Evolving Effective Micro Behaviors in Real-Time Strategy Games

Siming Liu, Student Member, IEEE, Sushil J. Louis, Member, IEEE, Christopher Ballinger, Student Member, IEEE,

Abstract—We investigate heuristic search algorithms to generate high quality micro management in combat scenarios for real-time strategy games. Macro and micro management are two key aspects of real-time strategy games. While good macro helps a player collect more resources and build more units, good micro helps a player win skirmishes and battles against equal numbers and types of opponent units or win even when outnumbered. In this paper, we use influence maps and potential fields as a basis representation to evolve short term positioning and movement tactics. Unit micro behaviors in combat are compactly encoded into fourteen parameters. A genetic algorithm evolves good micro behaviors by manipulating these fourteen parameters. We compared the performance of our evolved ECSLBot with two other state of the art bots, UAlbertaBot and Nova on several skirmish scenarios in a popular RTS game StarCraft. The results show that the ECSLBot tuned by genetic algorithms outperforms UAlbertaBot and Nova in kiting efficiency, target selection, and fleeing. Further experiments show that the parameter values evolved in one scenario work well in other scenarios and that we can switch between pre-evolved parameter sets to perform well in unseen scenarios containing more than one type of opponent unit. We believe our representation and approach applied to each unit type of interest, can result in effective micro performance against meme and ranged opponents and provides a viable approach towards complete RTS bots.

Index Terms—RTS game, genetic algorithm, micro, influence map, potential field.

I. INTRODUCTION

REAL-TIME STRATEGY (RTS) games have become a popular platform for computational and artificial intelligence (CI and AI) research in recent years. RTS players need to gather resources, build structures, train military units, research technologies, conduct simulated warfare, and hopefully, defeat their opponent. All of these factors and their impact on in-game, time-critical decisions are essential for a player to win a RTS game. In RTS communities, RTS players usually divide their decision making into two separate levels of tasks called macro and micro management, as shown in Figure 1. Macro is long term planning, like generating good build orders in the early game, technology upgrading, and scouting. Good macro management helps a player build a larger army or economy or both. On the other hand, micro is the ability to control a group of units in combat or other skirmish scenarios to minimize unit loss and maximize damage to opponents. We decompose micro management into two parts: tactical and reactive control [1]. Tactical control addresses the overall positioning and movement of a squad of units. Reactive control focuses on controlling a specific unit to move, fire, and flee in combat. This paper investigates heuristic search algorithms to find winning tactical and reactive control for skirmish scenarios as indicated by the dotted square in Figure 1. Micro management of units in combat aims to maximize damage given to enemy units and minimize damage to friendly units. Common micro techniques in combat include concentrating fire on one target, withdrawing seriously damaged units from the front of the battle, and kiting1 your units to take advantage of the enemy units’ attack-distance limitation. We are interested in generating competitive micro as part of an RTS game player that outperforms an opponent with the same or greater number of enemy units. In the future, we plan to incorporate these results into the design of more complete RTS game players.

Fig. 1: Typical RTS AI levels of abstraction. Inspired by a figure from [2].

Spatial maneuvering is an important component of combat in RTS games. We applied a commonly used technique called Influence Maps (IMs) to represent terrain and enemy spatial information. IMs have been widely used to attack spatial problems in video games, robotics, and other fields. An IM is a grid placed over a virtual world with values assigned to each square by an IM function described in [3]. Figure 2 shows an IM which represents a force of enemy units and the surrounding terrain in StarCraft: Brood War, our simulation environment. A unit IM is computed by all enemy units in the game. In our experiments, greater cell values indicate more enemy units in the cell and more danger to friendly units. In addition to the position of enemy units, terrain is

S. Liu, S. Louis, and C. Ballinger are with the Department of Computer Science and Engineering, University of Nevada, Reno, Reno NV, 89557-0148 USA (e-mail: simingl@cse.unr.edu; sushil@cse.unr.edu; cballinger@cse.unr.edu).

1Kiting refers to making your units pull back and shoot repeatedly in order to keep your units at an optimum distance from your target.
another critical factor for micro behaviors. For example, kiting enemy units near a wall is not a wise move. We then use another IM to represent terrain in the game world to assist micro management. We combine the two IMs and use this battlefield (or map) spatial information to guide our AI player’s positioning and reactive control. In our research, we use search algorithms to find optimal IM parameters that help specify high quality micro behaviors of units in combat scenarios. Once learned, the parameters that define an IM generalize well to other game maps and is one reason IMs are a useful representation for spatial information.

Fig. 2: An IM representing the game world with enemy units and terrain. The light area on the bottom right represents enemy units. The light area surrounding the map represents a wall.

While good IMs tell us where to go, good unit navigation tells our units how best to move there. We use Potential Fields (PFs) in our research to control a group of units navigating to particular locations on the map. PFs are used widely in robotics research for coordinating movement for multiple entities and are often applied in video games for the same purpose. In our research, we apply two PFs to coordinate units’ movement and use two parameters to specify each PF.

The long term goal of our research is to create a complete human-level RTS game player and this paper tackles one aspect of this problem: finding effective micro management for winning small combat scenarios. Our approach is to evolve (off-line) two sets of parameters for each unit type that we want to control. One set of parameters specifies behavior against melee units, the other set specifies behavior against ranged units. During a real-time skirmish, ECSLBot switches between these two parameter sets, and their corresponding behaviors, based on whether our ECSLBot controlled unit’s current target is a melee unit or a ranged unit. In this paper, we apply this approach to Vultures, a fast but fragile Terran unit.

Several challenges needed to be addressed in our research. First, how do we represent reactive control like kiting, target selection, and fleeing? Second, since the parameters for IM, PF, and reactive control are related, how do we tune these variables on different types of units and scenarios? Third, how well does our micro bot perform in selected sample scenarios compared with other state of the art bots? Finally, do the evolved behaviors generalize to scenarios not previously encountered?

To investigate these issues, we compactly represent micro behaviors as a combination of two IMs, two PFs, and a set of reactive control variables. In earlier work, we had compared the quality and robustness of movement behaviors produced by genetic algorithms (GAs) and two types of Hill Climbers (HCs). The results showed that our HCs were quick but unreliable, while GAs were slower but reliably found high quality solutions[4]. Going with higher quality, in this paper, we use GAs to evolve combinations of IM, PF, and reactive control parameters that lead to winning micro behaviors. Since StarCraft has become a popular platform for RTS AI research, we created our skirmish scenarios in StarCraft and used GAs to search for winning micro behaviors in these scenarios. We subsequently compared the performance of micro behaviors produced by our GAs with two state of the art StarCraft bots, UAlbertaBot [5] and Nova [6], on both training and testing scenarios. We chose these bots because their source code was available and they performed well in prior competitions. The results show that Nova performs well on kiting behavior against melee units, while UAlbertaBot does well against ranged attack units. Our ECSLBot tuned by GAs performs well against both melee units and ranged units in a variety of scenarios.

The remainder of this paper is organized as follows. Section II describes related work in RTS AI research and common techniques used in RTS micro research. The next section describes our RTS research environment. Section IV explains the specifics of our methodology. Section V presents results and compares solutions produced by our methods with two state of the art StarCraft bots. Finally, the last section draws conclusions and discusses possible directions for future work.

II. RELATED WORK

Typically, industry RTS AI developers create RTS AI not so much for beating opponents as to entertain and tutor users. Industry AI employs techniques such as finite state machines, rule based systems, and scripting [7], [8]. Some industry AIs may cheat. On the other hand, academic RTS AI research focuses on using learning or reasoning techniques to win an RTS game and reach human competitive performance. Our research falls in the academic category. Much work has been done in applying different techniques to build RTS AI players in academia. For example, some work has been done in our lab on co-evolving macro and robust build orders in WaterCraft [9]. Ontañón et al. used real-time case-based planning (CBP) in an RTS game Wargus [10]. Weber and Mateas presented a data mining approach to strategy prediction by learning from StarCraft replays [11]. Churchill et al. adopted Alpha-Beta search approach from board games for RTS combat scenarios of up to eight versus eight units [12]. This paper focuses on the work related to using IMs and PFs for spatial reasoning and unit movement. Miles et al. applied IMs to evolve a LagoonCraft RTS game player [13]. Sweetser
et al. developed a game agent designed with IMs and cellular automata, where the IM models the environment and helps an agent make decisions in their EmerGEnt game [14]. They built a flexible game agent that responds to natural phenomena and user actions while pursuing a goal. Bergsma et al. used IMs to generate adaptive AI for a turn based strategy game [15]. Su-Hyung et al. proposed a strategy generation method using IMs in the strategy game Conqueror. He applied evolutionary neural networks to evolve non-player characters’ strategies based on the information provided by layered IMs [16]. Avery et al. worked on co-evolving team tactics using a set of IMs, guiding a group of friendly units to move and attack enemy units based on the opponent’s position [17]. Their approach used one IM for each entity in the game to generate different unit movement. However, this method does not scale well to large numbers of units. For example, if we have two hundred entities, the population cap for StarCraft, we will need to recompute two hundred IMs every update. This could be a heavy load for our system. Preuss et al. used a flocking based and IM-based path finding algorithm to enhance group movement in the RTS game Glest [18], [19]. Raboin et al. presented a heuristic search technique for multi-agent pursuit-evasion games in partially observable space [20]. In this paper, we use an enemy units position IM combined with a terrain IM to gather spatial information and guide our units in producing winning micro behaviors for RTS games.

Potential fields have also been applied to AI research in RTS games. Most of the prior work in PFs is related to unit movement for spatial navigation and collision avoidance [21]. This approach was first introduced by Khatib in 1986 while he worked on real time obstacle avoidance for mobile robots [22]. The technique was then widely used in avoiding obstacles and collisions, especially in multiple unit scenarios with flocking [23], [24], [25]. Hagelbäck et al. applied this technique to AI research within an RTS game [26]. They presented a Multi-Agent Potential Field based bot architecture in the RTS game ORTS [27] and incorporate PFs into their AI player at both tactical and unit reactive control level [28]. We have also done some prior work in PFs [4], [29] and use two PFs for group navigation in our work.

Reactive control, including individual unit movement and behavior, aims at maximizing damage output to enemy units and minimizing the damage to friendly units. Common micro techniques in combat include fire concentration, target selection, fleeing, and kiting. Uriarte et al. applied IMs for kiting, frequently used by human players, and incorporated kiting behavior into his StarCraft bot Nova [6]. Gunnerud et al. introduced a CBR/RL hybrid system for learning target selection in given situations during a battle [30]. Wender et al. evaluated the suitability of reinforcement learning algorithms to micro manage combat units in RTS games [31]. The results showed that their AI player was able to learn the selected tasks like “Fight”, “Retreat”, and “Idle” during combat. We scripted our reactive control behaviors with a list of unit features represented by six parameters. Each set of parameters influences reactive control behaviors including kiting, targeting, fleeing, and movement.

III. RTS RESEARCH ENVIRONMENT

In our field, we believe a suitable RTS research environment should be popular, open source, and speed adjustable. StarCraft is one of the most popular RTS research platforms due to the StarCraft: Brood War Application Programming Interface (BWAPI) framework and the AIIDE\(^3\) and CIG\(^4\) StarCraft AI competitions [32]. BWAPI provides an interface which allows our program to interact with StarCraft game data through code instead of keyboard and mouse. However, StarCraft was designed for human players and as such has some limitations for evolutionary computing approaches. Researchers have thus developed RTS games such as SeaCraft, Wargus, and ORTS designed specifically for scientific research. In our work, we run all of our experiments on the popular platform, StarCraft, which allows us to compare our work with our peers.

A. StarCraft and Bots

StarCraft is one of the most well known RTS games with a huge player base and numerous professional competitions all over the world. The game has three different but well balanced races that players can choose from: Terran, Protoss, and Zerg. Thanks to the popularity of the StarCraft and recent StarCraft AI tournaments, many groups have been using it as a platform for their research. Typically, researchers develop RTS AI players called “bots” which are capable of playing StarCraft. In our research, we apply heuristic search algorithms to generate effective micro behaviors and compare the micro performance of our ECSLBot with two other state of the art bots: UAlbertaBot and Nova. The bots we used in this paper are listed below:

- **UAlbertaBot**: Developed by Churchill from the University of Alberta. UAlbertaBot won the AIIDE 2013 StarCraft competition as a Protoss player.
- **Nova**: Developed by Uriarte from Drexel University. Nova was ranked number 7 on the AIIDE 2013 StarCraft competition.
- **SCAI**: The passive StarCraft AI was used as our baseline in evaluating the micro performance of other bots.
- **ECSLBot**: Developed by us and currently only does micro, using parameters generated by our approach.

We are aware that UAlbertaBot is mainly a Protoss player and optimized for using Protoss units. However, UAlbertaBot is also capable of playing Terran and Nova was specifically optimized by Uriarte for kiting a Terran unit “Vulture”, so we select Vulture to evaluate our method. Another reason for selecting a ranged unit Vulture was that comparing to melee units, ranged units usually need more micro operations and could benefit more from good micro behaviors in a RTS skirmish.

The micro logic of UAlbertaBot is handled by two components called MeleeManager and RangedManager for all types of units rather than a separate component for each specific unit type. This means that UAlbertaBot uses the same logic for all military units and ignores the differences between units. For

\(^3\)http://www.StarCraftAICompetition.com

\(^4\)http://cilab.sejong.ac.kr/se_competition/
example, both Vulture and Dragoon are ranged attackers and can “kite” or “hit and run” against melee units, but they should kite differently based on their unique weapon cool down times. Nova uses IMs for unit navigation and kiting when controlling Vultures.

B. Scenarios

The first step of this research was building an infrastructure within which to run our bot. We designed a series of customized StarCraft maps by using StarEdit, a free tool provided by Blizzard Entertainment to build custom scenarios. As explained earlier, we want ECSLBot to learn to control a specific type of Terran unit (Vulture) to fight against different types of enemy units. More specifically, we evolve Vulture kiting behavior against melee enemy units and target selection and fleeing against ranged enemy units. Melee and ranged units are the two broad types of units in RTS games like StarCraft. After learning how to play against different types of enemies in two training scenarios, we test the generalizability of learned behaviors on eight previously unseen testing scenarios. Finally, we compare our ECSLBot, Nova, and UAlbertaBot against each other using Vultures. Players in RTS games are usually not able to access complete state information because of the “fog of war”. The “fog of war” influences longer-term planning for distant units, while we are focused on short-term planning for units in close proximity. Furthermore, good human players will start a skirmish only when they already have enemy and terrain information through scouting. Therefore, we allow all bots to access complete state information for all scenarios in this research. In another word, there is no “fog of war” in our scenarios.

1) Training Scenarios: Human players usually use different micro behaviors against melee units than they use against ranged units. Therefore, we evolve our bot against exemplars of these two broad types of enemy units, separately. In our experiments, ECSLBot learns how to control Vultures to defeat melee enemy units on a scenario containing 5 friendly Vultures and 25 opponent Zealots (a Protoss melee unit) as shown in Figure 3a. We call this training scenario, $Train_1$ and the parameters evolved on this scenario, $P_m$. The goal of this scenario is eliminating as many enemy Zealots as possible. Kiting efficiency is important in this type of combat and will be evolved by our GAs.

Our second scenario was created for fighting against ranged attack units. We call this scenario, $Train_2$, and the parameters evolved here, $P_r$. Figure 3b shows that our ECSLBot controls 5 friendly Vultures positioned at the top left to fight against 6 enemy Vultures positioned at the bottom right. Positioning and target selection become key contributors in this scenario. Both $Train_1$ and $Train_2$ run for 2500 frames or until one side is eliminated. Finally, once these parameter sets are evolved, ECSLBot switches between these two sets based on whether its target type is ranged or melee. This switching happens in real time during a game.

2) Testing Scenarios: We evolve micro behaviors for fighting against melee and ranged enemy units in the previous two training scenarios. However, we are interested in evaluating the generalizability of our evolved behaviors on scenarios never encountered before. Therefore, we created eight new testing scenarios with more units, mixed types of opponent units, and different terrain. We expect ECSLBot evolved on two simple training scenarios to perform similarly in other unseen situations because our evolved parameters represent a range of behaviors that are relatively position and terrain independent and simply switching between parameter sets takes into account the two broad opponent types in RTS games. We first test our evolved ECSLBot on the two test scenarios shown in Figure 4. Here, friendly Vultures spawn at the bottom left of the map instead of top left. Enemy units spawn at the top of the map and are split into two groups. These two scenarios ($Test_1$ and $Test_2$) test whether ECSLBot is able to adapt when the initial positions of friendly units and enemy units changes.

Next, we considered four scenarios where we change the number of units controlled by our bots. We evolved ECSLBot for controlling five Vultures against enemy units. We now investigate how ECSLBot performs when controlling more than five Vultures. Figure 5 shows four scenarios ($Test_3$, $Test_4$).
Train against each other. This scenario contains no obstacles like for this comparison where all three bots control five Vultures each other with identical units. A new scenario was designed also interested in how they perform when competing against different types of enemy units controlled by SCAI, we were io where our bots control different number of Vultures against SLBot, UAlbertaBot, and Nova on a variety of testing scenar-

eighteen Zerglings and ten Hydralisks were a good match for of the three bots. Too few and all three bots killed all the were chosen to show the difference in micro performance that the numbers and types of units used in the above scenarios (Zerg). These units have different melee damage and different ranged unit types respectively, from a different StarCraft Race (Zerg). These units have different melee damage and different weapons range when compared to Zealots and Vultures. Note that the numbers and types of units used in the above scenarios were chosen to show the difference in micro performance of the three bots. Too few and all three bots killed all the opponent units, too many and all three bots’ units died. Eighteen Zerglings and ten Hydralisks were a good match for ten Vultures.

3) Head-to-head Scenario: After we compared our EC- SLBot, UAlbertaBot, and Nova on a variety of testing scenar- ios where our bots control different number of Vultures against different types of enemy units controlled by SCAI, we were also interested in how they perform when competing against each other with identical units. A new scenario was designed for this comparison where all three bots control five Vultures against each other. This scenario contains no obstacles like Train1 and Train2.

IV. METHODOLOGY

In our scenarios, the micro bot attempts to defeat the opponent by eliminating enemy units while minimizing the loss of friendly units. A secondary objective is to do this as quickly as possible. Our scenarios contain two StarCraft unit types: Vulture and Zealot. A Vulture is a Terran unit with a ranged attack weapon, low hit-points (easy to destroy), and fast movement. On the other hand, a Zealot is a Protoss unit with a melee weapon, high hit-points (hard to destroy), and slow movement.

A. Influence Maps and Potential Fields

We compactly represent micro behaviors as a combination of two IMs, two PFs, and a set of reactive control parameters. The IM generated from enemy units combined with the terrain IM tells our units where to go and PFs are used for unit navigation. The unit IM and the terrain IM are functions of the weights and ranges of enemy units and untraversable terrain (walls and obstacles). Since computation time depends on the number of IM cells, we use a cell size of 32 × 32 pixels. The entire map consists of a 64 × 64 grid of such 32 × 32 pixel cells. Note that as enemy units move, the unit IM changes. The sum IM therefore also changes and no matter what the actual Starcraft map we play on and where on that map opponent units are positioned, the sum IM indicates vulnerable positions to attack as well as positions not to attack. The Algorithm 1 described below chooses on specific location to attack based on the current sum IM. We first find the lowest value IM cell that contains an enemy unit. Call this Cellt. This cell denotes the enemy that the IM indicates is most separated from the rest of the enemies. The algorithm then finds the IM cell with the lowest value within the Cellt’s immediately adjacent cells. We choose the first hit in case there are multiple cells found. We call this new IM cell, Cellnt. The algorithm then chooses Cellnt as the location to first move to and from which to then launch the attack. In case Cellnt is far from the Cellt, ECLBot may perform less well resulting in a lower fitness. The IM parameters that result in poor performance should be quickly eliminated by the GA.

Equation 1 shows a typical PF function where Force is the potential force on the unit, d is the distance from the source of the force to the unit. c is the coefficient and e is the exponent applied to distance and used to adjust the strength and direction of the vector force.

\[
\text{Force} = cd^e
\]
Fig. 5: Testing scenarios with ten friendly Vultures and different types of enemy units. All maps used in this paper are the same size of $64 \times 64$. 

Algorithm 1 Squad Targeting and Positioning Algorithm

\begin{verbatim}
Initialize TerrainIM, EnemyUnitIM, SumIM;
Cell$_t$ = the lowest value cell that contains an enemy unit on SumIM;
target = the enemy unit in Cell$_t$;
Cell$_{nt}$ = the cell with the minimum cell value and the smallest distance to target on SumIM;
movePos = Cell$_{nt}$.getPosition();
squad.moveTo(movePos);
if squad.getPosition() \approx movePos then
    squad.attack(target);
end if
\end{verbatim}

We use two PFs of the form described by Equation 1 to control the movement of units. Each PF calculates one force acting on a unit. The two potential forces in our game world are:

- **Attractor**: The attraction force is generated by the unit’s destination - the unit is attracted to its destination. This force is inversely proportional to distance.
- **Repulsor**: This keeps friendly units moving towards the destination from colliding with each other. It is usually stronger than the attractor force at short distances and weaker at long distances.

Each PF is determined by two parameters, a coefficient $c$ and an exponent $e$. Therefore, we use four parameters to determine a unit’s PFs:

$$PF = c_a d^{-ea} + c_r d^{-er}$$  \hspace{1cm} (2)

where $c_a$ and $e_a$ are parameters of the attractor force, $c_r$ and $e_r$ for the friend repulsor force. These parameters are then encoded into a binary string for our algorithms.

**B. Reactive Control**

Besides the group positioning and unit movement, reactive control behaviors must be represented in a way that our algorithms can process. In our research, we considered three reactive control behaviors: kiting, target selection, and fleeing frequently used in real games by good human players. We first developed and implemented parameterized algorithms for kiting (Algorithm 3) and target selection (Algorithm 2). We then used the genetic algorithm to evolve the values of these parameters for specific friendly unit and enemy unit types. These algorithms do not depend on position. The parameters affect target prioritization, relative kiting location, and determine when to flee based on our unit’s health. Once evolved, we expect to be able to use our parameterized micro behaviors tuned by genetic algorithms on any map with any distribution of friendly and enemy units. Note however, that we will need to evolve parameters for every pair of friendly and enemy unit types. This will take time but is feasible for most RTS games which usually implement less than twenty
(20) unit types. Figure 6 shows the six parameters used in these algorithms and Table I explains the details and purpose of each parameter (variable).

Fig. 6: Variables used to represent reactive control behaviors. The Vultures on the left side of map are friendly units. Two Vultures on the right are enemy units.

- **Target Selection**: Concentrating fire on one target, or switching to a nearby enemy. In Algorithm 2, each of our units selects the nearest enemy unit as a possible target, \( t_{\text{closest}} \). Within a distance \( R_{nt} \) (an evolved parameter as shown in Table I) from \( t_{\text{closest}} \), our unit will choose an actual target based on the following prioritized list:
  1. \( t_{\text{lowhp}} \), the lowest hit-point enemy unit below the evolvable threshold: \( HP_{ef} \).
  2. \( t_{\text{focus}} \), the enemy unit being targeted by the most friendly units within \( R_{nt} \) relative to \( t_{\text{closest}} \).
  3. \( t_{\text{closest}} \), the nearest enemy unit.

- **Kiting**: Also known as “hit and run”. This behavior is especially useful in combat where our units have a larger attack range than enemy units. In Algorithm 3, our unit moves towards and attacks the target whenever unit’s weapon is ready. unit will start kiting if the distance between unit and target is less than \( D_k \) and our unit’s weapon is not ready (for example, because it is in cool down). The unit moves back to a kitingPosition computed by a function getKitingPositionFromIM() in Algorithm 3, line 6. \( D_{kb} \) is the number of cells away from target cell. If \( Cell_i \) is the IM cell containing our target, getKitingPositionFromIM() finds a neighboring IM cell with the lowest value. We set this new IM cell to \( Cell_i \). We then repeat this process with the new \( Cell_i \), \( D_{kb} \) times. The algorithm then chooses \( Cell_i \) as the kitingPosition to move towards.

- **Flee**: Temporarily repositioning to the back of our forces away from the front line of battle when our units have low hit-points. \( HP_{fb} \) controls this “fleeing” behavior.

We encoded a candidate solution into a 60-bit string. The detailed representation of IMs, PFs, and reactive control parameters are shown in Table I. Note that the sum IM is derived by summing the enemy unit IM and terrain IM so it does not need to be encoded. When the game engine receives a candidate solution, it decodes the binary string into corresponding parameters according to Table I and directs friendly units to move to a location specified by Algorithm 1 and then attack enemy units. The fitness of this candidate solution at the end of each match is then computed and sent back to our GA. Once units engage an enemy unit, Algorithm 2 takes over to perform reactive control behaviors like target selection and kiting.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( W_U )</td>
<td>5</td>
<td>Enemy unit weight in IMs.</td>
</tr>
<tr>
<td>( R_{U} )</td>
<td>4</td>
<td>Enemy unit influence range in IMs.</td>
</tr>
<tr>
<td>( W_T )</td>
<td>5</td>
<td>Terrain weight in IMs.</td>
</tr>
<tr>
<td>( R_{T} )</td>
<td>4</td>
<td>Terrain influence range in IMs.</td>
</tr>
<tr>
<td>( e_a )</td>
<td>6</td>
<td>Attractor coefficient.</td>
</tr>
<tr>
<td>( e_f )</td>
<td>6</td>
<td>Repulsor coefficient.</td>
</tr>
<tr>
<td>( e_a )</td>
<td>4</td>
<td>Attractor exponent.</td>
</tr>
<tr>
<td>( e_f )</td>
<td>4</td>
<td>Repulsor exponent.</td>
</tr>
<tr>
<td>( S_t )</td>
<td>4</td>
<td>The waiting time to move after each firing. Used for kiting.</td>
</tr>
<tr>
<td>( D_k )</td>
<td>5</td>
<td>The distance from the target that our unit start to kite.</td>
</tr>
<tr>
<td>( R_{nt} )</td>
<td>4</td>
<td>The radius around current target. Other enemy units within this range will be considered to be a new target.</td>
</tr>
<tr>
<td>( D_{kb} )</td>
<td>3</td>
<td>The distance, in number of cells, for our unit to move backward during kiting. This is a parameter to getKitingPositionFromIM() in Algorithm 3.</td>
</tr>
<tr>
<td>( HP_{ef} )</td>
<td>3</td>
<td>The hit-points of our units, under which unit will flee.</td>
</tr>
<tr>
<td>( HP_{fb} )</td>
<td>3</td>
<td>The hit-points of nearby enemy units, under which target will be assigned.</td>
</tr>
<tr>
<td>Total</td>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>

C. Fitness Evaluation

We evolve the behaviors for fighting against melee units and ranged attack units separately. Our first fitness evaluation...
maximizes the damage to enemy units, minimizes the damage to friendly units, and minimizes the game duration in scenarios with ranged attack enemy units. In this case, a unit remaining at the end of game will contribute 100 to its own side. The fitness of an individual will be determined by the difference between the number of friendly units and the number of enemy units at the end of each game. For example, suppose three friendly Vultures and one enemy Vulture remain at the end of the game, the score will be \((3-1) \times 100 = 200\) as shown in the first term of Equation 3. Negative fitnesses when the number of enemy units is greater than the number of friendly units were not allowed to reproduce and typically stopped appearing after three generations. The detailed evaluation function to compute fitness against ranged units \((F_r)\) is:

\[
F_r = (N_F - N_E) \times S_u + (1 - \frac{T}{\text{MaxT}}) \times S_t
\]

where \(N_F\) represents how many friendly units remained, \(N_E\) is the number of enemy units remaining, \(S_u\) is the score for saving a unit (100) as defined above. Since Equation 3 is used in the scenario \(\text{Train}_2\) which contains only one type of unit (Vulture), \(S_u\) is sufficient to represent score of any unit. The second term of the evaluation function computes the impact of game time on score. \(T\) is the time spent on the whole game, the longer a game lasts, the lower is \(1 - \frac{T}{\text{MaxT}}\). \(S_t\) in the function is the weight of time score which was set to 100. Maximum game time is 2500 frames for experiments using this training scenario \((\text{Train}_2)\), approximately one and a half minutes at normal game speed. We took game time into our evaluation because “timing” is an important factor in RTS games. Suppose combat lasts one minute. This might be enough time for the opponent to relocate backup troops from other places to support the ongoing skirmish thus increasing the chances of our player losing the battle. Therefore, combat duration becomes a crucial factor that we want to take into consideration in our evaluation function. We kept the evaluation function \(F_r\) simple, while encapsulating all three main objectives. Our evaluation function leads ECSLBot to learn micro behaviors that maximize the damage to enemy units, minimize the damage to friendly units, and minimize the game duration.

In the melee training scenario, \(\text{Train}_1\), good players can be expected to maximize damage and destroy all opponents by hitting well. To reflect this bias towards damage we increase the weight given to destroying an enemy unit and reduce the weight for losing a friendly unit. Therefore, in \(\text{Train}_1\), where we want to see how many Zealots can be eliminated by Vultures during 2500 frames, we add 200 to the score for destroying an enemy Zealot while losing a friendly Vulture will subtract 150, therefore, the second melee specific fitness function \((F_m)\) is:

\[
F_m = N_E \times DS_{ET} - N_F \times DS_{FT}
\]

where \(N_F\) represents how many enemy units were killed, \(N_E\) is the number of friendly units being killed. \(DS_{ET}\) and \(DS_{FT}\) are the destroy scores for the types of unit being killed as defined in StarCraft. We apply this fitness function in experiments dealing with \(\text{Train}_1\) where we want to evaluate how fast our bots can eliminate melee attack enemy units. Although this equation does not specifically deal with time, we have 25 zealots in this scenario and the faster you can destroy a Zealot, the more Zealots can be destroyed. This implicitly drives evolution towards parameters that lead to faster elimination of enemy units. \(\text{Train}_1\) also runs for 2500 frames.

Summarizing, ECSLBot learns how to fight melee units in \(\text{Train}_1\) shown in Figure 3a using fitness function \(F_m\). ECSLBot learns how to fight against ranged units in the \(\text{Train}_2\) shown in Figure 3b using fitness function \(F_r\). After training and learning to handle both melee and ranged units, ECSLBot simply switches between the two sets of learned parameters, \(P_m\) and \(P_r\), according to the current target enemy unit type (melee or ranged). Note that we update 3 IMs for \(P_m\) and 3 IMs for \(P_r\) in every frame.

\[D. \ Genetic \ Algorithm\]

We used a CHC based GA in our experiments [33], [34]. CHC stands for Cross generational elitist selection, Heterogeneous recombination and Cataclysmic mutation. CHC selects the \(N\) best individuals from the combined parent and offspring populations \((2N)\) to create the next generation after recombination. Early experiments indicated that our CHC GA worked significantly better than the canonical GA on our problem.

Following prior experiments in our lab, we set the population size to 20 and ran the GA for 30 generations. The probability of crossover was 88% and we used CHC selection. We also used bit-mutation with 1% chance of each individual bit flipping in value. SCAI was used to control the opponent force in our evaluations. Standard roulette wheel selection was used to select chromosomes for crossover. CHC being strongly elitist, helps to keep valuable information from being lost if our GA produces low fitness children. These operator choices and GA parameter values were empirically determined to work well.

\[V. \ Results \ and \ Discussion\]

We used StarCraft’s game engine to evaluate our evolving solutions. In order to increase the difficulty of game play and because this is used in professional game play, the behavior of the StarCraft game engine was set to be non-deterministic for each game. In this case, some randomness is added by the game engine thus affecting the probability of hitting the target and the amount of damage done. This randomness is restricted to a small range so that results are not heavily affected. Non-determinism does not impact some scenarios such as Vultures against Zealots significantly, because, theoretically Vultures can “kite” Zealots to death without losing even one hit-point. But the randomness may have an amplified effect in other scenarios. To mitigate the influence of this non-determinism, individual fitness is computed from the average scores in five games. Furthermore, our results are collected from averaged scores over ten runs, each with a different random seed. Early experiments showed that the speed of game play affects outcomes as well. The definition of game speed in BWAPI is the wait time between two consecutive
frames. A number of 0 in game speed indicates that frames are executed immediately with no delay. We set our games to a slower speed of waiting 10 milliseconds between any two frames to reduce the effect of the randomness. Our experiment environment was a workstation with Intel Xeon E5-1620 CPU and 6 GB memory. Given the above experimental setup, the following subsections describe our results.

A. Training ECSLBot

Scenario $Train_1$ as shown in Figure 3a evaluates the efficiency of kiting behavior against melee attack units. $F_m$ as defined in Equation 4 is used as our evaluation function in this scenario. Figure 7 shows the average scores of ECSLBots running on this kiting scenario. We can see that the maximum fitness in the initial population is as high as 3217. However, the average of maximum fitness increases slowly to 3660 at generation 30. This results tell us that our GAs can quickly find a kiting behavior to perform “hit and run” against melee attack units while trading off damage output. Our ECSLBot trades off well between kiting for safety and kiting for damage to enemy.

Fig. 7: Average score of ECSLBot versus generations on scenario $Train_1$. X-axis represents time and Y-axis represents fitness by the fitness function $F_m$.

Besides performance against melee attack units, we are also interested in performance against ranged attack units. In this case, positioning and target selection become more important than kiting because the additional movement from kiting behavior will waste enemy damage output while avoiding enemy attack. We used our GAs to search for effective micro behaviors using the same representation as in the previous scenario. However, we changed our fitness evaluation function to $F_r$ as shown in Equation 3 to maximize killing of enemy units, minimize the loss of friendly units, and minimize combat duration. Figure 8 shows the average score of the evolving ECSLBots in scenario $Train_2$. The average maximum fitness found by GAs is 336, which means 3 friendly units remained at the end of the game and all enemy units were eliminated. Considering that the Vulture is a vulnerable unit and easily dies, 3 Vultures saved after a skirmish is, we believe, good performance.

We are interested in the differences in evolved parameters for the two training scenarios - against melee attack units and ranged attack units. Table II lists the details of optimal solutions in different scenarios. Videos of all learned micro behaviors can be seen online. We would like to highlight two findings in these results. The first concerns the learned optimal attack route in the scenario against six Vultures as shown in Figure 9. The IM parameters evolved by the GA and our control algorithms 1, lead to a gathering location at the left side of the map to move toward before the battle. Our ECSLBot then commands the five Vultures to follow this route to attack enemy units. The result is that only three of the enemy units are triggered in the fight against our five Vultures at the beginning of the fight. This group positioning helped ECSLBot minimize the damage taken from enemy units while maximizing damage output from outnumbered friendly units. Although we describe a behavior that is specific to this scenario, the IM parameters tend to guide our units to such vulnerable positions with respect to enemy forces located anywhere on any map. As performance in never-before-seen testing scenarios shows, using an IM helps generalizability of learned behaviors. This is detailed in Section V-C where our forces move to a location that the IM indicates is a location where enemy forces are less concentrated.

Fig. 8: Average score of ECSLBot over generations in scenarios $Train_2$. X-axis represents time and Y-axis represents fitness by the fitness function $F_r$.

The second interesting finding is that different micro behaviors are learned by ECSLBot in different scenarios. The result shows that our ECSLBot kited heavily against Zealots, but seldom move backward against ranged attack units. The

5http://www.cse.unr.edu/~simingl/publications.html
values of our parameters reflect this behavior. Table II shows the parameter values found by our GAs in the two training scenarios. We can see that $S_t$ (the first parameter in the reactive control section) is 1 frame in the scenario against melee attack units, which means a Vulture starts to move backward right after every shot. On the other hand, $S_t$ is much bigger (12 frames) against ranged attack units. This is because our units will gain more benefit after each weapon firing by standing still and firing again as soon as possible rather than moving backward immediately against ranged attack units.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>IM</th>
<th>PF</th>
<th>Reactive Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Train$_1$, 25 Zealots</td>
<td>3</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>Train$_2$, 6 Vultures</td>
<td>16</td>
<td>13</td>
<td>20</td>
</tr>
</tbody>
</table>

B. Comparing ECSLBot with State of the Art Bots in Training Scenarios

Next, we investigate the performance differences among our ECSLBot (using $P_m$ and $P_r$ evolved by GAs), UAlbertaBot, and Nova. ECSLBot switches between $P_m$ and $P_r$ depending on the current target type. We used UAlbertaBot and Nova to control the same number of Vultures (5) against 25 Zealots in training scenario Train$_1$. Table III shows the results for all three bots versus the baseline SCAI over 30 runs on the two training scenarios. We can see that the UAlbertaBot performed poorly against melee attack units. This seems to be mainly because UAlbertaBot uses the same logic for all its units and the logic is optimized only for Protoss units. It eliminated only 3.33 Zealots on average in each game, while losing all of its Vultures. Note that UAlbertaBot was designed to play as Protoss. On the other hand, Nova’s performance is good. Nova killed 20.03 Zealots and lost only 0.3 Vultures on average per run. This is because Nova has hard coded and tuned logic specifically for Vultures and is optimized to control Vulture kiting behavior against melee attack units. We then tested ECSLBot on scenario Train$_1$. The results show that ECSLBot got the higher score on average over 30 runs. 20.2 Zealots being killed in one match on average, while losing only 0.2 Vultures. Visually, ECSLBot and Nova seem to have very similar kiting behavior and performance and statistically, the difference in performance is not significant at $p = 0.79$ using the t-test. All significance tests reported in this paper are based on independent two-sample student's t-test, assuming unequal variance and equal sample sizes. The significance level is $\alpha = 0.05$. Considering all the comparing contains three datasets, the likelihood of rejecting the null hypothesis when it's true increases. We then applied the bonferroni correction and set the significance level to $\alpha = 0.05 / 3 = 0.0167$.

Table III shows the results from all of our three bots tested in scenario Train$_2$. All the bots run 30 times against SCAI. This time, both UAlbertaBot and Nova perform poorly. UAlbertaBot loses all 30 games against 6 Vultures, killing 2.67 enemy Vultures on average in each game, while losing all of its units. Nova performed slightly better than UAlbertaBot with 2 wins and 28 losses out of 30 runs. However, ECSLBot outperformed both the others with 60% winning rate. 5.2 enemy Vultures were eliminated and 1.8 friendly Vultures survived on average in each run. This is statistically significantly different at $p = 6.04 \times 10^{-7}$ using the t-test on the number of Zealot killings between ECSLBot and Nova. This result indicates that kiting against ranged units is not as effective as kiting against melee units. Positioning and target selection become more important than kiting in such scenarios. UAlbertaBot and Nova did not optimize micro behaviors in all scenarios and performed poorly in these cases. Note however, that ECSLBot needs about 21 hours to evolve either $P_m$ or $P_r$.

C. Comparing ECSLBot with State of the Art Bots in Testing Scenarios

Since we evolved micro behaviors for ECSLBot in two training scenarios, we investigate how the corresponding parameter sets perform in scenarios never encountered before. Therefore, we compared our evolved ECSLBot to Nova and UAlbertaBot on eight testing scenarios as shown in Figure 4 and Figure 5. Table IV shows the results of the three bots playing against SCAI on all eight scenarios. The table provides standard deviations.

The first two testing scenarios Test$_1$ and Test$_2$ as shown in Figure 4 are somewhat similar to the two training scenarios Train$_1$ and Train$_2$ with the only change being the units’ positions. The purpose of these two scenarios is to evaluate how well ECSLBot’s parameters work when the positions of both friendly units and enemy units are changed. The results show that ECSLBot and Nova are still good at kiting Zealots in these new scenarios. 17.4 Zealots were killed by Nova and 19.8 Zealots were killed by ECSLBot during 2500 frames on Test$_2$. This testing scenario performance is similar to performance on the training scenarios. UAlbertaBot only killed 2.63 Zealots and lost all 5 Vultures on Test$_2$. However, UAlbertaBot performs better when destroying split enemy Vultures on Test$_2$. 5.97 Vultures were killed and 2.0 Vultures survived on average over thirty runs. ECSLBot performs similar to UAlbertaBot on Test$_2$. Nova performs badly on fighting against ranged attack units which is similar to the results on training scenarios. Eleven out of thirty matches are lost against six split Vultures. The difference in performance between UAlbertaBot and ECSLBot in Test$_1$ is not statistically significant at $p = 0.25$. On Test$_2$, the difference in performance between ECSLBot and Nova is statistically significant at $p = 2.92 \times 10^{-5}$.

In order to test the micro performance of ECSLBot for controlling different numbers of Vultures, other than five we trained on, we conducted experiments where bots control ten Vultures to fight against different types of enemy units including ranged units, melee units, mixed units, and different terrain. The results on scenarios Test$_3$ and Test$_4$ show that all three bots do well in destroying enemy units. However, ECSLBot saved 8.1 and 6.4 out of 10 friendly Vultures in two scenarios. Nova saved 6.93 and 3.06 Vultures and UAlbertaBot saved only 3.97 and 1.13 Vultures. The difference
in performance on saved units between ECSLBot and Nova is statistically significant at \( p = 5.35 \times 10^{-5} \) on Test 3. These results show that ECSLBot outperformed Nova and UAlbertaBot both on training and testing scenarios against ranged opponents. The ECSLBot seems to find a good balance in the trade-off between concentrating fire and kiting.

On scenarios Test 5 and Test 6, Nova outperformed the other two bots on scenarios with and without obstacle as shown in Table IV. Nova destroyed on average 37.6 and 32.6 Zealots in the two scenarios. ECSLBot performs close to Nova and destroyed 34.87 and 30.8 Zealots. In scenario Test 5, the difference in performance between Nova and ECSLBot is statistically significant at \( p = 2.43 \times 10^{-8} \). The difference between UAlbertaBot and the other two bots is statistically significant. This results show that in terms of kiting efficiency against melee units, ECSLBot performs as well as Nova and better than UAlbertaBot on both training and testing scenarios. Moreover, ECSLBot is able to use the same set of parameters to perform well despite an untraversable obstacle in the center of the map by using the terrain influence map learned during training.

Since we test our bots on scenarios against melee units and ranged attack units independently, we next investigated how our bots fight against enemy units containing both melee and ranged units. We test our bots on scenario Test 7 where the enemy is composed of a mix of eight Zealots and eight Vultures as shown in Figure 5. Test 8 looked at a completely different set of units from the Zerg race in StarCraft. Eighteen Zerglings and ten Hydralisks, Zerg melee and ranged units, never encountered by ECSLBot during training made up the opponents in scenario Test 8. Results on these two scenarios are displayed in Table IV and show that Nova performs as well as ECSLBot. Both eliminate all enemy units and more than six friendly Vultures survive. The differences in performance of saved units between ECSLBot and Nova on both Test 5 and Test 6 are not statistically significant. However, the difference between UAlbertaBot and the other two bots is statistically significant. ECSLBot performs well in these mixed opponent type scenarios by switching between \( P_m \) and \( P_r \) depending on target type. For example, if the current target is a Zergling which is a melee unit, ECSLBot uses \( P_m \), the parameters evolved in the Train 1 melee scenario for the IM, PF, and reactive control. As soon as the target changes to an enemy Vulture, ECSLBot will use ranged reactive control parameters that refer to the ranged attack IM and PF (\( P_r \)). This mechanism performs well even when fighting against enemy units not seen during training by simply comparing the weapon attack ranges between the target unit and the friendly unit to determine ranged or melee and thus which parameter set to use.

### D. Comparing ECSLBot with State of the Art Bots in Head-to-head Scenario

We have compared the performance of three bots playing against SCAI on two training and eight test scenarios and the
results show that ECSLBot works well on all scenarios while Nova and UAlbertaBot perform well on some and perform badly on others. However, what are the results when they play against each other? To answer this question, we set up our last set of experiments with the Head-to-head scenario described in Section III-B3. Each bot plays against the other two bots thirty times with identical units. Since we evolved parameters for Vultures which are a ranged unit, we used Vultures as the unit type in this scenario. ECSLBot thus used $P_r$ for control against UAlbertaBot and Nova. The result shows that ECSLBot beats Nova but is defeated by UAlbertaBot. ECSLBot won only 8 and lost 22 out of 30 matches against UAlbertaBot. The replays show that ECSLBot’s positioning micro is driven by training against SCAI and does not generalize well to other bots. Thus although ECSLBot’s representation and control algorithms evolve to generalize over opponent positions, terrain, and opponent types, they are specifically evolved to beat SCAI.

Therefore, we evolved another set of parameters directly against UAlbertaBot and applied ECSLBot with this set of parameters against UAlbertaBot and Nova on the Head-to-head scenario. Table V shows the detailed results among all the bots. We can see UAlbertaBot wins 24 matches, draws 5, and loses 1 against Nova. After examining game replays for these games, we found that Nova’s micro kites against any type of opponent units. However, as our experiments with the scenario $Train_2$ showed, kiting too much against the same ranged attack units actually decreased micro performance. UAlbertaBot on the other hand, disabled kiting when fighting against the equal weapon range units and defeated Nova easily. Similarly, ECSLBot defeated Nova on all 30 games without a loss or draw. The average number of units surviving was 3.37 which is higher than UAlbertaBot’s 2.33. The final comparison was between ECSLBot versus UAlbertaBot. The results show that ECSLBot wins 17 matches, draws 1 match, and loses 12 matches out of 30. ECSLBot performed quite well on this scenario against the other bots.

Although our approach can evolve good micro against specific opponents while generalizing over maps and unit types, for the longer term, we are investigating co-evolutionary approaches to evolving micro that is effective against a variety of opponents. Co-evolutionary approaches have been shown to work well in board games and other video games and provide a promising computational intelligence approach to robust behavior evolution.

### Table V: Head-to-head scenario over 30 matches.

<table>
<thead>
<tr>
<th>Match</th>
<th>Win</th>
<th>Draw</th>
<th>Lose</th>
<th>Units Remaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>UAlbertaBot vs Nova</td>
<td>24</td>
<td>5</td>
<td>1</td>
<td>2.33</td>
</tr>
<tr>
<td>ECSLBot vs Nova</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>3.37</td>
</tr>
<tr>
<td>ECSLBot vs UAlbertaBot</td>
<td>17</td>
<td>1</td>
<td>12</td>
<td>0.30</td>
</tr>
</tbody>
</table>

**Conclusion and Future Work**

Our research focuses on generating effective micro management: group positioning, unit movement, kiting, target selection, and fleeing in order to win skirmishes in real time strategy games. We compactly represented micro behaviors as a combination of influence maps, potential fields, and reactive control parameters in a 60 length bit-string. Early work showed that evolutionary algorithms perform better than other heuristic search algorithms [4]. Here, we test the micro performances of our evolved ECSLBot against two state of the art bots, UAlbertaBot and Nova on several skirmish scenarios in StarCraft. We designed eight testing scenarios in which bots need to control a number of Vultures against different types of enemies, to evaluate micro performance. The results show that our genetic algorithm quickly evolves good micro for handling melee attack units and ranged attack units.

ECSLBot performs well by switching parameter values depending on the currently targeted unit. Simple parameter switching can be done in real-time and ECSLBot thus achieves good micro performance. The results also indicate that Nova is highly effective at kiting against melee attack units but performs poorly against ranged attack units. UAlbertaBot, the AIIDE 2013 champion, performs poorly against melee attack units but is excellent against ranged attack units in our scenarios. Compared to the UAlbertaBot, we generate unit specific micro behaviors instead of a common logic for all units. With the right parameters, our ECSLBot beats both UAlbertaBot and Nova.

Our representation leads to good generalization over different numbers of units, different initial positions, and different terrain obstacles. However, evolving against a specific AI (SCAI) means that ECSLBot performs well against SCAI, but not as well against other AIs. We plan to investigate co-evolutionary approaches while using the same representation in order to evolve robust micro performance. Moreover, the use of two fitness functions in our research to evolve micro behaviors against melee and ranged enemy units may lead to extra experiments for game developers.

We are also interested in micro management with mixed unit types instead of a single type of unit, the Vulture - one simple approach is to evolve or co-evolve parameter sets for each unit type in an RTS game. This can be done off-line as we have shown. In addition, we want to integrate the usage of unit abilities (abilities are different from weapons) like the Terran Ghosts EMP pulse, into our micro behaviors.

**Acknowledgment**

This material is based in part upon work supported by the National Science Foundation under grant number IIA-1301726 and by the Office of Naval Research grants N00014-12-1-0860 and N00014-12-C-0522.

**References**


IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES, SPECIAL ISSUE ON REAL-TIME STRATEGY GAMES, JUNE 2016


**Siming Liu** (S’13) is currently working towards his Ph.D. degree in the Evolutionary Computing System Laboratory, Department of Computer Science and Engineering at the University of Nevada, Reno, NV, USA. He is working on using evolutionary approaches to computer game AI and human behavior modeling. Major application areas include computer game AI and human computer interfaces. Dr. Louis is an associate editor of the IEEE transactions on Computational Intelligence and Artificial Intelligence in Games and was co-general chair of the 2006 IEEE Symposium on Computational Intelligence and Games. You can visit his website, http://www.cse.unr.edu/~simling/, for more.

**Sushil J. Louis** (M’01) is a full professor of Computer Science at the University of Nevada, Reno and director of the evolutionary computing systems laboratory. He works in the areas of evolutionary computation and its applications in search, optimization, and machine learning. His current interests are in evolutionary approaches to computer game AI and human behavior modeling. Major application areas include computer game AI and human computer interfaces. Dr. Louis is an associate editor of the IEEE transactions on Computational Intelligence and Artificial Intelligence in Games and was co-general chair of the 2006 IEEE Symposium on Computational Intelligence and Games. You can visit his website, http://www.cse.unr.edu/~sushil/, for more.

**Christopher A. Ballinger** (M’12) is currently working towards his Ph.D. degree in the Evolutionary Computing System Laboratory, Department of Computer Science and Engineering at the University of Nevada, Reno, NV, USA. He is working on using coevolution techniques for finding strong, robust build-orders in real-time strategy games. His research interests include computational intelligence and machine learning, with a focus on applications in computer games. You can visit his website, http://www.cse.unr.edu/~caballinger/, for more.