Cellular Encoding for Interactive Evolutionary Robotics

Frédéric Gruau and Kameel Quatramaran

University of Sussex,
School of Cognitive and Computing Sciences,
Evolutionary and Adaptive SYstems Group (EASY)
Falmer, Brighton, BN1 9QH UK
http://www.cogs.susx.ac.uk/
and
Centrum Voor Wiskunde en Informatica
Department of Algorithms and Architecture
Kruislaan 413 SJ Amsterdam
gruau@cwi.nl, http://www.cwi.nl/˜ gruau/gruau/gruau.html

Abstract

This work reports experiments in interactive evolutionary robotics. The goal is to evolve an Artificial Neural Network (ANN) to control the locomotion of an 8-legged robot. The ANNs are encoded using a cellular developmental process called cellular encoding. In a previous work similar experiments have been carried on successfully on a simulated robot. They took however around 1,000,000 different ANN evaluations. In this work the fitness is determined on a real robot, and no more than a few hundreds evaluations can be performed. Various ideas were implemented so as to decrease the required number of evaluations from 1,000,000 to 200. First we used cell cloning and link typing. Second we did as many things as possible interactively: interactive problem decomposition, interactive syntactic constraints, interactive fitness. More precisely: 1- A modular design was chosen where a controller for an individual leg, with a precise neuronal interface was developed. 2- Syntactic constraints were used to promoting useful building blocs and impose an 8-fold symmetry. 3- We determine the fitness interactively by hand. We can reward features that would otherwise be very difficult to locate automatically. Interactive evolutionary robotics turns out to be quite successful, in the first bug-free run a global locomotion controller that is faster than a programmed controller could be evolved.

1 Introduction

1.1 The motivation for Interactive Evolutionary Algorithm

In [3] Dave Cliff, Inman Harvey and Phil Husbands from the university of Sussex lay down a chart for the development of cognitive architectures, or control systems, for situated autonomous agent. They claim that the design by hand of control systems capable of complex sensorimotor processing is likely to become prohibitively difficult as the complexity increases, and they advocate the use of Evolutionary Algorithm (EA) to evolve recurrent dynamic neural networks as a potentially efficient engineering method. Our goal is to try to present a concrete proof of this claim by showing an example of big (> 16 units) control system generated using EA. The difference between our work and what we call the "Sussex" approach is that we consider EAs as only one element of the ANN design process. An engineering method is something which is used to help problem solving, that may be combined with any additional symbolic knowledge one can have about a given problem. We would never expect EAs to do everything from scratch. Our view is that EA should be used

interactively in the process of ANN design, but not as a magic wand that will solve all the problems. In contrast with this point of view, Cliff, Harvey and Husband seem to rely more on EAs. In [3] they use a direct coding of the ANN. They find ANN without particular regularities, although they acknowledge the fact that a coding which could generate repeated structure would be more appropriate. The advantage of the Sussex approach is that it is pure

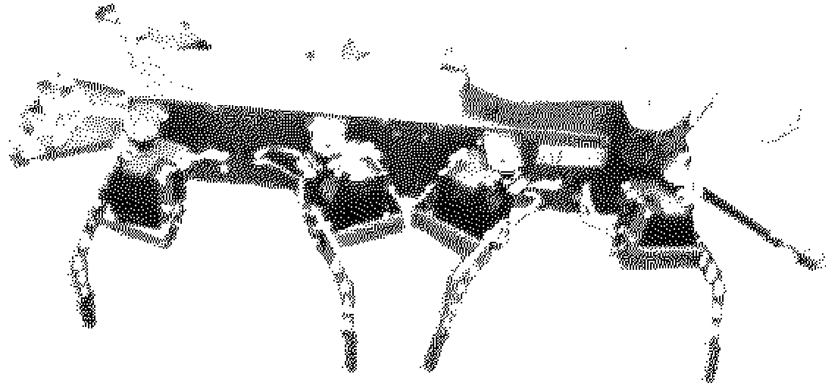
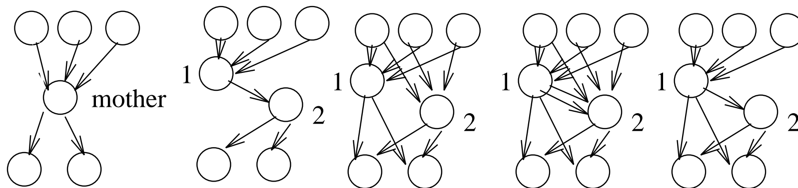


Figure 1: The OCT1 8-legged robot

The paper presents what is cellular encoding, cell cloning and link typing, how we use syntactic constraints, experiments done with only one leg, and then with all the legs. Automatically generated drawing represent different ANN that were found at different stages of the evolution. We discuss the behavior of the robot, and try to explain it based on an analysis on the architecture, whenever possible. The description tries to render how it feels to breed robots.

2 Review of Cellular Encoding

Cellular encoding is a language for local graph transformations that controls the division of cells which grow into an Artificial Neural Network (ANN) [5]. Other kind of developmental process have been proposed in the literature, a good review can be found in [8]. Many schemes have been proposed with partly the goal of modeling biological reality. Cellular encoding has been created with the sole purpose of computer problem solving, and its efficiency has been shown on a range of different problem, a review can be found in [4]. We explain the basic version of Cellular Encoding in this section. A cell has an input site and an output site and can be linked to other cells with directed and ordered links. A cell or a link also possesses a list of internal registers that represent local memory. The registers are initialized with a default value, and are duplicated when a cell division occurs. The registers contain neuron attributes such as weights and the threshold value. The graph transformations can be classified into cell divisions and modifications of cell and link registers.



combined. All the links are duplicated, and the two child cells are interconnected with two links, one for each directions. This division can generate completely connected sub-ANNs. The `CPO` division (CoPy Output) is a sequential division, plus the output links are duplicated in both child cells. Similarly, the `CPI` division (CoPy Input) is a sequential division, plus the input links are duplicated. Before describing the instructions used to modify cell registers it is useful to describe how an ANN unit performs a computation. The default value of the weights is 1, and the bias is 0. The default transfer function is the identity. Each neuron computes the weighted sum of its inputs, applies the transfer function and obtain s and updates the activity a using the equation $a = a + (s - a)/\tau$ where τ is the time constant of the neuron. See the figures 7 11 and 13 for examples of neural networks. The ANNs computation is performed with integers; the activity is coded using 12 bits so that 4096 corresponds to activity 1. The instruction `SBIAS` x sets the bias to $x/4096$. The instruction `DELTAT` sets the time constant of the neuron. `SACT` sets the initial activity of the neuron. The instruction `STEP` (resp LINEAR) sets the transfer function to the clipped linear function between -1 and $+1$ (resp to the identity function). The instruction PI sets the sigmoid to multiply all its input together. The `WEIGHT` instruction is used to modify link registers. It has k integer parameters, each one specifying a real number in floating point notation: the real is equal to the integer between -255 and 256 divided by 256. The parameters are used to set the k weights of the first input links. If a neuron happens to have more than k input links, the weights of the supernumerary input links will be set by default to the value 256 (i.e., $\frac{256}{256} = 1$).

The cellular code is a *grammar-tree* with nodes labeled by names of graph transformations. Each cell carries a duplicate copy of the grammar tree and has an internal register called a reading head that points to a particular position of the grammar tree. At each step of development, each cell executes the graph transformation pointed to by its reading head and then advances the reading head to the left or to the right subtree. After cells terminate development they lose their reading-heads and become neurons.

The order in which cells execute graph transformations is determined as follows: once a cell has executed its graph transformation, it enters a First In First Out (FIFO) queue. The next cell to execute is the head of the FIFO queue. If the cell divides, the child which reads the left subtree enters the FIFO queue first. This order of execution tries to model what would happen if cells were active in parallel. It ensures that a cell cannot be active twice while another cell has not been active at all. The `WAIT` instruction makes a cell wait for a specified number of steps, and makes it possible to also encode a particular order of execution.

We also used the control program symbol `PROGN`. The program symbol `PROGN` has an arbitrary number of subtrees, and all the subtrees are executed one after the other, starting from the subtree number one.

Consider a control problem where the number of control variables is n and the number of sensors is p. We want to solve this control problem using an ANN with p input units and n output units. There are two possibilities to generate those i/o units. The first method is to impose the i/o units using appropriate syntactic constraints. At the beginning of the development the initial graph of cells consists of p input units connected to a reading cell which is connected to n output units. The input and output units do not read any code, they are fixed during all the development. In effective these cells are pointers or place-holders for the inputs and outputs. The initial reading cell reads at the root of the grammar tree. It will divide according to what it reads and generate all the cells that will eventually generate the final decoded ANN. The second method that we often prefer to use, is to have the EA find itself the right number of i/o units. The development starts with a single cell connected to the input pointer cell and the output pointer cell. At the end of the development, the input (resp. output) units are those which are connected to the input (resp. output) pointer cell. We let the evolutionary algorithm find the right number of input and output unit, by putting a term in the fitness to reward the network which have a correct number of i/o units. The problem with the first method is that we can easily generate an ANN where all the output units output the same signals, and all the inputs are just systematically summed in a weighted sum. The second method works usually better, because the EA is forced to generate a specific cellular code for each i/o unit, that will specify how it is to be connected to the rest of the ANN, and with which weights. To implement the second method we will use the instruction BLOC which blocs the development of a cell until all its input neurons are neurons, and the instruction TESTIO which compares the number of inputs to a specified integer value, and sets a flag accordingly. The flag is later used to compute the fitness.

Last, the instruction `CYC` is used to add a recurrent link to a unit, from the output site to the input site. That unit can then perform other divisions, duplicate the recurrent link, and generates recurrent connections everywhere.

3 Enhancement of Cellular Encoding

We had to enhance cellular encoding with cloning division, and the use of types. We also implemented another way to obtain recurrent links. All these new elements are reported in this section.

The cloning operation is really easy to implement, it is done by encapsulating a division instruction into a PROGN instruction. After the division, the two child cells only modify some registers and cut some links, then they simply go to execute the next instruction of the PROGN, and since they both execute the same

```
<nn>[0..8];
<axiom> ::=  <nn>

<nn>  ::= ( PAR(<nn>)(<nn>) )
            | (  CPO(<nn>)(<nn>) )
            | ( SEQ (<nn>)(<nn>) )
            | (  <attribute> )

<attribute> ::=
          (PROGN : set[0..4] of
```

```
<nn>[6..20];
begin
<axiom> ::= (LABEL
  (SEQ (WAIT) (PAR
       (<nn>)
      (PROGN
         (PAR)
         (PAR(PAR(PAR(WAIT)(WAIT))(WAIT))(PAR(WAIT)(PAR(WAIT)(PAR(WAIT)(WAIT)))) )  ) ) ) )

<nn> ::=   (SEQ(<nn>)(<nn>))
         | (PAR(<nn>)(<nn>))
         | (SHARI1(<nn>)(<nn>))
         | (CPI(<nn>)(<nn>))
         | (CPO(<nn>)(<nn>))
         | (FULL(<nn>)(<nn>))
         | (<tunit>)
 | (<sunit>)

<tunit> ::= (PROGN  (STEP)
             (PROGN : set[1..3] of
           ( DELTAT (integer[1..40]))
    ( WEIGHT: list[8..8] of ( integer[ 256..+256]) )
    ( SBIAS  (integer[ 4096..+4096]))  )   )


<sunit> ::= (PROGN
                (LINEAR)
                (PROGN : set[2..2] of
    ( WEIGHT: list[8..8] of ( integer[ 1024..+1024]) )
    ( SBIAS  (integer[ 4096..+4096]))  )   )
```

Figure 5: Syntactic constraints used for the leg controller

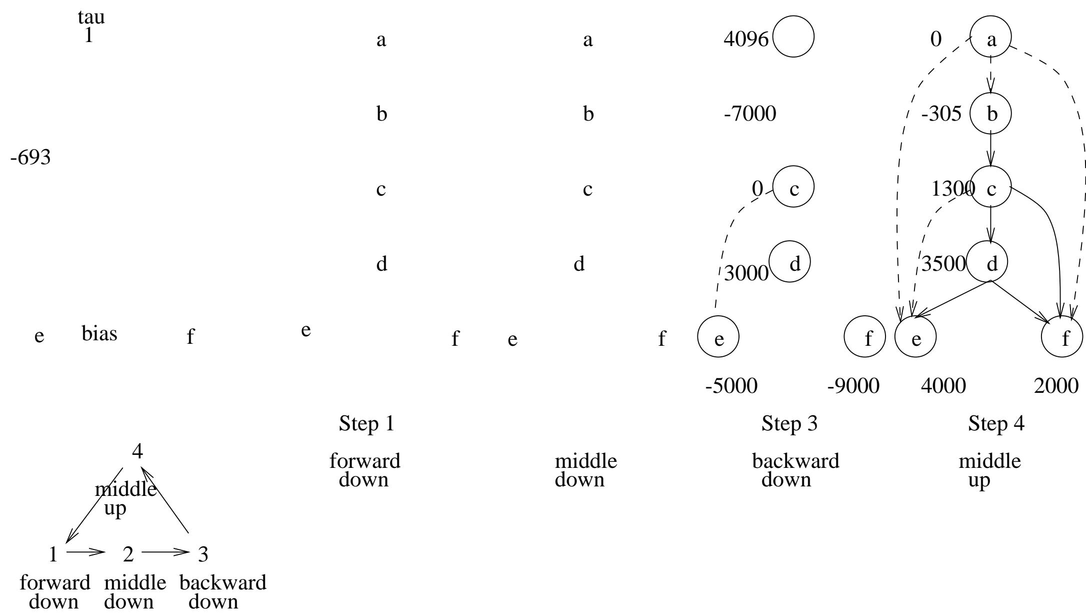
links are shared between the two child cells, the first child gets the first input, and the second child gets the other inputs. The ANN begins by generating 14 fake output units using parallel division (one clone and 7 normal). Those units reproduce the input signal. In this way, we can compare the movement on the leg whose controller is evolved, with the raw signal that moves the 7 other legs. The non-terminal <nn> is rewritten between 6 and 20 times, and finally the neuron specializes either as a temporal unit (non-terminal <t-unit>) or as a spatial unit (non-terminal <s-unit>.) The temporal units have a threshold sigmoid and a time constant that is genetically determined. They are used to introduce a delay in a signal, and the spatial units have a linear sigmoid, and a bias that is genetically determined. They are used to translate and multiply a signal by genetically specified constants. Those two types of units are the building blocks needed to generate a fixed length sequence of signals of different intensities and duration. The duration is controlled by the temporal units, and the intensity by the spatial units.

5.4 Explanation of the solutions

Figure 7 presents the leg controller found by the Evolutionary Algorithm, together with four different settings of the activities inside the ANN, and the corresponding leg positions. Figure 6 shows its genetic code after some hand-made obvious simplifications, such as merging a tree of PROGNs.. Neuron e controls the lift, neuron f controls the swing and neuron a is the input neuron. While the ANN is completely feed-forward, yet it can generate a short sequence of different leg commands. Because we use neurons with time constants, different events can happen at different time steps. In Step 1, the input to the ANN is 0, and the leg is forward down, the input is then changed to 4096 to initiate the power stroke. Step 2, we are in the middle

```
 SEQ (PROGN(STEP) (DELTAT(1))) (CPI  (SEQ (PROGN(LINEAR)(WEIGHT(-693 )
(-1024 )(360 )(-252 )(-300 )(-984 )(-849 )(610 ))(SBIAS(3497 )))
(CPO (PROGN(STEP)(SACT(3458 )))  (PROGN(STEP)(DELTAT(39 ))) ) ) (PAR
(PROGN(LINEAR)(WEIGHT(-693 )(-1024 )(360 )(-252 )(-300 )(-984 )(-706 )
(796 ))(SBIAS(1864 ))) (PROGN(LINEAR)(WEIGHT(-914 )(40 )(736 )(-622 )
(-1024 )(-984 )(-706 )(610 ))(SBIAS(-2321 ))) )  )
```

Figure 6: The genetic code of the champion leg controller



```
<axiom>::=(LABEL(SEQ
 (SEQ
  (PAR(<command>)(<command>))
  (PROGN
     (WAIT(4))  (CTYPEI( 1)) (RESTRICTI(0)(1)) (STYPEI(0)) (CTYPEI( 1))
     (STYPEI(1)) (CTYPEO( 1))  (STYPEO(0))
     (<evolved>)   )   )
  (PROGN
     (BLOC)
     (TESTIO8)
     (SHARI (JMP12)  (SHARI (JMP12)  (SHARI (JMP12)   (SHARI (JMP12)  (PROGN (SWITCH) (SHARI (JMP12)
          (PROGN (SWITCH) (SHARI (JMP12)   (PROGN (SWITCH) (SHARI (JMP12) (JMP12)
           (1) ))  (1) ))   (1) ))  (1) )   (1) )   (1) )   (1) )  ) ))
```

Figure 8: Syntactic constraint specifying the general structure

6 The locomotion controller

6.1 The Challenge

We have previously evolved ANN for a simulated 6-legged robot, see [6]. We had a powerful parallel machine, an IPS■860 with up to 32 processors. We needed 1,000,000 of ANNs to find an ANN solution to the problem. In this study, one ANN takes a few minutes to assess the fitness, because the fitness is manually given, and it takes some time to see how interesting the controller is. One time we even spent an hour trying to figure out whether the robot was turning or going straight, because of the noise that was not clear. The challenge of this study was to help the EA so as to be able to more efficiently search the genetic space and solve the problem with only a few hundreds of evaluations instead of one million.

6.2 General setting

There are some settings which were constant over the successful run 2, and run 4, and we report them here. The way we give the fitness was highly subjective, and changed during the run depending on how

7 Log of the experimental runs

We did only five run, and we are proud of it. That is a proof that the method works well, it doesn't need weeks of parameter tuning. That's useful because one run is about two days of work. So we report the five runs, even if only one of them was really success-full, and the other were merely used to debug our syntactic constraints.

7.1 Analysis of run 0 and run 1

The first two runs were done with only seven legs, because a plastic gear on the eighth leg had burned, the robot had stubbornly tried to run into an obstacle for 30 seconds. As a result, ANNs were evolved that could made the robot walk with only seven legs. Run 0 brought an approximative solution to the problem. But after hours of breeding we input accidentally a fitness of 23, which had the effect to stop the EA, the success predicate being that the fitness is greater than 1. Run 1 also gave a not so bad quadripod, as far as we can remember, but we realized there was a bug in the way we used the link types which were not used at all. Instead of selecting a link and then setting the type we were first setting and then selecting, which amounts to a null operation. We had simply exchanged the two alleles. When we found out the bug, we were really surprised that we got a solution without types, but we think that's an interesting feature of EAs, even if your program is bugged, the EA will take care of it!.

7.2 Syntactic Constraints used in the second run

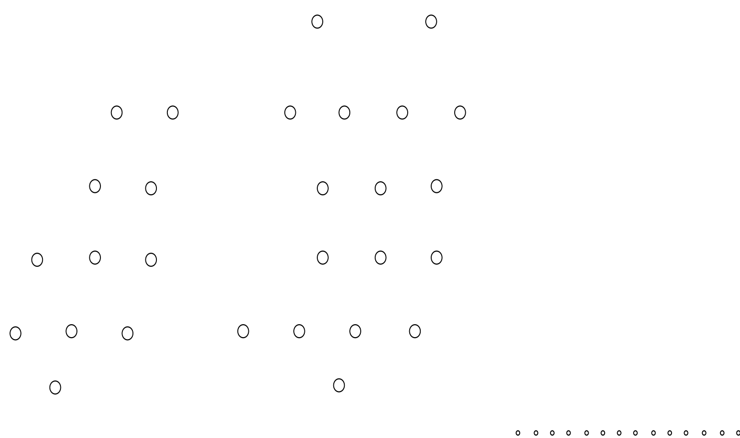
The syntactic constraints used for the second run are reported in 10. The core of the controller is developed by the cell executing the non-terminal

```
<clone> [3..3];
<nn>
```

get a fitness like 0.001. One of them produced an oscillation on four legs. During the next two generations, we concentrated on evolving oscillations on as many legs as possible, giving fitnesses between 0.01 and 0.02, depending on how many legs were oscillating, and how good was the period and the duration of respectively the return stroke and the power stroke. At generation 3, we started to have individuals with a little bit of coordination between the legs. We watched an embryo of wavewalk which very soon vanished because the leg went out of synchrony. The coupling was to weak. The ANN is represented in figure 11 on the second picture, you can see that it is very sparsely connected. The next ANN at generation 4 generated an embryo of quadripod. Four legs were moved forward, then four other legs, then the 8 legs were moved backward. At generation 6 we got an ANN made of four completely separated sub-ANNs each one was controlling two legs, due to a different initial setting of the activities, the difference of phase was correct at the beginning, and we obtained perfect quadripod, but after 2 minutes, the movement decreased in amplitude, four legs come to a complete stop, and then the four other legs. Generation 7 gave an almost good quadripod, the phase between the two pairs of four legs was not yet correct. The ANN is probably a mutation of the guy at generation 3, because the architecture is very similar. Generation 8 gave a slow but safe quadripod walk. The coupling between the four sub-ANNs is strong enough to keep the delay in the phase of the oscillators. Only the frequency needs to be improved, because that's really too slow. Generation (ftea9]TJ247.6cthat's).7(to)-99934

```
<evolved> ::= ( PROGN(<clone>)(<clone>)(<clone>)(<nn>))
<clone>::=  (FULL(<opi>)(<opo>))
<opo> ::= (PROGN  (<op>) (<settype>) (WAIT (integer[0..4]))  )
<opi> ::= (PROGN(<op>)(WAIT (integer[14..18])))
<op> ::=  (PROGN
           ( PROGN: set[0..4] of
                (PROGN: list[1..2] of (PROGN(CTYPEI (2) )(<cuti>)) )
                (PROGN: list[1..2] of (PROGN(CTYPEI (3) )(<cuti>)) )
                (PROGN: list[1..2] of (PROGN(CTYPEI (4) )(<cuti>)) )
                (PROGN: list[1..2] of (PROGN(CTYPEI (5) )(<cuti>)) )   )
           ( PROGN: set[0..4] of
                (PROGN: list[1..2] of (PROGN(CTYPEO (2) )(<cuto>)) )
                (PROGN: list[1..2] of (PROGN(CTYPEO (3) )(<cuto>)) )
                (PROGN: list[1..2] of list[1..2<clon3.1d[((PROGN15999.4(of)]TJ65.999.3492])
```

```
 LABEL(SEQ(SEQ(PAR(PROGN(WAIT(200 ))(DELTAT(1 ))(SBIAS(0 ))(PROGN(CTYPEO(0 ))
(SWEIGHTO(318 )(319 )(128 )(148 )(485 )(228 )(154 )(49 )(333 )(7 )(357 )(327 )
(314 )(444 )(171 )(448 )))(PROGN(CTYPEO(1 ))(SWEIGHTO(268 )(185 )(424 )(113 )
(54 )(357 )(316 )(110 )(259 )(102 )(90 )(43 )(299 )(367 )(477 )(78 )))(LINEAR)
)(PROGN(WAIT(200 ))(DELTAT(1 ))(SBIAS(0 ))(PROGN(CTYPEO(0 ))(SWEIGHTO(453 )
(461 )(82 )(56 )(283 )(111 )(385 )(43 )(409 )(312 )(391 )(210 )(491 )(347 )
(171 )(238 )))(PROGN(CTYPEO(1 ))(SWEIGHTO(67 )(403 )(483 )(458 )(104 )(219 )
(505 )(323 )(234 )(94 )(291 )(330 )(154 )(198 )(355 )(324 )))(LINEAR)))
(PROGN(WAIT(4 ))(CTYPEI(-1 ))(RESTRICTI(0 )(1 ))(STYPEI(0 ))(CTYPEI(-1 ))
```



7.6 Analysis of the fourth run

A selection of the champions of this run are reported in figure 13. We deleted systematicaly ANNs that do not have the right number of inputs, or produced no movement at all, as in the second run. We represent at generation 0 a guy which produced oscillation on one leg. At generation one, we had an individual that has a correct oscillation on all the legs. That guy has a quite simple 16 neuron controller made of 8 oscillator neurons with a recurrent connection, and 8 neurons that implement a weak coupling. The oscillators loose synchronisation. In generation 1, we had an ANN that produced oscillation, and coupling between the two sides but not within one side. That means that the two front legs for example, or the two right legs are synchronous, but not the left front leg with the left rear leg. Generation 2 produced a fun individual which moved the two front legs two times quicker than the other 6 legs. You can see figure 13 fourth picture, that the ANN that controls the two front legs is much more sparsely connected than the one which controls the other 6 legs. Generation 2 produced another champion: a slow but correct quadripod gait. The genotype is probably a mutation of the second guy at generation 1, because the architectures are very similar. There is one sub-ANN for each leg, as is specified by the syntactic constraints, the sub-ANNs within one side are coupled, but the two sides are independent, as is clearly shown in the picture. Generation 3 produced a funny gait, with four pair of two coupled oscillators, inside each pair, one oscillator has the double frequency of the other, due to the coupling. The figure clearly shows the structure of the ANN. Generation 5 produced a quadripod gait, not too slow, but there there still lacks some coupling between the ANNs controlling the legs of one side. There are diagonal connections between the four groups, which implement coupling between the two sides, but there are no connections from top to bottom. At generation 6 we had a nice ANN with all the connection needed for coupling, but the frequency on the right side was slightly greater than on the left side, as a result the robot was constantly turning right. We would have thought that a higher frequency result in turning in the opposite direction, but the reverse is true at least for those particular frequencies which were almost similar. We got a number of individuals which were always turning right. Generation 7, we finally got success, a perfect quadripod, smooth and fast. We had the robot run for 20 minutes to check the phase lock.

8 Comparison of the Interactively Evolved Solution with the hand-coded solution

8.1 performance

We now want to compare this solution with the hand-coded solution. The hand-coded program is a wavewalk that has been done with a loop and a ◘ program, by Koichi I7(th10Td(with)Tj23.e)]TJ/R175(is)Tj1ea999.654(y6)-13

May be the most unexpected thing out of this work, is that the breeding is worth its pain. We are not sure that an automatic fitness evaluation that would have just measured the distance walked by the robot would had been successful, even in one week of simulation. There are some precise facts that support this view. First, right at the initial generation, we often got an individual which was just randomly moving its legs, but still managed to get forward quite a bit, using this trick. This is because the random signal has to go through the leg controller and the leg is up when it goes forward and down when it go backward. Using automatic fitness, the guy would have just dominate all the population right at generation 1, and all potentially interesting building blocks would have been lost. With interactive fitness we just systematically eradicate this noisy and useless individual. When we say noisy, it really do much more noise than all the others, because it moves all the time by the maximum allowable distance. So after a while, we push the kill button as soon as we hear the noise, kill, kill, kill, that gives an (un-healthy?) feeling of power. Second, there are some very nice features which do not result at all in making the robot go forward. We are pretty happy if we see at generation 1, a guy which move periodically a leg in the air, because that means that somewhere there is an oscillatory sub-structure that we would like to see spreading. Typically, we spend the first generations tracking oscillatory behavior, and tuning the frequency, we then rewards individuals who get the signal on all the height legs, and last, we evolve coupling between the legs, with the right phase delay. That's a pretty simple strategy to implement when breeding on line, but that would be dificult to program.

In short, we developed in this work a new paradigm for using Evolutionary ■omputation in an interactive way. Syntactic ■onstraints provide a prior probability (machine learning terminology) on the distribution of ANNs. Modular decomposition allow to replace one big problem by two simpler problems, Interactive fitness evaluation can steer the EA towards the solution.

Our future direction will be to evolve a locomotion controller with three command neurons: one for forward/backward, one for right/left and one for the walking speed. In order to do that, we need to enhance our method so has to be able to optimize different fitnesses with different populations, and then build one behavior out of two behavior separately evolved. In the case of turning, or speed controling things can be stated more precisely. We thing that turning as well as varying speed is only a matter of being able to govern the frequency of the oscillators. Typically we would like to evolve separately an oscillator whose frequency can be tuned using an input neuron, and then recombining it with the locomotion controller evolved in this paper. The way how to successfully operate recombination is still an open subject of research.

Acknowledgment

This paper reports on work partially done by Dominique Quatravaux in fullfillment of the master's thesis requirements under supervision of Michel ■osnard and Frédéric Gruau, on Frédéric Gruau's project at ■WI in Amsterdam.

Since Dominique declined to be coauthor, it is proper to carefully delineate his contributions. He developed the gnu ■ compiler for the robot processor, and the protocol to handle the serial line, that is, to control the robot from the host computer. The programs can be downloaded from URL page `http://www.cwi.nl/ gruau/gruau/gruau.html`, together with user information, bugs and tips.

Dominique performed about half of the experiments (the leg controler, run0 and run1 of the locomotion controller) and developed the approach to first evolve a leg controler with a well defined interface. He can be reached at (quatrava@clipper.ens.fr). To represent Dominique's involvement we added the name of his workstations as co-author.

Phil Husbands acted as a remote adviser, and we thank him for letting us use their O■T1 robot. Paul Vitanyi and the Algorithmics and ■omplexity (AA1) Group at ■WI hosted the whole research, providing a modern computer environement and tools. We are grateful to Paul for his constructice and positive comments. F. Gruau was suppported by a postdoctoral TMR grant from the European community within the Evolutionary Adaptive SYstems group at the ■OGS department in Sussex University. We also acknowledge help from the pole Rhone Alpes de Science ■ognitive who supplied money to buy a prometheus Robot in the frame of a joint project with prof. Demongeot. We prefered to use the O■T1 Robot because it turned out we had the choice and the O■T1 Robot was 10 time more expensive, more reliable and faster. We did however use replacement part from the prometeus robot, namely the power supply. Applied AI system did a good job in sending us replacement part for the O■T1 robot, together with advices. We thank Ann Griffith

and Koichi Ide.

References

[1] Randall Beer. *Intelligence as adaptive behavior*. Academic Press, 1990.

[2] Randall Beer and John Gallagher. Evolving dynamical neural networks for adaptive behavior. *Adaptive Behavior*, 1:92–122, 1992.

[3]