The 2009 IEEE/RSJ International Conference on
Intelligent Robots and Systems
October 11-15, 2009 St. Louis, USA

Grammatical Evolution of a Robot Controller

Robert Burbidge, Joanne H. Walker and Myra S. Wilson

***Abstract*— An autonomous mobile robot requires an onboard controller that allows it to perform its tasks for long periods in isolation. One possibility is for the robot to adapt to its environment using some form of artificial intelligence. Evolutionary techniques such as genetic programming (GP) offer the possibility of automatically programming the controller based on the robot's experience of the world. Grammatical evolution (GE) is a recent evolutionary algorithm that has been successfully applied to various problems, particularly those for which GP has been successful. We present a method for applying GE to autonomous robot control and evaluate it in simulation for the Khepera robot.**

I. INTRODUCTION

An autonomous, mobile robot is a mechanical device with an onboard controller, sensors to observe the world and actuators to move around it; together with a power source [1]. In this work we take the mechanical architecture as given (although this could be evolved [2, Section 11.3]) and aim to optimize the controller. We consider a small, mobile robot with two wheels and various sensors together with an onboard, programmable micro-controller.

Software and hardware comprise the controller. There have been some attempts to use electronic hardware that can adapt to changes in the environment (e.g. [3]), although the hardware is in general fixed. The software can either be optimized in the lab, or during the lifetime of the robot, or both [4]. We initially consider the case where the controller is optimized in the lab in simulation. Prospects for ongoing adaptation during the lifetime of the robot are discussed in Section V-B.

The last two decades have seen a shift in robot control from an engineering perspective to a behavioral perspective [5] and more recently to adaptive and evolutionary perspectives [2]. Genetic programming (GP) was first applied to simulated robot control in 1994 [6]. The GP system evolved LISP S-expressions that executed a limited set of movements in order to push a box to the edge of an arena. The robot was provided with sonar (i.e. distance sensors) together with detectors that indicated whether the robot had bumped into something, or become stuck. This approach contrasted with an approach based on reinforcement learning [7]. It was argued that the GP was much closer to 'automatic programming' than was the reinforcement learning approach.

The first attempt to control a real robot using GP evolved low-level machine code [8], [9]. A linear GP (i.e. a variable length genetic algorithm) was used to evolve a sequence of instructions that took eight active infra-red (i.e. distance) sensors as input and gave motor speeds for each wheel as output. The task was the now-standard one of obstacle avoidance. Significant in these papers was the use of 'probabilistic sampling' in the fitness function in order to reduce training time. The evolution was further speeded up by a factor of 40 by using a GP to model the environment using symbolic regression [10], [11].

Grammatical evolution (GE), a technique related to GP (see Section II), was first applied to robot control in 1999 [12]. There has been little subsequent work on the use of GE for the automatic programming of robots [13], [14]. GE has not been fully evaluated for robot control but has been successfully applied to other noisy, dynamic optimization problems such as adaptive trading [15, ch. 14]. In this paper, we demonstrate the applicability of GE to robot control.

In Section III, we present a grammar for the automatic programming of robots. This is inspired by previous work in evolutionary robotics [10]. In Section IV, we describe the experimental conditions, define an objective function for a goal-finding task and present the results. We close with some conclusions and suggestions for further work.

II. BACKGROUND

Grammatical Evolution (GE) is a method for evolving computer programs in any language. The genotype is a binary string and the phenotype is syntactically correct code. The genotype is composed of 8-bit sequences coding for pseudo-random integers in 0–255. The mapping from these integers to compilable code is achieved through a set of production rules in a generative grammar. A generative grammar is a meta-syntax for a formal language, which in the context of GE is a programming language such as C. An example is given below in Section III.

This is in contrast to genetic programming (GP) [16] in which the genotype is a LISP S-expression so that there is little distinction between the genotype and the phenotype. In GP there is a one-to-one mapping from genotype to phenotype.

GE employs a generative grammar in Backus Naur Form (BNF). A BNF grammar is composed of a set of terminals which are items in the language, a set of non-terminals, and a set of production rules that map non-terminals to terminals together with a start symbol which is a non-terminal. The production rules are repeatedly applied to the start symbol until there are no remaining non-terminals. The result is a list of terminals, i.e. items from the language, that form syntactically correct code.

R. Burbidge, J.H. Walker* and M.S. Wilson are with the Department of Computer Science, Aberystwyth University, Penglais, Aberystwyth, SY23 3DB, UK {rvb, jnw, mxw}@aber.ac.uk
*Corresponding author.

The first step in GE is thus to decide upon a language and define a BNF for that language. One possibility is to define a domain specific language (DSL). This is an approach often taken in genetic programming (GP), e.g. for a symbolic regression problem the items of the DSL could be $\{+, -, \times, /, \sin, \cos, \exp, \log, x\}$. The input to the program is x and the program implements a function of x.

GE was introduced in [17] in which a BNF grammar is given for the above DSL for a symbolic regression problem described in [16]. Since the genotype is a binary string there is no need for problem specific genetic operators, and GE employs the standard operators of variable-length genetic algorithms (GAs). For an introduction to GAs see [18].

GE has been applied to symbolic regression, finding trigonometric identities, and symbolic integration [19]; to the Santa Fe Trail problem [20], [21]; and to the generation of caching algorithms in C [22]. Steady state selection[1] was found to outperform generational selection[2] on these and other problems by increasing the likelihood of finding a solution in a given number of generations.

In several cases, GE has outperformed GP in terms of the probability of finding a solution [20] and generalization [22]. As noted above (Section I), there have been few applications of GE to robot control, although this is an area in which GP has been successfully applied [6], [8], [9], [10], [11], [23], [24]. Note that [24] used a graph grammar to grow neural networks for robot control and as such is a precursor to the GE approach.

III. Grammatical Evolution for Robot Control

One of the advantages of GE over GP is that the population is simple to initialize and few constraints need be imposed on the genetic operators of crossover and mutation [25, p. 57]. The obverse of this is that the inductive bias is in the grammar; the population may evolve in an unexpected way. Nevertheless, given this feature of GE it is simple to specify grammars that mimic other evolutionary learning systems.

One successful application of GP to control of a Khepera robot generated programs for a register machine [10]. An example individual program written as a C program follows.

```
m[1]=s[13] >> s[10];
r[3]=s[11] - +0x3a2;
m[1]=s[9] | s[15];
r[8]=s[9] ^ r[19];
m[1]=s[8] - r[5];
m[0]=r[10] << r[2];
m[0]=s[10] << r[16];
m[1]=r[21] >> r[6];
m[0]=r[0] << s[11];
m[1]=r[6] << r[0];
r[7]=r[21] >> r[2];
m[0]=r[12] << +0xd3f;
m[1]=s[13] << r[7];
```

Each individual is composed of simple register instructions operating on input sensors s_i, registers r_i and 13-bit integer constants (represented in hex). The output is the motor speed values m_0, m_1. The program is treated as the genotype of a linear GP.

In the GE approach, the phenotype is the same but the genotype is a binary string. The mapping from genotype to phenotype is achieved by the grammar. A grammar that generates programs for a register machine is given below[3].

```
<code> ::= <line><code> | <line>
<line> ::= <lhs>=<rhs>;\n
<lhs> ::= <var> | <motor>
<motor> ::= m[0] | m[1]
<var> ::= r[0]|r[1]|...|r[13]
<rhs> ::= <arg1> <op> <arg2>
<arg1> ::= <var> | <sensor>
<sensor> ::= s[0]|s[1]|...|s[15]
<arg2> ::= <const> | <var> | <sensor>
<const> ::= <pm>0x<hex><hex><hex>
<pm> ::= +|-
<hex> ::= 0|1|...|f
<op> ::= + | - | *
       | & | BITOR | ^ | << | >>
```

The binary genotype is mapped to an integer string, $n_1, n_2, \ldots$, in analogy with transcription of DNA to RNA. The phenotype is generated by starting with the first non-terminal, `<code>`, and replacing it with one of the options on the RHS of the production rule. Suppose the integer sequence is $(14, 131, 36, 89, 191, 20, 65, 2, 238)$. For each non-terminal we look up the corresponding production rule and count the number of options on the RHS. If there is only one option, the LHS non-terminal is replaced by the RHS. If there are $r > 1$ options then we take the next integer, n_i, and calculate $n_i \bmod r$ to select an option to replace the RHS. Since $n_1 \bmod 2 = 0$, `<code>` becomes `<line><code>`. The non-terminal `<line>` becomes `<lhs>=<rhs>`. $n_2 \bmod 2 = 1$, so that `<lhs>` becomes `<motor>`. Then we have $n_3 \bmod 2 = 0$, so that `<motor>` becomes `m[0]`, which is a terminal, i.e. an element of the final program. This example and the next few steps are shown below.

```
start: <code>
n1%2:  <line><code>
       <lhs>=<rhs>;\n<code>
n2%2:  <motor>=<rhs>;\n<code>
n3%2:  m[0]=<rhs>;\n<code>
       m[0]=<arg1> <op> <arg2>;\n<code>
n4%2:  m[0]=<sensor> <op> <arg2>;\n<code>
n5%8:  m[0]=s[7] <op> <arg2>;\n<code>
n6%8:  m[0]=s[7] & <arg2>;\n<code>
n7%3:  m[0]=s[7] & <sensor>;\n<code>
n8%8:  m[0]=s[7] & s[2];\n<code>
...
```

[1]Steady state selection involves evaluating only a subset of the population of potential solutions each generation.

[2]Generational selection involves evaluating the entire population each generation.

[3]`BITOR` is a macro for the C operator '|', to avoid confusion with '|' as a separator in the grammar.

where % is the mod operator. This process continues until there are no non-terminals remaining. Header and footer code are added and the result is a compilable C function mapping sensors to motors that can be used to control the robot.

The mapping of integers to production rules and the generation of terminals is analogous to the translation of RNA into amino acids and proteins. The resulting program, i.e. the phenotype, is then compiled and used to control the robot for a given task. A fitness score is calculated and ascribed to the genotype for evolutionary optimization.

It is possible for an individual to run out of genes in the mapping process. In this case the genotype is wrapped, i.e. we go back to the beginning of the integer sequence and scan through again. If there are still non-terminals remaining then the phenotype is invalid and the genotype is given a fitness of $-\infty$.

IV. Experiments

A genetic algorithm (GA) is used to optimize the phenotype. The phenotype is used to control a Khepera robot in simulation. The task is to navigate toward a point light source whilst avoiding obstacles. The various components are described in more detail below.

A. Genetic Algorithm

The genotype is evolved using the genetic operators of single-point crossover, with probability 0.9, and bitwise mutation, with probability 0.01. The population size is fixed at 500. A steady state GA is used as is common for GE. Each generation, four individuals are selected uniformly at random without replacement. These are evaluated on the robot and the best two are chosen as parents to produce two children. The children replace the worst two of the four. This is the same as in [10]. The maximum number of generations is 3500, giving 1400 evaluations in total. The GA is implemented using GAlib 2.4.6 [26].

The genotypes are mapped to phenotypes using the grammar above, modified to ensure the phenotypes are sufficiently complex. Specifically, we set the maximum number of register instructions to 256 as in [10]. The GE is implemented with libGE 0.26 [27] using the in-built sensible initialization.

B. Khepera

The Khepera is a standard, autonomous, miniature robot [28]. It is circular with a diameter of 5.5 cm and a height of 3 cm. It has two wheels whose speed and direction can be set independently. The maximum speed for each wheel is 127 rad s^{-1}. There are eight active infra-red sensors (see Fig. 1) that can detect objects and measure distances up to about 5 cm away. The IR sensors also detect ambient light up to around 25 cm away for a 1 W source. Multiplicative noise of 10% is added to the sensor readings to provide a more realistic simulation.

The environment for the Khepera is a 1 m^2 arena with ambient overhead lighting and a number of obstacles. The obstacles are 5 cm $\times$ 5cm and are randomly positioned in the arena. The number of obstacles for a particular trial is $\sim \mathrm{Binomial}(40, 0.5)$. The robot starts in a corner with a random orientation. The robot and its environment are simulated in 3-d with Webots 6.0.1 [29] (see Fig. 2). Note that these are the same experimental conditions as the '10% general world' used in [4] and [30].

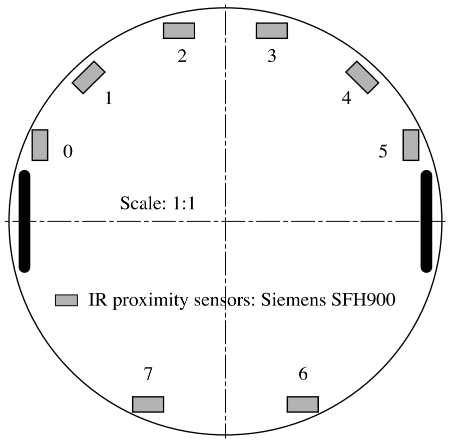

Fig. 1. Position of the infra-red sensors on the Khepera.

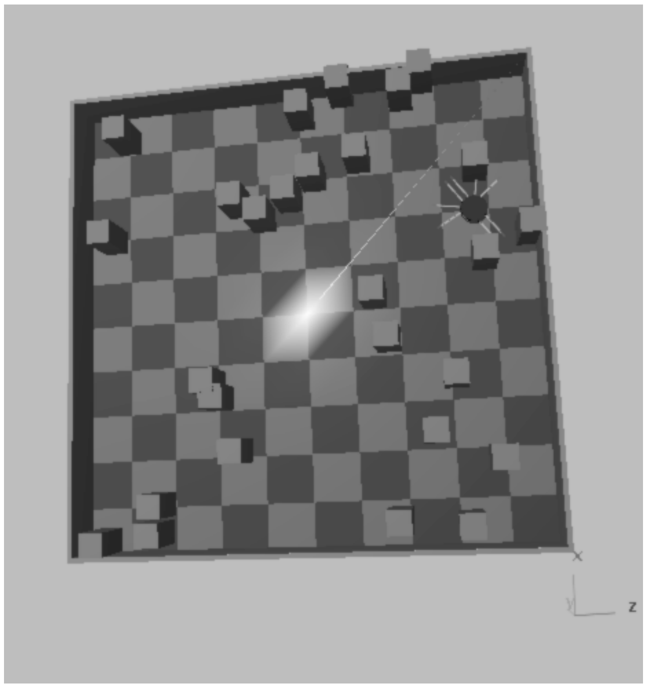

Fig. 2. The Khepera Robot in its environment.

C. Find Light Task

The objective of the find light task is to navigate from the corner of the arena to the centre, where there is a point light, whilst avoiding obstacles. A trial lasts for 1000 steps or until the robot finds the light or stops moving, if this is sooner. A step is 64 ms as this is the minimum time for a detectable change in the sensor values. The fitness has two components: one to reward finding the light and one to penalize collisions.

The ambient light sensors, ls_i, return values in $[0, 511]$ with a value of around 450 in the dark and 0 directly at the light. The reward, $r(ls)$, is the decrease in the average light reading, $\sum ls_i/8$, over a trial. The reward takes values in $[-511, 511]$.

Although it is possible to detect a collision in simulation, this is not possible for the physical robot and so the number of collisions was not used in the fitness function. Instead, in each time step the robot is penalized for being close to an obstacle, with the penalty increasing exponentially with proximity. The distance sensors, ds_i, return values in $[0, 1023]$ with a higher value indicating proximity. The formula used is:

$$p(ds) = \frac{\sum_{\text{steps}} \frac{\exp\left(\frac{\max ds_i}{1023}\right)-1}{e-1}}{1000} \times 511.$$

This takes values in $[0, 511]$ and is subtracted from the reward.

The overall fitness is:

$$f(ls, ds) = \alpha r(ls) - p(ds),$$

where α is a prior weight for maximizing the reward over minimizing the penalty; we use $\alpha = 4$ in the experiments reported here. The fitness thus takes values in $[-2555, 2044]$ with 0 representing a trial where the robot moves from darkness to the light without coming close to any obstacles. Owing to the ambient light, the value of $\sum ls_i/8$ is around 270 at the start of the trial and this is thus the maximum possible $r(ls)$. The highest attainable fitness under these experimental conditions is thus around 1080.

There is no explicit pressure to move fast, in a straight line, or to minimize the number of steps to find the light (although there is an implicit pressure in the penalty for the last of these). Note also that there is a reward for partial task completion.

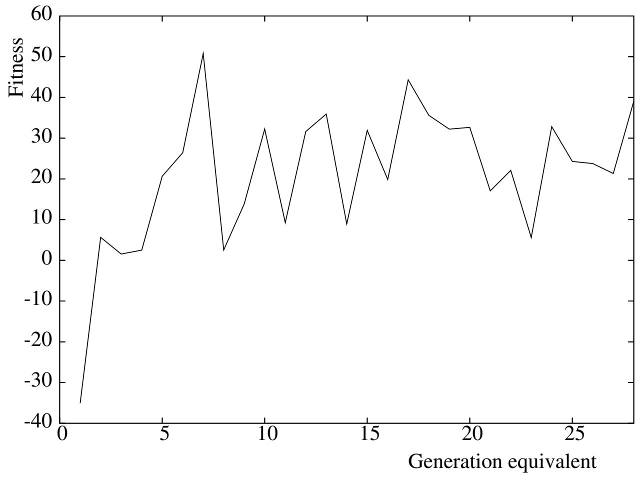

Fig. 3. The mean fitness per generation equivalent.

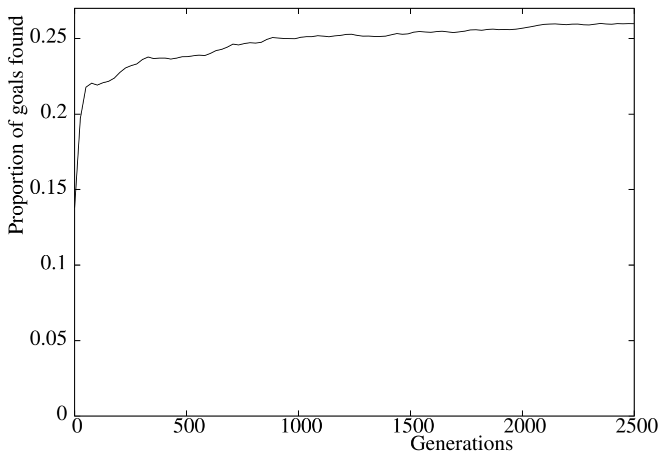

Fig. 4. The cumulative proportion of successful trials per generation, smoothed with a cubic spline.

D. Results

Each generation, four individuals are given a fitness. The population size is 500 and we use *generation equivalent* to mean 500 genotype evaluations. Overall, about 8% of phenotypes are invalid and these are not evaluated on the robot. The following analysis excludes these. All results presented are the average of two independent runs.

The best fitness each generation equivalent is always greater than 1080, indicating that the robot had no collisions and terminated at the goal. In 500 random trials there will be one that is easy and good performance on one trial does not imply good performance on other trials. The mean fitness is a better indicator of robust learning. The mean fitness is shown in Fig. 3. The fitness increases in the early stages then levels off. The fitness is very noisy since the environment is dynamic. Similar results were reported in [30] and [4] for the same problem, albeit with a different fitness function.

It is possible that the initial population became overfitted to the environments seen in the early stages and was then not able to generalize to the wide range of environments subsequently seen. This illustrates the usefulness of simulation when evaluating an algorithm for robot control.

The fitness is only a proxy for the ultimate objective, which is to find the light. If all of the light sensors return 0 then the robot has reached the goal. The cumulative proportion of successful trials increased from 5.5% to 12.5% during the run.

Observation of the robot showed that on a number of occasions the robot moved very close to the light then stopped without all light sensors returning 0. Such a robot gets a high fitness so there is little incentive to fine-tune the position to be exactly above the light. If we include these trials then the cumulative proportion of successful trials increases from 14% to 26%. Almost all of this improvement is in the early stages of the evolution, see Fig. 4. Since only four of 500 individuals are evaluated each generation, the early stages are mostly random search. There is continuing improvement up to around 2500 generations (i.e. 20 generation equivalents), after which there is no further improvement.

The number of collisions per minute decreases gradually throughout a run as shown in Fig. 5. This is the number of collisions incurred by the robot for all phenotypes that are evaluated. The performance of the best phenotypes is discussed below.

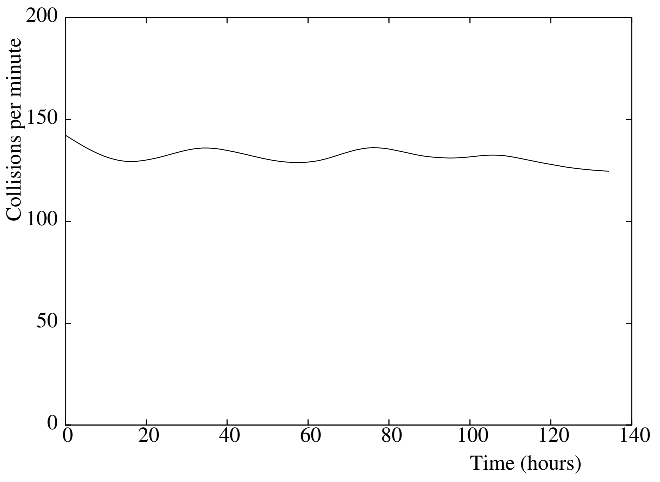

Fig. 5. The number of collisions per minute during the life of the robot.

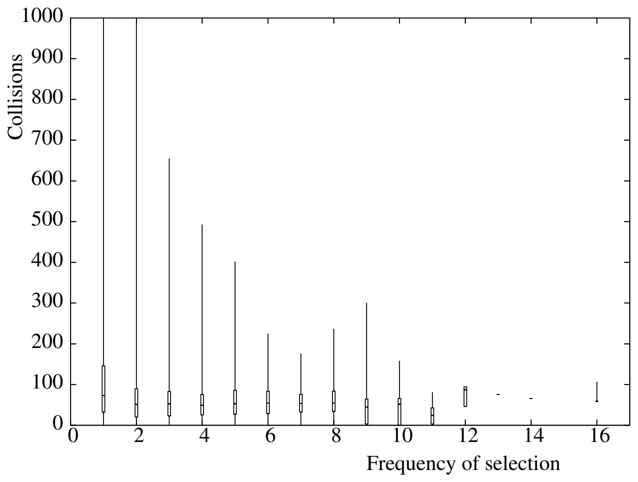

Fig. 6. The number of collisions per phenotype as a function of the number of times the phenotype was selected. A phenotype lives for up to 64 s.

E. Bloat

The average length of the genotype increases linearly during a run, a phenomenon known as bloat in the GP community [25, p. 101]. However, GE has an advantage over GP in that the size of the phenotype depends also on the grammar used. In our case, the phenotype is constrained to be exactly 256 register instructions. The grammar could be defined to allow variable length phenotypes and designed such that long phenotypes are rare.

Alternatively, the genotypic bloat in GE can be reduced by only applying the crossover operator within the effective part of the genotype, or by pruning the genotype [17].

F. Generalization

Although the results so far look promising there is no reason to believe that the evolutionary search is better than random (a point often overlooked). If a phenotype performs well in one trial it will be selected and possibly reevaluated later in the run, in a different environment. If the search is better than random then the phenotype should perform well in subsequent trials.

In Fig. 6, the number of collisions per phenotype is plotted as a function of the number of times the phenotype was selected. The more frequently selected phenotypes incur fewer collisions. The average number of collisions for phenotypes that were selected at least six times is half the overall average.

Similarly, the proportion of successful trials for phenotypes that were only selected once is 5%. The proportion of successful trials for phenotypes that were selected at least six times is 41%. This shows that the evolved controllers are able to generalize to new environments.

G. Discussion

The optimal controller is one that will navigate toward the light without collision for any initial conditions. Since it is not possible to evaluate the controller in all possible scenarios we must sample the problem space. Previous work on similar problems [31], [4] evaluated the controller over three trials of 3000 steps. At the opposite extreme to this, [10] used each individual to control the robot for one time-step of 300 ms (although for a simpler task). The approach taken here falls between these two. Morevoer, the reward component of our fitness function focusses on task completion rather rather than the ambient light during the robot's lifetime and is thus implicit rather than explicit [2, p. 66].

The optimization problem tackled here is harder than in previous work. The space of possible phenotypes to search over is the same as in [10]. The behavior required is more complex in that the robot must find the goal whilst avoiding obstacles, whereas in [10] only obstacle avoidance was required. Moreover, the environment is more cluttered in the work presented here.

A similar optimization problem was tackled in [31] and [4]. The controller was a problem-specific behavior-based schema with only five parameters to be optimized. A direct comparison with those works is not possible, but the results achieved with GE are qualitatively the same and were achieved in the same amount of computational time. The advantage of the GE approach is that it is not necessary to design a problem-specific controller.

V. Conclusions and Further Work

In the following we summarize the main points of the paper and suggest some directions for further work.

A. Conclusions

Evolutionary optimization and learning has been successfully applied to robot control. One such technique is genetic programming (GP). Grammatical evolution (GE) is similar to GP in that the phenotype is a computer program. It differs in that the genotype is a binary string. Since GP has been successfully applied to robot control and GE has been shown to be a useful alternative to GP in several problem domains it is natural to evaluate GE for robot control.

We have presented a BNF grammar that generates C code for robot control and used a GA to search the space of such programs. We have demonstrated the applicability of GE to robot control on a simple goal-finding task in simulation. Compared to random controllers, the evolved controllers incur fewer collisions, have greater success in navigating to the goal, and generalize better to new environments.

B. Further Work

Up to now, we have been vague as to what 'autonomous' means. The controller presented here requires access to a file store and a C compiler. It could be immediately ported to a physical Khepera provided the optimization and compilation was done on a computer attached to the Khepera by a serial tether. To use it as is on a truly autonomous robot would require the robot to have its own OS, for example the Pioneer robot. In any case, the controllers found by the algorithm in simulation should be evaluated on a physical robot since this is how they will be used.

The compilation and system calls are computationally intensive. Alternative approaches include (i) using an interpreted language such as LISP, (ii) directly evolving machine code as in [10], and (iii) using TinyCC which is a fast compiler with a backend to compile and execute evolved code.

We have only evaluated one grammar, producing a program of register instructions that is difficult to interpret. It is simple to write grammars for other control architectures, such as neural networks or behavior-based schemas. The inductive bias induced by the grammar is an interesting avenue for further work.

We can easily take advantage of the portability of GE by applying the algorithm presented here to other tasks in other environments for other robots with little change in the grammar and no modifications to the code.

References

[1] U. Nehmzow, *Mobile Robotics: A Practical Introduction*. Springer, 2000.

[2] S. Nolfi and D. Floreano, *Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines*. The MIT Press, 2000.

[3] D. Keymeulen, M. Iwata, Y. Kuniyoshi, and T. Higuchi, "Comparison between an off-line model-free and an on-line model-based evolution applied to a robotics navigation system using evolvable hardware," in *Proceedings of the Sixth International Conference on Artificial Life*, C. Adami, R. Belew, H. Kitano, and C. Taylor, Eds. 26–29 June, Los Angeles, USA: The MIT Press, 1998, pp. 109–209.

[4] J. H. Walker, S. M. Garrett, and M. S. Wilson, "The balance between initial training and lifelong adaptation in evolving robot controllers," *IEEE Transactions on Systems, Man, and Cybernetics—Part B: Cybernetics*, vol. 36, no. 2, pp. 423–432, April 2006.

[5] R. C. Arkin, *Behavior-Based Robotics*. The MIT Press, 1998.

[6] J. Koza and J. Rice, "Automatic programming of robots using genetic programming," in *Proceedings of the Tenth National Conference on Artifical Intelligence*. 12–16 July, San Jose, CA, USA: AAAI Press / MIT Press, 1992, pp. 194–201.

[7] S. Mahadevan and J. Connell, "Automatic programming of behaviour-based robots using reinforcement learning," in *Proceedings of the Ninth National Conference on Artificial Intelligence*, vol. 2. 15–19 July, Anaheim, CA, USA: AAAI Press / MIT Press, 1991, pp. 768–773.

[8] P. Nordin and W. Banzhaf, "Genetic programming controlling a miniature robot," in *Working Notes of the AAAI-95 Fall Symposium on Genetic Programming*, J. Koza and E. Siegel, Eds. MIT, Cambridge, MA, USA: AAAI Press / MIT Press, 1995, pp. 61–67.

[9] ——, "An on-line method to evolve behaviour and to control a miniature robot in real time with genetic programming," *Adaptive Behaviour*, vol. 5, no. 2, pp. 107–140, 1997.

[10] ——, "Real time control of a Khepera robot using genetic programming," *Cybernetics and Control*, vol. 26, no. 3, pp. 533–561, 1997.

[11] P. Nordin, W. Banzhaf, and M. Brameier, "Evolution of a world model for a miniature robot using genetic programming," *Robotics and Autonomous Systems*, vol. 25, pp. 105–116, 1998.

[12] M. O'Neill and C. Ryan, "Steps towards Khepera dance improvisation," in *Proceedings of the First International Khepera Workshop*, A. Löffler, F. Mondada, and U. Rückert, Eds. 10–11 December, Paderborn, Germany: HNI-Verlagsschriftenreihe, 1999.

[13] M. O'Neill, J. Collins, and C. Ryan, "Automatic generation of robot behaviours using grammatical evolution," in *Proceedings of the Fifth International Symposium on Artificial Life and Robotics (AROB)*, M. Sugisaka and H. Tanaka, Eds. 26–28 January, Oita, Japan: AROB, 2000, pp. 351–354.

[14] ——, "Automatic programming of robots," in *Proceedings of the 11th Irish conference on Artificial Intelligence and Cognitive Science (AICS2000)*, 23–25 August, Galway, Ireland, 2000.

[15] A. Brabazon and M. O'Neill, Eds., *Biologically Inspired Algorithms for Financial Modelling*. Springer, 2006.

[16] J. Koza, *Genetic Programming: On the Programming of Computers by means of Natural Selection*. John Wiley & Sons, 1992.

[17] C. Ryan, J. Collins, and M. O'Neill, "Grammatical evolution: evolving programs for an arbitrary language," in *Genetic Programming, First European Workshop, EuroGP'98*, ser. Lecture Notes in Computer Science, W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, Eds., vol. 1391. 14–15 April, Paris, France: Springer, 1998, pp. 83–95.

[18] M. Mitchell, *An Introduction to Genetic Algorithms*. The MIT Press, 1998.

[19] M. O'Neill and C. Ryan, "Grammatical evolution: a steady state approach," in *Proceedings of the Second International Workshop on Frontiers in Evolutionary Algorithms*, 1998, pp. 419–423.

[20] ——, "Automatic generation of high level functions using evolutionary algorithms," in *Proceedings of SCASE 1999, Soft Computing and Software Engineering Workshop*, Limerick, Ireland, 1998.

[21] ——, "Evolving multi-line compilable C programs," in [32], pp. 83–92.

[22] ——, "Automatic generation of caching algorithms," in *Evolutionary Algorithms in Engineering and Computer Science*, K. Miettinen, P. Neittaanmäki, M. Mäkelä, and J. Périaux, Eds. 30 May–3 June, Jyväskylä, Finland: Wiley, 1999, pp. 127–134.

[23] G. Adorni, S. Cagnoni, and M. Mordonini, "Genetic programming of a goal-keeper control strategy for the RoboCup middle size competition," in [32], pp. 109–119.

[24] F. Gruau, "Automatic definition of modular neural networks," *Adaptive Behavior*, vol. 3, no. 2, pp. 151–183, 1994.

[25] R. Poli, W. B. Langdon, and N. F. McPhee, *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008, (With contributions by J. R. Koza).

[26] GAlib, a C++ Library of Genetic Algorithm Components, written by Matthew Wall at MIT. [Online]. Available: http://lancet.mit.edu/ga/

[27] libGE, a Grammatical Evolution Library, written by Miguel Nicolau and Darwin Slattery at the University of Limerick. [Online]. Available: http://bds.ul.ie/libGE/

[28] F. Mondada, E. Franzi, and P. Ienne, "Mobile robot miniaturisation: a tool for investigation in control algorithms," in *Proceedings of the Third International Symposium on Experimental Robotics*, T. Yoshikawa and F. Miyazaki, Eds., Kyoto, Japan, 1993.

[29] Webots, commercial Mobile Robot Simulation Software, from Cyberbotics Ltd. [Online]. Available: http://www.cyberbotics.com

[30] S. Darby, J. H. Walker, and M. S. Wilson, "Transfer of evolutionary methods between robots," in *Proceedings of Towards Autonomous Robotic Systems 2007*, M. S. Wilson, F. Labrosse, U. Nehmzow, C. Melhuish, and M. Witkowski, Eds. University of Wales, Aberystwyth, 2007, pp. 62–69.

[31] A. Ram, R. Arkin, G. Boone, and M. Pearce, "Using genetic algorithms to learn reactive control parameters for autonomous robotic navigation," *Adaptive Behavior*, vol. 2, no. 3, pp. 277–304, 1994.

[32] R. Poli, P. Nordin, W. Langdon, and T. Fogarty, Eds., *Genetic Programming, Second European Workshop, EuroGP'99*, ser. Lecture Notes in Computer Science, vol. 1598. 26–27 May, Goteborg, Sweden: Springer, 1999.