the value sets V_1,...,V_p, then an individual on the parameter value level has the form (v_1,...,v_p) with $v_i \in V_i$. On the action sequence level, as before, an individual has the form (a_1,...,a_m). The fitnesses are then evaluated as follows.

On the action sequence level, we evaluate each individual (a_1,...,a_m) with a selection from the current population of the parameter value level, i.e. with individuals (v_1^1,...,v_p^1),...,(v_1^l,...,v_p^l). As before a single run of (a_1,...,a_m) starts from a start state s_0, performs action $a_{param}(v_1^i,...,v_p^i)$ for an i to get to state s_1^i and then uses a_1,...,a_m to produce states s_2^i,...,s_{m+1}^i. We then compute $single\_fit$((s_0,s_1^i,...,s_{m+1}^i)) and sum up over k runs for each (v_1^i,...,v_p^i), as before.

On the parameter value level, we use the evaluation runs done for the action sequence level. This means that for an individual (v_1,...,v_p) we have game state sequences (s_0,s_1^j,...,s_{m+1}^j) produced by running the action sequence $a_{param}(v_1,...,v_p)$,a_1^j,...,s_m^j (again, we might use the same action sequence k times, but j should range from 1 to $k \times o$ to have more than one action sequence, namely o, used and to achieve an even distribution of the evaluation runs among the individuals on the parameter value level). For the fitness $fitp$ of (v_1,...,v_p), we sum up the $single\_fit$-values for the produced state sequences, but we also consider especially the "quality" of the outcome of $a_{param}(v_1,...,v_p)$ by computing a weighted sum of the evaluation of this quality and the fitness of the action sequences. More precisely, if w_{seqfit} and w_{nexst} indicate the weights, then

$$\begin{aligned} fitp((v_1,...,v_p),(s_0,s_1^1,...,s_{m+1}^1),...,(s_0,s_1^{k\times l},...,s_{m+1}^{k\times l})) \\ = w_{seqfit} \times \sum_{i=1}^{k\times o} single\_fit((s_0,s_1^i,...,s_{m+1}^i)) \\ + w_{nexst} \times \sum_{i=1}^{k\times o} near\_goal((s_0,s_1^i)) \end{aligned}$$

With regard to Genetic Operators, on the action sequence level we use the operators introduced in Section 3, includ-

ing the targeted operators. On the parameter value level, we use the rather well-known operators for lists of numerical values, i.e. crossover similar to the one on the action sequence level (but not targeted) and a mutation that varies the value of a parameter by adding/subtracting a randomly chosen value within a mutation range r_{V_i} for the i-th parameter (the r_{V_i} are parameters of the evolutionary learning algorithm).

The two layers of the learning system proceed at the same pace, i.e. new generations for both levels are created, the individuals in both levels are evaluated and based on this evaluation the next generation is created.

5 Case study: FIFA-99

We instantiated the general methods described in Sections 3 and 4 for the Electronic Arts' FIFA-99 game. In this section, we will first give a brief description of FIFA-99, then we will provide the necessary instantiations of the general method, and finally we will report on our experiments and their results.

5.1 FIFA-99

FIFA-99 is a typical example of a team sports games. The team sport played in FIFA-99 (in fact, in all games of the FIFA series) is soccer. So, two teams of eleven soccer players square off against each other and the players of each team try to kick a ball into the opposing team's goal (*scoring a goal*). In the computer game, the human player is assigned one of the teams and at each point in time controls one of the team's players (if the team has the ball, usually the human player controls the player that has the ball). In addition to actions that move a player around, the human user can let a player also handle the ball in different ways, including various ways of passing the ball and shooting it. As parameterized actions on the offensive side, we have, for example, cornerkicks and penalty kicks. On the defensive side the actions include various tackles and even different kinds of fouls.

The goal of the human user/player is to score more goals than the opponent during the game. FIFA-99 allows the human player to choose his/her opponent from a wide variety of teams that try to employ different tactics in playing the game. In addition, Electronic Arts has included into FIFA-99 an AI programming interface allowing outside programs to control NPC soccer players (i.e. the soccer players controlled by the computer). For our experiments, we extended this interface to allow for our evolutionary testing system to access various additional information, set the game to a cornerkick situation and to feed the action sequences to the game.

5.2 Behavior testing for cornerkicks

A human player of FIFA-99 (and other team sport games) essentially controls one soccer player at each point in time, allowing this player to move, pass and shoot and there is also the possibility for the human player to switch his/her

Figure 1: Cornerkick start situation

control to another of his/her players. But this switch is also performed automatically by the game if another soccer player of the human player is nearer to the ball (and switch goes through a predefined sequence of the players in the team to accommodate the way the controls for the human player work). The set of actions without parameters is

- NOOP
- PASS
- SHOOT
- SWITCH
- UP
- DOWN
- RIGHT
- LEFT
- MOVEUPLEFT
- MOVEUPRIGHT
- MOVEDOWNLEFT
- MOVEDOWNRIGHT

There are several other, parameterized actions that all are associated with special situations in the game. For our experiments, we concentrated on the action

CORNER(x,z,angle)

that performs a cornerkick. Here, the x parameter gives the x-coordinate, a value between 0 and the width of the field. The z parameter describes the z-coordinate (at least in FIFA-99 it is called z), a value between 0 and the length of the field. Finally, angle provides the angle to the field plane that the kick is aimed at (within 90 degrees).

The unwanted behavior we are looking for in our test is either scoring a goal or getting a penalty shot. So, $\mathcal{G}_{unwanted}((s_0,...,s_i))$ = true, if FIFA-99 reports a goal scored in one of the states s_0,...,s_i or a penalty shot is awarded in one of these states. In [CD+04], s_0 was the kick-off, while in our experiments in the next subsection, we start from the cornerkick situation depicted in Figure 1.

The fitness functions of both layers are based on the function $near\_goal$. We use the same definition for this function as in [CD+04]. This definition is based on dividing 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.

We associate with each zone a penalty value (pen_1 to pen_4) and decide which value to apply based on the position of the player with the ball (resp. if none of the own players has the ball, then we look at the position of the player that had the ball in the last state). If the position is in zone 1, we apply pen_1 as penalty, if the position is not in zone 1, but in zone 2, we apply pen_2 and so on. By $dist(s_i)$ we denote the distance of the player position from above to the opponent's goal. Then for a state sequence (s_0,...,s_i), we define $near\_goal$ as follows:

$$near\_goal((s_0,...,s_i)) = \begin{cases} dist(s_i) \times \text{penalty}, \\ \quad \text{if the own players had the} \\ \quad \text{ball in } s_{i-1} \text{ or } s_i \\ max\_penalty \\ \quad \text{else.} \end{cases}$$

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. For the targeted operators we set $lose\_goal$ to $max\_penalty$ and did not consider $near\_goal$((s_0,...,s_{j-1})) at all, so that we target those actions that result in losing the ball. As in [CD+04], we used $Mtarg = COtarg = 1$.

For the parameters guiding the computation of $fitp$, we set w_{nexst} to 20 and w_{seqfit} to 1. Since a lower fitness means a better individual, the influence of the state directly after the cornerkick on $fitp$ is substantial, so that parameter value combinations that result in the own team not being in possession of the ball have a very bad fitness.

5.3 Experimental evaluation

We used the instantiation of our general method from the last subsection to run several test series. As described in [CD+04], we flagged action sequences that fulfilled $\mathcal{G}_{unwanted}$ in one evaluation run and then did 100 additional runs with this action sequence-parameter value combination. And we only consider sequence-value combinations that were successful in 80 of these additional runs for reporting to the developer/tester. The population size in the parameter value level was 20 and an individual on the action sequence level was evaluated using each of the individuals of the current value level once (i.e. $l = 20$, $k = 1$). On the action sequence level, the population size was 10.

In our experiments, every run of our test system quickly produced one or several action sequences that were flagged for further testing and from every run we got at least one parameter value-action sequence combination that passed the additional test (and scored a goal; we usually got several combinations that produced a penalty shot). Many of these

Figure 2: First sequence, ball in air

Figure 3: First sequence, ball under control

combinations are quite acceptable, i.e. they present a game behavior that could be observed in a real soccer game, except for the fact that the game does allow for the scoring by this combination so often (or even all the time).

In the following, we will present a few rather short action sequences that scored in our experiments all the time, with screenshots for the shortest sequence we found. We will then also present some screenshots showing a not very intelligent game behavior consistently leading to a penalty shot. This is the kind of game behavior that a developer will want to change, respectively allow to be exploited only once or twice. In all figures, the light-colored soccer players are the attackers controlled by the action sequences produced by our system and the dark-colored players are the opponents.

Since a cornerkick is a dangerous situation for defenders in a soccer game, it can be expected in a computer soccer game that there are many opportunities for scoring goals. Here are the 3 shortest action sequences we have found using our testing method, so far:

Figure 4: First sequence, shoot

Figure 5: First sequence, goal!

Figure 6: Second sequence, ball in air

Figure 7: Second sequence, ball under control?

- CORNER(-1954,-620,28.1), LEFT, SHOOT
- CORNER(-1954,-620,28.1), MOVEDOWNRIGHT, UP, SHOOT
- CORNER(-2097,-880,27.5), LEFT, DOWN, MOVEDOWNRIGHT, SHOOT

The first two were produced by the same run, after 3:15, resp. 3:13 minutes of search, while the third was found after 7:22 minutes (in a different run). The first two are a good example for how the fitness function fit aims at producing shorter sequences.

Figure 2 shows the effect of the first value-sequence combination, resp. the effect of the cornerkick parameter values. In Figure 3, the attacker has the ball under control and moves to the left to get some distance to the goalie that will come out of its goal soon. Figure 4 shows the situation after the goalie has come out and the attacker has shot towards the goal. And Figure 5 shows the goal.

When we added being awarded a penalty shot as a successful attack action sequence, we were not aiming at cornerkicks (this was more of interest for the sequences starting from the kickoff, in our opinion). But surprisingly, our system evolving action sequences found more sequences leading to penalty shots than it found leading to a goal. These sequences are longer (but found earlier by the evolutionary search) and while some of the situations could happen in a real soccer game (although you do not see many penalty shots developing out of corners), there are quite a few sequences that we would consider unrealistic, due to the foul committed being rather unnecessary. The following sequence is an example for such a sequence:

- CORNER(-1954,-775,28.1), NOOP, DOWN, DOWN, RIGHT, SWITCH, MOVEUPLEFT, MOVEDOWNRIGHT, LEFT, LEFT, SWITCH

Figure 6 shows the ball in the air. Figure 7 shows the attacker trying to get the ball under control. In Figure 8, the attacker is still trying to get control, while the action sequence now switches to a different attacker, essentially leaving the player with the ball with the built-in control, which

Figure 8: Second sequence, still trying to get control

Figure 9: Second sequence, stupid foul!

drives it nearly out of the penalty box. In Figure 9, the control switches back just when the attacker is brought down from behind just a little inside of the penalty box. This is a rather stupid move by the defender and as consequence, Figure 10 shows the referee indicating a penalty shot. We had several other sequences leading to a penalty shot, where actively moving control away from the key attacker to let the game take over its behavior resulted in a penalty. While this naturally is not a behavior expected by a human player (and therefore not intensively tested) it shows an advantage of our testing method.

6 Conclusion and Future Work

We presented an extension to the behavior testing method proposed in [CD+04] that allows us to deal with actions that have parameters. By adding a lower evolutionary layer for evolving good parameter value combinations, we are able to co-evolve parameter value combinations and action sequences resulting in value-sequence combinations that produce unwanted behavior in games. Our tests with the FIFA-99 game produced many value-sequence combinations scoring goals and resulting in penalty shots. Several of the penalty shot situations are not very realistic both from the attacker and the defender side, so that we revealed a weakness of the AI controlling the defender that should have been avoided.

Figure 10: Second sequence, penalty kick!

Our future work will focus on developing fitness functions for unwanted behavior that do not focus on scoring goals. In newer versions of FIFA, we have special players with special moves and special abilities (modeled after real, and well-known human soccer players) for which the game designers have special expectations. Bringing the game into situations where these expectations can be observed (and reaching such situations in different ways) is not easy to achieve using human testers, but an obvious application for our method.

Bibliography

[CD+04] 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.

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

[PJ00] M.A. Potter and K.A. De Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents, Evolutionary Computation 8, 2000, pp. 1–29.

[St00] P. Stone. Layered Learning in Multi-Agent Systems: A Winning Approach to Robotic Soccer, MIT Press, 2000.

[vR03] J. van Rijswijck. Learning Goals in Sports Games, Game Developers Conference, San Jose, 2003. http://www.cs.ualberta.ca/ javhar/research/ LearningGoals.doc

[WS03] S. Whiteson and P. Stone. Concurrent Layered Learning, Proc. AAMAS-03, Melbourne, 2003, pp. 193–200.