Co-Evolving Robocode Tanks

Robin Harper
Sydney, Australia
rharper2@bigpond.net.au

ABSTRACT

Robocode is a Java based programming platform where robot tanks, controlled by programs written in Java, compete. In this paper Grammatical Evolution is used to evolve Java programs to control a Robocode robot. This paper demonstrates how Grammatical Evolution together with spatial co-evolution in age layered planes (SCALP) can harness co-evolution to evolve relatively complex behaviour, including robots capable of beating Robocode's sample robots as well as some more complex human coded robots. The results of the co-evolution are similar to the results obtained by direct evolution against a range of human coded robots. This indicates that co-evolution alone is able to evolve robots of a similar standard to those evolved against graded human coded robots.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming – *program synthesis*

General Terms: Algorithms, Design, Experimentation.

Keywords: Grammatical Evolution, Genetic Programming, Robocode, Co-Evolution, SCALP.

1. What is Robocode

Robocode is a Java based programming game platform developed by IBM Alphaworks[1], where the aim is to program a robot battle tank to fight (and beat) other robot battle tanks. The programs controlling the tanks are written in Java and are executed as threads in the main program.

During the game the tank must navigate around its environment, avoiding the walls, bullets fired by the other tanks and (unless it has chosen to ram a tank) the other tanks. At the same time it must locate the other tank, anticipate its likely position (since the bullets take time to move) and try to hit it. Programs implementing simple strategies such as wall following or moving in circles while firing at an opponent are provided as sample programs to help the novice Robocode programmer get started. In addition there are multiple resources on the internet[2] illustrating advanced strategies such as pattern matching, gravity movement and wave surfing, to name a few. These allow robotic tanks of high complexity to be built. There is a Robocode league, where robots compete against each other with the view of taking league table honours.

2. Aim of this paper

Although the writing of the Java programs to control such robots is in itself an interesting task, the focus of this paper is to test the ability to evolve the controlling Java programs. That is - using genetic programming principles - is it possible to evolve a Java program to compete successfully against hand-coded robot tanks? In particular it looks at the ability to successfully harness co-evolution to derive interesting strategies with the view ultimately to evolving a tank that is competitive with a human coded tank. The current sophistication of the better human coded tanks is such that the evolved tanks require a high level of complexity if they are to be competitive. Unless the context makes it otherwise clear a reference to a Robocode tank or robot or tank is a reference to the Java program controlling the simulated Robocode tank.

3. Structure of this paper

This paper begins with a brief overview of the problem area (i.e. what is Robocode), discusses the form of the evolution attempted and the reasoning behind the design decisions taken. It then discusses the environment in which the evolution is to take place, that is what opponents are selected, how competitive co-evolution is attempted and how evolution versus human written Robocode tanks is attempted and finally it presents the results.

4. Brief Overview of a Robocode Tank

A tank consists of the tank body, a turret containing the gun, which can rotate with or separately from the body, and a scanner used to detect enemies that can rotate with or separately from the gun. A tank can move forward and/or turn. While turning its maximum forward speed is reduced. Each tank starts with a certain amount of energy. It loses energy each time it crashes (either into a tank or a wall) and each time it is hit (either by a bullet or another tank). It gains energy if one of its bullets hits an opposing tank. When it fires its gun, it can vary the amount of energy used to fire the bullet, the more energy put into the bullet the slower it travels but the more damage it does and the higher the energy gain to the firing tank (if it hits). Guns also need to cool down before they can be fired, the higher the energy of the bullet – the longer the gun takes to cool. Tanks can detect other tanks, their energy level, speed and direction. They cannot detect bullets nor can they directly detect the fact that a bullet has been fired, although the energy loss in an opposing tank is noticeable and provides an indication that the tank may have fired (or suffered an energy loss by other means). The structure of a Robocode tank program is explored later. The tanks fight in an empty arena of a fixed size, surrounded by walls. Typically the start position and orientation of the tanks is randomised for each round. A typical battle consists of a number of rounds. Although

[1] http://robocode.sourceforge.net/

[2] For example, http://robowiki.net/wiki/

GECCO'11, July 12–16, 2011, Dublin, Ireland.

there are various leagues, including for battles involving multiple tanks in the same battleground, this paper only looks at one versus one battles. The scoring system is discussed in section 7.5.

5. Previous Work

In [3] Eisenstein used a form of genetic evolution, based on a fixed size genome that encoded a table version of Leslie Kaelling's REX Language. Effectively the genome encodes a number of computation elements, which form the rows of the table. Each element can take as inputs either a value from the robot's sensors or from the output of other computation elements (referenced by their row in the table). Eisenstein's work had some success in evolving robots to win against the sample robots provided. However, he experienced difficulty in getting robots to actually fire their gun. Eisenstein hypothesises that this is because a miss costs the robot energy and therefore, early in evolution when a miss is likely, not firing is a good starting strategy. Principally because of this difficulty in getting randomised robots to fire, his attempt at using co-evolution to commence an "arms-race" was not successful. However, using a more traditional generational approach he was, amongst other things, able to evolve robots that could each beat the starter robot opponent they were evolved to compete against. He was less successful in his attempts to evolve a robot that could beat all of the five different starter robots he used. His best-reported robot was able to beat four out of these five robots, the robot it was not able to beat being the "tracker" robot[3].

In the same year another study was published relating to the use of Machine Learning in a Robocode robot [4]. This study looked at splitting up the required functions of a Robocode tank (i.e. movement, target acquisition, radar control and gun control) on the basis of a subsumption architecture. Reinforcement learning was used for target acquisition (unlike this paper, the robots were tested in multi-tank fights) a neural network was used to control the gun and, relevantly, genetic programming was used for the movement and radar control aspects of the robot. The module resolving conflicts between layers as well as wall avoiding, randomisation of movement and attempts at bullet dodging were all hand-coded. Although the robot itself was quite successful, the genetic programming part of the robot failed to improve over time and did little to help the over all competitiveness of the robot.

[16] used a more canonical Genetic Programming (GP) style system [11] to evolve a robot for the category of HaikuBots. HaikuBots are Robocode robots that are limited to four lines of code (although there are no length limits imposed on the lines). GP was used to evolve numerical expressions that were given as arguments to the four lines of code written in the shell program. Each line of code controlled one of the main actuators; namely the gun/radar, forward/backward movement, turn rate and the scanner. Firing the gun was controlled by "side-effects" in the numerical evaluation. Like Eisenstein it was noted that if only one adversary was included in the evolutionary run specialised strategies were evolved which did not generalise well to different opponents. However, where multiple adversaries were included in the runs this did lead to better generalisation and robot tanks able to fight previously unseen opponents. One of the bots took third place in the HaikuBot league in 2004. Limited success was reported with their attempts to use co-evolution to evolve tanks. Their belief as to the reasons for this is that as early generations evolved for idleness (conserving energy by not firing or moving and hitting walls) the genes responsible for movement and gun firing were lost – hampering the later evolution of more complex strategies.

[15] compares the use of NEAT and XCS to evolve controllers to control the scanning and targeting of Robocode tanks and [9] used evolutionary strategies to combine various hand programmed behaviours. Neither of the last two papers explored the use of co-evolution

[3] The tracker robot is one of the starter robots provided with Robocode. It tracks its opponent, moving close and firing when possible.

6. Structure of a Robocode tank program

The program to control a Robocode tank (a robot) is usually written in Java. It is executed as a separate thread that is activated from the main Robocode program. Each thread has a certain amount of time per "tick" of the competition to send its command instructions (i.e. what actuators it wants to activate of the tank) and these are then executed simultaneously. If a robot thread does not respond in time then it misses its turn. If no response is received for a period of time (e.g. the thread is stuck in a loop) the tank has its energy reduced to zero and is eliminated from the round.

The main run function of a Robocode tank thread should not exit. Accordingly, after any initialisation, it typically consists of an endless loop containing whatever instructions are relevant. As well as the main run() thread each robot receives event notifications. Relevant to this paper the following events can be received onScannedRobot, which is called when the robot scans an enemy robot (i.e. the scanner is pointing at an enemy robot), onHitRobot which is called if the robot is hit (or hits) an enemy tank, onHitByBullet which is called if the robot is hit by a bullet and onHitWall which is called when the robot hits a wall. Each of these event notifications is passed a data structure that contains information relevant to the notification e.g. OnScannedRobot is called with a ScannedRobotEvent parameter, which contains the velocity, heading and energy levels of the scanned tank.

The structure of typical robot therefore looks like this

```
public void run()
   {
   [Initialisation statements];
   while ( true )
    [Main Loop statements];
    }
  }
public void onScannedRobot(ScannedRobotEvent e)
   {
   [onScanned statements;]
   }
public void onHitByBullet(HitByBulletEvent e)
   {
   [onHitByBullet statements;]
   }
public void onHitRobot(HitRobotEvent e)
   {
   [onHitRobot statements];
   }
public void onHitWall(HitWallEvent e)
   {
   [onHitWall statements;]
   }
```

A very simple tank might therefore use the main thread to turn its gun/radar and the onScannedRobot event to fire its gun. Given the travel time of bullets such a tank is likely to hit a stationary tank but miss a moving one. Obviously human coded robots have additional functions and, often, additional classes to carry out such things as pattern matching, avoidance, strategies to aim ahead of a moving robot etc – but the structure of human designed programs will have all of the above elements.

7. Main elements of the evolutionary design

7.1 The grammar

In order to evolve a the Java program controlling a Robocode tank the process can be simplified slightly by taking the overall structure of the program outlined in section 6 as a given and focus on evolving the statements which appear in square brackets of that structure. While canonical Genetic Programming (GP) as introduced by Koza [11] might be the best known method of evolving programs a newer form of evolving programs, namely Grammatical Evolution (GE) [14], which uses a grammar based on a Backus Naur Form (BNF) grammar is, arguably, better suited to evolving programs that comply with the requirements of a fully fledged Java program.

7.2 Grammatical Evolution

GE is a form of GP that separates the underlying representation of each candidate solution (the genotype) from the representation (the phenotype). Associated with each GE problem is a BNF grammar that dictates the eventual form of all candidate solutions. The genotype (which is a bit string) is decoded by the grammar. All phenotypes begin as the start non-terminal of the grammar. The genotype is read from left to right. The bit string of the genotype, grouped into codons (typically 8 or 12 bits) is used to select between the different possible expansions of each non-terminal (again read in a left to right manner) in the current phenotype string. If the phenotype string fully expands to terminals the individual is valid. If the genotype terminates before the phenotype is fully expanded the individual is invalid.

7.3 Designing the BNF Grammar

7.3.1 - A quick note about the difference between Robots and AdvancedRobots

The Robocode Robot (which is designed to introduce Robocode to beginner tank designers) codes all movement and turning commands as blocking actions. That is, if part of the code tells the tank to move forward 10 units, the next line of that section of code will only be executed once the move has been carried out. This means that a Robocode Robot (as distinct from an AdvancedRobot) can't be instructed to turn and move at the same time in the one sequence of instructions (although the turn command could be executed in one of the event handlers and the move command in the main run() loop). An AdvancedRobot on the other hand has access to a group of "set" commands which can "set" the tank to move forward, turn etc and these are then executed when a blocking command is executed (or the execute() command is issued). As discussed later (section 8.1) attempts at evolving a Robot proved unsuccessful, so the AdvancedRobot was used.

7.3.2 - Commands to be included in the grammar

The following commands that could be sent to a Robocode tank were identified as being relevant:

Movement Related	Turn Related	Miscellaneous
setAhead(float)	setTurnRight(float)	setAdjustRadarForRobotTurn(bool)
setback (float)	setTurnLeft(float)	setAdjustRadarForGunTurn(bool)
Stop()	setTurnGunLeft(float)	setAdjustRadarForRobotTurn(bool)
Resume()	setTurnGunRight(float)	Fire(float)
	setTurnRadarRight(float)	execute()
	setTurnRaderLeft(float)	

The first three commands in the miscellaneous column control whether the radar/gun turn with or independently of the gun/tank.

7.3.3 - Functions

There are a number of functions which are globally valid and which return relevant information: getX() and getY(), which return a tank's position, getGunHeading(), getGunHeat(), getRadarHeading(), getBattleFieldWidth(), getBattleFieldHeight() and getEnergy() – all of which should be self explanatory.

Each of the events (passed to the OnEvent handlers) contain information relevant to that event.

7.3.4 - Additional Data Structures

Finally in an attempt to make the tanks as flexible as possible, they were given access to simple stacks, two global stacks and two stacks local to each of the event handlers. Although the standard Java stack structure was used, it was wrapped in functions to make it more "friendly" to evolution, i.e. if an attempt was made to peek at or pop from an empty stack, 0 was returned rather than an error. The impact of this change is include as "Actions" in the grammar (see section 7.3.5) the ability to push a value on to the relevant stacks and as a "globalVariable" the appropriate pop and peek function for the global stacks and as a "localVariable" the appropriate pop and peek functions from the relevant stacks. It was also decided to make available the previous event (i.e. in each event handler there is a local variable which has the value of the event in the previous call).

7.3.5 Putting it all together

From the Java code framework contained in section 6 it can be seen there are six places (each a "Statement") where evolved code is required namely: [Initialisation Statements], [Main Loop Statements], [onScanned Statements], [onHitByBullet Statements], [onHitByRobot Statements] and [onHitByWall Statements]. One could write a BNF grammar that can serve to expand these statements into Java code. However, since there are local variables unique to each of these Statements, each Statement will in effect have to be treated as a separate type. This has two relevant effects: 1) the BNF grammar is very long (since it has to have near identical rules for each of the six Statements listed above); and 2) because the non-terminals used to expand each of the Statements will be different it will prevent the GP style crossover of GE [5] from swapping expressions between event handlers.

To illustrate:

We might write the following BNF grammar extract:

```
Statements:-        Statement; Statements |
                    Statement;
Statement:-         if ( Bool ) { Statements }
                    else { Statements } |
                    Action
Action:-            turnAction |
                    moveAction |
                    setAction |
                    fireAction |…
…
turnAction:-        setTurnRight( Variable ) |
                    setTurnLeft( Variable ) | …
Variable:-          CompoundVariable |
                    globalVariable |
                    localVariable
CompoundVariable:- ( Variable + Variable ) |
                    (Variable – Variable) | ….
globalVariable:-    getX() |
                    getY() |
                    getHeading() ….
localVariable:-     ?
```

So as can be seen the possible expansions of localVariable are different depending on which type of Statement we are expanding. For instance in the onScanRobot Statements, localVariable might include e.getEnergy() – which returns the energy of the scanned tank, but in the onhitWall Statements, the event has no e.getEnergy() but only has e.getBearing() – the bearing of the wall which was hit. One solution is to have separate expansions for Statements for each of the five different parts of the code e.g. for the Main Loop:

```
MLStatements:-      MLStatement; MLStatements |
                    MLStatement;
MLStatement:-       if ( Bool ) { MLStatements }
                    else { MLStatements } |
                    MLAction
MLAction:-  MLturnAction | MLmoveAction…
```

And so on until we get to

```
MLLocalVariable:- [local variables for the Main Loop]
```

This is repeated for each on the OnScannedStatements, the OnHitByBulletStatemets etc.

Alternatively we can provide some means of allowing the grammar to determine which local variables the particular derivation of Statements should use. A Christiansen grammar such as that in [15] would be up to this task, but is probably more complex than required. A simpler and quite neat solution, using a similar methodology to Dynamically Defined Functions [6], was used. Key words were embedded in the grammar. By encoding, say, HITBYBULLETCODE when the local variables relating to the onHitByBullet event are available, it is a simple matter of allowing the genotype to decode the correct localVariables. What is more by allowing each part of the grammar to share the same non-terminals, GP-style typed sub-tree crossover is enabled across each of the different types of statements.

With GE a single genome can be used to decode the grammar resulting in a phenotype which is a valid java program containing each of the event handlers. By virtue of the key statements the expansion of "localVariables" is mapped to the correct set of local variables for each particular part of the program.

7.4 The Environment

Spatial Co-Evolution in Age Layered Planes (SCALP) [7] was used as the evolutionary environment to foster co-evolution. SCALP is a blend of the spatially separated co-evolutionary framework described in such works as [8] and [13] and the age-layering concepts introduced by Hornby [10]. A full description of the SCALP methodology is beyond the scope of this paper, but the following is a brief summary. A SCALP layer consists of a grid like layer of nodes – which wrap around, each node connected to its 4 orthogonal and 4 diagonal neighbours. See Figure 1.

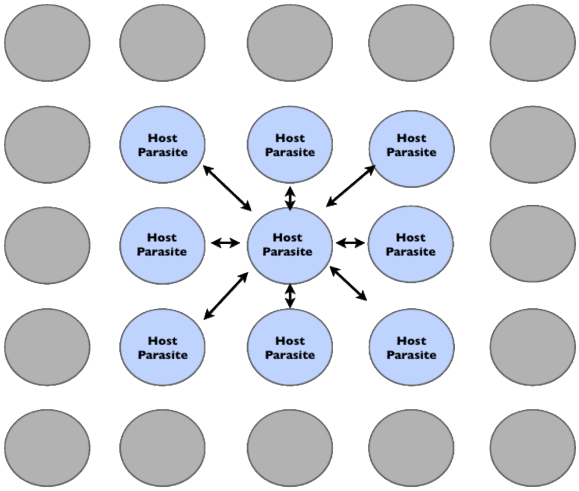

Figure 1 – A SCALP Layer

A creature (a host) exists on each node together with a parasite[4]. Each generation and for each node the host in that node competes with the parasite in the node and the parasites in the 8 surrounding nodes.

Hosts and parasites receive different scores. As explained in [2] this differentiation between the scores of parasites and hosts has at its foundation the dinner/life principle of [1]. The idea behind this principle is that for a predator it is less important that there exists prey which can outrun it – provided there is some prey it can catch. On the other hand for prey the fact that it can outrun some predators is of scant interest if it comes across the predator it can't outrun. In fact for many prey it is less important that it can outrun predators than it is that it can outrun the other prey.

The implementation of this in a SCALP environment means a host is given a score related to the sum of its battles with the

[4] The terms hosts and parasite are as used in [13] and [2] and are kept here for the sake of consistency, although in the case of this paper, we perhaps more intuitively, have two teams, say team "A" and team "B" and the host is a member of team "A" while the parasite is a member of team "B".

parasites, whereas a parasite is given a score solely related to its worst battle. As I discuss in section 8 I believe that this different method of scoring leads to the different strategies that emerged from the hosts and the parasites.

After each node has been evaluated the host and parasite allowed to occupy a particular node in the next generation is determined stochastically based on the scores of the surrounding hosts or parasites, as appropriate. 40% of the nodes are occupied not by the selected host but with a child of the host (the other parent is selected randomly from the surrounding nodes). Similarly with the parasites. Finally in 20% of the nodes the occupying host is mutated and in 10% of the nodes the occupying parasite is mutated.

The layered part of SCALP means that there are a number of these layers of nodes, stacked above each other. For each layer the hosts are tested against not only the parasites in their own node and the eight surrounding nodes but (other than in the case of the bottom layer) also with the node directly below them and the eight nodes surrounding that node. Candidates selected to stay in a node or to be used as parents for the child that will occupy the node are chosen from this extended neighbourhood. This allows successful hosts on lower layers to move up layers. Similarly with the parasites. Each layer has a genetic age limit, which increases as we move through the layers. The genetic age of a host/parasite is the number of generations that host or parasite has survived. A child inherits the genetic age of its oldest parent, incremented by one. If a host or parasite exceeds the maximum genetic age of a particular layer it is eliminated and replaced with the host (or parasite) in the node directly below it (or in the case of the bottom layer a freshly randomised individual). The idea behind this is that the lower levels serve as a "nursery" allowing code which has not evolved much to compete against other less evolved code and begin evolving towards its own "basin of attraction" without being overwhelmed by the more evolved code in higher layers (which is potentially trapped in a local maxima). The creation of random individuals in the lowest layer serves to continually introduce new genetic material into the system.

For the co-evolutionary part of this experiment both the hosts and the parasites were evolved. They were kept apart in the sense that there are two different populations which do not "inter-breed" (even though – since they both used the same grammar – they could). Where the system was used to try to evolve tanks to beat various human coded opponents, the parasites were those human coded tanks. Each of the different possible tanks was numbered, the parasite just being a number corresponding to one of the opponents. There was no "breeding" of parasites but mutation was possible, with the number (and therefore the tank) changing. More details of the tanks used and the way this was integrated with the layering of SCALP is discussed later (section 9).

7.5 Scoring

The Robocode system scores each participant in a fight. The score is based on a number of factors, there are 1) survivability points if the tank survives a round (i.e. wins); 2) points for bullet damage which related to the damage done by the tank's bullets; and 3) points for ramming which are awarded at twice the rate of points for similar damage done by a tank's bullets.

Although it appeared to make good sense just to use the points scored in each battle as the fitness score for a particular combatant, one side-effect of this was noticed when the system was run with human coded opponents. Because a tank gains energy when it hits an opponent – if you allow your opponent tank to hit you sometimes (but still ultimately win the round) then you will need to do more damage to it than if your opponent never hit you. This means that a tank which dodges all incoming fire and kills its opponent in short order will score fewer raw points than a tank which allows itself to be hit a few times and takes a long time to kill its opponent. With the human coded opponents it became apparent because the weaker human coded tanks were getting better scores than the human tanks that annihilated the hosts. Although this "oddity" of the scoring system may well actually be an interesting dynamic, for the purposes of the paper – in the second part (where tanks are evolved against human coded tanks) the percentage of the total points scored was used as the fitness function. Thus a tank which keeps its opponents score to a minimum (while killing it) will get a far higher percentage of the points scored that round than a tank that allows itself to be hit.

7.6 Setup

The experiments were run on a 15x16 SCALP grid (240 nodes), consisting of (up to) 4 Layers. The layer lives (the maximum genetic age allowed in the layer) were: [40,70,130,258]. Initialisation of individuals (hosts and, for co-evolution, parasites) was via Luke's PTC2 initialisation [12]. Each individual was given a unique name. Each generation a sub-directory was created and the phenotype of each host (and for co-evolution each parasite) was constructed and saved as a .java file in that sub-directory. The .java files were then batch compiled and the .class files created (this only took a second or two). In order to assess the fitness each of the requisite battles was scheduled (nine battles for each node on the first layer, 27 per node for each subsequent layer[5]) and farmed off to helper applications that would run the Robocode battle as a separate process and return the result. On an eight-core MacPro running six helper applications approximately 100 battles could be processed each minute (this slows down as the battles become more intense). A battle consisted of a number of rounds of a Robocode competition. Each round the robots are placed in a random place in the arena with a random orientation. Placement can give a large advantage to one of the opponents. For co-evolution each "battle" consisted of 5 rounds. Where the opponents were human-coded robots, in order to minimise the stochastic nature of battles, the battles consisted of 15 rounds.

8. Co-Evolution results.

8.1 Using the "Robot" Class

The initial experimental setup used the Robocode "Robot", rather than the "AdvancedRobot" – see section 7.3.1 for a brief discussion of the differences. With the "Robot" attempts at co-evolution were not particularly successful, but for different reasons than those experienced by [3]. Early on both the hosts and the parasites developed a strategy of spinning in place and firing every time the onScannedRobot() event handler was called. Although there were variations in the strength of the bullet, this appeared to be a very difficult local maximum for the system to escape from. Because of the blocking nature of movement calls any attempt to move during the main loop slowed the spin and

[5] Being the host (layer1) v 9 parasites (layer1), host (layer1) v 9 parasites (layer0) and host (layer0) v 9 parasites(layer1)

thus the frequency with which the onScannedRobot event handler was called and any movement in that event handler delayed the firing rate. It was possible for movement to occur in other event handlers and indeed some individuals who moved when onHitByBullet was called were observed, but overall the results were not satisfactory, with limited variation and no different strategies observed even over 100 generations. Although SCALP (and ALPS) is designed to obviate the need for multiple runs, five runs were carried out, with similar results.

An analysis of the human coded robots showed that many of the competitive robots and even the more interesting sample robots (such as spinBot which moves in circles and fires when it sees an opponent) all used the AdvancedRobot class.

8.2 Using the "Advanced Robot" class

Moving the system to use the "AdvancedRobot" class proved to be successful. Within a few generations robots emerged that moved and fired at each other. Within 20 or 30 generations different behaviour between hosts and parasites could be observed, presumably directly as a result of their different fitness functions (see section 7.5). The hosts very aggressively tracked and charged at their opponents (in a similar way to the sample human coded robot "ramfire" – see table 1), firing as they went, whereas the parasites tended to rely on moving in circles and firing at their opponents (in a similar way to the sample robot "spinBot" – see table 1). By about 60 generations, the parasites had developed a tactic of moving backwards, in a circular motion, away from the on coming hosts and firing at their opponent as it charged them. This is the same tactic that many human coded robots use to defeat robots using the ramFire strategy (the strategy works because the retreating robot can be pretty sure where the attacking robot will be, but the attacking robot has to guess the direction the fleeing robot will turn in). After about generation 195 the hosts abandoned their ramfire strategy and had developed a circular movement, with strafing shot strategy. By generation 245 (when the simulation was stopped) the hosts and parasites were involved in some quite complex looking battles, although from a qualitative perspective the behaviour of the robots still appears less "purposeful" than human coded robots. One interesting behaviour that was noted is that when the evolved robots (the hosts in this case) were just about to run out of energy they slowed their turn and fired off one last shot directly at the opponent – allowing them to win if their opponent was disabled – an all or nothing approach.

After the run was complete, in an attempt to achieve some empirical data as to how well the co-evolution was working, the individuals in each generation were made to compete in a knockout competition and the top 24 individuals (the "Generational Winners") were retained. This was the first time that hosts had competed directly against hosts and parasites against parasites. It was noted that in each generation the Generational Winners were a mixture of parasites and hosts, the number of parasites or hosts in a particular set of Generational Winners varied depending on the generation, presumably reflecting whether the parasites or hosts were stronger in that particular generation. The Generational Winners were allowed to compete against each other in a knockout competition. This was carried out five times. Although the top 10 individuals in the tournament varied each time the tournament was run (with closely matched robots, such is the random nature of Robocode) in all cases the top 10 consisted only of parasites. In each of the 5 knockout tournaments the robots which made up the top 10 robots were dominated by parasites from the last 15 generations (the youngest parasite to make any of the top 10 lists was from generation 191). This provides evidence that the robot populations (both the hosts and parasites) were continuing to evolve throughout the process.

8.3 Assessing against the Sample robots

These top 24 individuals from each generation were then assessed against each of the sample robots listed in table 1.

TABLE 1 – DESCRIPTION OF ROBOCODE SAMPLE ROBOTS

Robot	Behaviour
Crazy	Moves randomly
RamFire	Attempts to ram and fire on an opponent.
SpinBot	Moves in circles.
Tracker	Tracks, moves close then fires
TrackFire	Sit still firing at an opponent.
VelociRobot	Varies speed.
Walls	Moves along the walls (which makes it quite difficult to hit).

Although none of the co-evolved robots had seen these sample robots before, they performed well against them. Unlike the tournament situation (where the co-evolved robots competed with each other), there was a mixture of hosts and parasites in the robots that preformed best against the sample robots. Using the percentage of points scored in a battle (so 50% indicates matched opponents), then, of the robots tested, the robot that had the highest average of scores across all categories was a host in generation 243.

TABLE 2 – SAMPLE CO-EVOLVED ROBOTS (AT VARIOUS GENERATIONS) .V. ROBOCODE SAMPLE ROBOTS

Opponent	Gen 243(H)	Gen 244(P)	Gen 193(H)
Crazy	86%	88%	78%
RamFire	74%	80%	52%
Spin Bot	82%	84%	75%
Tracker	89%	85%	54%
TrackFire	78%	77%	40%
VelociRobot	79%	81%	69%
Walls	38%	26%	58%

Walls did appear to give the co-evolved robots some difficulty – presumably because they had not faced that type of movement before. Earlier generation hosts (which exhibit more dramatic chase and kill behaviour) could defeat Walls; they trapped it in a corner and rammed it to death. However, as can be seen from table 2, the behaviour which allowed them to defeat walls led to them losing to Trackfire (a stationary robot) because their chasing behaviour meant that they did not dodge bullets on their way and lost out on the shoot-out with Trackfire. To defeat Trackfire a robot needs to dodge at least some of its bullets.

8.4 Assessing against competitive human coded robots

As previously mentioned there are many Robocode tournaments and a lot of the robots that compete in the tournaments are available for download. A selection of robots were downloaded (see table 3), and the co-evolved robots where assessed against them. In selecting the robots for download it was decided to avoid adaptive robots (that is those that adapted to their opponents behaviour) since it was difficult to enable a repeatable result against such robots (it was likely they would perform better as the number of rounds increased).

TABLE 3 – DESCRIPTION OF HUMAN CODED ROBOCODE ROBOTS USED

Robot	Comments and ranking[6]
Guess Factor	Uses a predictive gun – current ranking about 783 in the robot leagues[7]
Peryton	A robot that is marked as "exemplary" and was competitive in the earlier days of Robocode, but is not so competitive now.
Squigbot	Like Pertyon, very competitive once, not so much now.
Sparrow	A micro bot (limited size) ranked 434th
Duelist	Ranked 483rd.
NanoLauLectrick TheCannibal	Ranked 367th
Tron 2.0.2	Ranked about 148th
Aspid	Ranked about 167th
Cigaret	A mini robot (limited size), ranked 107th

Table 4 shows some of the results for the better individuals against these hand coded robots. As can be seen ones that do well against one type of robot, might not do as well as against some of the others. Robots did evolve (through co-evolution alone) which could beat two of the easier ones and some evolved that were competitive against robots that were ranked in the high 400s.

TABLE 4 – RESULTS OF CO-EVOLVED ROBOTS .V. SELECTED ROBOTS

Generation and (H)ost or (P)arasite	214 (P)	243 (P)	200 (H)	195 (H)	Best score by any robot
GuessFactor	53%	41%	21%	24%	60%
Peryton	54%	59%	22%	26%	59%
SquigBot	24%	20%	50%	43%	50%
Sparrow	30%	35%	9%	9%	43%
Duelist	20%	29%	6%	6%	44%
Cannibal	15%	17%	35%	45%	45%
Tron	9%	7%	22%	25%	25%
Aspid	11%	8%	16%	20%	20%
Cigaret	12%	8%	15%	12%	16%

Significantly the co-evolution results are better than any previous result reported where co-evolution has been attempted.

[6] http://darkcanuck.net/rumble/Rankings?version=1&game=roborumble as at 14 January 2011

[7] http://robowiki.net/wiki/GuessFactor_Targeting_Tutorial

9. Evolution against human coded robots

One concern was that the grammar chosen might limit the robots that could be evolved. For instance it was noted that the human coded robots tended to make extensive use of trigonometric functions (to calculate angles of fire) and random number generation (for non-predictable movement). Although, in theory, approximations to the trigonometric functions might be evolved – this could potentially place undue demands on the evolution of human competitive robots.

It was decided to try and see if direct evolution against the human coded robots would yield different results from the co-evolution – it was hoped to evolve robots that were competitive with the more advanced robots coded by humans.

Since the principle behind SCALP is that the earlier layers are "nursery" layers, it was decided to implement a system whereby the parasites (now hand coded robots) increased in difficulty as the layers increased. This would help to prevent a flat fitness function (i.e. all the hosts being beaten by the better human coded robots and scoring very few points). The first layer was reserved for the sample robots and GuessFactor. After that an attempt was made to select robots of increasing sophistication and difficulty. The next layer also used Sparrow, Peryton and Squigbot in addition to those previously used.. The third layer included Duelist and Cannibal and each subsequent layer allowed any of the hand coded robots as parasites. One parasite could mutate into any other parasite, subject to the layer limits. Initially the first layer was "seeded" with the co-evolved hosts, although subsequently when the first layer was re-generated (after generation 40 in the system used) randomly created hosts (by PTC2 initialisation [12]) were used.

TABLE 5 – RESULTS OF INDIVIDUAL EVOLVED ROBOTS

Opponent	Gen 440	Gen 389	Gen 330	Best score by any robot
Crazy	86%	79%	83%	90%
RamFire	67%	58%	81%	86%
Spin Bot	72%	79%	66%	90%
Tracker	83%	60%	81%	94%
TrackFire	64%	64%	78%	87%
VelociRobot	74%	79%	77%	85%
Walls	65%	52%	25%	77%
GuessFactor	51%	53%	46%	66%
Peryton	55%	20%	44%	56%
SquigBot	28%	41%	35%	48%
Sparrow	25%	27%	22%	41%
Duelist	19%	15%	22%	48%
Cannibal	21%	32%	27%	42%
Tron	8%	9%	15%	31%
Aspid	8%	9%	13%	25%
Cigaret	15%	9%	27%	27%

Interestingly the "seeded" layer was out evolved after about 80 generations. The entire system was allowed to evolve for 440 generations. Once again it was noted that robots that performed well against some robots tended to perform worse against others. This was entirely expected. Encouragingly robots emerged that could

defeat every one of the sample robots as well as GuessFactor and Peryton on a regular basis emerged. Robots capable of defeating Squigbot almost 50% of the time also emerged, but such robots tended to perform, relatively, poorly against Pertyton and Walls (and often the trackers). This was the same with the co-evolved robots. The robots which preformed best against the later robots (Tron/Aspid and Cigaret) were only capable of winning about 6 out of 30 fights and also had poorer performance against the "easier" robots. It appeared that in the time the system was given it was finding it hard to find robots that could generalise against the harder opponents. Given this is probably the hardest part of the challenge – it is not surprising. Table 5 shows some of the better sample robots that illustrate the points made above. The first column is one of the best robots against the sample robots, Peryton and GuessFactor. The second column contains one of the better robots that could achieve a decent score against Squigbot while still beating the sample robots. The third column shows the best against Cigaret (note how poorly it performs against Walls and Peryton) and the final column shows the highest score any robot achieved against each opponent (each score in this column is from a different robot). By way of comparison Peryton scores 20-26% against Aspid and Cigaret, Squigbot about 30-35%.

10. Conclusions and Future Work

The system was able to successfully co-evolve robots of sufficient complexity to win battles against unseen human coded robots that historically used to top the league but not robots that are currently ranked in the top 500 of the league. However, the successful co-evolution of robots that are able to beat human coded robots that were, at one time, leading contenders is very encouraging. This is particularly the case as successful co-evolution in this domain does not appear to have been previously reported. The predator/prey scoring system used with SCALP did appear to lead to different strategies being adopted by the two competing populations and it is believed helped the successful co-evolution of relatively robust strategies (i.e. robots able to compete against unseen opponents). Although the co-evolutionary system (unlike the direct evolutionary system) did not find a strategy that would beat all the sample robots (the unusual movement of "Walls" defeated it) it may just have required more time. Of some concern was the fact that even with direct evolution against a number of selected human coded robots a strategy was not developed that could beat any of the robots used in this paper that are in the top 500 of the league. It may be that, once a certain sophistication is reached, robots need to be trained against specific opponents and not always the ones they are weakest against – as that may lead to cycling rather than improvement. It might also be the case that the grammar itself was insufficient to allow sufficiently complex robots to be evolved; it may be that access to random numbers (to allow random movement patterns) and trigonometric functions (to allow easier firing calculations) is required. One oddity that was noticed is that the Java programs did not appear to suffer from bloat. The reason for this is unclear and tracking this down might suggest other ways to allow the robots to evolve even more complexity.

REFERENCES

[1] Dawkins, R. and Krebs, J (1979) Arms races between and within species. In Proceedings of the Royal Society of London. Series B, Biological Sciences, volume 205 of the Evolution of Adaptation by Natural Selection, pages 489-511.

[2] Folkert De Boer, Master's Thesis, The role of speciation in Spatial Co Evolutionary Function Approximation. Utrecht University 2007.

[3] Eisenstein, J 2003,'Evolving Robocode TankFighters', Technical Report 2003-026, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, 2003.

[4] Gade, M, Knudsen, M,Kjær, RA,Christensen, T,Larsen, CP, Pedersen, MD & Andersen, JSK 2003, Applying Machine Learning to Robocode, Aalborg University, Aalborg, Denmark.

[5] Robin Harper and Alan Blair, A Structure Preserving Crossover in Grammatical Evolution, Proceedings of the 2005 IEEE Congress in Evolutionary Computation, Vol.3, pp. 2537-2544, IEEE Press, 2-5 September 2005.

[6] Robin Harper and Alan Blair, Dynamically Defined Functions in Grammatical Evolution, Proceedings of the 2006 IEEE Congress on Evolutionary Computation, Vol.3 pp 2537-2544. IEEE Press, 2-5 September 2005.

[7] Robin Harper, Spatial co-evolution in Age Layered Planes (SCALP), 2010 IEEE Congress on Evolutionary Computation (CEC 2010), IEEE Press, 18-23 July.

[8] Daniel Hillis, Co-evolving parasites improves simulated evolution as an optimization procedure. Artificial Life II Santa FE Institute Studies in the Sciences of Complexity, Vol. X pp. 313-324, Addison-Wesley, February 1990.

[9] Hong, J, & Cho,SB2004,'Evolution of Emergent Behaviours for Shooting Game Characters in Robocode', Congress on Evolutionary Computation 2004, vol. 1, pp. 634-638

[10] Gregory Hornby, ALPS: The Age-Layered Population Structure for Reducing the Problem of Premature Convergence, GECCO 2006, July 8-12, 2006, Seattle Washington.

[11] John R. Koza, Genetic Programming, On the programming of computers by means of natural selection, 1992, The MIT Press. p.93.

[12] Sean Luke, Two Fast Tree-Creation Algorithms for Genetic Programming, IEEE Transactions on Evolutionary Computation, 4(3), pp. 274-283, September 2000.

[13] Melanie Mitchell (2006) Co evolutionary learning with spatially distributed populations. In G.Y. Yen and D.B. Fogel (editors) Computational Intelligence: Principles and Practice. New York: IEEE.

[14] Michael O'Neill and Conor Ryan, Grammatical Evolution, IEEE Transactions on Evolutionary Computation, 5(4), pp. 349-358, August 2001.

[15] A. Ortega, M. de la Cruz, M. Alfonseca. Christiansen grammar evolution: grammatical evolution with semantics. IEEE Trans. Evol Comput. 11(1), 77-90 (2007).

[16] David Nidorf, Luigi Barone, A Comparative Study of NEAT and XCS in Robocode, WCCI 2010 IEEE World Congress on Computational Intelligence July 18-23 CCIB Barcelona Spain, page 86.

[17] Yehonatan Shichel, Eran Ziserman and Moshe Sipper, 2005,'GP- Robocode: Using Genetic Programming to Evolve Robocode Players', Proceedings of the 8th European Conference on Genetic Programming, vol. 3447, pp. 143–154.