Chapter Number

Progressive design through staged evolution

Ricardo A. Téllez & Cecilio Angulo
Technical University of Catalonia
Spain

1. Introduction

Evolutionary robotics is a good method for the generation of controllers for autonomous robots. However, up to date, evolutionary methods do not achieve the generation of behaviors for complex robots with a fixed body structure composed of lots of sensors and actuators. For such cases, no satisfactory results exist due to the large search space that the evolutionary algorithm has to face. Furthermore, the bootstrap problem does not allow convergence on the first generations, preventing the generation of simple solutions with a minimum fitness value that could guide the evolutionary path towards the final solution. Solutions like incremental evolution try to overcome the problem, but they do not scale well in complex robots with lots of devices.

The question is then, why natural evolution succeeded evolving complex animals, but evolutionary robotics does not. One answer to that question may be that, while natural evolution gradually evolved at the same time the animals body plan, their sensors and actuators, their nervous system, and even their environment, artificial evolution tries to evolve the nervous system for a robot with a fixed given body, sensors and actuators, within a fixed complex environment.

Our proposal states that when, as it happens in most cases, none of the evolutionary constraints can be relaxed (those are, the robot body, the sensors and actuators, the behavior to implement or the environment), then the use of external knowledge to guide the evolutionary process should be mandatory. Evolutionary approaches try to avoid the use of such knowledge, also called bias, because it directs the evolutionary search towards specific places of the space, not allowing the algorithm to find its own solutions. In this paper, we advocate instead for the use of bias as an inevitable situation when the robot body, task and environment are complex and fixed.

Based on this idea, we develop a modular architecture for evolutionary controllers based on neural networks, which allows the selective introduction of bias knowledge in the neural controller during the evolutionary process. The architecture allows the introduction of external knowledge on selected stages of the evolutionary process, affecting only selected parts of the controller that need to accommodate that information. The evolutionary controller is progressively designed in a series of stages, almost in a surgical way, independently of the complexity of the robot (in terms of number of sensors and actuators). This approach allows to avoid the bootstrap problem completely, and to obtain a completely distributed neural controller for the robot using only artificial evolution.

We will show how our method applies to general robots. Additional results will show its application to a complex Aibo robot.
The rest of the chapter is divided as follows: section two provides a description of the problem of evolutionary robotics that we try to solve, including a review of existing solutions. It follows a description of the architecture we developed to solve such problem, and a simple application to a simple wheeled robot. Then an application of the architecture to an Aibo robot, where more than 20 devices needed to be used to generate a behavior. The chapter ends with the a discussion of the results, the conclusions and future lines of application.

2. Evolutionary robotics for complex robots

Evolutionary robotics is a framework for the automatic creation of autonomous robots. It reproduces on robots selective reproduction of the fittest, that means, it uses evolutionary algorithms to develop the control program of an autonomous robot (Nolfi & Floreano, 2000). The idea behind this framework is that the controller for a robot may be so complex that we don't know how to construct it. Instead we reproduce the mechanisms that evolution took for the generation of animal minds. The control system of the robot is then represented as an artificial chromosome subject to the laws of genetics and natural selection, which evolves on a hostile environment. Evolutionary robotics pays more attention to the processes that build a controller than how the final controller is. This liberates the designer of the necessity to specify how the controller must exactly work in every situation, and, more important than that, allows the evolutionary process to find novel solutions for the controller that the designer could not have thought of.

2.1 Complex robots

One of the most interesting problems of evolutionary robotics is the creation of behaviors for complex robots. We measure the complexity of a robot by its number of devices, that is, the number of sensors and the number of actuators that it has, which require a coordination in order to achieve a task, or generate a behavior. As this number increases, more difficult will be to generate an evolutionary controller for it.
At present, evolutionary robotics is mainly relegated to the realm of simple wheeled robots, where just a very small number of actuators (about two or three), and a small number of sensors (between two and six) are used. Even if quite complex behaviors have been evolved for such robots (Nolfi, 2004), there is no general solution for its implementation in complex robots with tens or hundreds of devices to be coordinated.

2.2 Problems faced when evolving complex robots

Two main problems arise in the evolution of controllers for complex robots: first, the search space that the evolutionary algorithm has to face to find a suitable solution is very large. Due to the large number of elements that need to be coordinated even for the simplest task, the number of weights of the neural controller is large. Then, the space of possible solutions has a high dimensionality, what makes computationally difficult to find a solution. The second problem is the bootstrap problem, which prevents the generation of simple solutions at the beginning of the evolutionary process with a minimum fitness value that could guide the evolutionary path towards the final solution (Nolfi & Floreano, 1998).

Up to date, there is no general method which avoids such problems, even if partial solutions have been proposed. For instance, incremental evolution (or incremental learning) is an evolutionary method that evolves a neural network by doing successive steps of teaching with an increase of the complexity of the task been teach on every step (Elman,1991; Gomez & Miikkulainen, 1996). This approach requires the use of the designer's knowledge of the task to correctly define the incremental evolutionary steps. It has worked well in complex tasks in simple robots, but a successful use in complex robots has not been reported. Modifications to this approach have also been proposed (Doncieux & Meyer, 2004), but their applications have still been applied to simple robots.

Co-evolution, is proposed as another solution. It consists of the evolution of competing controllers using a coupled fitness function (Nolfi & Floreano, 1998). However, co-evolution includes its own series of problems, and as (Doncieux & Meyer, 2004) say, they have to materialize yet.

In (Nelson et al. 2002), the authors propose a fitness function with two modes. The first mode rewards the controller when it has not been able of completing the task, using a subjective measure (completely determined by the designer) of the uncompleted task. A second mode only provides a reward when the task is achieved. A similar approach was also used by (Nolfi, 1997) on the generation of a garbage collector controller or on the generation of a hand able to grasp things (Bianco & Nolfi, 2004).

Another line of research involves the evolution at the same time of the robot body plan and the neural controller. Impressive results have been obtained in (Hornby and Pollack, 2002; Pollack et al., 2003). However, those approaches lack the ability to direct the evolutionary path of the body plan towards a body structure that has been designed before hand. But it is its lack of directness towards a given and fixed body what makes this approach useless for the case of a robot body which is fixed and complex.

We could however, introduce additional terms in the fitness function and restrict the possible morphologies to evolve a body plan similar to the one we want to control (the given robot body). This is the approach followed by (Muthuraman, 2005). Muthuraman performed a human directed evolution of a robot body and its neural controller in progressive steps. Initially, a simple controller for a simple version of the robot is evolved. When the controller cannot improve its fitness anymore, new neural modules are added to it, and the evolutionary process continued, until fitness improvement is not obtained. Then the body of the robot is complexified in some sense, and the previous evolutionary process repeated. The algorithm will continue complexifying body and controller until it reaches the robot body and task required. It seems difficult, though, to see how this method could be applied to any robot on any task, aside the one described by the author.

Additional solutions for the evolution of complex controllers include all sort of tricks for the particular controller to evolve. For example, (Mojon, 2004) generated the initial weights of the nets for a walking humanoid with small values, allowing the robot to obtain some controllers that didn't fall at the first generations. (Reeve, 1999) used symmetry on the evolution of a walking robot to reduce the search space and avoid initial useless controllers. (Ijspeert, 1998) used staged evolution for the generation of a controller for a salamander. In a similar way, (Lara et al., 2001) evolved two separated and independent neural controllers for a Khepera robot and, once they had those two modules, they evolved a neural interface between them, allowing the robot to show both behaviors on a unique controller.

3. Solution proposed

What all the previous approaches share in common is their introduction of human knowledge into the process in some or another way, that is, there is an introduction of bias by the designer. The introduction of bias helps to direct the evolutionary process towards a solution, reducing by hence the search space. However, this directness is usually not welcomed in evolutionary processes, since it performs a restriction on the solutions that the process will be able to find. The fact is that the bias leaves a short space for the algorithm to find a different solution from that that the researcher knows.
In our solution, we see the massive use of external knowledge as something inevitable in evolutionary robotics when the robot body, task and environment are fixed and complex. Starting from this idea, we propose a massive modular architecture for the control of robots, which allows the introduction of information in a selective manner, including certain information only on specific parts of the controller, without affecting others. The generation of the final controller is performed on a series of stages where the modules are progressively evolved using their specific information as fitness function, until the global controller is obtained. In fact, the global neural controller is progressively designed, through a series of evolutionary stages.

3.1 Description of the architecture

To generate a controller for a complex robot we think unavoidable the use of the concept of modularization. In neural controllers, modularization can be performed at two levels: at the structural level and at the learning level (Auda & Kamel, 1999). A modularization of the neural structure means that the controller will be composed of several neural networks. A modularization of the learning procedure implies that the learning will be performed in different stages. The problem is that it is difficult to train globally a chained functionality on a modular neural structure, because, even that we want a global intelligent behavior, the learning capabilities only exist at the organization level of the neural nets. Then, if modularization at the neural structure is taken, a modularization of learning may also be required.
Hence, for the progressive design of a neural controller we take modularization at both levels. We provide a standard building block and evolutionary method for any robot, independently of its number of sensors or actuators. The key point of this modularization is in which way the controller modularization is performed. While most of the existent modular architectures perform modularization at a functional level (that is, the global task to be implemented by the robot is decomposed into simpler subtasks (Davis, 1996; Lara et al. 2001; Dorigo & Colombetti, 2000), in the progressive design method, modularity is created at the level of the robot device by creating a small and independent neural module around each of the sensors and actuators of the robot.
The evolution of the neural weights for each module is performed in several stages. On a first stage, only a small set of the sensors and actuators available are used for the evolution of a simple task, called an evaluation task, which must be related with the final global one. The neural modules associated to the set of sensors and actuators are evolved at the same time on the evaluation task. As the small group of controllers gains proficiency in their evaluation task, new sensors and actuators together with their neural modules are added to the group, and the evolution process started again with a new evaluation task. The re-started evolutionary process only affects the evolution of the weights of the newly added

modules and their connections with the evolved in the previous stages. As this process goes along, more and more sensors and actuators of the robot are progressively added, and the evaluation task they perform modified, until they reach the total number of sensors and actuators and the final global task required.
The evolution of modules by stages allows us to minimize the effects of the two main problems described in section 1: first, by evolving a limited number of modules at each evolutionary stage on a simplified task, we keep reduced the searching space the evolutionary algorithm has to face. Second, by evolving new modules keeping the functionality obtained in previous stages, the likelihood of obtaining a bootstrap is reduced, since the evolution of the newly added modules depart from a previous stable solution which can be scored, and by hence, direct the evolutionary path from the initial generations.

3.2 Sensor and actuator neural modules

Modularity is implemented at the level of the robot device and works as follows: each sensor and actuator will have an associated neural module, that we call in general, an Intelligent Hardware Unit (IHU). Each IHU module will belong only to that particular sensor or actuator. We will call those modules in particular as *SENEMO* (from SEnsor NEural MOdule) for the union of a sensor and a neural net, and *ACNEMO* (from ACtuator NEural MOdule) for the union of an actuator and a neural net. Each neural network will *take care* of its associated device. This means that it decides which commands will be sent to its actuator, in the case of an ACNEMO, or how will the information coming from its sensor be processed, in the case of a SENEMO.

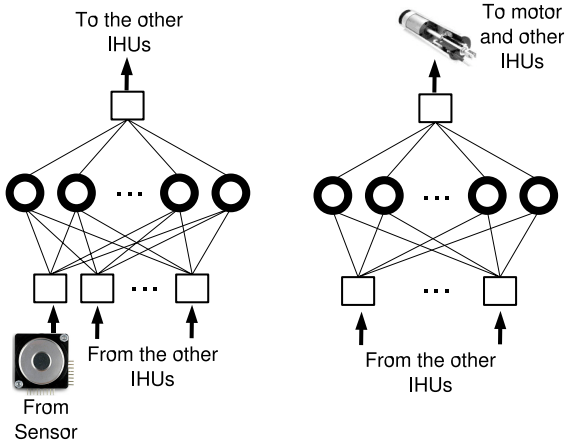

Figure 1: Left: Connection schematics of the neural modules for a sensor (SENEMO) and an actuator (ACNEMO).

The generation of a coordinated global robot behavior will require the cooperation between all the sensors and actuators. This is what is called sensorimotor coordination (Pfeifer & Scheier, 1997). In order to achieve such coordination, a communication channel is established that allows the modules the interchange of information. The communication

channel is implemented by connecting the neural modules' output to one input of the other modules. This connection mechanism makes all the modules aware of what the other modules are doing before deciding their own action. More complicated connection schemes are possible where two or more outputs were allocated in the modules, establishing one of the outputs as the action of the module, and the other as the communication for the other modules. We have kept this scheme simpler by allocating one single output for each module which acts at the same time as the output of the module and the communication for the other modules. Schematics of the SENEMO and ACNEMO module connections are included in figure 1. Figure 2 shows a connectivity example of the final controller obtained for a simple robot composed of two sensors and two actuators.

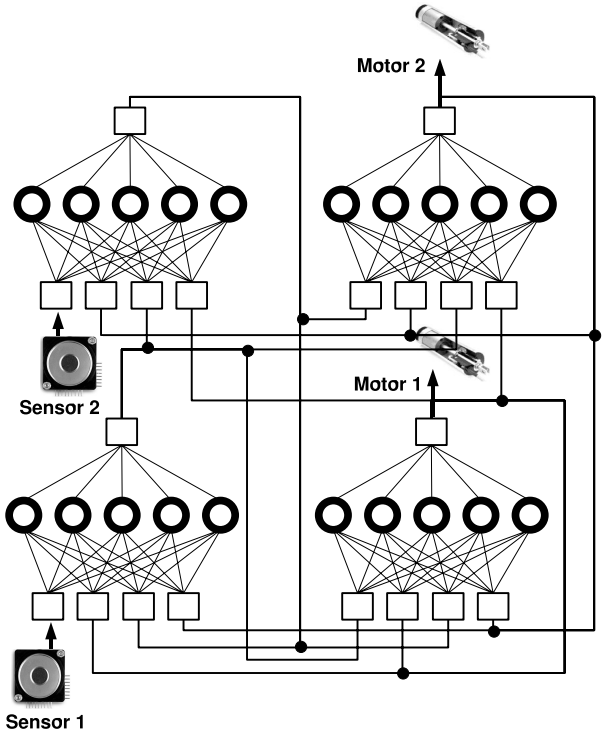

Figure 2: Application of the approach to the control of a simple robot composed of two sensors and two motors. Four modules are required, two SENEMO and two ACNEMO.

It should be stated that when put several modules together on a control task, each module has its own particular vision of the situation since each one is in charge of its own sensor or actuator. Since there is no central coordinator, each module selects an action for its associated actuator or sensor output, based on the information sent by the other modules and its current status of its associated device, for the generation of the global robot behavior.

3.3 Evolutionary process

In the context presented above, the neural modules need to learn the control of their associated devices at the same time that they cooperate between each other for the generation of the global robot task. This is accomplished through an evolutionary process using a neuro-evolutionary algorithm. Due to the fact that the evolutionary process has to evolve different neural networks for different roles on a common task, a co-evolutionary algorithm is required, that is, the simultaneous evolution of several nets, each one with its own population, but with a common fitness for all of them. By using such kind of algorithm it is possible to teach to the networks how they must cooperate to achieve a common goal (i.e. the global robot behavior to implement) when every network has its own an different vision of the whole system, since their inputs and/or outputs are different.

The algorithm selected to evolve the nets is the ESP (Enforced Sub-Populations)(Gómez & Miikkulainen, 1996), which has been proved to produce good results on co-evolutionary processes (Yong & Miikkulainen, 2001). A chromosome is generated for each module network, coding in a direct way the weights of the network connections, and the whole group of neural nets is evolved at the same time with direct interaction with the environment. The fitness function which guides the evolutionary process is created by the designer, depending on the problem that the robot has to solve.

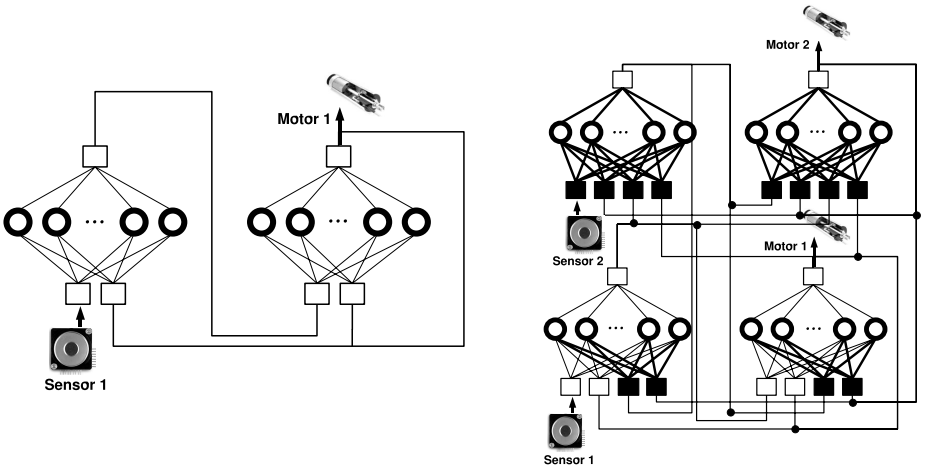

Figure 3: Progressive design of a neural controller for a simple robot with two sensors and two actuators. On stage one (left), only one SENEMO for sensor 1, and one ACNEMO for motor 1 are evolved obtaining the controller of the top of the figure. On stage two (right), another SENEMO (sensor 2) and ACNEMO (motor 2) are added to the controller. At that stage, only those newly added modules and their connections to the previously evolved modules are evolved (in thick black).

The progressive design method goes as follows: first, a goal-task for the robot is decided. This goal-task will be the behavior required for the robot. Then, the robot is divided into modules, one module for each sensor (a SENEMO) and actuator (an ACNEMO). Then the staged evolution begins. At the first stage, a subset of the modules are evolved to perform a

reduced task related in some way with the goal-task. This reduced task is called an evaluation-task. The subset of modules to be evolved and the evaluation-task are selected by the designer, depending on the evolutionary strategy he is going to perform. The evaluation-task is usually a task that is relevant for the goal-task and that concerns the modules selected for this stage.
Once this evaluation-task has been achieved by the subset of modules with a reasonable fitness value, a new evolutionary stage starts where a new set of modules is added to the controller. For this new stage, the evaluation-task may be changed, by incrementing its complexity, by changing it completely, or by not changing it at all. During this second stage, previously evolved modules are not evolved, and only the new ones added and their connections to the already existing ones are evolved on the new evaluation-task. Once the evolutionary process obtains a reasonable fitness value, a new stage begins. This procedure is repeated until all the modules are included in the controller and the final goal-task is achieved.
Figure 3 shows the two stages progressive design of a controller for a robot composed of two sensors and two actuators.

3.4 Application example

In this section we will show an application example of the methodology to a simple robot, in order to clearly see how it works. The example will be the Khepera garbage collector. An application to a complex robot in a complex task will follow in the next section.
Experiments consisted of the implementation of the progressive design method for the control of a Khepera robot while performing a cleaning task. The selected test bed task is called the garbage collector, and follows the following description:

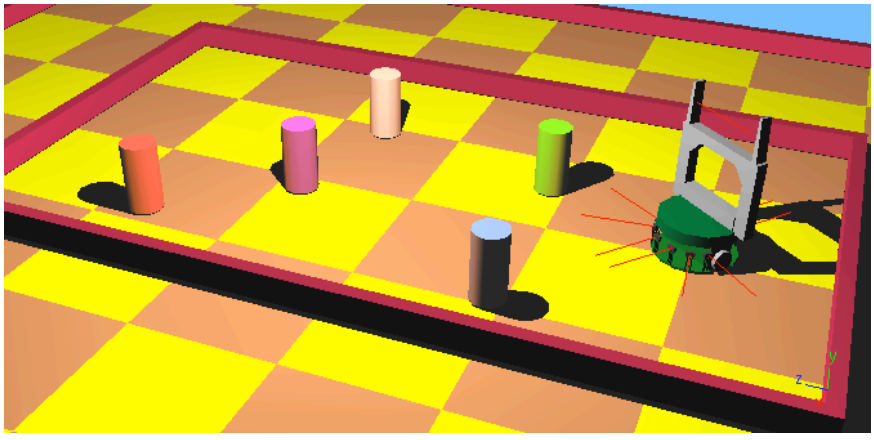

Figure 4: The garbage collector experiment on Webots simulation environment

a Khepera robot is placed inside an arena of dimensions 60x35 cm surrounded by walls (figure 4), where it must look for any of the sticks randomly distributed in the space, grasp it, and take it out of the arena. The robot was also placed randomly in the arena at the beginning of each epoch.

The robot has eight infrared (IR) sensors, six at the front of the robot, two at the back. It also has one presence detector sensor in the gripper fingers. For the task of the garbage collector, only the six front sensors will be used, as well as the gripper sensor. IR sensors have a limited range of detection (from 0 to 75 mm). The sensors have 1024 different values from 0 to 1023, which indicate the presence of an object at the corresponding distance. Objects placed at a distance above 75 mm are not detected at all.
As actuators, the robot has two motors (left and right), but it is also possible to control the position of the gripper arm and the status of the gripper fingers (open or closed). The gripper turret is controlled by two procedures that activate a complete behavior. When procedures are activated, they perform a complete series of movements of the gripper which cannot be interrupted until finished. In the case of the first procedure, the movements are: move the arm down, close the gripper fingers. If there is nothing between the fingers, then open the fingers. Then move the arm up again. This procedure should be the one activated, in order to pick up a stick, and by hence we will call it the pick-up procedure. The second procedure moves the arm down, opens the gripper fingers, and moves the arm up again. It should be activated when a stick has to be released, so we will call it the release procedure. Of course, the controller does nothing neither about the behavior of those procedures nor when they have to be activated. The evolutionary process will have to create the appropriate controller that activates those procedures when it is necessary. Procedures are activated on a on-off basis, i.e., they will be taken as if they were actuators which can be controlled with two possible values: an on value when require activation, or an off value when no activation is required.

Experiment setup
Basically, experiments consist of 15 epochs of 200 time steps each, where an evolved controller was tested over the task. The duration of each time step was of 100 ms. Each epoch ended after the 200 steps or after a stick had been correctly released out of the arena.
The final goal is the generation of a modular controller for the Khepera robot solving the garbage collector. This is a controller composed of eleven modules: six infra-red sensors, one gripper sensor, two motors, and two gripper procedures. Each neural module was implemented using a feed-forward neural net with no hidden units and only one output. At the beginning, the number of inputs of those nets will depend on the number of elements selected for the first evolutionary stage. In any case, at the end of the progressive design, the neural net of each module will have eleven inputs. Figure 6-left shows the model of neural network used for each module. Figure 6-right shows the final controller that we want to obtain by progressive design.
The architecture is evolved using the progressive design process described above. A three stages procedure is designed. The evolutionary strategy decided by us consisted on using a reduced set of sensors to perform the garbage collector task. This means that, on a first stage, not all the sensors will be used in order to reduce the number of weights to be evolved. Then, in progressive stages, new sensors will be added that may allow increase the proficiency of the robot for the task, and by hence the fitness obtained. We decided to divide the progressive design of the controller into three evolutionary stages. On the first one, only seven modules are evolved (three sensors and four actuators are used). On the second stage, two additional sensors are included. On the final stage, the remaining two sensors are added.

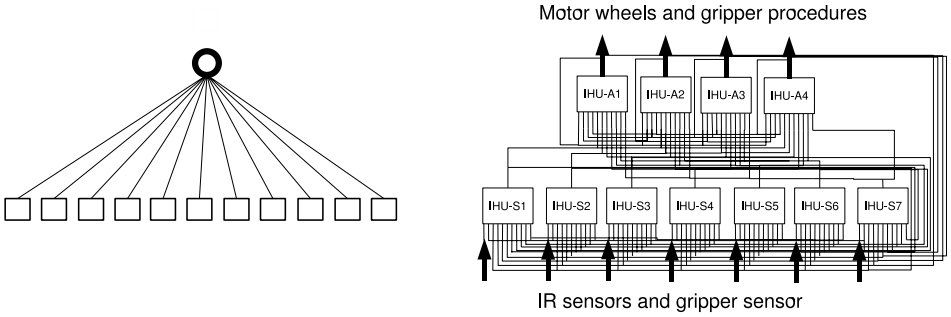

Figure 6: Left: The neural network used for each module at the end of the evolutionary process. Right: Modular representation of the architecture implemented for the Khepera robot.

First evolutionary stage

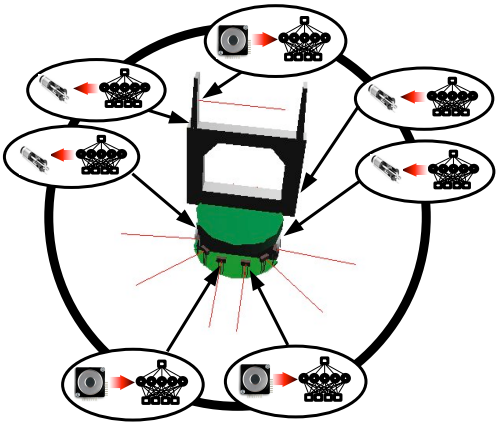

Figure 7: First stage evolutionary controller. At this stage, only seven modules are evolved corresponding to three sensors and four actuators.

On the first stage, we start using as less sensors and actuators as possible, in order to have an initial search space small. However, we want the robot be capable of performing more or less the garbage collector task. So we decide that using the two front IR sensors would be the minimum necessary for the robot that will allow it to discriminate between sticks and walls. For grasping and releasing sticks the gripper sensor is required, as well as the two wheel motors and the take and release procedures. This makes a total number of seven IHUs (figure 7).
A fitness function was created for the evolutionary process which rewarded controllers capable of releasing one stick out of the arena. An additional term was added to the function which rewarded robust controllers, those are, controllers that performed the stick releasing

behavior without performing any error. Errors included, crashing into walls, releasing sticks inside the arena or over another stick, or trying to pick a wall. Controllers that were able to only pick up one stick were also rewarded with a lower fitness.

$$fitness = \begin{matrix} 1 & if\ pick\ up\ stick \\ 100 & if\ stick\ released\ outside\ arena \\ 110 & if\ stick\ released\ without\ errors \\ 0 & otherwise \end{matrix}$$

After 1000 generations, the maximum fitness obtained was of 1531 (out of 1650). The behavior obtained, even that a little bit non-optimal, allowed the robot to solve the task. Due to the randomness of the method employed, we performed the evolutionary process ten times, obtaining a mean fitness value of 1021. Eight out of ten evolutionary processes were able to generate the garbage collector behavior. Figure 10 shows the evolution of the mean fitness value over generations.

Second evolutionary stage

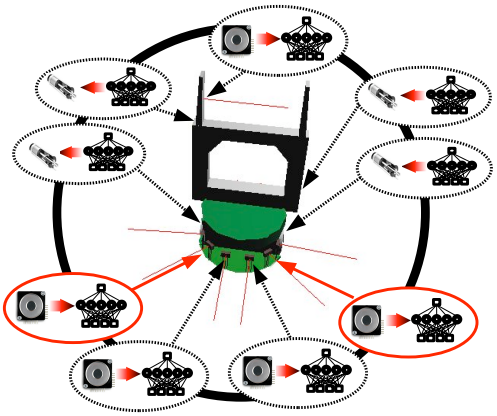

Figure 8: At the second stage, two additional sensor modules were added (in red).

On the second stage, the best group of modules evolved on the previous stage were freeze in their evolution. Two new sensor IHUs were added to the controller, the ones corresponding to the control of the two diagonal IR sensors (figure 8, in red). During this stage, only the weights of those two new modules and their connections to the already existing modules were evolved. New connections between modules are random initialized by the evolutionary algorithm, and evolved in separated sub-populations.
For this task, the fitness function used at this stage is the same as in the previous one. If the progressive design method were used in the evolution of other controllers, the fitness function may have changed at this point if necessary, in order to implement a different task starting with the knowledge of the task learned in the previous stage. For the garbage

collector problem, it is not the case, and the same fitness function can be used, even if the structure of the controller has changed. An example of task that requires the change of the fitness function at different stages will be discussed in section 4.
Figure shows the evolution of the mean fitness value over generations for this stage. If we look at the evolution of fitness, we observe that, at the beginning of this new evolutionary stage, the fitness obtained at the first generation is lower than the maximum obtained at the previous stage. This is due to the interference that the newly added connections are producing in the solution found at the previous stage. However, the fitness of this stage first generation is not low, since the neural weights obtained in the previous stage represent a point in the fitness landscape were a solution is near, that is, the task learned in the previous stage represents a good starting point for the learning of this stage task. After 1000 generations the maximum fitness reached is 1650. We performed the evolutionary process ten times, obtaining a mean fitness value after 1000 generations of 1570. All ten evolutionary processes were able to generate the garbage collector behavior.

Third evolutionary stage

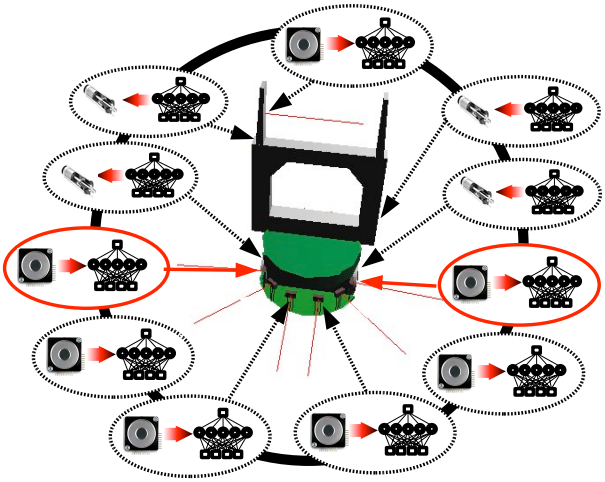

Figure 9: At the final stage, two additional sensors were included (in red).

At the third stage, the IHU for the two lateral sensors are introduced (figure 9, in red). The same procedure as in the previous stage is used. The already existing modules are freeze, and only the two new modules and their connections to the already existing ones are evolved. The same fitness function was used for this stage.
For this stage, we observe the same behavior as in the second stage, that is, the fitness obtained in the first generation of this stage is a slightly reduced value from the maximum obtained at the end of the previous stage. However, at this stage, the reduction is smaller than the reduction experienced at stage 2. Our hypothesis is that the solution found at the previous stage is very related to the solution with two more sensors. In fact, the task is the

same but using two more sensors, so the solution has only to accommodate the action of the newly added sensors.
After 1000 generations the maximum fitness value is 1650. We performed the evolutionary process ten times, obtaining a mean fitness value after 1000 generations of 1647. All ten evolutionary processes were able to generate the garbage collector behavior. Figure 10 shows the evolution of the mean fitness value over generations.

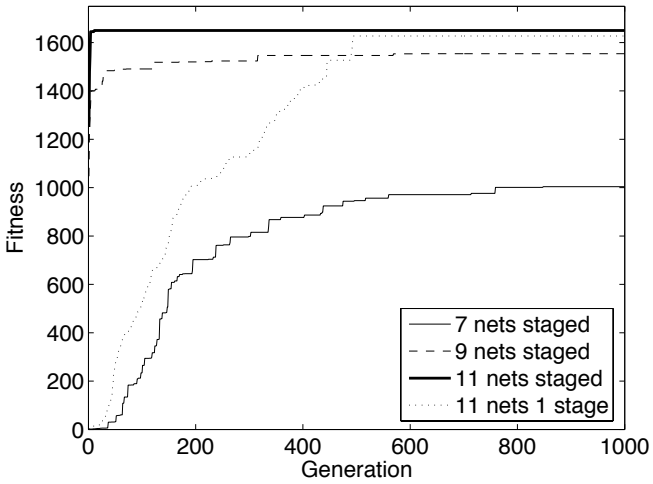

Figure 10: Mean fitness evolution through generations for all the stages. Dotted line shows the fitness evolution obtained when the eleven modules are evolved in one single stage. Values have been averaged over ten evolutionary runs in all cases.

Comparison with one single stage evolution
Due to the simplicity of the robot used, the simplicity of the task to be solved, and the power of the evolutionary algorithm, it is possible to evolve the whole controller of figure 6 for the garbage collector in one single stage. This implies to evolve at the same time, in one single stage, the eleven modules.
If we compare the case of one single stage evolution with progressive design, we observe that the mean fitness value of the former is 1645, a fitness slightly below the mean fitness obtained by progressive design (1647). Figure 10 shows the evolution of the mean fitness value over generations, compared with the evolution of fitness of the other stages. Furthermore, in the case of single stage evolution, nine out of ten evolutionary processes were able to evolve a garbage collector behavior, meanwhile in the case of the progressive design, all the controllers of the last stage were able to generate the garbage collector behavior (10 out of 10). Even if both approaches obtained similar results, a small improvement seems to be on the side of the progressive design. This results seems reasonable, since in the progressive case, the controller is carefully built step by step, starting from previously known domains, which allows for a better evolutionary process.

However, the improvement in fitness obtained it may not be worth it because of the needed complexification of the evolutionary process. This complex procedure may only be required in more complex situations where the single stage evolution is not possible. This is the case of the next section.

4. Complex robot application

The results presented in the previous section show how the progressive design method works for a simple case. However, the main aim of this methodology is its use in complex robots. This section shows the results we obtained when we applied the methodology to the Aibo robot for the generation of a walking pattern. The generation of such behavior is a complex issue, since Aibo has 12 degrees of freedom (DOF) related to walking (no head or queue DOF have been taken into account). Each leg has 3 DOF that must be coordinated with the other legs in order to obtain a walking gait. It is easy to see that an error of coordination in one single joint may cause the robot not walking at all.
For the generation of the walking gait we will only take into account the 12 motors of the leg joints as well as those joints sensors, what makes a total of 24 devices which need to be controlled. The whole process is carried in simulator and once the global controller is obtained, it is transferred to the real robot.
The generation of a walking gait for a quadruped has been solved in the literature using different methods. In order to feed our evolutionary controller with the required information, we studied the different solutions already available. One of the most interesting was the use of non-linear dynamic central pattern generators (CPGs) (Collins & Richmond, 1994; Lewis, 2002). The idea is to use the information about how the robot should walk to feed the evolutionary process.

4.1 The Central Pattern Generator concept for walking robots

It has been shown that animals perform rhythmic movements such as walking or running by means of CPGs (Grillner, 1985). CPGs are groups of neurons that generate an oscillatory pattern from a tonic input signal. Depending on the value of the tonic signal, CPGs can change its frequency of oscillations as well as its amplitude.
CPGs can be reproduced artificially using artificial neural networks. Those CPGs are the ones that will be created in the robot by using the distributed architecture. In real animals, each joint has a CPG for its control and these are interconnected only with the nearest CPGs. In the robot case, Aibo's joints are composed of a sensor (which obtains the position of the joint at each instant) and an actuator (which moves the joint). In our architecture a CPG is implemented for each joint by coupling a neural net for the joint sensor and a neural net for the joint actuator (see figure 13). The final robot controller can be seen then as a group of CPGs coupled ones with the others.
Because such a controller cannot be evolved in one single round by any current evolutionary algorithm, a progressive design of it is proposed. The different stages for the generation of the walking gait are: generation of the CPG oscillator, where a segmental oscillator is evolved for each type of joint; generation of a layer of joints of the same type that oscillate in counter phase by using the previously generated CPGs; and coupling of the three layers to obtain the final walking behavior.

4.2 Neural model used

We begin by analyzing the type of neural network that we will use in the IHU models for the generation of the gait controller. Due the dynamic nature of the task to evolve, a neural network capable of capturing dynamics is required. We find in (Reeve & Hallam, 2005) an exhaustive analysis of the characteristics, advantages and drawbacks of different types of neural networks for walking behaviors. We select the model called Continuous Time Recurrent Neural Nets (CTRNN).

CTRNNs are composed of a set of neurons modeled as leaky integrators that compute the average firing frequency of the neuron.

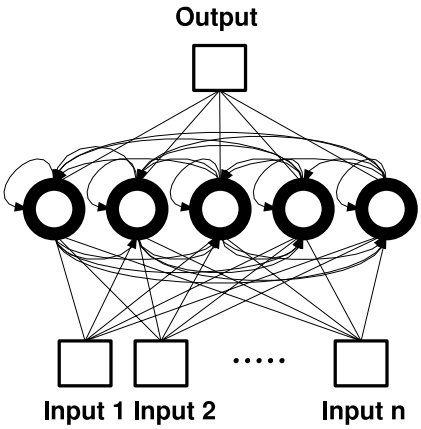

Figure 11: Schematics of the CTRNN used in the walking controller. The hidden units are modeled as leaky integrators

The output of each hidden unit of a CTRNN network is computed using the following equations (from (Reeve & Hallam, 2005)):

$$\tau_i \frac{dm_i}{dt} = -m_i + \sum w_{ij} x_j \tag{1}$$

$$x_i = (1 + e^{(m_i + \theta_i)})^{-1}$$

where m_i represents the mean membrane potential of neuron i, that is, the output of the network. x_i is the short-term average firing frequency of neuron i, θ_i is the neuron bias, τ_i is a time constant associated with the passive properties of the neuron's membrane, and ω_{ij} is the connection weight from neuron j to neuron i. Calculation of each neuron output is performed using the Euler method for solving differential equations with a given step of 96 ms.

A neural net similar to the one shown in the figure 11 is used for each IHU neural element, where: the number of inputs equals the number of devices to control, i.e. 24; the number of hidden units is five; and the number of output units is one. Inputs represent the connections of a given IHU to the others, and the output is the answer from that IHU to its associated element (sensor or actuator).
For each hidden unit of the networks it is necessary to evolve its connection weights to inputs and outputs, its bias term, and its time constant. So, the genotype for each hidden unit will contain those values in a direct encoding scheme. Initially, the values of the parameters for all the neurons are randomly created around certain values. Weights are initialized between -16 and 16. However, for the bias term and the time constant different initializations were done based on the data of (Seys & Beer, 2004). For the bias, the initial values were taken between -16 and 16. For the time constant, values were between 0.5 y 10.

4.3 Progressive design of a walking gait

The evolutionary strategy for the evolution of the controller makes intensive use of the CPG concept for the generation of the controller. Using that information, we decide that the controller of the robot will be made of a group of CPGs, one per each joint, that will drive its corresponding joint with an oscillation signal. Joint oscillators will be coordinated in the appropriate way to produce a walking gait.
Once those points were clear, its practical implementation was divided into four stages: on a first stage, a controller that performs an oscillation is generated for each joint using its associate motor an sensor. At that point each joint has its own an independent oscillator that drives it. On a second stage, two oscillators of the same joints in different legs are coupled in the form that they generate a single controller that drives both joints with an oscillation, but with a required phase shift between them. On the third stage, two groups of two coupled oscillators are again coupled, obtaining a group of four oscillatory joints driven by one single controller. Those three stages are repeated for the three types of joints that the robot has, what leads to a situation where three controllers exist, each one driving four joints on a synchronized oscillation. The fourth stage evolves the connections between those three controllers, generating a single controller for the whole robot, which finally is able to walk.

First stage: generation of the joint oscillator

The aim of this stage is to obtain a controller for a joint of the robot, capable of generating an oscillatory pattern for each type of robot joint. Joints in the robot's legs are of three different types, which we will call J1, J2 and J3 (see figure 12). J1 is in charge of the rotatory movement of the shoulder, J2 of the lateral movement of the shoulder and J3 of the knee movement. Each joint is physically implemented using different PID controllers, and, in addition, their movement limits are different. For this reason, a different type of oscillator must be implemented for each type of joint, this means, we are going to evolve three types of oscillators. The evolution of the oscillator for each joint type is performed separately. It means that each joint type will evolve its associated controller on a different and independent evolutionary process. Nevertheless, the process for the generation of each type is exactly the same, with the only practical difference being the range of movements, and the way the leg is moved. So, at the end of this stage we will obtain three different controllers, one for each joint type.
For each joint, an oscillator is implemented by coupling two CTRNN networks: one for the joint sensor and another for the joint actuator (the motor). The resulting controller is shown

in figure 13. Both nets are interconnected as the architecture specifies, but each one is in charge of a different element (the sensor net is in charge of the sensor, and the motor net is in charge of the motor). At each time step of the evaluation process, the sensor reading is taken and entered into the IHU sensor. Then the output is computed and sent to the IHU actuator. The output of the IHU actuator specifies the velocity to be applied to the motor, and, once de-normalized, it is applied directly to the motor.

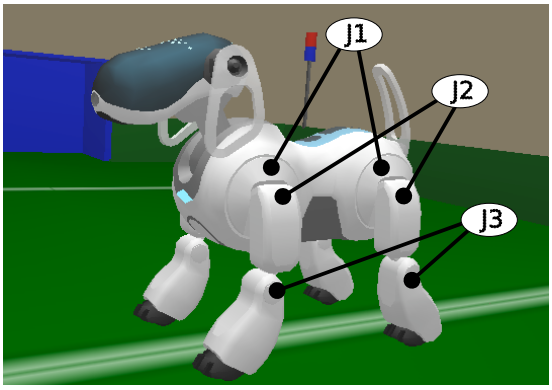

Figure 12: Detail of the three different joint types for the robot legs.

The weights of the nets are evolved using the ESP algorithm and a fitness function that rewards the production of an oscillatory pattern in the motor joint. The type of pattern that should be obtained is not specified, only that it has to be periodic and between certain oscillatory limits. The following fitness function was defined for the evolution of such oscillations:

$$mean = \frac{1}{N}\Sigma x_i \qquad (2)$$

$$fitness = \frac{1}{N}\Sigma(x_i - mean)^2$$

Basically, what this fitness function calculates is the variance of the trajectory followed by the joint. Thus, the evolutionary process will try to maximize it, and the value of the variance will be maximum when the position of the joint changes from one limit to the other. By using the previous fitness function, we obtained a single oscillation within the full range of the joint, that is, the joint oscillated one single time from one limit to the other. Aibo joints can oscillate between very large limits, but those are too large for an appropriate walking behavior, and we needed to limit them, by defining some oscillation limitations. In order to keep it simple we took those limits by observing the walking limits of the walking style that comes with the Aibo robot from Sony. Those limits are included in table 1.
A new fitness function was defined to reward regular oscillations within the limits of each joint. In order to obtain this, it is required that the system should generate a joint movement

around the mean value in the table, with maximal variance within the limits for each joint. The fitness function is then a product of two factors:

$$fitness = fit\_var * fit\_cross \tag{3}$$

where *fit_var* is the variance of the position of the joint during the 200 evaluation steps, as calculated in equation (2). *fit_cross* is the number of crossings that the joint performs through the mean position value. The addition of the second term to the fitness function ensured that the final oscillation obtained would not be limited to a single oscillation. However, this term also ended in the generation of very high frequency oscillations, which increased the second term of the fitness function. In order to avoid that, a limitation in the number of possible crossings was established to 20, what means that bigger number of oscillations were taken as 20.

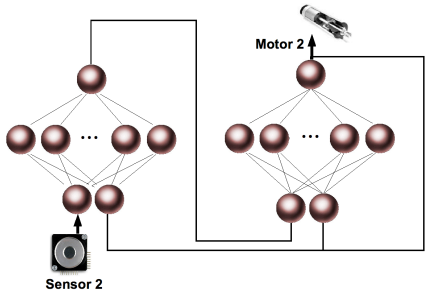

Figure 13: Schematics of the coupling between two neural nets of a joint. The figure shows the coupling between the joint sensor IHU (SENEMO) and the joint motor IHU (ACNEMO).

Joint	Max	Min	Mean
J1, fore	0.3936	-0.5837	-0.0950
J2, fore	0.3702	-0.2163	0.0769
J3, fore	1.1732	0.1435	0.6583
J1, rear	0.0059	-0.7848	-0.3894
J2, rear	0.4215	-0.2163	0.1026
J3, rear	1.6599	0.9907	1.3253

Table 1: Limits taken for Aibo joints oscillations. It includes a calculated mean value of the range of each joint. This mean value establishes the central position of the joint around which the joint will have to oscillate. Values are in radians.

On addition, each fitness factor was limited by a function that linearly varies between 0.01 and 1.0 when its corresponding variable fluctuates between a good and a bad boundary (see table 1). This means that normalization was applied to each factor in order to have a fitness value between 0.01 and 1 for bad or good results. This transformation allowed a clear vision of the evolutionary state at any time.
For the results, ten runs were carried out for each type of joint starting with different initial random populations. Each run was composed of 200 simulation steps of 96 ms. After 13 generations all runs converged to networks capable of maintaining oscillations within the range specified.

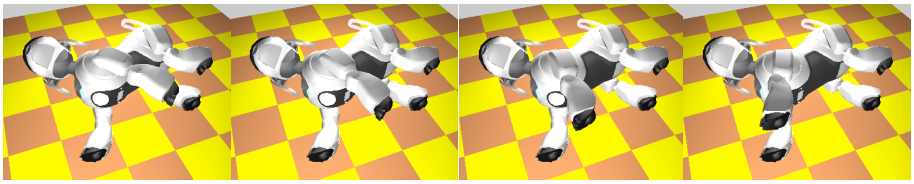

Figure 14: Sequence of the oscillatory movement obtained in simulation for the J1 joint type. The robot is lying on his side to allow the movement of the joint freely. The evolution of the other joint types was performed with the same setup.

Second stage: generation of two coupled oscillators
In the previous stage, three different and independent controllers were obtained, each one for the control of one type of joint of the robot. The aim of the second stage is to obtain a controller for two joints of the same type, which makes them oscillate with a given phase relationship.
For this case, if we apply the architecture, this means to have a controller composed of four neural networks (for each type of joint): two for controlling the two motors and two for controlling the two sensors. Since we evolved in the previous stage the controller for one joint and its sensors, in this case it will only be required to evolved the neural modules for the newly added motor and sensor, as well as the connections between those newly added modules and the already existing for the previous joint. However, a quicker and simpler solution exists, and it consists of performing a duplication of the modules created in the previous stage for the new joint. This means that the oscillator that controls one type of joint of the previous stage, is going to be copied to control the joint of the same type that is in the opposite side to the original one.
Then connections between both modules are established in order to apply the architecture definition. Since the interior of the copied networks were already evolved in stage one, only the connections between modules are going to be evolved in this stage (see figure 15). It is important to note that at this stage, the three types of oscillators are still completely independent yet from each type and evolved in different processes. So each evolutionary process has to cope with a reduced search space only related to its associated joint type. Also note, that it could also be possible to formally apply the tactical modular method not copying the modules and evolving completely the new joint controllers, but in this case we could take advantage of this situation to reduce the search and the evolutionary time.
Therefore, for each type of joint, the controller obtained in the previous stage is copied from the left leg joint to the right leg joint, obtaining by hence two isolated oscillators like the one

of figure 13, each one controlling their associated joint. However, the controller to be created requires that all the joints oscillate with a determined phase relationship. This phase relationship is determined by the type of gait that one wants to implement. The coordination between joints is achieved by creating the connections between both modules and evolving them. Figure 15 shows the final controller for the control of two joints in counter phase. Weights in red are the ones to be evolved.

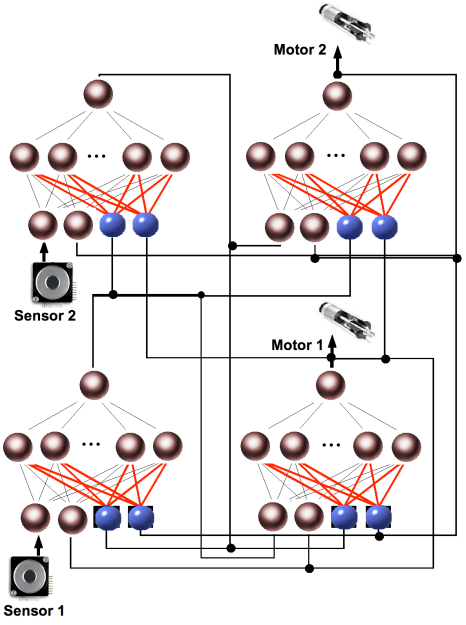

Figure 15: Connections between four IHUs corresponding to two joints of the same type.

Leg	Walk	Trot	Bound
Leg-Fore	0º	0º	0º
Right-Fore	180º	180º	0º
Left-Rear	270º	180º	180º
Right-Rear	90º	0º	180º

Table 2: Phase relationship for three common quadrupedal gaits (Collins & Richmon, 1994).

Then, connections are made between the two groups of nets. This implies that each neural net will have to add two more inputs coming from the outputs of the other two neural nets

duplicated. Once in the evolutionary process, only the new connections between IHUs will be evolved: neither the internal connections of the neurons nor the time constant and bias obtained from the previous stage will be evolved. Since the oscillation has already been obtained in the previous stage, it will not have to be evolved in this stage, although the phase relationship between the two oscillators will be required.
In this case, we are interested in the implementation of a walking gait, which implies a phase relation of 180º between those two legs (in all types of joints). The fitness function will be that which punctuates the phase difference between the legs close to 180º, and rewards a continuous oscillatory movement of both legs. The fitness function is composed of three parts: two parts are the fitness function of the first stage for each leg; the third part is the fitness factor that measures the variance between the movements of both legs, and tries to maximize it.

$$fitness = fit\_cross_1 * fit\_cross_2 * fit\_var \tag{4}$$

where *fit_var* is the variance of the difference of positions between both legs during the 400 evaluation steps, and fit_cross_1 and fit_cross_2 are the number of crossings that each joint performed through their mean position value. Similar to the fitness function for the first stage, these factors vary between 0.01 and 1.0 when their corresponding variable fluctuates between a good and a bad boundary.
For the results: ten runs were carried out for each type of joint to evolve the connections between CPGs. Each run was composed of 400 simulation steps of 96 ms. After 14 generations, 90% of the networks were capable of a counter-phase oscillatory pattern.

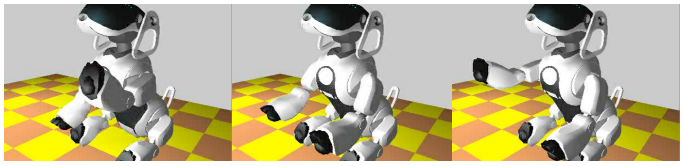

Figure 16: Sequence of the oscillation obtained in simulation for the J1 joint type. The robot is sit on its butt to allow a free movement of the joint. The evolution of the other joint types was performed with the same setup and similar behaviors were observed.

Third stage: coupling the oscillation of four joints of the same type
In this stage, the same procedure as in stage two was implemented. Now the control for the rear two joints of the same type is added to the controller. This means that four new neural modules will be added to the modular controller, two for the control of the two rear joints and two for the two rear sensors. However, since for a walking gait the two front joints have the same phase relationship as the two rear joints have, it is possible to clone the controller obtained for the front joints to the rear joints, and then evolve the connections between them. Since four new modules are added for the control of the two rear joints, it is necessary to evolve 4 connections per network, with a total number of 8 IHUs per joint type.
The evolution of the new connections is performed using a fitness function which rewards controllers that achieve an oscillation of all the legs with a given phase relationship, as

specified in table 2. For a typical walking gait the relation from left to right and from front to rear is 0º,180º,270º,90º.
Hence, making use of that algorithm, the fitness function designed to obtain coordination was composed of three parts: a part for each leg that expresses the oscillation requirement; a part that expresses the maximal variance requirement between the fore legs; and a final part that expresses the maximal variance requirement between fore-rear differences.
This is specified in the following fitness function:

$$fitness = fit\_cross * fit\_oscil * fit\_phases \tag{5}$$

where *fit_cross* is the product of the number of crossings for each joint performed through their mean position value, *fit_oscil* is the variance of the left fore joint which indicates how well that joint oscillates (if this joint oscillates, the others must follow), and *fit_phases* is the part of the fitness that indicates the phase relationship between all the joints. Similar to the fitness function for the previous stage, these factors vary between 0.01 and 1.0 when their corresponding variable oscillates between a good and a bad boundary.
For the results: the evolutionary process was carried out ten times. Each time, it was composed of 400 simulation steps of 96 ms each. After 26 generations, 92% of the networks were capable of the typical oscillatory walking pattern 0º, 180º, 90º, 270º (for the legs sequence fore_left, fore_right, rear_left, rear_right).

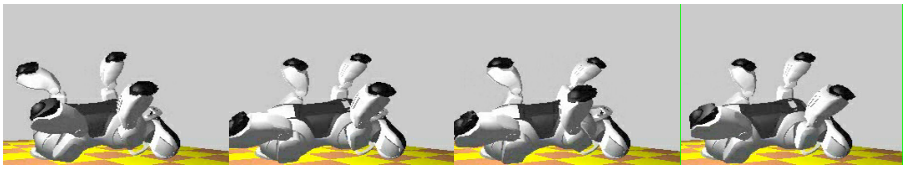

Figure 17: Sequence of the oscillation obtained in simulation for the J1 joint type. The robot is lying on its back to allow a free movement of the joint. The evolution of the other joint types was performed with the same setup and similar behaviors were observed.

Fourth stage: coupling between types of joints
The last stage is the coupling between the three groups of neural controllers obtained. From the previous stage three different oscillating modular controllers were obtained, one per joint type, with four joints of the same type oscillating together with a walking phase relationship. It is now required to interconnect the three layers in order to obtain a coordination between the different joint types, that enables the robot to walk, and completes the architecture as a whole. The next step will be the evolution of the connections between the three groups of controllers. In terms of walking, connection between groups should produce coordination between the different types of joints that have been evolved separately.
The connection between the three groups of controllers implies that 16 new inputs will be added to each IHU neural module. Those inputs represent the connection to the other 16 modules of the other two groups. Only those connections between groups are evolved to

generate the required coordination between the groups for the generation of a stable walking.
On a first approach, we tried to evolve the coordination between groups with a simple fitness function composed of the distance walked by the robot. However, the walking behavior obtained by that approach, even if correct, was very sudden and induced instabilities that made sometimes the robot fall. Analyzing the behavior obtained, we observed that the coordination between groups was correctly achieved but some of the joints had loose their oscillation pattern.
Because of that, a new fitness function was proposed were the oscillation of the joints was still imposed, together with the distance walked. If the robot does not fall over, the fitness function is composed of two multiplying factors: the distance d walked by the robot in a straight line and the phase relationship between the different joints. In the event of the robot falling over, the fitness is zero.

$$fitness = \begin{cases} d * fit\_phases & when \quad final \quad height > 0 \\ 0 & otherwise \end{cases} \tag{6}$$

For the results: a walking behavior was obtained after 37 generations for around 87% of the populations. A walking sequence obtained is shown in figure 18.

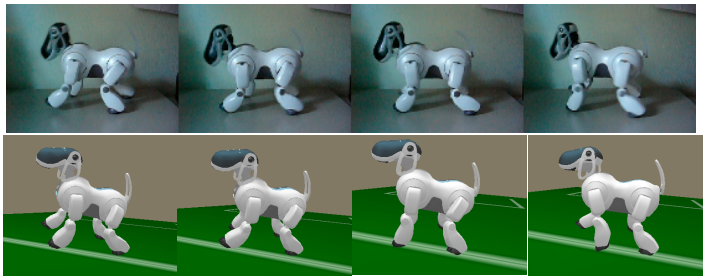

Figure 18: Top: Real Aibo walking sequence. Bottom: Simulated Aibo walking sequence

Once this walking behavior was obtained in the simulator, the resulting ANN based controller was then transferred to the real robot using the Webots simulator cross-compilation feature. The result was an Aibo robot that walks in the same manner as the simulated robot with some minor differences. A walking sequence obtained is shown in figure 18.

5. Discussion

The progressive design method allows for the evolution of complex controllers in complex robots. However, the process is not performed in a complete automatic way as the evolutionary robotics approach aims to have. Instead, a gradual shaping of the controller is performed, where there is a human trainer who directs the learning process, by presenting increasingly complex learning tasks, and deciding the best combination of modules over

time, until the final complex goal is reached. This process of human shaping seems unavoidable to us if a complex robot body, sensors/actuators, environment and task are imposed before hand. This point has also been suggested by other researchers (Urzelai et al. 1998; Muthuraman et al., 2003). But, different to other approaches, progressive design, by implementing modularity at the level of devices and also at the level of learning, allows for a better flexibility in terms of shaping of complex robots. The main reason is that, due to the modularization at the level of device, the designer can select at any evolutionary stage which small group of sensors and actuators will participate, under which task, and only evolve those modules. This would not be possible on a complex robot if modularization at the level of behavior is used.

Progressive design can be seen as an implementation of the incremental evolution technique but with a better control of who is learning what, at each stage of the evolutionary process. If incremental evolution were used on a controller with several inputs and outputs which controls every aspect of the robot, it would be possible to produce genetic linkage by which, learning some behaviors on early stages would prevent learning other behaviors in following steps, because the controller is so biased that it cannot recover from. This effect may be specially important in complex robots where several motors have to be coordinated. The learning of one coordination task may prevent the learning of another different one. Instead, the use of progressive design allows for the evolution of only those parts required for the task that they are required. This allows a more flexible design.

In the case of the Khepera robot, when the results obtained in the evolution of the eleven modules in one single stage process are compared with the results obtained by three stages, we observe that the progressive design of the controllers obtained a slightly better mean fitness value than the mean fitness obtained in the single stage case. Furthermore, the multiple staged approach generated a valid solution 100% of the time, meanwhile the one stage process did 90% of it. Then, progressive design showed to be more stable finding good controllers than the single stage process. The reason is that progressive design does the evolution at small steps in reduced searching spaces, and builds new solutions in new stages starting from an already stable solution provided by the previous stage.

But this fact has a good side and a bad side: the good side is what has been said about building more stable solutions because one stage starts evolving from the last stage stable solution. The bad side of this approach is that only a good enough solution can be provided. Due to the fact that previously evolved modules are freeze from evolving in the new stages, new stages have to carry the solutions found in previous ones. Therefore, it will be very difficult for progressive design to find the best possible controller. Only a good enough controller can be obtained, if a correct evolutionary shape strategy is implemented.

It is not clear wether the progressive design method will be useful in more complex robots with hundreds of modules. Even that progressive design allows the evolution of just a few modules at one stage, in the case of hundreds of modules, the last modules to be evolved will have hundreds of connections to evolve during that stage, what makes the search space large again. It will have to be analyzed in future work if the solution found until that moment will be able to direct the new stage towards a point of the fitness landscape where a solution is near. In both the Khepera and the Aibo experiments, it was observed that the solutions for one stage rapidly evolved from the solutions found in previous stage, manifesting this good landscape starting point effect. This makes us think that the method will be valid for more complex agents, if a progressive enough strategy is performed.

Drawback of the method: it is necessary the use of a simulator, to evolve at least, the first stages until a more or less stable controller is obtained.

6. Conclusion and future work

In this paper we have described the progressive design method for the generation of controllers for complex robots. In the progressive design method, modularity is created at the level of the robot device by creating an independent neural module around each of the sensors and actuators of the robot. This small conceptual modification from functional modularization is the responsible of the reduction of the dimension of the search space and of the bootstrap problem, by allowing a separate evolution of each device (or group of them) by stages.

This special type of staged evolution, evolves the neural controller by stages using evaluation-tasks, which are conditioned to the devices to be evolved. It must be stressed that determining the evaluation-tasks and the set of modules to evolve on each stage is the designer's job, and no general formula is provided. In general, the designer's knowledge of the problem will play a relevant role on it, introducing by hence a bias in the evolutionary process, which we think is unavoidable when working with complex robots. As a drawback, the introduction of knowledge reduces the likelihood of finding an original solution by the evolutionary process may find.

The architecture has been successfully used in several sensory-motor coordinations and compared in performance with other but further experiments show how the architecture would enable its use in more deliberative tasks. In (Téllez & Angulo, 2007), the ability of the architecture to express its current status is described. The architecture may be used in the future for the complete control of a robot, where the current status is sensed by a superior layer and used to deliberate and modify the robot behavior.

7. References

Auda, G. and Kamel, M. (1999). Modular neural networks: a survey. International Journal of Neural Systems, 9, 2, 129-151

Bianco, R. and Nolfi, S. (2004). Evolving the neural controller for a robotic arm able to grasp objects on the basis of tactile sensors. Adaptive Behavior, 12, 1, 37-45

J.J. Collins and S.A. Richmond (1994). Hard-wired central pattern generators for quadrupedal locomotion. Biological Cybernetics, 71, 375-385

Davis, I. (1996). A Modular Neural Network Approach to Autonomous Navigation, PhD thesis at the Robotics Institute, Carnegie Mellon University.

Doncieux, S. and Meyer, J.-A. (2004). Evolution of neurocontrollers for complex systems: alternatives to the incremental approach. Proceedings of The International Conference on Artificial Intelligence and Applications.

Dorigo, M. and Colombetti, M. (2000). Robot shaping: an experiment in behavior engineering. The MIT Press

Elman, J.L. (1991). Incremental learning, or the importance of starting small. Proceedings of the 13th Annual Conference of the Cognitive Science Society

F. Gomez and R. Miikkulainen (1996). Incremental Evolution of Complex General Behavior. Technical report of the University of Texas AI96-248

Grillner, S. (1985). Neurobiological bases of rhythmic motor acts in vertebrates. Science, 228, 143-149

G. Hornby and J. Pollack (2002). Creating high-level components with a generative representation for body-brain evolution. Artificial Life, 8, 223-246

A.J. Ijspeert (1998). Design of artificial neural oscillatory circuits for the control of lamprey- and salamander-like locomotion using evolutionary algorithms. PhD thesis at the Department of Artificial Intelligence, University of Edinburgh

Lara, B. and Hülse, M. and Pasemann, F. (2001). Evolving neuro-modules and their interfaces to control autonomous robots. Proceedings of the 5th World Multi-conference on Systems, Cyberbetics and Informatics

M.A. Lewis (2002). Gait adaptation in a quadruped robot. Autonomous robots, 12, 3 301-312

Mojon, S. (2004). Using nonlinear oscillators to control the locomotion of a simulated biped robot. Master thesis at École Polytechnique Fédérale de Lausanne

S. Muthuraman and C. MacLeod and G. Maxwell (2003). The development of modular evolutionary networks for quadrupedal locomotion. Proceedings of the 7th IASTED International Conference on Artificial Intelligence and Soft Computing

Muthuraman, S. (2005). The Evolution of Modular Artificial Neural Networks. PhD thesis at The Robert Gordon University, Aberdeen, Scotland

Nelson, A. and Grant, E. and Lee, G. (2002). Using genetic algorithms to capture behavioral traits exhibited by knowledge based robot agents. Proceedings of the ISCA 15th International Conference: Computer Applications in Industry and Engineering

S. Nolfi (1997). Using Emergent Modularity to Develop Control Systems for Mobile Robots. Adaptative Behavior, 5, 3-4, 343-364

S. Nolfi and D. Floreano (1998). Coevolving Predator and Prey Robots: Do ``Arms Races'' Arise in Artificial Evolution?. Artificial Life, 4, 4, 311-335

S. Nolfi and D. Floreano (2000). Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines. The MIT Press

S. Nolfi (2004). Evolutionary Robotics: Looking Forward. Connection Science, 4, 223-225

Pfeifer, R. and Scheier, C. (1997). Sensory-motor coordination: the metaphor and beyond. Robotics and Autonomous Systems, 20, 157-178

Pollack, J. B. and Hornby, G. S. and Lipson, H. and Funes, P. (2003). Computer Creativity in the Automatic Design of Robots. Leonardo, 36, 2, 115-121

R. Reeve (1999). Generating walking behaviours in legged robots. PhD thesis of the University of Edinburgh

R. Reeve and J. Hallam (2005). An analysis of neural models for walking control. IEEE Transactions in Neural Networks, 16, 3

C. W. Seys and R. D. Beer (2004). Evolving walking: the anatomy of an evolutionary search. Proceedings of the eighth international conference on simulation of adaptive behavior

Téllez, R. and Angulo, C. (2007). Acquisition of meaning through distributed robot control. Proceedings of the ICRA Workshop on Semantic information in robotics

Urzelai, J. and Floreano, D. and Dorigo, M. and Colombetti, M. (1998). Incremental Robot Shaping. Connection Science, 10, 341-360

H. Yong and R. Miikkulainen (2001). Cooperative coevolution of multiagent systems. Technical report of the Department of computer sciences, University of Texas AI01-287