Evolution of a World Model for a Miniature Robot using Genetic Programming

Peter Nordin*   Wolfgang Banzhaf   Markus Brameier

Fachbereich Informatik
Universität Dortmund
44221 Dortmund, GERMANY
email: nordin,banzhaf,brameier@ls11.informatik.uni-dortmund.de

Abstract

We have used an automatic programming method called Genetic Programming (GP) for control of a miniature robot. Our earlier work on real-time learning suffered from the drawback of the learning time being limited by the response dynamics of the robot's environment. In order to overcome this problem we have devised a new technique which allows learning from past experiences that are stored in memory. The new method shows its advantage when perfect behavior emerges in experiments quickly and reliably. It is tested on two control tasks, obstacle avoiding and wall following behavior, both in simulation and on the real robot platform Khepera.

Keywords: Evolutionary Robotics, Genetic Programming, World Model, On-line Learning, Planning

---

*Now at: Dacapo SA, Goeteborg, Sweden

1 Introduction

A very general way of representing and specifying an autonomous agent's behavior is by employing a computer language. If one uses a suitable computer programming language every behavior of an autonomous agent might be specified. The question, then, arises how to (automatically) program in that computer language.

Genetic Programming (GP) is a method that recruits an evolutionary algorithm to evolve computer programs. It is thus a probabilistic method of automated programming. If a goal has been defined in form of a fitness function and a set of instructions has been determined for use, the genetic programming system will try to evolve programs that solve the task specified by the fitness function.

To use a selective process derived from evolution as a guiding principle for the design of a controller architecture could, at first, be considered an awkward approach. As far as we know today genetics is only indirectly involved in information processing of the brain, although the idea of genetics as a model of mental processing is not new. Indeed, it has recently gained more prominence through work by Richard Dawkins [7], Douglas Hofstadter [13] and William Calvin [3, 4].

From the time of William James who argued that mental processes could operate in a Darwinian manner [15] scientists have continued to discover the strength of selective principles in adaptive systems. A case in point is the following statement by immunologist Niels Jerne:

"Looking back into the history of biology, it appears that wherever a phenomenon resembles learning, an instructive theory was first proposed to account for the underlying mechanisms. In every case, this was later replaced by a selective theory. Thus the species were thought to have developed by learning or by adaptation of individuals to the environment, until Darwin showed this to have been a selective process. Resistance of bacteria to antibacterial agents was thought to be acquired by adaptation, until Luria and Delbrück showed the mechanism to be a selective one. Adaptive enzymes were shown by Monod and his school to be inducible enzymes arising through the selection of preexisting genes. Finally, antibody formation that was thought to be based on instruction by the antigen is now found to result from the selection of already existing patterns. It thus remains to be asked if learning by the central nervous system might not also be a selective process; i.e., perhaps learning is not learning either." [16]

More recently, selectionist approaches to learning have been studied in detail by Gerald Edelman and his collaborators (see [8] and references therein). Why would it be impossible to consider machine learning and man-made adaptive systems from much the same perspective?

In the wake of the spread of Darwinian thinking in Computer Science the use of evolutionary methods has been reported in robotic control in a number of publications. Robot controllers have, for instance, been evolved using dynamic recurrent neural nets [5, 12]. Genetic Algorithms have been used in [9] for generating wall-following behavior. Several experiments have also been performed where a controller program has been evolved directly through genetic programming [11, 17, 28].

We, too, have reported earlier on our first experiments using GP to control a real robot that has been trained in real-time with actual sensor values [23, 25]. In a real environment

our system had to evolve robust controllers because noise was present everywhere and the number of real-life training situations was virtually infinite. Consequently we were forced to devise an online learning method which ensured learning of behavior while each individual was tested against *different* real-time (and noisy) fitness cases which were sampled probabilistically from the environment. This procedure might have resulted in "unfair" fitness comparisons where individuals were confronted with very different situations. However, our experiments showed that — over time — the fluctuations of the environment cancelled out and the system approached a robust state. The advantage of our method, which we call stochastic sampling, was a considerable acceleration of the evolutionary process.

Using this technique the evolution of a successful control program was driven by continuous interaction with and feed–back from the environment (reinforcement learning). There was, however, no memory of past experience other then the information implicitly stored in the genetic material. Hence the main disadvantage of the approach was the learning speed being limited by the response dynamics of the environment, i.e. most of the learning time was spent waiting for the reinforcement signal from the enviroment.

In order to overcome this problem we have proposed an extension of this approach in [1, 24] that might be called *online learning from past experiences*. It will be the subject of this contribution. It is built on past experiences held in a memory buffer and associated with a positive or negative evaluation with respect to the control task. The learning task here is not to evolve a controller as a function — directly mapping sensor inputs to actions that corresponds to a certain behavior. Rather a generalising world model should be derived from the experiences in memory to serve as a basis for planning future actions while remembering past experiences. We show that, besides speeding up learning and allowing for a much more systematic exploration in the first place, this memory-based approach reaches perfect behavior in a very high percentage of the experiments.

The rest of this paper is organized as follows: In section 2 we start by briefly introducing the genetic programming paradigm in general. We present the real miniature robot Khepera and the simulator used in the experiments in section 3. The memory-based control architecture is presented in section 4. The objectives of training in the experiments, obstacle avoiding and wall following behavior, are described in section 5. We present our results in section 6. Finally we summarize the results, discuss conclusions and present ideas for future work.

2 Genetic Programming

Evolutionary algorithms mimic aspects of natural evolution to optimize a solution towards a defined goal. Darwin's principle of natural selection plays a key role when differential fitness advantages are exploited to lead to better solutions.

A general evolutionary algorithm may be summarized as follows:

1. A population of solutions is initialized to random content.

2. In an iteration loop: Individuals are selected from the population randomly and are compared, based on their fitness. The fitness measure defines the problem which

should be solved by the algorithm. Only the fitter individuals are modified by the following genetic operations while the population size is kept constant.

- Identical *reproduction*
- Exchange of a (small) substructure in an individual at a random position (*mutation*)
- Exchange of substructures between two individuals (*crossover*)

3. The currently best individual in the population represents the best solution found so far.

Different research subfields of Evolutionary Algorithms have emerged, such as Genetic Algorithms [14], Evolution Strategies [26] and Evolutionary Programming [10]. In recent years these methods have been applied successfully to a wide spectrum of problem domains, especially in optimization.

A comparatively young and growing research area in this context is Genetic Programming (GP) [17]. Genetic Programming uses the mechanisms behind variation and selection for the evolution of computer programs. The approach has been formulated originally using tree structures that were represented by variable length LISP S-expressions as individuals. The inner nodes of these trees are functions while the leafs are terminals that represent input variables or constants. The operators applied to generate variants, i.e. crossover and mutation, must guarantee syntactic closure during evolution. In other words, no syntactically incorrect programs are allowed to be generated. Figure 1 illustrates the crossover operation in *tree-based* genetic programming.

In recent years, the scope of Genetic Programming has expanded considerably and now includes evolution of linear and graph-like representations of programs as well, in addition to tree respresentations [2].

2.1 Machine Code Genetic Programming

In the experiments described below we use a genetic programming system with a *linear* program representation. In our *Automatic Induction of Machine Code by Genetic Programming (AIMGP)* system (formerly known as CGPS [20, 21]) an individual is composed of a variable number of 32-bit machine code instructions for a register machine. Individuals are directly manipulated as binary machine code in memory and directly executed without passing an interpreter during the fitness calculation. This results in a significant speed-up compared to interpreting GP systems in, e.g., tree representation. Another advantage is the memory efficiency of this system.

The *function set* consists of the arithmetic operations ADD, SUB and MUL, the shift operations SLL and SLR and the logic operations AND, OR and XOR. All these instructions operate on registers or integer constants (*terminal set*) and correspond to simple C-expressions such as `i0=i1+i2` or `i0=i1*27`. In addition to the configuration of the system in [21] we use the *branch* instructions BG (*branch on greater*) and BLE (*branch on less or equal*) here. The conditions of these branches are composed of specific bits in the Processor Status Register (PSR) that can be modified by certain operations (here SUB). If a condition is false the instruction directly following the branch instruction is skipped.
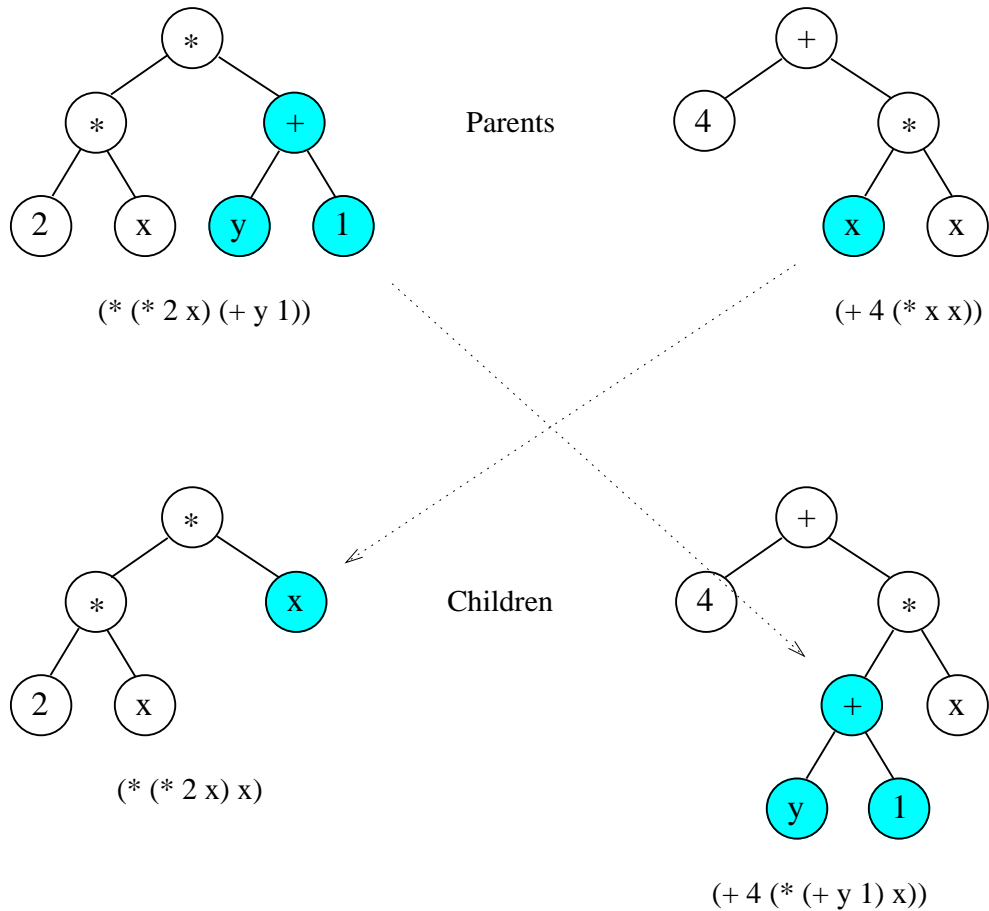
4

Figure 1: Crossover in hierarchical GP

Using branches has proved to be very important specifically for the wall follwing behavior (see section 5.3).

The following is an individual program as it looks if disassembled into a C-program.

```
unsigned int ind(i0,i1,i2,i3,i4,i5)
  unsigned int i0,i1,i2,i3,i4,i5;
{
  ...

  i1=i0 * i2
{ i5=i1 - 3; greater = (((int)i5) > 0); }
  i4=i0 & 12;
  i2=i4 + 53;
  if (!greater) i4=i1 << 2;
  i2=i1 * 77;
  i3=i2 + i4;
  i5=i4 + 0;
  i0=i5 << 4;
  if (greater) i0=i3 << 8;
  return(i0);
}
```

Similar to tree-based GP, AIMGP uses a crossover (and mutation) operator to generate variants. Figure 2 illustrates the two-point string crossover employed in the linear genetic programming system. Crossover can occur *between* instructions (atomic units) only but not within. The mutation operation flips bits *inside* the 32-bit instructions. It is ensured that only instructions from the function set can be created with valid ranges of registers and constants.
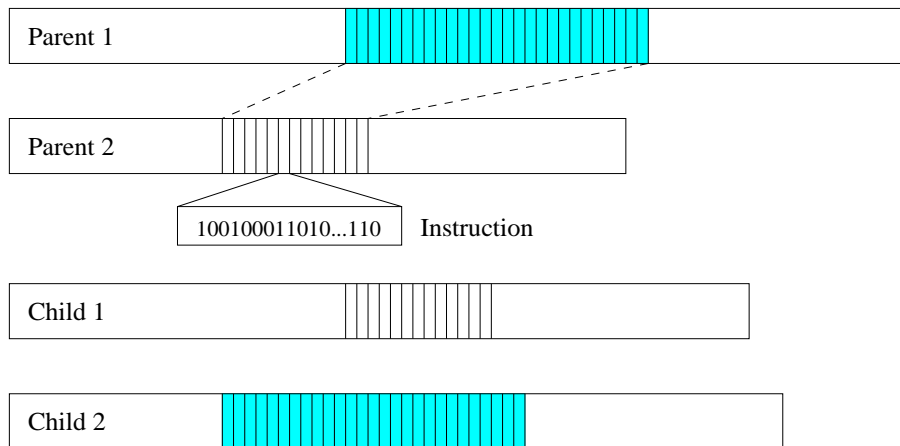

Figure 2: Crossover in linear GP

The evolutionary algorithm of the system realizes a simple steady state tournament selection:

1. Select four arbitrary individuals from the population.

2. Compare the fitness of the individuals in pairs.

3. Copy the two winners and let the copies undergo modifications by crossover and mutation.

4. Replace the two loosers with the two new offspring.

3 The Khepera Robot

Our experiments were performed with a standard autonomous miniature robot, the Swiss mobile robot platform Khepera [19]. It is equipped with eight infrared proximity sensors. The mobile robot has a circular shape, a diameter of 5.5 cm and a height of 3 cm. It possesses two motors and on-board power supply. The motors can be independently controlled by a PID controller. The eight infrared sensors are distributed around the robot in a circular pattern. They emit infrared light, receive the reflected light and measure distances in a short range of 5 cm. Figures 3 and 4 show the robot and its sensor positions.

The robot is equipped with a Motorola 68331 micro-controller which can be connected to a workstation. It is possible to control the robot in two ways. The controlling system may be run on a workstation with data and commands communicated through the serial line. Alternatively the controlling system is cross-compiled on the workstation and down-loaded to the robot which then runs the complete system in a stand-alone fashion. The version
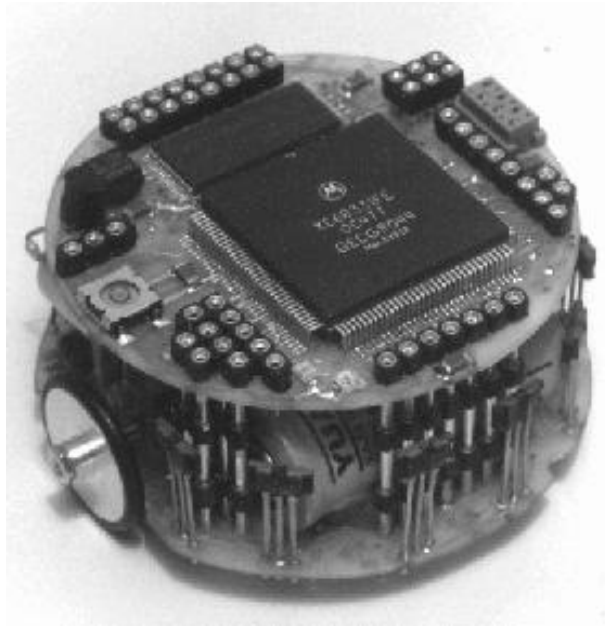
Figure 3: The Khepera Robot

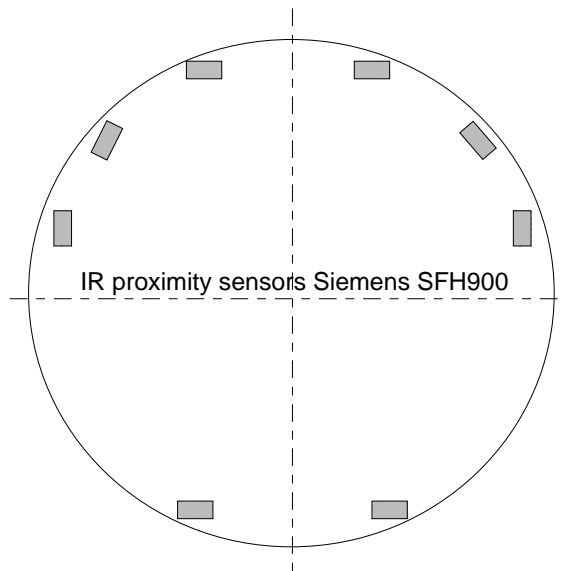IR proximity sensors Siemens SFH900

Figure 4: Positions of the IR proximity sensors

of the controlling GP-system described in this papers runs on the workstation due to high memory requirements involved. The micro-controller has only 256 KB of RAM and a ROM containing a small operating system. The operating system has simple multi-tasking capabilities and manages the communication with the host computer.

The robot also has several extension ports where peripherals such as grippers and TV cameras might be attached.

3.1 The Training Environment

The training environment for the robot is about 70 cm × 90 cm in size. It has an irregular boarder with different angles and four deceptive dead-ends in each corner (see Figure 5). In the large open area movable obstacles may be placed. Friction between wheels and surface is low enabling the robot to slip with its wheels during a collision with an obstacle. The walls and obstacles are choosen white here to give a perfect reflection to the infrared sensor light.
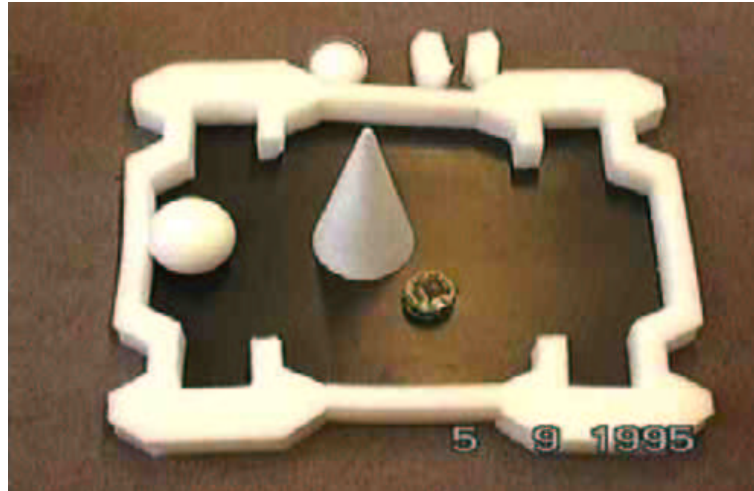

Figure 5: The Training Environment

3.2 The Khepera Simulator

In addition to the experiments done with the real Khepera robot we tested our approach in a software simulation using the Khepera Simulator [18]. The simulation allows test runs without looking at the robot all the time. In this way much more and longer tests have been performed than had been possible with the real robot (see section 6).

Figure 6 gives an impression of the simulator interface. The user is allowed to build an enviroment for the simulated robot from little "bricks" that can be arranged in arbitrary positions. While the robot is moving, the current sensor and motor values may be observed on the screen. The interprocess communication between the GP system and the simulator works with two UNIX pipes – one for each direction.

For two reasons the simulation is quite realistic. First, the proportions of robot size and sensor range correspond to reality. Second, noise has been artificially added to the simulated sensor and motor values. The deviations from exact values range up to ±5%. Noise is extremely important in simulations for the diversity of occuring events and for the degree of generalisation reachable by the learning system. Reynolds showed that a noisy fitness enviroment allows for the development of more robust controllers in genetic programming [27].
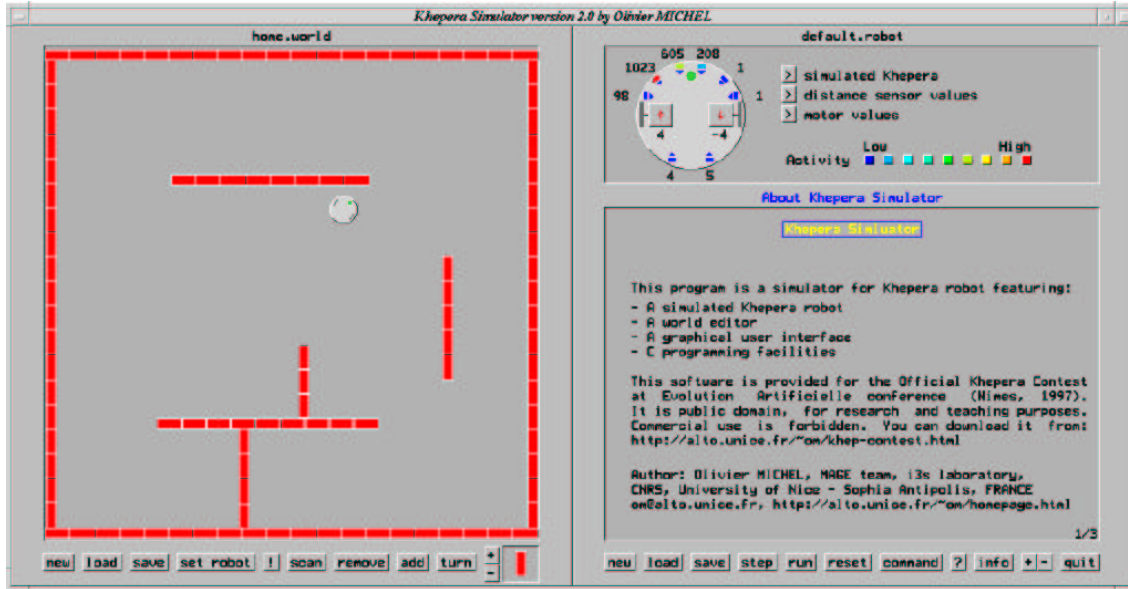
8

Figure 6: The Khepera Simulator

4 The Memory-based GP Control Architecture

The memory-based control architecture consists of two separate processes. One process communicates with the robot and stores past events into memory. The other process is constantly trying to learn and to induce a model of the world consistent with the data in memory. We call the former process the *planning process* because its main objective is deciding what next action to perform given the current best model of the world supplied by the learning process. The latter process is called *learning process* and denotes the evolutionary process of the GP system here. Figure 7 gives a schematic illustration of the architecture of the control system.

4.1 The Planning Process

The main execution cycle of the planning process may be divided into the following five steps:

1. Read the current sensor values from the robot.

2. Calculate the desired evaluation of the *last* event from the *current* sensor values (reinforcement signal).

3. Store the last event and its evaluation in memory as an event vector.

4. Use the current sensor values and search through all motor actions for the best predicted evaluation in the currently best world model.

5. Send the best motor speed actions to the robot. Sensor values and motor speed form a *new* event.
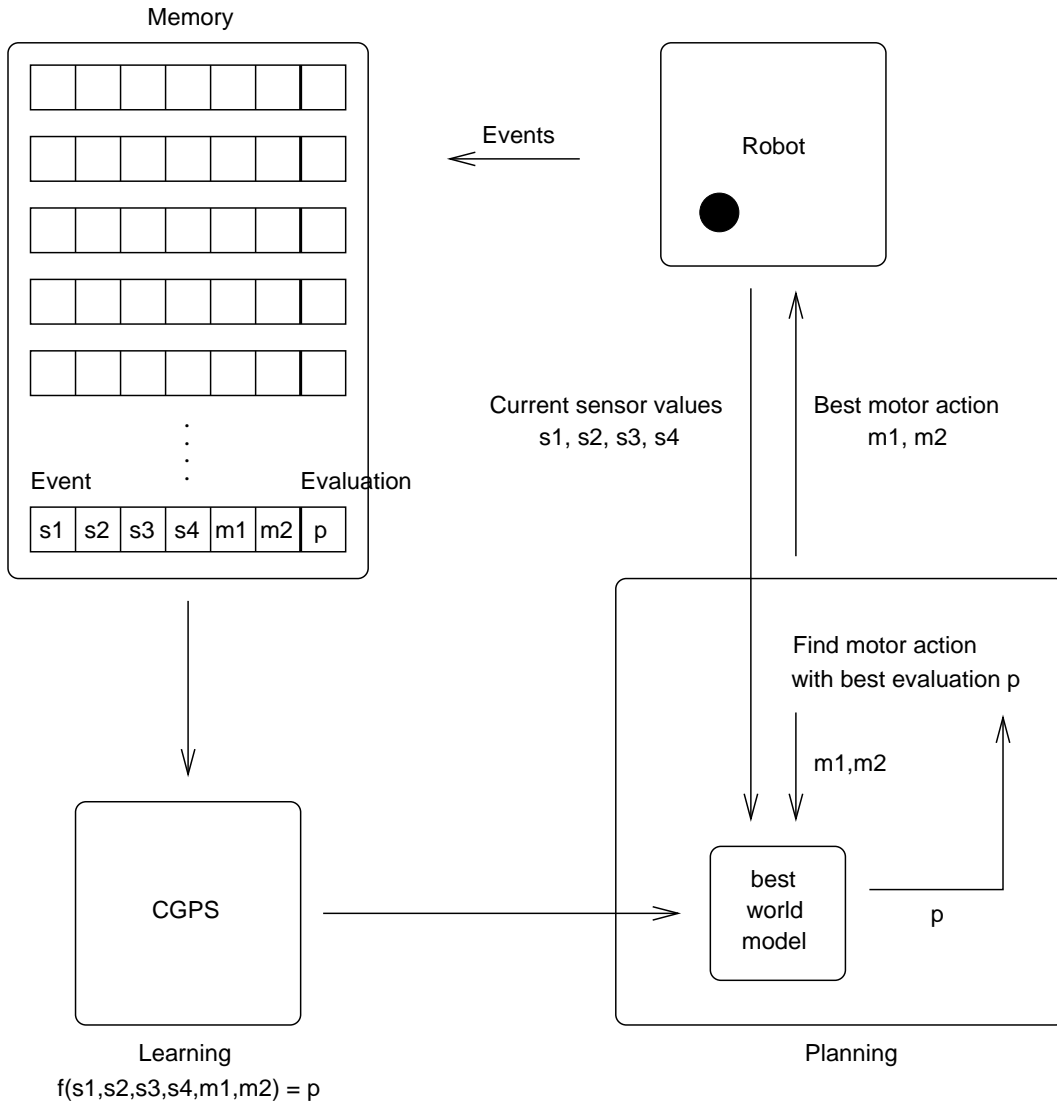
9

Figure 7: Schematic view of the memory based control architecture

6. Sleep 300ms.

The planning process starts with reading the robot sensors. The current sensor values are used to evaluate the last event, i.e. the sensor situation and the resulting action of the last planning phase. The evaluation is stored, together with the last event as an event vector (fitness case). An event vector consists of eight sensor values, two motor speeds and the measured evaluation for this event. Thus, the vector represents what the agent experienced, what it did and what the results were of its action.

The current sensor values instantiate the corresponding variables in the currently best world model (best program). The objective is to find a favorable action to this current sensor situation. With the Khepera, the possible actions are 16 different motor speeds for each of the two motors. Each motor has 8 speeds forward and 7 backwards and a zero speed. Combining all alternatives of the two motors there are 256 different actions altogether to choose from. This comparatively small figure means that we can easily afford

to search through all possible actions while the world model provides us with a predicted evaluation for each of them. The induced model in the form of a computer program from the learning process can thus be seen as a simulation of the environment consistent with past experiences, where the robot can simulate different possible actions. The action which gives the best evaluation is remembered and sent as motor speed actions to the robot.

In order to get feed-back from the environment the planning process has to sleep and wait for results, i.e the new sensor situation, of the chosen action. The planning process sleeps 300 ms while the robot performs the movement determined by the motors speed signals. This delay time is nearly a minimum in order to get usable feed-back from changes in sensor values.

It is then the responsibility of the learning process to evolve a program that simulates the environment as good and as consistent as possible with respect to the events in memory. As we will see below, this can be done by a straight-forward application of symbolic regression through genetic programming.

4.2 The Learning Process

The objective of the learning process is to find a program (function) p which will predict the measured evaluation v of an action (m_1, m_2) given the initial conditions in form of sensor values $(s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8)$:

$$p(s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, m_1, m_2) = v$$

This is a problem of symbolic regression, i.e. inducing a symbolic equation fitting the given numerical data. Genetic programming is ideal for symbolic regression and most GP applications can be reformulated as variants of symbolic regression.

The central idea we discuss in this paper is the evolution of a world model resulting in a relation between an event and an evaluation with reference to a specific control task. The approach is *memory-based* in that past events or experiences are "associated" with values that might be termed "feelings" (feedback of the fitness function).

4.3 Giving the System a "Childhood"

Our first approach to managing the memory buffer when all locations (here about 60) had been filled was to simply shift out the oldest memory entries as the new ones came in. However, we soon realized that the system would then forget important early experiences. Hence we gave the robot a "childhood" – an initial period whose entries are not forgotten.

Another important factor for successfully inducing an efficient world model was to have a "stimulating" childhood. It is important to have a wide set of experiences to draw conclusions from. Noise is therefore added to the behavior in the childhood phase in order to avoid stereotypic behavior very early, in the first seconds of the system's execution. As long as experiences are too few to allow for a meaningful model of the world, this noise is needed to assure enough diversity in early experiences.

5 Training Objectives

5.1 Obstacle Avoidance

A perfect obstacle avoiding behavior may be defined as follows: the robot turns away from an obstacle in the range of its sensors without touching it and runs straight and fast otherwise (see Figure 8).
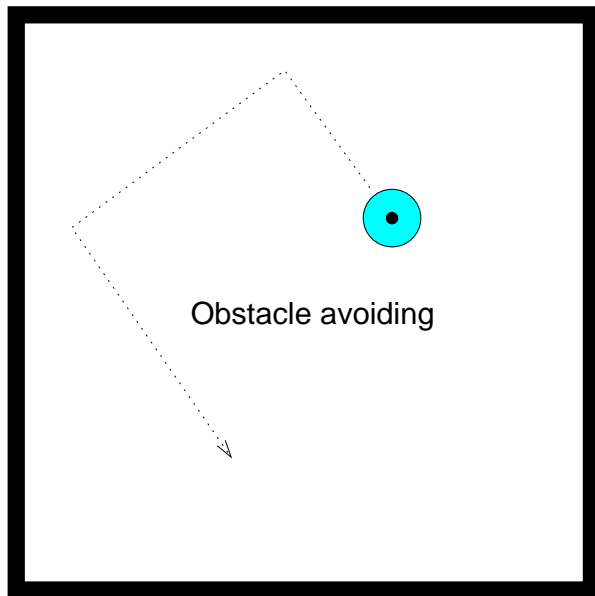

Figure 8: Perfect obstacle avoiding behavior

5.2 Fitness Calculation

Throughout this paper, the fitness of an individual (world model) p is calculated as the square error between the event evaluations $v_{pred} = p(e)$ predicted by the individual and the desired event evaluations $v_{des} = f(e)$ summed over all n events in memory:

$$F(I) = \sum_{i=1}^{n} (f(e_i) - p(e_i))^2$$

The fitness function f defining the obstacle avoiding task has a "pain" and a "pleasure" part. The negative contribution to fitness, "pain", is simply the sum of all proximity sensor values. The closer the robot's sensors are to an object the more pain it experiences. In order to keep the robot from standing still or spinning a positive contribution to fitness, called "pleasure", is necessary. The robot receives "pleasure" from going straight and fast. Both motor speed values minus the absolute value of their difference is thus added to the fitness.

As mentioned above an event e is composed of a sensor situation and a reaction of the robot. Let s_i denote the sensor values ranging from 0 to 1023 where a higher value means

closer to an object. Also let m_l and m_r be the resulting left and right motor speeds in the range of 0 to 15. Then the fitness function can be expressed formally as:

$$f(e) = \alpha * (|m_l - m_r| + |m_l| + |m_r| - (m_l + m_r)) + \sum_{i=1}^{4} s_i$$

5.3 Wall Following

In this task the robot has to follow a continuous, arbitrary formed wall without touching it, allowing navigation around corners as well as navigation in simple mazes.

The difficulty of this control task is to keep a minimum and maximum distance to the wall while moving the robot forward. Accordingly there are three sections in the sensor range to be distinguished in the fitness function (see section 5.4). As shown in Figure 9 the robot normally moves on a zigzag course along the wall. This behavior is due to course corrections that keep the robot within the "corridor".

By comparison, in the definition of the obstacle avoiding behavior stated above there is only a distinction between two basic sensor states necessary, i.e. between perceiving and non-perceiving sensors.
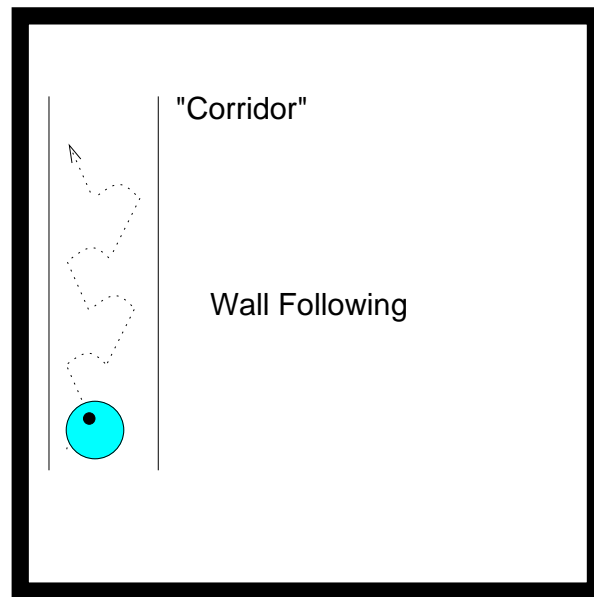

Figure 9: Wall following behavior

5.4 Fitness Calculation

The wall following behavior is defined by a *conditional fitness function* that punishes negative events and rewards positive events with regard to the control task in a more differentiated way. This is necessary because of the high complexity of this behavior. Here every of the four possible atomic actions – turning left, turning right, moving forward, moving backwards – is given either a constant positive or a constant negative evaluation

13

depending on the section that the corresponding sensor situation of the event belongs to. Figure 10 illustrates positive event classes for the wall following behavior when considering the sensors on the *right* side of the robot. Note again that the closer the distance to the wall the higher the sensor values are.
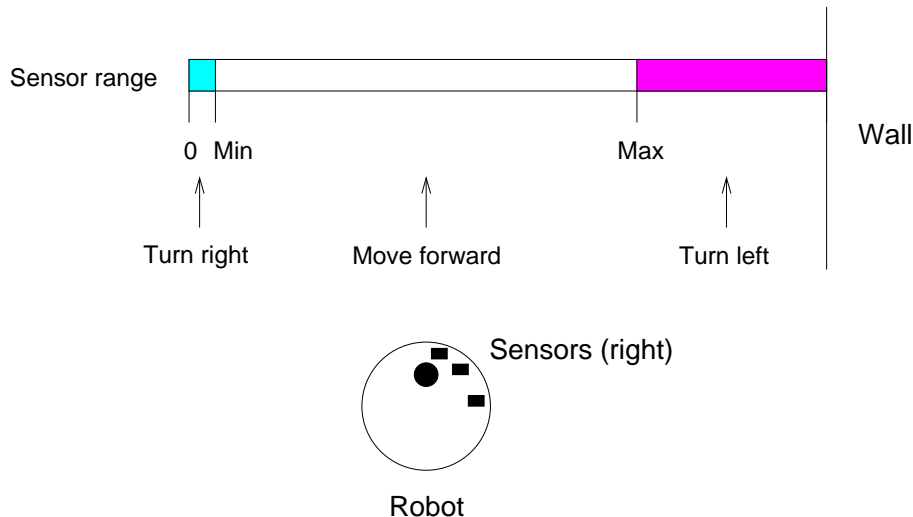

Figure 10: Positive events (wall following)

The complexity of the problem requires that the composition of the memory content is tightly controlled, in order to sufficiently represent all positive and negative event classes.

6 Results

During test runs the occuring events are classified as positive events or as negative events (errors) with reference to the control task. At the end of a successfull run, i.e. a run showing perfect behavior of the robot, only positive events happen and no collisions are registered anymore.

Table 1 gives on overview of the parameter settings of the AIMGP system. Apart from the function set these settings are identical for both control tasks. The branch instructions BG and BLE have proven to be absolutely necessary for the wall following problem. Obstacle avoidance also works by substituting instruction BG with non-linear functions, e.g. SHL and AND, in the function set, but with a lower success rate.

Most behaviors evolved with the memory-based approach show a perfect solution with respect to the fitness function (see Table 2). The obstacle avoiding task is learned with a success rate of about 80%, counting the number of runs that show a perfect behavior after 250 generations. The average number of generations until a behavior definitely becomes perfect, has been calcuated to about 50 generations for the obstacle following behavior.

The more complex wall following behavior shows a lower success rate of about 60% as well as a lower learning speed of about 150 generations on average.

Objective	Symbolic regression of controllers
Terminal set	Integers in the range 0-4096
Function set (obstacle avoiding)	ADD, SUB, MUL, BG
Function set (wall following)	ADD, SUB, MUL, BG, BLE
Population size	10000
Crossover probability	90%
Mutation probability	50%
Selection	tournament selection
Random seed	system time
Maximum program size	256 instructions

Table 1: Koza tableau of parameter settings for the AIMGP system

Behaviour	Obstacle avoiding	Wall following
Number of generations	250	500
Generations until success	50	150
Number of runs	100	100
successfull	82	64
with errors	4	10
failed	14	26

Table 2: Results with the memory-based control architecture

7 Summary and Conclusions

We have demonstrated that a GP system can be used to perfectly control a robot in a noisy environment. The evolved programs show robust performance even if the robot is placed in a completely different environment or if obstacles are moved around. We believe that the robust behavior of the robot partly can be attributed to the built in generalization capabilities of the genetic programming system [21, 22].

We have further demonstrated that the memory-based GP control system can evolve much smoother and less chaotic behavior than the non-memory GP control system described in [23].

We would like to evaluate the usefulness of our approach with agent systems that have a wider set of possible actions. In such systems it would be infeasible to use exhaustive search for finding the best action according to a world model. Handley has previously demonstrated the feasibility of GP for evolution of plans for simulated robots [11]. Equipped with a real robot and the memory-based system, the planning process could incorporate its own GP system to evolve a suitable plan which optimizes the outcome given the currently best world model.

Finally exploration of different strategies of "active learning" [6] is warranted. The decision which memory entries to keep and which ones to discard will have a profound influence on the resulting world model.

ACKNOWLEDGMENT

References

[1] W. Banzhaf, P. Nordin and M. Olmer (1997) Generating Adaptive Behavior using Function Regression within Genetic Programming and a Real Robot. In: *Genetic Programming 1997: Proceedings of the Second Annual Conference, 1997, Stanford University.*, J. Koza, K. Deb, M. Dorigo, D. Fogel, M. Garzon, H. Iba, R. Riolo (eds.) Morgan Kaufmann, San Francisco, 35 - 43.

[2] W. Banzhaf, P. Nordin, R. Keller, F. Francone (1998) *Genetic Programming — An Introduction.* dpunkt.verlag, Heidelberg and Morgan Kaufmann, San Francisco, CA.

[3] W.H. Calvin (1996) *How Brains Think: Evolving Intelligence, Then and Now.* Basic Books, New York.

[4] W.H. Calvin (1996) *The Cerebral Code: Thinking a Thought in the Mosaics of the Mind.* MIT Press, Cambrigde, MA.

[5] D. Cliff (1991) Computational Neuroethology: A Provisional Manifesto. In: *From Animals To Animats: Proceedings of the First International Conference on simulation of Adaptive Behavior.* J.A. Meyer and S. Wilson (eds.) MIT Press, Cambridge, MA.

[6] For a recent report, see, e.g. D. Cohn and D. Lewis (1995) *Working notes from the AAAI-95 symposium on Active Learning.* MIT, Cambridge, MA.

[7] R. Dawkins (1982) *The Extended Phenotype.* Oxford University Press, Oxford.

[8] G. Edelman (1987) *Neural Darwinism.* Basic Books, New York.

[9] Floreano D., F. Mondada, (1996) Evolution of Homing Navigation in a Real Mobile Robot. *IEEE Transactions on Systems, Man and Cybernetics — Part B, Special Issue on Learning Autonomous Robots* **26** 396 - 407.

[10] L.J. Fogel, A.J. Owens and M.J. Walsh (1966) *Artificial Intelligence through Simulated Evolution.* Wiley, New York.

[11] S. Handley (1994) The automatic generation of Plans for a Mobile Robot via Genetic Programming with Automatically defined Functions. In: *Advances in Genetic Programming.*, K. Kinnear (ed.) MIT Press, Cambridge, MA.

[12] I. Harvey, P. Husbands and D. Cliff (1993) Issues in evolutionary robotics. In: *From Animals To Animats 2: Proceedings of the Second International Conference on simulation of Adaptive Behavior.* J.A. Meyer and S. Wilson (eds.) MIT Press, Cambridge, MA.

[13] D.R. Hofstadter (1985) *Metamagical Themas.* BasicBooks, New York.

[14] J. Holland (1975) *Adaption in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI.

[15] W. James (1890) *The principles of psychology. Vol.1.* Originally published: Henry Holt, New York.

[16] N.K. Jerne (1967) Antibodies and learning: Selection versus instruction. In: *The Neurosciences: A Study Program.* G. C. Quarton, T. Melnechuk and F. O. Schmitt (eds.) Rockefeller University Press, New York, 200-205.

[17] J. Koza (1992) *Genetic Programming.* MIT Press, Cambridge, MA.

[18] O. Michel (1995). *Khepera Simulator v2.0, User Manual.*

[19] F. Mondada, E. Franzi, and P. Ienne (1993) Mobile robot miniaturization. In *Proceedings of the third international Symposium on Experimental Robotics.* Kyoto, Japan.

[20] P. Nordin (1994) A Compiling Genetic Programming System that Directly Manipulates the Machine-Code. In: *Advances in Genetic Programming.* K. Kinnear (ed.) MIT Press, Cambridge, MA.

[21] P. Nordin and W. Banzhaf (1995a) Evolving Turing Complete Programs for a Register Machine with Self-Modifying Code. In: *Proceedings of Sixth International Conference of Genetic Algorithms, Pittsburgh, 1995.* L. Eshelman (ed.), Morgan Kaufmann, San Mateo, CA.

[22] P. Nordin and W. Banzhaf (1995b) Complexity Compression and Evolution. In: *Proceedings of Sixth International Conference of Genetic Algorithms, Pittsburgh, 1995,* L. Eshelman (ed.) Morgan Kaufmann, San Mateo, CA.

[23] P. Nordin and W. Banzhaf (1997a) An On-Line Method to Evolve Behavior and to Control a Miniature Robot in Real Time with Genetic Programming. *Adaptive Behavior* **5** 107 - 140.

[24] P. Nordin and W. Banzhaf (1997b) Real Time Control of a Khepera Robot using Genetic Programming. *Control and Cybernetics* **26** (3), 533 - 561.

[25] M. Olmer, P. Nordin and W. Banzhaf (1996) Evolving real-time behavioral modules for a robot with GP. In: *Robotics and Manufacturing.* M. Jamshidi, F. Pin, P. Dauchez (Eds.) Asme Press, New York, 675 - 680.

[26] I. Rechenberg (1975) *Evolutionstrategien.* Fromann-Holtzboog, Stuttgart.

[27] C.W. Reynolds (1994) Evolution of Corridor Following Behavior in a Noisy World, in: *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior (SAB94),* D. Cliff, P. Husbands, J.-A. Meyer and S. Wilson (eds.), MIT Press, Cambridge, MA.

[28] C.W. Reynolds (1994) Evolution of Obstacle Avoidance Behavior, in: *Advances in Genetic Programming.,* K. Kinnear (ed.) MIT Press, Cambridge, MA.