Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment

Behavior Learning-Based Testing of Starcraft Competition Entries

Michael Blackadar and **Jörg Denzinger**
Department of Computer Science
University of Calgary, Canada

Abstract

In this paper, we apply the idea of testing games by learning interactions with them that cause unwanted behavior of the game to test the competition entries for some of the scenarios of the 2010 StarCraft AI competition. By extending the previously published macro action concept to include macro action sequences for individual game units, by adjusting the concept to the real-time requirements of StarCraft, and by using macros involving specific abilities of game units, our testing system was able to find either weaknesses or system crashes for all of the competition entries of the chosen scenarios. Additionally, by requiring a minimal margin with respect to surviving units, we were able to clearly identify the weaknesses of the tested AIs.

Introduction

Testing of games has always been a crucial and very costly part of the game development process. While using conventional testing methods from SE might be sufficient to test components of a game like the graphics engine, the physics engine or sound, other components like the game AI and the interactions between components require other additional testing. This is done by human test players that have to be rather good at playing games finding unwanted behavior. In addition, the improvements in game AIs over the last decade make this testing more and more difficult, due to the increased abilities of these AIs and the consequently increased risks for AI misbehaviors.

Automating at least some of the kinds of tests human testers currently have to do would not only reduce the costs due to game testing, it would also eliminate the chances for a "bad day" that human testers have. Recently, the use of machine learning as the means to automate testing of games and especially the game AIs has been suggested (see, for example, (Chan et al. 2004) or (Xiao et al. 2005)). The method from (Chan et al. 2004) was refined and improved in (Chan et al. 2005) and (Atalla and Denzinger 2009) to deal with more complex game situations and another game genre. The other game genre was Real-time Strategy games (RTS) that require game AIs to control a number of units and the infrastructure that supports those units, with the goal of battling and destroying units and infrastructure of the opponent(s). With games like the Warcraft series, or the StarCraft series, RTS games represent a sizable market segment and therefore have drawn the attention of a lot of researchers as evidenced by two competitions, namely the ORTS competitions (see (Buro 2007)) and recently the StarCraft AI player competition (see (Weber 2010)). (Atalla and Denzinger 2009) used the ORTS competition and AIs developed for this competition as the application area for the learning based testing concept and showed the need for so-called macro actions to deal with the complexity of having to test AIs controlling groups of units (and the learner having to come up with cooperative behaviors for groups of agents).

In this paper, we extend the method from (Atalla and Denzinger 2009) to deal with testing AIs developed for the StarCraft 2010 competition (respectively the first two tournaments of this competition that deal with battles between groups of units). The new challenges we had to face with this competition were a game interface that is not turn-based (in contrast to ORTS), the need for more types of macros than the cooperation macros, dealing with different types of units in the same type of tournament scenario and sometimes having to find wins by some margin of units to be able to really see the problematic behavior in the tested AI player. Our solutions to these challenges are heavily based on the macro concept and on how we represent and interpret a strategy for the units controlled by the learner. It should be noted that for testing competition AIs, the main goal is obviously finding weaknesses in them to fix these weaknesses. For testing game AIs intended to create interesting game experiences for human players, human testers will have to decide if a found weakness should be fixed or not.

In our experimental evaluation, our testing system was able to either crash or defeat all of the competition AIs from the first two tournaments of the 2010 StarCraft competition by a margin sufficient enough to show off a weakness. This included beating the winner of the most complex scenario losing only 4 of the 28 units.

Testing by learning behavior

In this section, we present the general idea of testing systems by learning behavior, as first introduced in (Chan et al. 2004), using an evolutionary learner and also how this idea was extended in (Atalla and Denzinger 2009).

When testing a system or multi-agent system A_{tested} = $\{\mathcal{A}g_{tested,1},...,\mathcal{A}g_{tested,m}\}$ that acts in a given environment $\mathcal{E}nv$, the tester takes over control of his/her own group of agents $A_{attack} = \{\mathcal{A}g_{attack,1},...,\mathcal{A}g_{attack,n}\}$ and tries to create behaviors for the agents in A_{attack} that result in the agents in A_{tested} showing an unwanted behavior. We see this even for a human tester as a learning process that can be automated using a machine learner.

More precisely, our learner creates an attack strategy for testing of A_{tested} by creating for each $\mathcal{A}g_{attack,i}$ a sequence of actions $(a_{i,1},...,a_{i,l})$, where each $a_{i,j} \in Act_{attack,i}$ for the set of possible actions $Act_{attack,i}$ of $\mathcal{A}g_{attack,i}$. The attack strategy is then evaluated by having the attack agents execute their particular action sequences in $\mathcal{E}nv$ together with the agents of A_{tested}. The result of this evaluation is a trace $e_0,e_1,...,e_x$ of environmental states that the learner uses to determine how good the attack strategy is.

While in theory every learner that can produce sequences of actions for all attack agents can be used for this approach to behavior testing, we found evolutionary learners very useful for this task, since they mimic rather closely how human testers approach the problem. Our evolutionary learner works on individuals as described above, i.e. a vector of action sequences $((a_{1,1},...,a_{1,l}),...,(a_{n,1},...,a_{n,l}))$ that forms an attack strategy. An individual's fitness is evaluated by measuring the environmental trace produced by the individual.

In (Atalla and Denzinger 2009), the agents in A_{attack} are allowed to interpret an action according to the environmental state e_j (with $j \leq= x$) that the agent is currently observing. Additionally, (Atalla and Denzinger 2009) extended each of the sets $Act_{attack,i}$ by a set Act_{macro}, i.e. $Act^{new}_{attack,i} = Act_{attack,i} \cup Act_{macro}$, $Act_{attack,i} \cap Act_{macro} = \emptyset$, and each agent $\mathcal{A}g_{attack,i}$ itself is extended by some internal data fields Dat_i for memory and a *decision function* $f_{\mathcal{A}g_{attack,i}} : Act^{new}_{attack,1} \times ... \times Act^{new}_{attack,n} \times \mathcal{E}nv \times Dat_i \rightarrow Act_{attack,i}$ to determine what "normal" action or sequence of "normal" actions it is taking during a fitness evaluation run. Dat_i contains two variables, namely an element $a^i \in Act_{macro} \cup \{\perp\}$ that contains the macro action the agent is currently working on (respectively $\perp$ if the agent is not involved in such an action at the moment) and an element $k^i \in \mathbb{N}$ that indicates how many "steps" to do are left in the action sequence that implements the macro action in a^i. We will extend this agent definition once more in the next section.

The general scheme for computing the fitness fit of an individual given the environment states it produced is

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

Here, $\mathcal{G}_{test}$ is a predicate that evaluates to true, if the environmental states $e_0,...,e_j$ showed the unwanted behavior we are testing for. And *near_goal* will measure how near its argument trace comes to fulfilling $\mathcal{G}_{test}$ for the particular system and application that is tested.

In an evolutionary learner, a set of individuals forms a so-called generation. After each individual in a generation is evaluated and has a fitness value, new individuals are generated using genetic operators. For vectors of sequences, there are several variants of the two standard classes of operators, namely mutation and crossover, known in the literature. As in (Atalla and Denzinger 2009), we added so-called targeted variants of these operators that use the computation of fit for an individual to identify positions in the action sequences of individuals, where the *near_goal*-values heavily drop. This indicates that one or several agents did perform bad actions and therefore the genetic operators should target the few positions before that position to try to create individuals that do better than the old individuals.

Testing Starcraft AI players

In this section, we first briefly introduce the StarCraft game and the competition setting used in (Weber 2010). Then we present our extensions/instantiations of the testing method from the last section to test AI players for the competition.

StarCraft

Starcraft is an RTS game where each player is in control of a number of agents (units). In a typical game, the goal of each player is to destroy the other players' buildings. Units have different types and abilities, so the player must know how to use these appropriately. Each unit has different attack capabilites, such as only being able to attack in close range, or only being able to attack ground units from the air. These units also have abilites such as cloaking, which hides the unit until detected by a special enemy unit, or the ability to heal friendly units. Exploiting weaknesses of the opposing player by using the best units and abilities for the situation is part of the skill involved while playing. At the beginning of a complete game, each player has a number of worker units, a base which can produce more workers, and nearby resources. The player must harvest these resources using the workers, and use the workers to create structures. These structures are what allows for the creation of new units. In this way, each player creates an army which fights with the opponent anywhere on the map. For our testing, we concentrate on battles between groups of units only, which is also the setting of the first 2 tournaments in the AI player competition.

For the StarCraft competition, the competition AIs interacted with the game using the bwapi API (see (BWAPI 2010)). Given that StarCraft is a commercial product and that therefore it is not possible to easily change the source code of it, bwapi needs to emulate a human user interacting with the game, which essentially means that it sends mouse positions, mouse button events and keyboard events to the game. In return, bwapi sends back the game state as it can be constructed from what the game presents to the human player, which contains the positions of the units and their current health. In order to realize the real-time aspect of the game, bwapi interacts with the game via 24 frames per second, which theoretically allows for a game AI to influence its units much more than a human player could. It should also be noted that the game itself already provides each unit with a rather simple control AI, that lets a unit react to some events in its immediate neighborhood in cases when the unit is not following explicit commands from the (human or AI)

player. This essentially results in the unit defending itself when attacked by either counterattacking or running away.

Instantiating the testing approach

In order to apply the testing approach from the last section to StarCraft AIs, we needed to extend the macro concept from (Atalla and Denzinger 2009) to on the one hand side deal with the bwapi API and also to allow for a more selective way how macros influence the game units (i.e. the $\mathcal{A}g_{attack,i}$s). In order to evaluate the different kinds of macros, we created 3 variants of our testing system: variant one, the basic variant, is not using any macros, variant two only uses macros that influence single units (macros that did not exist for ORTS) and variant three adds to variant two cooperation macros, i.e. macros that influence several units at the same time.

More precisely, an agent $\mathcal{A}g_{attack,i}$s in A_{attack} can choose its possible actions from the union of three sets, $Act_{attack,i}$, $Act_{indmacro,i}$ and $Act_{coordmac,i}$ with empty intersection sets. $Act_{attack,i}$ contains actions that move the agent a given distance to either the north, east, south, or west and that allow it to attack either the closest or weakest enemy unit in range.

$Act_{indmacro,i}$ contains two attack macros that either attack the closest unit of the enemy or the weakest unit (as known to the testing system). In contrast to the attack actions in $Act_{attack,i}$, these macros will first move near enough to the particular unit to get it in range before they then perform the attack. If the enemy has different types of units, we have these two macro actions for each possible type. For the unit type medic, we also have the heal macro that causes the medic to find the weakest other unit, move towards it and heal it, while also healing any other own units on the way.

$Act_{coordmac,i}$ contains macro actions that involve several attack agents and that coordinate the efforts of these agents. Similar to (Atalla and Denzinger 2009), we have a gather macro that causes all units to move towards the center of the group and a team attack macro that has a group of agents moving in range of the weakest enemy unit and attacking it. We also have for agents of type marine a stim macro that has every marine who has enemy units nearby perform the unit-specific stim action that allows these agents to move faster at the cost of loosing health points. And we have the team heal macro that causes injured units to move away from the enemy and towards a medic, while this medic moves towards those injured units.

In order to enable $Act_{attack,i}$ to use these macros, we needed to extend its internal data fields Dat_i by a field $A^i \subseteq A_{attack}$ that tells the agent which other attack agents it is performing a macro currently with. A^i is empty, if the agent is not performing a macro, it is $\{Act_{attack,i}\}$ if the agent performs a macro out of $Act_{indmacro,i}$ and it contains $Act_{attack,i}$ and all other agents that perform the macro, if it is from $Act_{coordmac,i}$. The agent's decision function $f_{\mathcal{A}g_{attack,i}}$ has been extended as explained above to initialize the macro behavior and to then perform the macro as intended. Please note that not only the initialization of a macro from $Act_{coordmac,i}$ requires that several agents coordinate with each other, this is also necessary during the performance of the macro, for example when a targeted enemy unit is destroyed and the target needs to be changed. As in (Atalla and Denzinger 2009), a macro is activated if a single agent has this macro as next action in its sequence and performing this macro overrides the actions intended for the other agents involved.

The fitness fit used by our learner instantiates the general scheme from the last section as follows. Our predicate $\mathcal{G}_{test}$ is not just having won the game against the tested StarCraft AI. In our initial experiments we had to realize that for some of the tournament scenarios just winning was not enough to identify the general problem the AI has. Therefore our $\mathcal{G}_{test}$ allows to define a margin *surv* of units by which to win, i.e. at least *surv* agents of A_{attack} need to still be alive after the last enemy unit has been destroyed. Given our definition of *near_goal*, this extension was not difficult to do.

For *near_goal*, we followed the general ideas from (Atalla and Denzinger 2009), but used only two of the components from there, namely the surviving unit count SU and the health criteria HE. $SU(e_j)$ for an environmental state e_j is the difference between the agents in A_{attack} still alive and the agents in A_{tested} still alive, with different weights associated with different types of units. Similarly, $HE(e_j)$ adds up the health points of each unit in A_{attack} and A_{tested}, multiplied by weights associated with the type of a unit, and then subtracts the sum for A_{tested} from the sum for A_{attack}. *near_goal*$((e_0,...,e_j))$ then just sums up $SU(e_j)$ and $HE(e_j)$.

As evolutionary operators, we used some slight modification of the ones used in (Atalla and Denzinger 2009). This modification is that on the level of agents, our operators randomly selected some of the agents (and not just one) to be affected by the operator. Then the standard operators for strings, respectively the targeted variants, are applied to the action sequences for these selected agents. For selecting the individuals that act as parents to the operators, we used roulette-wheel selection. And naturally the best individuals of one generation are copied over to the next generation without change.

Experimental evaluation

In this section, we present our evaluation of the 3 variants of our testing system described in the last section using AI StarCraft players from the 2010 AI player competition. We will first provide the exact settings for the parameters of our system and a short description of the test scenarios and then look at our experimental results.

Setup

For the evolutionary learning system, we used 10 generations with 20 individuals in each of them. One in four genetic operator applications was a mutation the others crossovers. For each type of operator, 4 out of 5 applications were of the targeted kind. Our attack agents provided an action to the game every second (i.e. each 24th frame) and the action sequence for an agent in an individual consisted of 300 actions, so that a single game had a maximum duration of 5 minutes and consequently a whole run of our testing system could last 1000 minutes (although this was seldomly the case). In our experiments, there were 5 types of units in the

scenarios: zealot, mutalisk, scourge, marine and medic. For determining the weights used in the computation of the SU-value, we used the costs that building such a unit requires. This resulted in weights of 100, 250, 137.5, 50, and 87.5. The weights used in the computation of HE reflect the initial health points each type of unit has, which results in weights of 160, 120, 25, 40, and 60.

For performing our experiments, we used two computers interacting in a network. The first computer ran the newest version of bwapi, the game itself and our testing system. The second computer ran the proper version of bwapi for the AI we are testing and had it join the game hosted by the first machine.

The StarCraft AI player competition consisted of several tournaments and some of the tournaments had several scenarios. The first two tournaments deal with battles between two opponents. We selected two scenarios from the first tournament (Scenario 1 and 2 in Table 1) and one scenario from the second tournament (Scenario 3). Scenario 1 involves two small groups of melee units battling over a small map. This was chosen as a starting point for our system, as it is the simplest scenario from the competition. There is only one type of unit to control, namely zealots which have no special unit abilites, and the map has no terrain features. In Scenario 2, both players control two types of flying units. One type, mutalisks, has a ranged attack that can hit multiple enemies if they are close together, and the other type, scourges, can explode when in contact with the enemy. This scenario takes place on the same simple map as Scenario 1. To be successful in this scenario, a player must know how to use each type of unit properly, which makes this scenario more interesting than the other scenarios in the first tournament and the reason why we selected this scenario.

In Scenario 3, each player controls a large group of marines, which are a ground unit with a ranged attack, and a group of medics which are support units. The marines have a special ability, namely the already mentioned stim action, that increases their fire and movement rates at the cost of their health. The medics are capable of healing each other and the marines as long as they are close enough to each other. This scenario takes place on a larger map that has terrain features such as ramps and different levels of elevation. This is important because units on higher ground are less likely to be hit by ranged attacks from lower ground. This scenario was chosen because it is the most interesting of the scenarios from the two tournaments, since it involves multiple unit types, terrain, unit abilites, and requires cooperation between units in order to be successful.

In the 2010 StarCraft AI player competition, there were 7 competitors in the first tournament (for all scenarios), namely FreScBot, Sherbrooke, MSAILBOAT, Windbot, ItClusters, UTPABroncScript, and ChaosNeuron (which is called ArixSheeBot on the participant page at (Participants 2010)). FreScBot and Sherbrooke also participated in the second tournament (in all scenarios). In the 2010 competition, FreScBot won both tournaments.

Scenario/Player	One	Two	Three
Scenario 1			
FreScBot	X	X	8(6)
ChaosNeuron	X	X	8(3)
UTPABroncScript	X	X	8(6)
Windbot	X	X	2(6)
ItClusters	X	X	3(4)
Scenario 2			
FreScBot	X	5(7)	0(6)
ChaosNeuron	X	X	3(8)
UTPABroncScript	X	X	5(8)
ItClusters	X	X	5(9)
Scenario 3			
FreScBot	X	X	3(24)

Table 1: Generation of sucess of our learner

Results

We applied all three variants of our testing system to all competitors for the scenarios they participated in in the 2010 competition. Unfortunately, we had problems with some of the competitors due to crashes, but since it is the goal of our testing system to test systems, we see this also as a success of our system. We were able to reproduce the crashes at will with all three variants of our system. The Sherbrooke AI crashed on every scenario within the first five games of a test run (i.e. within creating the initial generation). This crash occured immediately when a new game was started, before either player could issue orders to their units. A possible explanation for this behaviour may be not clearing memory properly. It should be noted that our system relies on playing multiple games against the opponent to find a weakness, which makes clearing the memory an issue.

The Windbot AI always crashed in Scenario 2 immediately after our system had destroyed all of the AI's units. This may have happened because the developers may have assumed that the game would be immediately over when a player loses its units, but we found that it took a frame after this event to end the game. The last AI that had crashing problems was MSAILBOAT. Strangely, Scenario 1 would crash the first time our testing system only had one unit left. It is possible that there is a calculation in the tested system that depends on the enemy having two or more units. In Scenario 2, the AI crashed a few seconds after our units came in view during the first game. We do not have a reasonable explanation for this behaviour.

The results of our tests with the other competition AIs are reported in Table 1. An X indicates that the system variant was not able to win against the competitor within the given number of generations. An entry of the form X(Y) indicates that our testing system produced an individual that won against the competitor and that allowed us an idea what the problem of the competitor is in generation X with the testing system having Y units left alive.

As can be seen in Table 1, variants One and Two of our testing system are not really very successful. It takes all types of macros to play the game well enough to iden-

tify weaknesses. But if all macro types are used, i.e. variant Three, our testing system was able to win against all of the competition AIs with margins high enough to allow for an analysis of the weaknesses of the opponents.

Let us first look at FreScBot, the winner of both tournaments. In Scenario 1, we found the following unwanted behavior. Our, i.e. the testing system's, units stayed at the starting point for a while, then moved slowly towards the centre. The opponent moved towards the centre, and upon reaching it, split its units into three groups. One was sent to the top of the screen, the other stayed in the middle, and the third moved towards us to attack. The intention may have been to search for our units, but once our units were in sight, the opponent did not bring its units together again. Instead, all of our units fought the small group of units that came forward, then the ones in the centre, then finally the ones at the top of the map. The top group never attempted to rejoin the others, so our system was able to win with 6 of the original 9 units surviving due to having greater numbers in the battle. In Scenario 2, the opponent moves all units towards the centre, which causes them to be very close together. Our strategy spread out our units as they moved towards the centre, simply due to the random generation of move actions by the learner. Because the opponent had many units close together, our selfdestructing scourge units were able to cause a great deal of damage. In contrast, our units were rather spread out, making it difficult for the opponent to chose the proper unit to attack with its scourges. Additionally, fewer of these caused damage, because we were able to shoot them down before they detonated. When both sides only had ranged units left, we had the greater numbers.

In Scenario 3, both groups moved towards each other, however our system moved slowly due to frequently gathering the units together via the gather macro (see Figure 1, we are the yellow units, the opponent's are red). The opponent had most of its units moving in a long line, without stopping to regroup. This caused the battle to happen at the top of our ramp (see Figure 2), which gave the opponent units less of a chance to hit us. The opponent did not seem to consider the terrain, which gave us an advantage. Because our units were together while the opponent's came in small groups, we were able to cause much more damage. The stim ability was not used by the opponent, which may be a weakness since it provides an attack speed bonus, and our agents used it heavily. Finally, as a side effect of having its units spread out, the opponent's medics were not very effective at healing, whereas ours were close to the marines, supporting the main army (see Figure 3). Due to all these factors, we were able to win with 24 of the original 28 units surviving.

In Scenario 1, when testing ChaosNeuron, it took until the 6th generation for our system to win by a significant margin. In the game, our system and the opponent both sent their units towards the centre of the map. Both players formed a line with an equal number of units and some units behind, unable to fight since they are melee units. Our units in the back then quickly moved to the top of the line and surrounded the top units of the opponent's line. The opponent attempted the same technique at the bottom of the line, but did it at a slower rate. This meant our system was able to defeat the units at the top of the line, giving us superior numbers, which later won us the game. At the end, we won by the smallest margin of all our experiments, but were able to demonstrate a weakness nonetheless. In Scenario 2, the opponent spread its units in a v-shape, while ours were shaped in a line as they moved towards the centre. This meant that more of our units were in range to fire on the opponent's units when the groups first met. Then our units formed two groups, one to attack the top of the v-shape, and one to attack the bottom. Some of the opponent's unit were out of range to attack us, so we were able to overwhelm the units at the top and bottom. This technique caused us to win with still 8 of the 12 range units left.

In both of the scenarios tested, UTPABronScript left all of its units in the corner in a very tight group instead of moving to the centre. This seems to rely on facing an opponent that will move its units into the corner to avoid having a draw. In Scenario 1, we moved our units in a group towards the opponent's corner. As we approached, the opponent sent two of its units to explore the map. When we came in contact however, the other units did not react. Our group was able to destroy the two units, and then moved back to the corner. Again, as we approached only a few of the units came to fight, so we were able to win due to having more units fighting at the same time. In Scenario 2, the opponent had its units in the corner spread in a line. This weakness is difficult to describe due to the fast pace of the battle, but it appeared as though our units were able to fire first, destroying several of the scourges before they could do any damge to us. Our system also focused on attacking specific units more than the opponent, quickly decreasing its numbers which gave us the win.

Regarding Windbot in Scenario 1, both our system and it moved their units towards the middle of the map. In a similar manner to the ChaosNeuron test, our units quickly tried to surround the opponent's units. The opponent had some of its units moving around frequently in order to get into better positions, which allowed us to do more damage than it. Because we were able to surrond an opponent unit, we created an advantage in numbers and that allowed us to win.

The weaknees found in the ItClusters AI for Scenario 1 is quite simple. Both players moved towards the centre in a group. When the battle occured, equal numbers of units were attacking, and our system tried to surround one of the opponent's units. The opponent's weakness appeared to be the technique it used of pulling back hurt units. When units were hurt, the opponent would have them retreat and then come back to the fight, hoping we would start attacking units with higher health. Although this is a good idea, the problem was that we had a greater number of units attacking at one time, so some of the opponent's units were being attacked by several of our units. The weakness leading us to win Scenario 2 is difficult to describe despite winning with nine of the twelve ranged units. Both players moved their units to attack in a large clump, and after a couple of seconds we have a larger number of units, which caused us to win the battle. The best explanation we have is that our units were about twice as spread out when the battle occured. This meant that the ranged attack was unable to hit as many of our units at

the same time. We were able to destroy several of the opponent's self-destructing units before they could cause damage, but the opponent could not destroy ours because they were too spread out.

As these descriptions show, the ways how the different opponents can be beaten are rather different (even completely opposite). It should also be noted that the 2010 competition was the first StarCraft AI competition, so there were no AIs from "old" competitions that the designers could use to test their AIs. But this shows how important having testing systems like ours is!

Conclusion and future work

We presented an extension of the testing method of (Atalla and Denzinger 2009) for testing competition entries for the StarCraft AI competition. By adding more types of macros and aligning those macros with the real-time requirements of StarCraft we were able to identify weaknesses in all of the entries into the 2010 competition for different game scenarios. While (Atalla and Denzinger 2009) only claimed that the concept can be used for heterogeneous unit types, we demonstrated how this could be done.

Future work should be directed to integrate the economic side of StarCraft into our testing method, which will require dealing with changing numbers of agents (as new buildings and units are created). Also, our current method focusses on the first weakness it detects. Without fixing the weakness it is usually not possible to find other weaknesses. This is not always in the interest of a developer and we want to find ways how to keep the learner away from already detected problems.

References

Atalla, M. and Denzinger, J. 2009. Improving Testing of Multi-Unit Computer Players for Unwanted Behavior using Coordination Macros. In Proc. CIG-09, 355–362, IEEE.

Buro, M. 2007. 2007 ORTS RTS Game AI Competition. http://www.cs.ualberta.ca/∼mburo/orts/AIIDE07/, as seen on Apr. 24, 2011.

BWAPI: Setting up the Broodwar API. 2010. http://eis.ucsc.edu/StarCraftBWAPI, as seen on Apr. 24, 2011.

Chan, B.; Denzinger, J.; Gates, D.; Loose, K.; and Buchanan, J. 2004. Evolutionary behavior testing of commercial computer games. In Proc. CEC 2004, 125–132.

Denzinger, J.; Loose, K.; Gates, D.; and Buchanan, J. 2005. Dealing with parameterized actions in behavior testing of commercial computer games. In Proc. CIG-05, 51–58, IEEE.

StarCraft Participants. 2010. http://eis.ucsc.edu/StarCraft Participants, as seen on Apr. 24, 2011.

Weber, B. 2010. StarCraft AI Competition. http://eis.ucsc.edu/StarCraftAICompetition, as seen on Apr. 24, 2011.

Xiao, G.; Southey, F.; Holte, R.C.; and Wilkinson, D. 2005. Software Testing by Active Learning for Commercial Games. In Proc. AAAI'05, 898–903, AAAI Press.

Figure 1: Our units stay together

Figure 2: Opponent is spread out

Figure 3: Our units pick off the enemy