

# GENETIC PROGRAMMING FOR ROBOT SOCCER \*

Vic Ciesielski, Dylan Mawhinney, and Peter Wilson  
Department of Computer Science  
RMIT  
GPO Box 2476V  
Melbourne 3001  
vc@cs.rmit.edu.au, Tel: (03) 9925-2926, Fax: (03) 9662-1617

April 17, 2003

## Abstract

RoboCup is a complex simulated environment in which a team of players must cooperate to overcome their opposition in a game of soccer. This paper describes two experiments in the use of genetic programming to develop teams for robocup. Experiment 1 used just the basic actions of turn, move and dash that a player can send to the soccer simulator. Experiment 2 used higher level functions such as dashing towards the ball and kicking towards the goal. The teams generated in experiment 2 were clearly better and reached the level of 'school boy soccer' where the players follow the ball and try to kick it. The teams generated in experiment 3 were quite good, however they were not as good as the teams evolved in experiment 2. The results suggest that genetic programming could be used to develop viable teams for the competition, however, much more work is needed on the higher level functions, fitness measures and fitness evaluation.

## 1 Introduction

Genetic programming is a process of generating programs by a process of simulated evolution[3, 4, 5]. To date the technique has generally been used for relatively simple tasks or constrained artificial problems. This paper considers the question of whether genetic programming would be useful in a complex, changing, uncertain environment such as simulated robot soccer.

Genetic programming has previously been used for robo soccer [1, 6]. The team evolved in [6] won two games in the 1997 tournament. Both these attempts have been somewhat competitive. Our focus is on finding a good set of high level functions and associated fitness measures prior to working on a team for the competition. Other machine learning techniques have been used previously for robocup including agent oriented programming [8], neural networks [9], and decision trees [10].

Strongly typed genetic programming (STGP) is an extension of standard genetic programming. All functions and terminals in a STGP system must return a particular data type, all function arguments must also be restricted to a specific type. The main benefit of STGP is the reduced search space it offers. We have developed a strongly typed genetic programming system which can evolve players for the robocup competition, full details can be found in [12].

---

<sup>1</sup>In Andreas Birk, Silvia Coradeschi, and Satoshi Tadokoro, [Editors], *Proceedings of the RoboCup 2001 International Symposium, Lecture Notes in Artificial Intelligence 2377*, pages 319-324. Springer-Verlag, 2002. Mpeg files of some games can usually be found at <http://www.cs.rmit.edu.au/~vc/robocup>

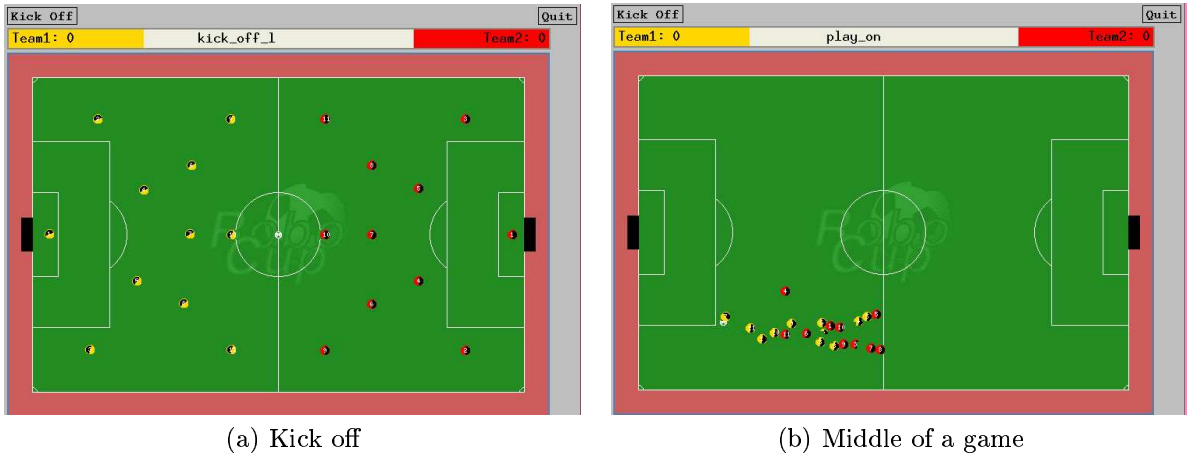


Figure 1: The simulator field of play

## 1.1 Goals

The overall goal of this work is to determine whether a competitive robo cup team can be developed by genetic programming. In this investigation we are particularly interested in:

1. Can a viable team be generated by evolving one player and making 11 copies to get the team?
2. Can a team be generated using just the basic soccer simulator primitives?
3. What fitness measures are useful?

## 2 RoboCup

RoboCup is designed to be an environment for the development of co-operative agents operating in a changing, uncertain domain. There are two separate leagues in the RoboCup competition — the robot league and the simulator league. Competitions in the robot league are played with real-world robots, moving a ball around a small soccer field. The simulator league allows researchers to test ideas regarding artificial intelligence and team-based strategies without the limitations of today’s robotics. This paper deals only with the simulator league. The field of play is shown in figure 1.

In the simulator league, play is facilitated by a software soccer server. The soccer server is responsible for providing the artificial world in which games are played. Each player in the simulation is its own process, with communication between players limited to messages which must pass through the soccer server via Unix sockets. A brief description of the RoboCup server follows; a full description can be found in [7].

Messages, represented as strings, allow communication between players and the server. Messages sent by players inform the server of actions they wish to execute. Messages from the server inform players of the position of the ball, positions of other players, lines, flags and goals which a player can see on the field. A list of commands used in the evolution is shown in table 1. The full set of commands can be found in [7].

After a game has begun, the server sends updated percept information to each player. This occurs once every 150 milliseconds, but players are expected to send actions to the server once every one-hundred milliseconds. Because of the length of time between percept updates, players are often acting upon out of date percept information. Many researchers have attempted to overcome

this problem by incorporating a method of estimating the changed positions of objects on the field as a player’s percepts become more and more out of date [6, 11].

Players receive sensory information from the server via a percept vector. Full details of the percept vectors can be found in [7]. The following example illustrates the basic mechanism. A player who can see the ball 5 units away directly in front of itself, a teammate (number 11) off to its left and an opponent, too far away to its right for its number to be visible, would receive the percept vector:

```
((ball) 5 0 0 0) ((player MyTeam 11) 10 -30 0 0) ((player TheirTeam) 10 0 5))
```

### 3 Experiment 1 – Basic Robocup Functions

Experiment 1 was the basic starting point for evolving genetic programs to play soccer. Each chromosome represents a control program for a soccer player. A team consists of 11 copies of this player. The only action functions available to programs were those already provided by the soccer server as shown in table 1. It was hoped that programs would use these very basic functions and terminals to evolve some useful soccer-playing behaviours. Experiment 1 also provides a baseline for measuring improvement in evolved programs due to changes in functions, terminals and fitness evaluation.

#### 3.1 Terminals and Non Terminals

Terminals correspond to inputs to a conventional program[4] and will always be leaves of a program tree. The terminals used are shown in table 3. Terminals essentially extract information from the percept vector. Non terminals form the interior nodes of a program tree and are the basic primitives that can be combined in the evolving programs. Those used in experiment 1 are shown in table 3.

#### 3.2 Genetic Operators

The basic genetic operators are crossover and mutation. These operators work on the tree representations of the evolving programs. The crossover operator selects two parents and swaps randomly selected subtrees to get the children, as shown in figure 2. The mutation operator selects a subtree at random and replaces it with a new, randomly generated tree.

#### 3.3 Fitness

For this experiment, each generation proceeded in an elimination-round style tournament [2]. Programs were randomly paired up and played off. The winner of the meeting was then advanced to the next round. This process continued until there was only one program left. The team represented by this program was the winner of the tournament, and therefore the fittest of the

Command	Description
(turn <i>Dir</i> )	Changes the direction a player is facing by <i>Dir</i> degrees.
(dash <i>Pow</i> )	Increases the momentum of the player with the power <i>Pow</i> , causing it to run in the current direction.
(kick <i>Pow Dir</i> )	Kicks the ball with a power of <i>Pow</i> and in the direction <i>Dir</i> . Players may only kick the ball if they are close enough to it.

Table 1: Basic RoboCup player control commands

Terminal Name	Type	Description
Random Number	f	A random number $n$ where $0 \leq n < 360$ .
NearTeam	v	Returns the location of the player's nearest teammate.
2ndNearTeam	v	Returns the player's 2nd nearest teammate.
3rdNearTeam	v	The player's 3rd nearest teammate.
4thNearTeam	v	The player's 4th nearest teammate.
NearEnemy	v	The player's nearest opponent.
2ndNearEnemy	v	The player's 2nd nearest opponent.
3rdNearEnemy	v	The player's 3rd nearest opponent.
4thNearEnemy	v	The player's 4th nearest opponent.
BallPos	v	The position of the ball.
MyGoalPos	v	The position of a player's goal (the goal a player must defend).
TheirGoalPos	v	The position of a player's opponent's goal (the goal a player needs to kick the ball through to score).

Table 2: Terminal Set used in experiment 1. In the type columns, f denotes a floating point value and v a percept vector.

generation. The fitness of the other programs was determined by the round in which they were eliminated. The best individuals of each generation were then played off in a best-of tournament.

The winner of a game was chosen based upon the number of goals a team scored — the one with the highest number was deemed to have won. If the number of goals scored by each team was equal, the winner was chosen based on the number of successful passes between players (a successful pass occurred if a player kicked the ball within five time steps of a teammate kicking the ball). If the number of passes was equal, the winner was based on the number of kicks a team achieved. If none of the preceding tests could not break the tie, the winner was chosen at random.

### 3.3.1 Results

The results for this experiment were unexpectedly disappointing. Not only did every program fail to score a proper goal, most failed to even kick the ball. Even less encouraging was the fact that some teams even managed to score goals against themselves. Since very few teams scored goals or executed successful kicks or passes, the winner of a tournament was often determined at random.

One winner was a program with 196 nodes, and, after simplifying the program by evaluating the functions which will always return the same value (because their arguments are constant), the program with 77 nodes shown in figure 3 was produced. This program provides its player with one action — kick. The large section of text beginning with `(max (max . . .` is the first argument to kick, or the value of the power argument. The second argument to the kick function, direction, is a constant value (23.977). As a result, even if the convoluted combination of ball, player and goal position caused the ball to travel the intended distance, it would always travel in the same direction (as the direction of the kick is constant). However, the player is unlikely to ever get near the ball as it contains no dash function to move it around and no turn function to face it in the right direction.

The unexpected results of this experiment are most likely due to two factors: the tournament selection method and the function and terminal sets. Tournament selection would appear to work well when programs are given a representation which allows them to achieve basic behaviours early in the process that should propagate them through to following generations. In a complex domain like RoboCup, however, behaviours like kicking the ball to a teammate or into a goal are often not readily available in early generations, especially when using such a low-level function and terminal set as the one used in this experiment. Perhaps a lower-level check of breaking a tie

Name	Returns	Args	Description
<i>ifltef(a, b, c, d)</i>	f	f, f, f, f	If $a$ is less than or equal to $b$ , return $c$ , else return $d$ .
<i>ifltea(a, b, c, d)</i>	f	f, f, a, a	If $a$ is less than or equal to $b$ , evaluate $c$ , else evaluate $d$ .
<i>kick(pow, dir)</i>	a	f, f	Send a kick message to the soccer server with the parameters $pow$ power and $dir$ direction.
<i>turn(moment)</i>	a	f, f	Turn the player within a range of -180 to 180 degrees.
<i>dash(pow)</i>	a	f	Cause the player to dash in the direction they are currently facing.
<i>min(a, b)</i>	f	f, f	Return the minimum of $a$ and $b$ .
<i>max(a, b)</i>	f	f, f	Return the maximum of $a$ and $b$ .
$+, -$	f	f, f	Standard arithmetic operators.
<i>getDist(v)</i>	f	v	Return the distance element of a vector.
<i>getDir(v)</i>	f	v	Return the direction element of a vector.
<i>getCDist(v)</i>	f	v	Return the change in distance element of a vector.
<i>getCDir(v)</i>	f	v	Return the change in direction element of a vector.
<i>execTwo(a, b)</i>	a	a, a	Unconditionally execute both actions and return an action.

Table 3: The function set used in experiment 1. In the type column, **f** denotes a floating point value, **v** a percept vector and **a** an action.

between programs, such as giving preference to a program which had all three action functions within it over programs which may have had only two, would have produced better results.

## 4 Experiment 2 - Higher Level Functions

The terminals, genetic operators and fitness evaluation were the same as in experiment 1. The non terminal functions used in experiment 1 (table 3) were replaced by those shown in table 4. These functions correspond to more complex soccer actions than just turning, dashing and kicking. A number of additional, lower level, support functions were also needed. Full details are in [12].

### 4.1 Results

The higher level representations used in this experiment allowed the generated programs to exhibit basic soccer-playing behaviour — that of chasing the ball and kicking it towards something. Like early programs described in Luke *et al.* [6], the programs evolved exhibited behaviour which might be expected at a children’s soccer game (ie. all players believe that they alone can win the game). All programs which made it into the final best-of-the-best elimination-round tournament flocked to the ball, irrespective of the location of teammates and opponents on the field. Most programs could kick the ball, some could not. Those that could not kick the ball surrounded it and typically blocked their opponents’ efforts to get near the ball. In evidence that the genetic process was working, those that could kick the ball all attempted to get it into the opposition’s goal. Watching a match between these players was almost like watching a game of tennis — the ball would be kicked to one side of the field, only to be chased down and kicked to the other side. Few goals were scored in this best-of tournament, mainly due to the ball-smothering behaviours of the players. The winner of the tournament is shown in figure 4a. This program scored a goal in four of the eight games it played.

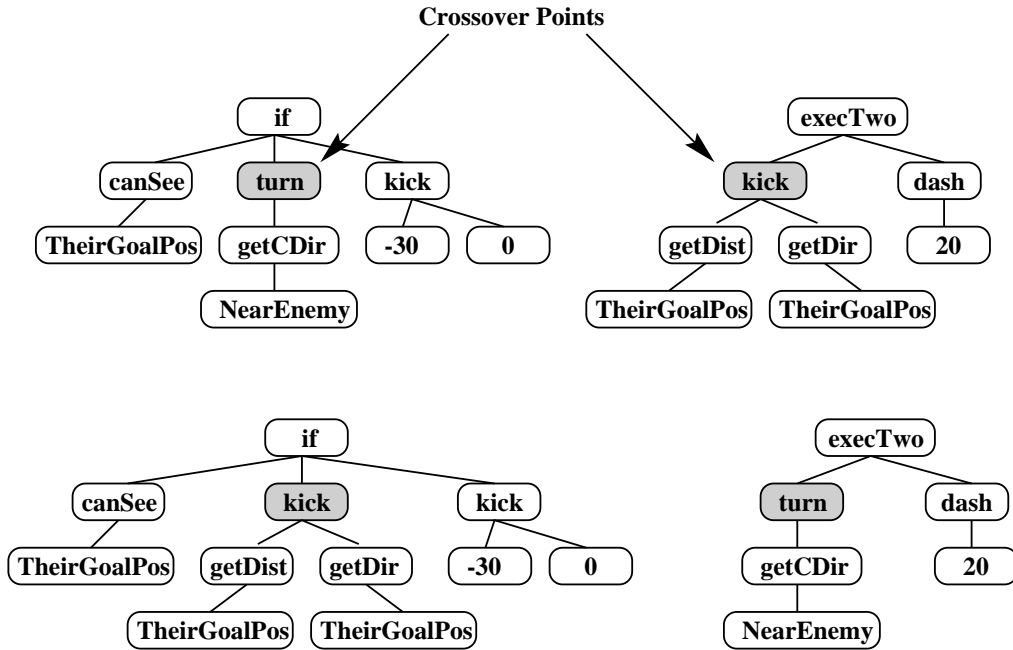


Figure 2: The genetic crossover operation. The top two trees are the parents, selected at random from the population. The bottom two trees are the children. The chosen subtrees have been swapped.

Figure 4b shows another program that won several games. If it can see an enemy or the nearest teammate has the ball, it will kick the ball short to the opposition's goal. If neither of those conditions holds, the condition inside the second *if* function (ie. *if* (*canSee* *NearEnemy*)) will not hold either, so the program turns to its goal. Regardless of the outcome of the *if* statements in the program, it will always run towards the ball and kick it at the opposition's goal anyway, since *kickTo* is the second argument of the root node, *execTwo*. The final effect of this program is essentially the same as that of program (a). In fact, 15 of the 32 best programs played off in the tournament were either identical copies or slight variations of the above two programs. Another point worth noting about the importance of *kickTo* to the best-of population was that 27 of the 32 programs had this function as their root node.

```
(kick
  (max (max 164.031 (ifltof (ifltof 184.781 (ifltof (getCDist BallPos) 74.5583
    (- -138.5811 (ifltof 281.591 (- 138.156 (min 15.5594 (ifltof (getDir
      TheirGoalPos) 31.2062 105.764 349.634))) 270.359 116.989)) 207.59)
    (+ (- 376.85 (getDir NearTeam)) 60.8147) 296.405) 132.398 -138.5811 126.811
    (getCDist BallPos))) (+ (ifltof 55.2772 295.528 (+ -138.5811 (getDist
      Near2ndTeam)) (max (max 346.21 (ifltof 308.454 (getDist Near2ndEnemy)) (max
        251.4319 (min (+ (getCDist Near4thTeam) 278.756) (getDir MyGoalPos)))
        182.467)) (- 303.597 (- 200.1987 (+ 329.773 (min 193.252 (getDist
          Near3rdTeam)))))))))) 25.9877)
  23.977
)
```

Figure 3: A (simplified) program from experiment 1 which won a tournament

Name	Returns	Args	Description
<i>if(a, b, c)</i>	a	b, a, a	If <i>a</i> is true, execute <i>b</i> , else execute <i>c</i> .
<i>canSee(v)</i>	b	v	Return true if the player can see its argument, otherwise return false.
<i>turnTo(v)</i>	a	v	If the percept vector is visible, turns the player to face it and runs towards it. If it isn't visible, the player scans for it by turning 90° and waiting for a percept update.
<i>moveTo(v)</i>	a	v	This function causes the player to turn towards its argument in the same fashion as <i>turnTo</i> (including the scanning if it isn't visible). If the percept vector is visible, the player then runs towards it.
<i>kickTo(v)</i>	a	v	If the player can see the ball and is within kicking distance, it will kick it in the direction and distance of its argument, else this function acts like <i>moveTo</i> with the ball's position as its argument.
<i>shorten(v)</i>	v	v	Shorten the distance <i>v</i> is away from the player by 20%.

Table 4: Experiment 2, main non terminal functions

```
(kickTo TheirGoalPos)
    (execTwo
      (if (or (hasBall NearTeam)
             (canSee NearEnemy))
          (kickTo (shorten TheirGoalPos))
          (if (canSee NearEnemy)
              (moveTo TheirGoalPos)
              (turnTo (shorten MyGoalPos))))
      (kickTo TheirGoalPos)\
    )
```

(a) (b)

Figure 4: Two programs from experiment 2

## 5 Experiment 3 - Augmented Basic Robocup Functions

In experiment 3 the terminals and low level functions in experiment 1 were used along with some additional terminals and functions. The new terminals added for this experiment can be found in table 5. The new functions added for this experiment can be found in table 6. Experiment 3 used the roulette wheel selection method, each team played one game against another team in the population, fitness was calculated as follows (larger fitness is better):

```
fitness = (number of goals scored * 200) +
          (number of own goals scored * -200) +
          (5 * (num of kicks)^0.5
```

If the team won the game they played, they are given an increase of 500 in their fitness.

### 5.1 Results

The programs generated during this experiment were able to get to the ball, and then kick it, however the ball did not always go towards the goal. Most of the teams which were evolved did

Name	Returns	Description
<i>GameState</i>	f	Returns the current state of the game (play_on, kick_off etc) as a number.
<i>PlayNum</i>	f	Returns the shirt number of the current player.
<i>IsLeft</i>	b	Returns true if the player is on the left hand side team, false otherwise.
<i>IsRight</i>	b	Returns true if the player is on the right hand side team, false otherwise.
<i>ClosestToBall</i>	b	Returns true if the player is the closest to the ball out of the players it can see.

Table 5: Experiment 3, added terminals

Name	Returns	Args	Description
<i>equal(d, d)</i>	b	f,f	Returns true if the two numbers passed to it are equal.
<i>or(b, b)</i>	b	b,b	Returns true if either of it's arguments are true.
<i>and(b, b)</i>	b	b,b	Returns true if both it's arguments are true.
<i>*, %(f, f)</i>	f	f,f	Multiplication and protected division.
<i>if(a, b, c)</i>	a	b,a,a	If a is true return b, else return c.

Table 6: Experiment 3, non terminal functions

not run directly towards the ball, they instead developed a “swirling” behaviour where the team would run around in small circles as a group, the group of swirling players would move towards the ball eventually kicking in what seemed to be a random direction. Strangely this behaviour was common, it was observed in many independent evolutionary runs which had each started from random populations. Teams which used this strategy were able to kick goals some of the time, however this was only the result of kicking the ball hard in random directions, most teams scored just a many own goals as they did goals. We are currently looking at modifying the fitness function to encourage more goals.

One interesting behaviour which emerged from some of the evolutionary runs, was that of goalie behaviour. Some of the teams had a player which would stand in front of the goals and not move around or follow the ball. This player was acting as a “stupid” goalie, one which did not move or kick but just sat in the way of the goals. This behaviour was able to be evolved because of the addition of the *PlayNum* terminal. By using one of the *if* type functions the genetic programming process had isolated a piece of code which would only be executed by a particular player. Essentially what had been evolved was a piece of code which would read something like the following:

```

if I am player number 1 then
    stand still in front of the goals
else
    do something else (chase the ball etc)

```

The players evolved in experiment 3 were not as good as those evolved in experiment 2. However the actual “learning” that took place in experiment 3, was much more advanced than in experiment 2. In experiment 2 the majority of the programs which were evolved simply made use of the *kickTo* function which was available, so all that was really learnt in experiment 2 was that the *kickTo* function was a good one to use. In experiment 3 however the programs evolved were very complex and contained many different functions and terminals. An example of an evolved program from experiment 3 can be found in figure 5.

```
(execTwo (if (and IsRight (or (and (and (or IsRight (or (equal 106.398 GameState)
) closestToBall)) (equal 227.719 (getCDist NearEnemy))) closestToBall) (equal (m
in (getDir Near3rdEnemy) (- (getDir GetTheirGoal) (ifltof GameState 330.826 (max
PlayNum (ifltof (+ 327.465 PlayNum) 51.3406 (min (/ PlayNum GameState) GameStat
e) GameState)) 270.59))) (* (* (* (min 264.267 9.12845) (min 158.671 GameState))
PlayNum) (- GameState (ifltof (/ (ifltof GameState 220.696 301.992 259.202) 118
.493) 222.581 330.413 143.037)))))) . . . . .
```

Figure 5: Example program from experiment 3

## 6 Conclusions

The goals of this work were (1) to determine whether a viable team could be generated by evolving one player and making 11 copies, (2) to determine whether a viable team could be generated using just the robocup basic functions, and (3) to determine appropriate fitness measures.

The strategy of evolving one player and making 11 copies worked reasonably well, particularly in experiment 2. From viewing the games it was clear that the teams would benefit from having a designated ‘goalie’ who stayed in the goal area. Further work is needed to determine whether evolving 11 different specialist players would result in better teams.

Experiment 1 showed that very low-level behaviours by themselves are not enough to evolve a set of high-level and complex soccer-playing strategies. Players all but lost the ability to dash by generation 7. No goals were scored, some own-goals were scored, but most players failed to even kick the ball.

Experiment 2 utilised higher-level functions and resulted in players which exhibited ‘school boy soccer’ abilities, that is, each player followed the ball around as it was kicked back and forth.

Experiment 3 utilised low-level functions, however it had quite a few different functions than in experiment 1. The players exhibited reasonable abilities, however they were not up to the level of the players from experiment 2.

The form of the fitness function and the method fitness evaluation remain unresolved issues. It seems clear that different fitness functions are needed early in the evolutionary process when the teams are not very good and later, when the teams have improved. Further work is under way in this area.

Although the generated teams to date are somewhat limited in soccer playing ability it has been established that performance improves over a number of generations no matter which functions are used. This suggests that with suitable attention to the functions and fitness evaluation genetic programming could be used to generate teams of much higher ability.

## References

- [1] David Andre and Astor Teller. Evolving team darwin united. In Minoru Asada, editor, *Robocup-98: Robot Soccer World Cup II. Lecture Notes in Computer Science*. Springer-Verlag, 1999.
- [2] Peter J. Angeline and Jordan B. Pollack. Competitive environments evolve better solutions for complex tasks. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 264–270, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.
- [3] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic programming: An introduction on the automatic evolution of computer programs and its applications*.

San Francisco, Calif. : Morgan Kaufmann Publishers ; Heidelberg : Dpunkt-verlag, 1998.  
Subject: Genetic programming (Computer science); ISBN: 1-55860-510-X.

- [4] John Koza. *Genetic Programming: on the Programming of Computers by means of Natural Selection*. The MIT Press, 1992.
- [5] John Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.
- [6] Sean Luke, Charles Hohn, Jonathan Farris, Gary Jackson, and James Hendler. Co-evolving soccer softbot team coordination with genetic programming. In *Proceedings of the First International Workshop on RoboCup, at the International Joint Conference on Artificial Intelligence*, Nagoya, Japan, 1997.
- [7] Itsuki Noda. *Soccer Server Manual*.  
<ftp://ci.etl.go.jp/pub/soccer/server/manual.r2.00.ps.gz>, February 1998.
- [8] Itsuki Noda. Team gamma: Agent programming on gaea. In H. Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*, 1998.
- [9] Itsuki Noda, Hitoshi Matsubara, Kazuo Hiraki, and Ian Frank. Soccer server: a tool for research on multi-agent systems. *Applied Artificial Intelligence*, 12(2-3), 1998.
- [10] Peter Stone and Manuela Veloso. A layered approach to learning client behaviors in the robocup soccer server. *Applied Artificial Intelligence (AAI)*, 12, 1998.
- [11] James Westendorp. Behaviour-based reasoning with dynamic, partial, uncertain information. Honours thesis, Dept. of Computer Science, Royal Melbourne Institute of Technology, November 1997.
- [12] Peter Wilson. *Development of a Team of Soccer Playing Robots by Genetic Programming*. Honours Thesis, RMIT, Department of Computer Science, 1998  
<http://www.cs.rmit.edu.au/~vc/papers/wilson-hons.ps.Z>.