ELSEVIER

Information Sciences 121 (1999) 1–25

INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL

www.elsevier.com/locate/ins

Evolving complex robot behaviors

Wei-Po Lee [*]

Research Lab. 301, Microelectronics and Information Systems Research Center, National Chiao Tung University, Hsin-Chu, Taiwan, ROC

Received 1 January 1998; received in revised form 25 September 1998; accepted 15 December 1998

Communicated by Kaoru Hirota

Abstract

Building robots is a tough job because the designer has to predict the interactions between the robot and the environment as well as to deal with them. One solution to such difficulties in designing robots is to adopt learning methods. The evolution-based approach is a special method of machine learning and it has been advocated to automate the design of robots. Yet, the tasks achieved so far are fairly simple. In this work, we first analyze the difficulties of applying evolutionary approaches to synthesize robot controllers for complicated tasks, and then suggest an approach to resolve them. Instead of directly evolving a monolithic control system, we propose to decompose the overall task to fit in the behavior-based control architecture, and then to evolve the separate behavior modules and arbitrators using an evolutionary approach. Consequently, the job of defining fitness functions becomes more straightforward and the tasks easier to achieve. To assess the performance of the developed approach, we evolve a control system to achieve an application task of box-pushing as an example. Experimental results show the promise and efficiency of the presented approach. © 1999 Elsevier Science Inc. All rights reserved.

Keywords: Evolutionary computing; Genetic programming; Computational intelligence; Robot learning; Automatic robot programming

[*] Present address: Department of Management Information Systems, National Pingtung University of Science and Technology, Nei-Pu, Pingtung, Taiwan, ROC.
E-mail address: wplee@mail.npust.edu.tw (W.-P. Lee)

1. Introduction

Building behavior-based control systems for robots has become a major alternative to the traditional robot design nowadays. Many behavior-based robots have been built in recent years, and this approach has been proven successful [1–3]. In contrast to the classical robot control (which divides the task of implementing intelligent behaviors along functional lines to form a *sense–think–act* cycle), behavior-based control performs task decomposition to structure the overall control mechanism as layers of behavior modules in an incremental manner. A behavior module is a task-achieving controller; it is a simple but complete computational mechanism which only uses sensory information related to this partial task to compute actuator outputs. To build a behavior-based system is to develop the behavior modules first, and then to design mediators to combine the individual results from different behavior modules into an unified command for actuators. By this design methodology, researchers are able to break the computational bottleneck, and successfully build robots acting in the real world in real time.

However, the extension of this approach to design complete autonomous robots for more complex tasks still present some challenges, as its initiator Brooks has already informed us [4]. The first concern is about how to code a single behavior module always capable of dealing with the information in the uncompletely known world to achieve a specific task. The second one is actually the action selection problem; it concerns how to decide, for a system including multiple behaviors to handle a variety of situations, which behavior or behaviors should be active at any particular time to achieve various tasks. All these problems will become even more challenging when the number of behavior modules involved increases to follow the increase in task complexity.

Both of the above problems are in fact caused by the difficulties in predicting complicated robot–environment interactions and in dealing with them. One solution to these problems is to adopt *learning* methods by that the robot is expected to self-improve its own behaviors to adapt to the different environmental situations it experiences, without explicit programming. Evolution-based algorithms, inspired by the Darwinian principle of selective reproduction of the fittest, are a specific kind of machine learning approach. They simulate the natural evolution process in which members of populations change their structures to adapt to the environment, in order to survive. The evolutionary algorithms have been proposed to synthesize (evolve) robot control systems [5,6] and there are some successful results [6,8,9]. As can be seen, researchers in this application field have mainly focused themselves on exploring the power of evolutionary algorithms and on analyzing the dynamics of evolutionary mechanisms, so their experiments always tried to use the least built-in knowledge to evolve a monolithic control systems from the scratch (i.e., evolving the controller as a whole). Admittedly, the tasks achieved so far, such as obstacle avoidance or light seeking, are relatively simple.

To investigate how the evolutionary computation technique can be used to develop robot controllers for more complicated tasks, in this paper we first analyze the problems of direct use of this technique to solve complicated control tasks, and then present a practical approach at the intermediate level: instead of hand-coding a behavior-based system or insisting on a strong learning principle as others do, we take a distributed control architecture similar to the behavior-based system, meanwhile, use the evolutionary technique to learn the individual behavior modules and the coordination strategies between them. This will take the advantages of both approaches and mediate between the two to overcome their own disadvantages. To demonstrate the proposed approach can speed up the learning process and can be scaled up to develop control systems for more complex tasks, we use it to undertake an application task of box-pushing as an example. In addition, we also use the GP system to evolve monolithic controllers to solve the same task in comparison to our approach. The results prove that only our approach can successfully evolve controllers to fulfill the target task.

2. Background

2.1. Related work

In general, the process of evolving control systems is similar to that of the traditional evolution-based work; but the major difference lies in that evolving controllers involves a robot (simulated or real). Evaluating a controller is to execute the corresponding control code on a robot in the experimental environment for a period of time and then to measure its performance. Based on the evolutionary technique used, research work in this field can be categorized into genetic algorithm (GA)-based work [6–8] and genetic programming (GP)-based work [10–12]. A GA-based work normally involves encoding the parameters of a controller, such as the weights and thresholds of a neural network, to a linear string and then used a standard GA to determine values for the relevant parameters. A GP-based work, on the other hand, uses a tree structure to represent a control program and employs GP techniques to determine the tree-like program. According to the literature work, it can be observed that using GP to evolve robot controllers has the innate advantage of operating variable-size genotypes. This is an important feature in developing control systems because it provides complete freedom to evolve the morphology of controllers. However, the GP-based work has been criticized for its need of a relatively large population size which in fact means more computational resources. This is especially an unfavorable feature in evolving robot controllers for its time-consuming characteristic. But, in this work we will show that by defining an appropriate representation, GP can restrain itself to use a relatively small population size to evolve reliable and robust controllers in a short period of time.

2.2. Genetic programming

Genetic programming is a new evolutionary computation technique which was recently invented by Koza [13], and its popularity is now increasing in the community of evolutionary computation research. It is an extension of the traditional GAs with the basic distinction that in GP, the individuals are dynamic tree structures rather than fixed-length vectors. GP aims to evolve dynamic and executable structures often interpreted as computer programs to solve problems without explicit programming.

As in computer programming, a tree structure in GP is constituted by a set of non-terminals as the internal nodes of the trees, and by a set of terminals as the external nodes (leaves) of the trees. The construction of a tree is based on the syntactical rules which extend a root node to appropriate symbols (non-terminals and/or terminals) and each non-terminal is extended again by suitable rules accordingly, until all the branches in a tree end up with terminals. Hence, the first step in applying GP to solve a problem is to define appropriate non-terminals, terminals and the syntactical rules associated for the program development. The search space in GP is the space of all possible tree structures composed of non-terminals and terminals.

The next step is to evaluate tree-individuals to determine their fitness for the creation of a new population. This is normally done by pre-defining a fitness function which quantitatively describes the requirements of a target task first, and then executing the corresponding codes for tree-individuals in the environment of the particular problem. After that, the genetic operators are applied to the fitters selected (based on a certain selection criterion) to generate new trees. The evaluation and recreation cycle is repeated until the termination criterion is met. Fig. 1 illustrates the general flow of such an evolution mechanism.

In GP, three kinds of genetic operators – reproduction, crossover, and mutation – are normally used to create new tree individuals. The reproduction simply copies the original parent tree to the next generation; the crossover randomly swaps subtrees for two parents to generate two new trees; and mutation randomly regenerates a subtree for the original parent to create a new individuals. Among them, crossover is the major one to create most of the offspring; when it is performed, all syntactic constraints must be satisfied to guarantee the correctness of new trees. Fig. 2 shows an example of the operation of crossover in GP. Considerable details about GP are referred to [13].

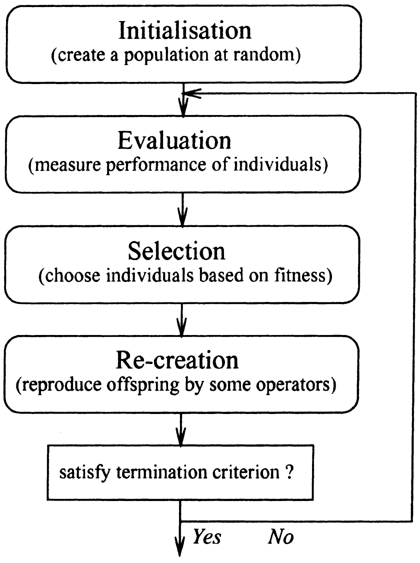
Initialisation
(create a population at random)
Evaluation
(measure performance of individuals)
Selection
(choose individuals based on fitness)
Re-creation
(reproduce offspring by some operators)
satisfy termination criterion ?
Yes
No

Fig. 1. The general flow of a simulated evolution mechanism.

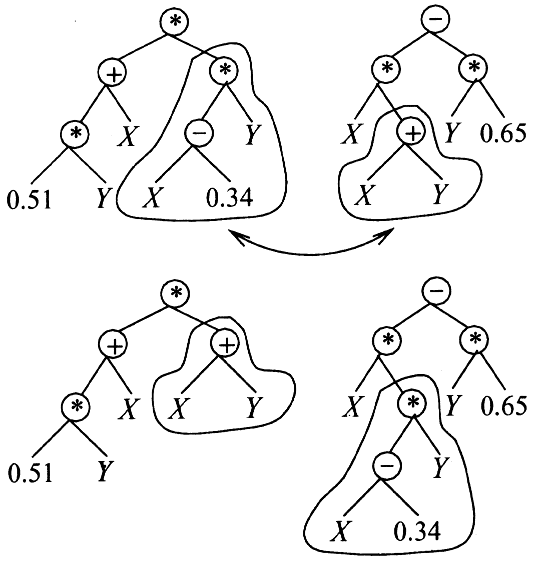

Fig. 2. An example of crossover in GP. Two new trees, representing two functions different from the original ones, are created by swapping subtrees of parents.

3. Evolving controllers for complex tasks

3.1. The difficulties

Generally speaking, the evolutionary approach is a kind of search-based method in which genetic operators are used in the hope of finding a satisfactory solution in a space; and the dimensionality of this space is determined by the number of basic elements constituting a controller. When the size of the controller is reasonably increased to match the increase in task complexity, the solution space will grow exponentially and then leads the search to be more and more difficult. This is particularly apparent in work that uses recurrent neural networks as control systems, since the characteristic of recurrence enlarges the search space even faster.

Increasing task complexity also introduces certain difficulties in defining fitness functions to guide the search direction during evolution. In evolving controllers, it is preferred to define a fitness function in terms of lower-level quantities, such as sensor or motor activities. This is because of that these quantities vary gradually and continuously when the robot is acting in the world; different controllers thus have distinguishable fitness values and this makes an evolutionary system easy to converge, and the task easy to achieve. However, an increase in task complexity implies a high-level goal to achieve, which almost always involves the interaction of multiple subgoals. Under such circumstances, defining a fitness function means to describe multiple subgoals and the interaction between them simultaneously – this is never a easy job.

Another way is to directly define a fitness function at the higher-level for a complex task. This is relatively straightforward, but it makes the task difficult to achieve. For example, in the work [14], the authors have shown that, in their grasping task, if the fitness function was simply defined as the number of objects grasped and deposited correctly, the desirable behavior could not be evolved successfully. This is due to the fact that during the earlier generations, none of the individuals can achieve the complete task; this results in equally bad fitness for all the population members (all scored zero) and made all the control systems indistinguishable in performance. On the other hand, in the same example, if lower-level subgoals were introduced to the fitness function, such as rewarding the behaviors of recognizing objects and picking objects up, the performance of controllers became more distinguishable and then the target task could be achieved, but at the expense of making great efforts to define a proper fitness function. It shows that manipulating fitness at lower levels can assist the evolutionary system to converge; nevertheless defining an appropriate fitness function at a lower level is difficult. Furthermore, such difficulty will increase as the consequence of the increase in task complexity.

The above analyzes the difficulties one will encounter in evolving monolithic control systems for complex tasks. From the point of view of controlling a robot, we may want the evolved control systems to be distributed for their corresponding advantages, but this is generally difficult to obtained by the strategy of evolving a control system as a whole. In a distributed architecture, the perceptual processing is distributed across multiple independent modules, and every module only deals with the sensory information directly related to its particular need. This not only reduces the sensory bottleneck but also allows each control module to be developed with the most suitable representation and approach with least restriction. Owing to the modular and distributed characteristics, the performance of the overall system will degrade gradually, even if some of the devices or control strategies do not function properly. Also, with an explicitly distributed architecture, an overall system will be easily integrated from different subsystems which could be designed independently; it can also be easily maintained. Therefore, from the point of view of developing robot control systems, a centralized control structure from the strategy of evolving a control system as a whole is not suitable.

3.2. Task decomposition

In order to reduce the search space to make the search easier, to simplify the job of defining fitness functions, and to obtain a distributed control system, a breakthrough is to adopt the divide-and-conquer problem-solving methodology. In such approach, the designers break tasks from complex (higher-level) down to simple (lower-level) recursively and then achieve the tasks in the reverse sequence. How to decompose a task generally depends on the designers' experiences, and human designers are normally quite capable of that. The tasks are arranged to be achieved in a sequence of increasing complexity and, at each level, the control systems are evolved on top of the ones evolved at lower-levels. Hence, fitness functions will become easier to define, and the tasks easier to achieve (the fitness function of a certain level task can be defined simply as the goal at this level, to reduce the difficulty in embedding the lower-level subgoals into it; and evolving control systems on top of other lower-level controllers can exploit their corresponding control skills to achieve the current goal). In addition, each subtask only needs to deal with the perceptual information directly related to it, which also makes the tasks easier to achieve.

As a matter of fact, the concept of such approach is much like behavior-based control described in the first section, while the main difference is that the approach here employs evolutionary techniques to *evolve* new behaviors and behavior coordinators, rather than to *hand-code* them. By the use of evolutionary techniques, the human designer can concentrate on the system-level design and let the evolutionary system take care of the implementation details. In addition, since the tasks in this approach are decomposed in the horizontal way proposed in [15], the corresponding control architectures will be explicitly distributed and then fully exploit all the advantages of distributed architectures as analyzed in the above section. The task decomposition technique has also been applied to other robot learning domains; examples are Dorigo and Colombetti's [7] and Colombetti et al.'s [16] work in learning classifier systems, and Mataric's [17] and Lin's [18] work in reinforcement learning.

We are especially interested in investigating some ways to reduce the load of robot programmers and in evolving distributed architectures for complex tasks. After analyzing the advantage of the use of task decomposition, we will now investigate how to combine this technique, with our GP system to evolve control modules and coordinators to achieve complex tasks.

4. Evolving distributed task-achieving controllers

As described above, to evolve distributed control systems to achieve complex tasks, we intend to use the technique of task decomposition to break the overall tasks and use the GP techniques to evolve separate behavior controllers and coordinators for integration. In this section, we will describe the general structure of the control architecture corresponding to the task decomposition, and then present the genetic representation of the controller to be evolved.

4.1. Control architecture

Since we will decompose tasks in a hierarchical way, the corresponding control system is organized in multiple layers. After decomposition, the overall control system includes a set of *behavior primitives* and *behavior arbitrators.* Here, a behavior primitive is a reactive controller with the representation described in the section below; it involves the lowest-level sensory–motor control. Unlike the fixed priority network in the subsumption architecture [15], a behavior arbitrator here is not hardwired in advance; it is also treated as an adaptive controller and implemented as a switcher as shown in Fig. 3(b). The behavior arbitrator has the same structure and representation as the primitives; the only difference between them is: the output of a primitive is used to control the motors, but the output of an arbitrator is used to activate one of the subcontrollers involved. Thus, in a similar manner to a reactive planner in [19] or a conditional sequencer in [20], an arbitrator here allows the binding between environment conditions and activations of lower-level behaviors to take place at run time. This provides adaptiveness not only at the lower-level sensory–motor control but also at the behavior level.

Depending on whether the computing system used supports parallel computation, the control flow in the control system with the above architecture can be implemented as bottom–up (if parallel computation is supported) or top–down (if not). For bottom–up flow, all reactive controllers are active and run in parallel. The behavior primitives send outputs to the

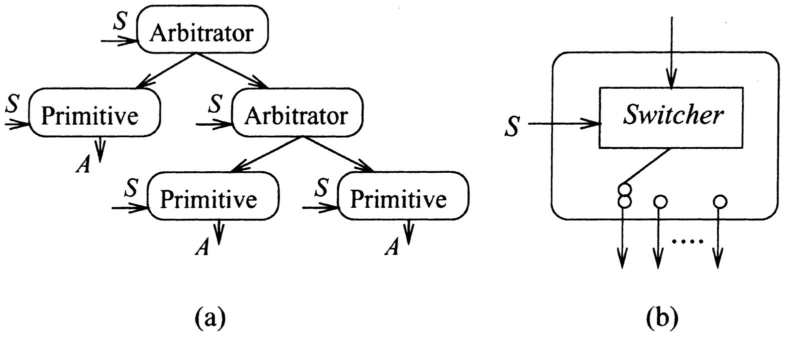

Fig. 3. (a) The general architecture of a control system. *S* and *A* represent the sensors and actuators related to a certain control work. (b) The implementation of an arbitrator.

arbitrators as their inputs, and each arbitrator selects one of its inputs (according to the environmental stimuli) as its output, and then sends this value to higher-level arbitrators. In this way, the output of the highest-level arbitrator will be the output of the overall control system. In contrast to this, for top–down flow, all the control modules are passive. At each time step, the highest-level arbitrator invokes one of its subcontrollers to be in charge of the control, according to certain sensory information. If the invoked subcontroller includes an arbitrator, this arbitrator will be evaluated first and its output can then be used to activate another controller. This process continues until a control primitive at the lowest level is invoked and drives the actuators. Because our system does not support parallel computation, top–down flow is used. Fig. 3(a) illustrates the general architecture of our control systems.

4.2. Genetic representation of a behavior controller

In the behavior-based control paradigm, the *circuit network* has been proven to provide a finer-grained view to represent a behavior controller. In the *circuit approaches* [21–23], a behavior controller exists in the form of digital hardware; and it is made up by two types of components, pure functions and delays, depending on what kind of tasks (reactive or sequential) it is achieving. Pure functions mean logic gates, and delays correspond to the flip-flops or registers. The output of one component may be the input to one or more other components, thus forming a network. Signals propagate through the network and sensing is thus linked to action. As is well known, any finite state transduction can be carried out by such a network. In this work, we concentrate on the reactive controller; afterwards the same approach can be extended to evolve sequential ones with minor modification.

The genetic representation of our reactive controller is inspired by the logic representation in the circuit approaches. By duplicating and separating those components, the output of which serve as inputs of multiple components, and by introducing a dummy root node to connect the outputs of a circuit network together, we find it very straightforward to convert a circuit network to a circuit tree. Fig. 4 shows an example. In this figure, an input variable X_i represents a sensor response thresholded by a pre-defined value. But in a real control work, the threshold is normally unknown and, when the robot continuously senses the environment to monitor the variation, it not only checks whether certain sensors reach some kind of threshold, but also observes the difference between sensors (if they are comparable) to determine its actions. Therefore, in our representation, we structure the perception information into *sensory conditionals* and connect them to the inputs of a logic circuit.

In our design, structured sensory conditionals involve comparing the responses of different sensors or comparing sensor response to numerical thresholds. For these purposes, both sensor responses and numerical thresholds are normalized to be between 0 and 1 inclusive. Thus, a sensor conditional has a constrained syntactic structure; it exists in the form of $X > Y$, where X, Y can be any normalized sensor response or threshold which is determined genetically.

Depending on the characteristics of the specific tasks, different kinds of sensors will be required. In general, a sensor is defined to be associated with a value between 0 and 1 which indicates the angle between the direction which the sensor is pointing at and the robot's heading. Thus, whenever a sensor is called in the control system, the normalized sensor response is returned, in the direction indicated by the value associated with that sensor. For instance, a

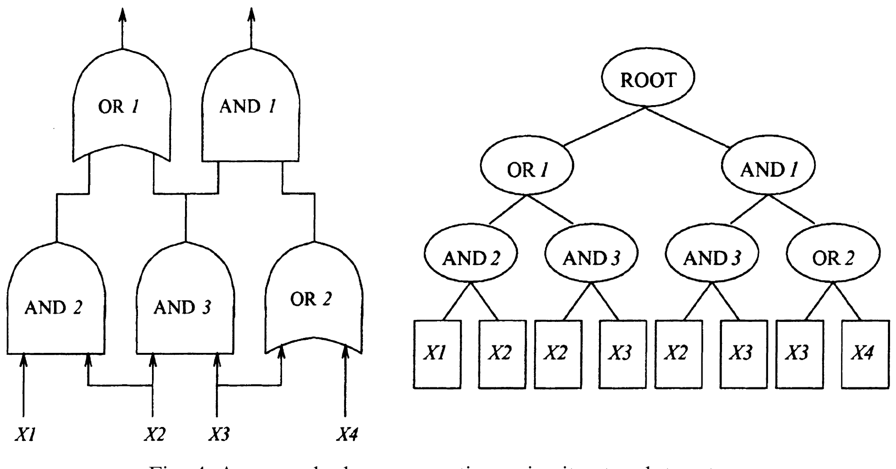

Fig. 4. An example shows converting a circuit network to a tree.

sensor with the value 0.3 will return the normalized sensor response in the direction 0.3 revolution (108°) anti-clockwise, relative to the robot's heading. In this way, the sensor positions and directions are also allowed to be co-evolved if the sensors are adjustable [24]. For a robot with fixed sensors, the values associated with the sensor are constrained, subject to the availability of sensors. In the experiments below, we use a robot with fixed sensors.

After organizing our genetic representation, we then define *non-terminals* and *terminals* which constitute a circuit tree for our GP system. In general, three types of non-terminals are defined: the dummy root node, the logic components, and the comparator. The dummy root node is to collect the main outputs of a control system for convenient manipulation by a GP system; the logic components are to constitute the main frame of the controller to map the structured sensor information into appropriate actuator commands; and the comparator is to construct the sensor conditionals. As is mentioned above, because the elements in a sensor conditional can be normalized sensor response or numerical thresholds, both of them are defined as terminals. The tree representation of a typical controller is illustrated in Fig. 5. In such a structure, the outputs of the subtrees of a circuit tree are interpreted as actuator commands to drive actuators.

To evolve instances to solve different control tasks, we have to define different sensor terminals, depending on the requirements of the specific tasks. For example, we may define infra-red sensors as sensor terminals to detect walls and objects for an obstacle avoidance task; we may also define ambient

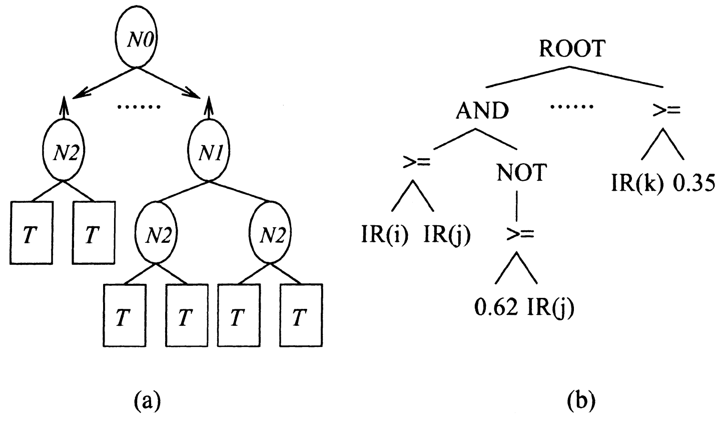

Fig. 5. (a) The general structure of a controller. In this figure, *N0* is a dummy root node, *N1* represent logic components, and *N2* is the comparator >=. *T* can be a normalized sensor response or a threshold between 0 and 1 inclusive. The outputs of the subtrees are used to drive the actuators or activate another control system. (b) An example of a typical controller in which *i*, *j*, *k* are values indicating the sensor directions.

light sensors as terminals to sense the light for a phototaxis task. The following experiments will give an account of how to evolve such kind of controllers in detail.

4.3. Other implementation issues

Because there are some constrained syntactic structures defined in this work, the crossover operation must be constrained to protect the defined structures. If the selected crossover point in the first parent is the root node, the second crossover point must also be a root node; if the chosen crossover point in one parent is an internal node, then the crossover point in the other parent must be an internal node as well; otherwise if the selected crossover point in the first parent is a terminal node, the crossover point for the second parent is restricted to be a terminal node.

In order to maintain the diversity and to reduce the computation cost, an *island model* [25] GP system is implemented. The subpopulations in our distributed genetic system are configured as a binary *n*-cube. Migration will happen only between immediate neighbors, along different dimensions of the hypercube, and the communication phase is to send a certain number of the best individuals of each subpopulation to substitute the same number of the worst individuals of its immediate neighbors at a regular interval.

5. Experimental setup

5.1. The robot and simulator

The robot used in this work is the miniature mobile robot, *Khepera* [26] (Fig. 6). It has two wheels driven by two DC motors and both motors can revolve forward and backward independently. It is originally equipped with eight infrared proximity sensors which can also be used in a different mode to measure ambient light around the robot. For the purpose of identification, the sensors are numbered from 0 to 7 as illustrated in Fig. 6.

The main goal of this paper is to demonstrate how our approach can be used to overcome the difficulties in building behavior-based robots manually and in evolving monolithic control systems, so we only present the simulation result. The simulator in this work was built by a look-up table approach and its performance, in bridging the gap between the simulated and real robots, has been shown in our preliminary study: the evolved controllers can be downloaded to the real robot without the loss of performance [27].

In addition to the eight sensors mentioned earlier, we assume that there are another set of eight sensors on the top of the simulated robot and those top sensors are higher than the box. This is to ensure that the robot can detect the

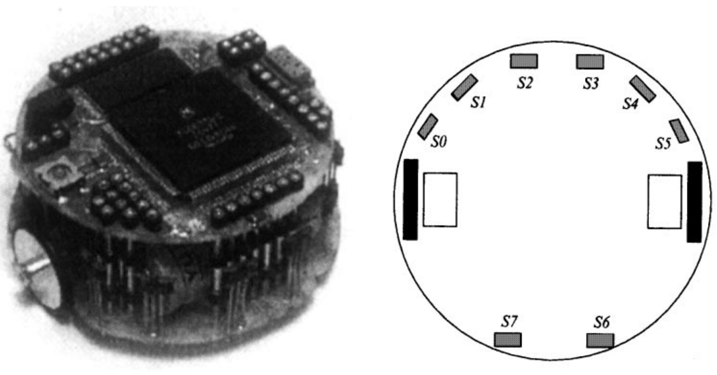

Fig. 6. The Khepera miniature robot and its sensor arrangement. In the right figure, a sensor S_i can function as an infra-red or an ambient light sensor.

light by the upper set of sensors, even in the situation where the box is between the robot and the light. Besides, the two sets of sensors can be used to construct box recognizers: a recognizer BR is defined to give the normalized reading difference between a pair of upper and lower IRs that point at the same direction.

5.2. Task description and decomposition

In the following experiments, we will follow the approach described in Section 4 to develop a behavior-based style control system for a moderately difficult box-pushing task. In this task, the robot has to explore the given arena in order to find a box; once it detects the box, it is then required to push the box toward a goal position indicated by a light source.

The task to be achieved is difficult for the following reasons. First of all, the robot is round, so that it only contacts the box at one point while pushing it; when the pushing force exerted by the robot is not directed straight through the center of the box, it tends to slide and rotate unpredictably. Therefore, the robot has to adjust its own position occasionally in order to push the box forward. Furthermore, as there is no particular restriction on the initial relative positions of the robot, the box, and the ambient light, the robot could approach and detect the box at any position and orientation around the box; under such circumstances, the robot needs to deliberately move to a proper position in order to perform an efficient pushing to satisfy the final goal.

To accomplish this task, we can decompose it into two subtasks, *exploration* and *push-box-toward-light*. The former is to control the robot to explore the given arena in order to find the box without bumping into a wall; and the latter, to push the box detected to the light center. As is mentioned above,

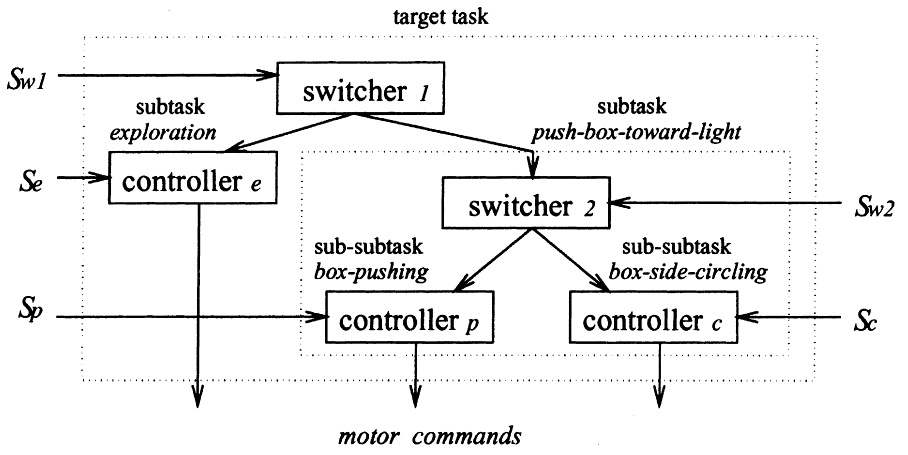

Fig. 7. The decomposition and integration of the target task; S_i indicates the sensory information relevant to control work *i*.

when the robot finds the box, it has to move to a proper position before pushing it, so the task *push-box-toward-light* can be decomposed again into two even lower-level subtasks, *box-pushing* and *box-circling*. *Box-pushing* is to keep the robot pushing a box forward, while *box-circling* is to keep the robot moving along the side of a box in order to provide the opportunity for the robot to move to suitable positions for pushing. Fig. 7 shows the result of decomposition and the design of the corresponding architecture for the target task. Each of the atomic subtasks is controlled by a separate behavior primitive, and the different subcontrollers can be merged by an arbitrator, which is implemented as a switcher as described in Section 4.1.

6. Experiments and results

In our experiments, a fitness function is in fact a penalty function. This differs from the traditional GAs, but is often used in GP-based work. As we emphasized previously, in our approach, the fitness function for a lower-level task should be defined in terms of sensor and motor activities; but for a complex task, the fitness function can be directly defined at the high level without describing all of the subgoals involved. The experiments below will show how fitness functions at different levels are defined and how our approach works.

To be objective, 10 independent runs (with different random seeds) were conducted for each task. In a single evolutionary run, 2 populations of 50 individuals were used. For the behavior primitives, each run lasted for 50 generations; but for the behavior arbitrators the GP system was run for 100 generations, in order to clearly demonstrate the differences between strategies with and without the use of task decomposition (we have also conducted experimental runs without using task decomposition for comparison). During the experiment, an individual was trained in multiple trials with different starting positions, to ensure the robustness and reliability of the evolved result [24]. Experimental details are described below.

6.1. Evolving a primitive for the box-pushing task

The task of *box-pushing* is that the robot keeps pushing a box forward as straight as possible. To achieve such a task, the robot needs to use its IR sensors to acquire perception cues for the location of the box. Therefore, we defined two kinds of terminals, IRs and numerical thresholds, for our GP system to evolve controllers capable of achieving this task.

The second step is to define a fitness function to guide the evolution. For this low-level task, the fitness function was formulated, through the quantitative description of the expected behavior, as keeping the activation value of the robot's front IR sensor high, the robot moving fast forward, and the speed difference between two motors low. The pressure from keeping the front IR sensor with high activation value was to reinforce the robot to approach a box; and the pressure from keeping robot moving fast forward with low speed difference between two motors was to prevent it from getting stuck in front of a box and to encourage the robot to move straight. The combination of these can lead to a pushing-forward behavior. Thus, the fitness function for evolving a behavior controller of box-pushing was defined as

$$f = \sum_{t=1}^{T} [\alpha * (1 - s(t)) + \beta * (1 - v(t)) + \gamma * w(t)],$$

in which $s(t)$ is the average of normalized sensor activations of the front sensors IR*2* and IR*3*, $v(t)$ the normalized forward speed, $w(t)$ the normalized speed difference of two motors at each time step t and α, β, γ are the corresponding weights expressing the relative importance of the above three criteria (and determined by preliminary testing).

In order to illustrate the performance of our GP system, the fitness curves over the 10 runs are shown in Fig. 8(a). It indicates that our GP system is considerably efficient: the performance was improved quickly and stably. The typical box-pushing behavior of the simulated robot, when performing the evolved controller, is shown in Fig. 8(b). To prove its reliability and robustness, the controller evolved was tested many times and at each time the robot started from an arbitrary position and heading around the box. During the tests, the robot always generated the consistent behavior: it turned to face the box,

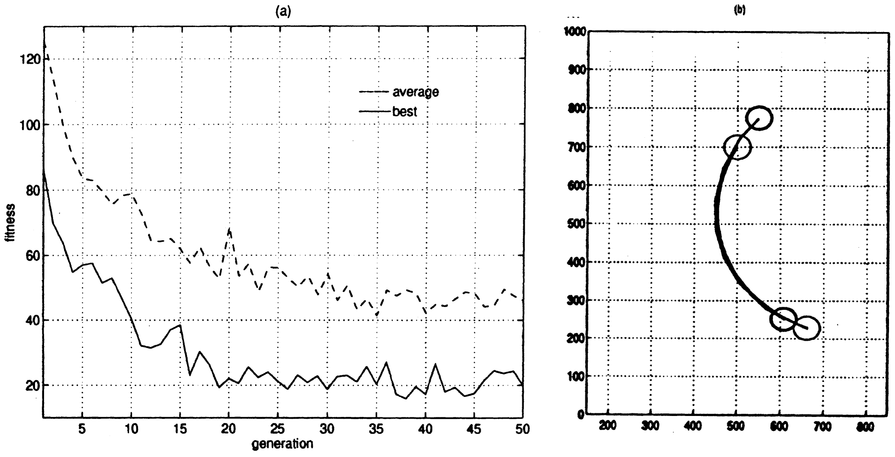

Fig. 8. (a) Population average fitness and best individual fitness at each generation. Values are averaged over 10 runs. (b) The trajectories of the simulated robot when it was pushing a box (the darker circles represent the boxes; the boxes are pushed from top to down).

approached it, and then pushed the box; while at some specific time steps, the robot produced prompt turns to drive back from path deviations and to head towards the box again.

6.2. Evolving a primitive for the box-circling task

The task of *box-circling* is defined as that the robot keeps moving forward and circling along the sides of a box. As performing the *box-pushing* task, the robot needs to use its IRs to capture the location of the box in this task. Thus, terminals for evolving a controller to achieve this task were defined as the same as those in the *box-pushing* task: IRs and numerical thresholds.

Again, a fitness function was needed, and it was formulated as keeping the side sensor IR*0* with a certain activation value and the speed positive. The former was to encourage the robot to keep a certain heading relative to the box and a certain distance away from the box; and the latter was to reinforce the robot moving forward. The combination of these would be able to produce a box-circling behavior. Thus, the fitness function was defined as

$$f = \sum_{t=1}^{T} [\alpha * \text{abs}(s(t) - k) + \beta * (1 - v(t))],$$

where abs is the absolute function, $s(t)$ gives a normalized activation value of the specific sensor IR*0*, k a pre-defined constant (between 0 and 1) indicating

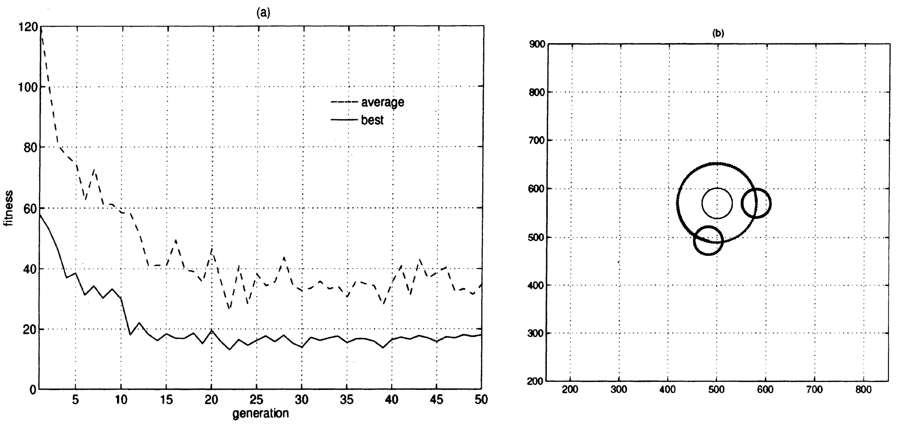

Fig. 9. (a) The fitness curves averaged from 10 independent runs. (b) The box-circling behaviors of the robots.

the distance between the robot and the box; and v is the normalized forward speed of a robot.

Fig. 9(a) shows the fitness curves over the 10 independent runs and Fig. 9(b) presents the evolved box-circling behavior of the simulated robot, which demonstrates that the task was achieved successfully. We tested the evolved controller several times by putting the robot around the box with an arbitrary heading each time. In all tests, the robot had the similar behavior: it performed turning to adjust its heading first and then moving along the side of the box with a certain turning rate.

6.3. Evolving a primitive for the exploration task

In this task, the robot is required to wander safely in an enclosure and visit as much of the enclosed space as possible. It can be described quantitatively as that the space is divided into some grid squares and the robot must visit as many squares as possible during a fixed period of time. There are different ways to achieve this task. For instance, it can be achieved by using a map to provide location information to the robot. Having the location information, the robot can realize its own location and the locations of those squares that have been visited already. It can then head to those squares which have not been visited according to the records of the map. There is also another kind of strategy without using a map, when no location information is available. In such a strategy, the robot does not know where it is and which squares it has not visited yet. To carry out the exploration task, the robot needs to determine its turning angle carefully in the situations when it senses the boundary of the enclosure. In our experiment, we intended to evolve a reactive controller for exploring a space without using location information.

Because the controller to be evolved was reactive and there was no location information provided here, to achieve this task the robot must fully exploit its IR sensors to determine the turning angle carefully. Since IR sensors were the only mechanism for providing perception cues, the terminals for the exploration task were then defined to include IRs and numerical thresholds as the above two tasks. Unlike the experiments presented above, the fitness measurement for this task was not the sum of penalties over each time step but rather could only be assigned after a complete trial. The main concern for the fitness was to minimize the number of squares which had not been visited, while an extra pressure on the speed was added to encourage the robot to move forward when exploring. Thus, the fitness function was defined as

$$f = \alpha \times (1 - P) + \beta \times (1 - \mathrm{Avg}),$$

where P is the proportion of the space visited, i.e., *visited-grids/total-grids*, and Avg is the average speed of the robot during a complete trial.

Fig. 10(a) illustrates the converging fitness curves for evolving controllers of exploration. It indicates the fast and smooth converging behavior of the evolutionary runs and again, shows the performance of our GP system. The typical exploration behavior produced is presented in Fig. 10(b), which shows that the robot was able to visit most of the specified arena during a fixed period of time. We should note that it is not important how the robot moved, when it did not sense anything; but the appropriate match between the turning angle (when the robot sensed the wall) and the way it moved (when it did not sense anything) is nevertheless crucial for a reactive controller to perform

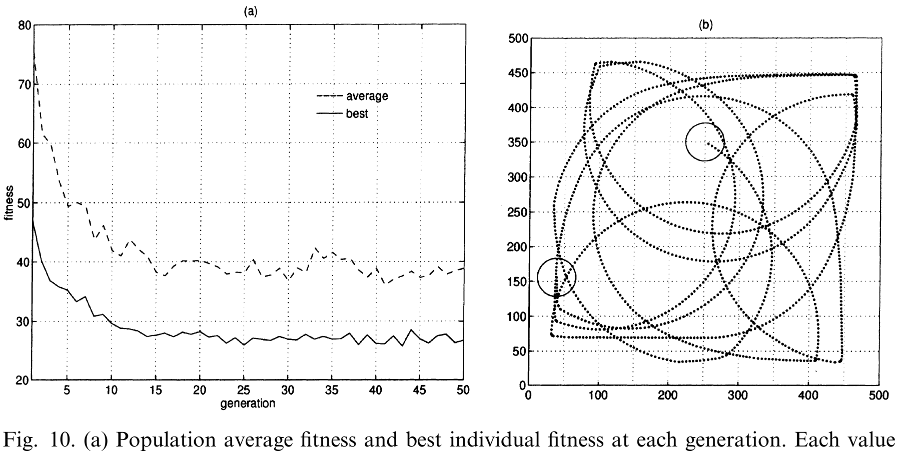

Fig. 10. (a) Population average fitness and best individual fitness at each generation. Each value represents the average over 10 runs. (b) The exploration behavior evolved.

exploration. As we can see in Fig. 10(b), a successful match has been evolved and it enabled the robot to achieve the task.

6.4. Evolving an arbitrator for the push-box-toward-light task

As mentioned above, an arbitrator is also implemented as a reactive controller; its inputs are from sensors, and its outputs are used to trigger other controllers. For the arbitrator here, two kinds of sensors – IRs and LDRs – are needed to detect the locations of the box and the light, so both kinds of sensors and the numerical thresholds were defined as terminals to the GP system to construct the conditionals for the arbitrator. Since there were only two subcontrollers involved, the arbitrator was designated to have a single output to activate them: if the output was 0, then the controller for subtask *box-pushing* dominated the control, otherwise the controller for subtask *box-circling* did. During the experiment, the two subcontrollers were frozen and only the arbitrator was evolved.

In this task, the robot was expected to push the box as close as possible to the center of the area brightened by the light. As analyzed in Section 3.2, we can directly define the fitness function at this level: to measure the distance rather than to describe the low-level quantities. Thus, the fitness function was defined as

$$F = \sum_{t=1}^{T} D_{b,l}(t),$$

in which $D_{b,l}(t)$ represents the distance between the box and the light source at each time step t. In this experiment, each step was a complete cycle: both the arbitrator and the activated controller were executed once.

Fig. 11 illustrates, step by step, the typical behavior of the robot. As can be seen, the arbitrator first activated the primitive *box-circling* to move the robot along the side of a box. Once the robot reached an appropriate position in which the box was between the light and the robot itself, the control was immediately switched to the other primitive, *box-pushing*, to drive the robot to push the box forward. The *box-circling* and the *box-pushing* primitives were activated again in the same order if the pushing path deviated. After the box was pushed to the goal position, the arbitrator continuously activated the primitive *box-circling* to make the robot circle the box in order to prevent it pushing the box away from the goal position. From Fig. 11, we can see that the box was successfully pushed to the center of the bright area.

6.5. Evolving arbitrators for the overall task

After evolving an arbitrator to combine two pre-evolved lower-level primitives, we can regard the integrated control system (including one arbitrator

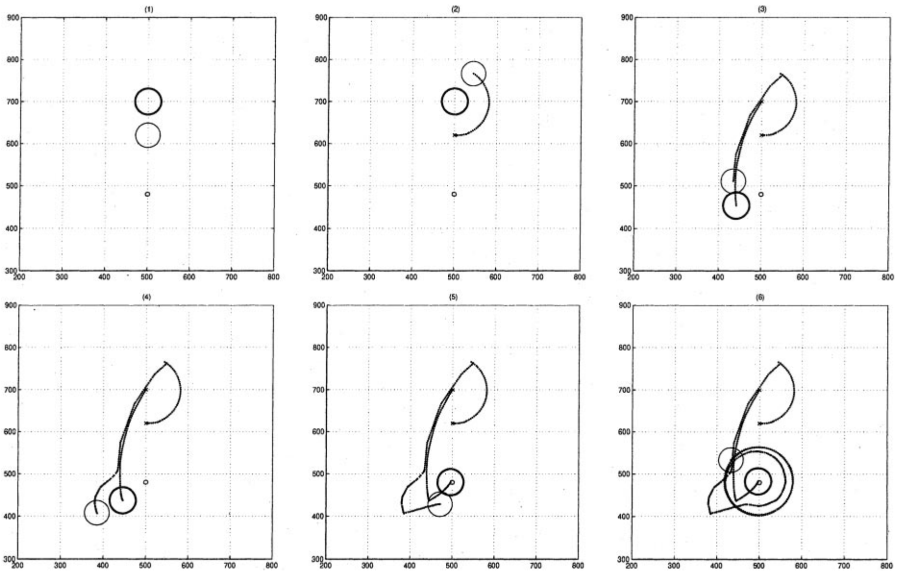

Fig. 11. The behavior sequence of the robot during a typical test: (1) the initial positions of the box (dark circle), the light (smallest circle) and the robot; (2) the robot moved along the side of the box; (3) it pushed the box forward; (4) it then circled again to an appropriate position; (5) it pushed the box again to the goal position; (6) and it continuously circled the box after the box has been pushed to the goal position.

and two primitives) as a building block, and then evolve a new arbitrator to combine this building block and the *exploration* controller to achieve the overall task. In order to generate proper output sequence to coordinate two control systems involved, this arbitrator needs the perceptual information to recognize the appearance of the box. Therefore, the box recognizer BRs and the numerical thresholds were defined as terminals to evolve the desired arbitrator. As in the above section, this arbitrator was to coordinate two controllers, so it was designated to have one output: if the output was 0, the controller for *exploration* was activated, otherwise the controller for *push-box-toward-light* was activated. Again, the controllers to be combined were frozen and only the arbitrator was evolved.

This task is to train the robot to push the box as close as possible to the specified position after it detects the box, so the fitness function can be described as to accumulate the distance between the box and the goal position at each time step, starting from the time when the the robot finds the box. Hence, the fitness function was defined as

$$f = \sum_{t=k+1}^{k+T} D_{b,l}(t),$$

in which $D_{b,l}(t)$ is the distance between the box and the goal position at time t, k the time when the robot finds the box and T is the number of time steps for fitness measurement; each time step was a complete cycle from the highest-level arbitrator to the lowest-level primitive.

The typical behavior of the robot, when executing the whole control system, is shown in Fig. 12. From these figures, we can see that the arbitrator first kept activating the controller *exploration* to drive the robot to explore the given environment and to avoid the walls. Once the robot found the box, the arbitrator began to activate the other controller, *push-box-toward-light*, according to the sensory stimuli. Since the arbitrator was able to activate this control

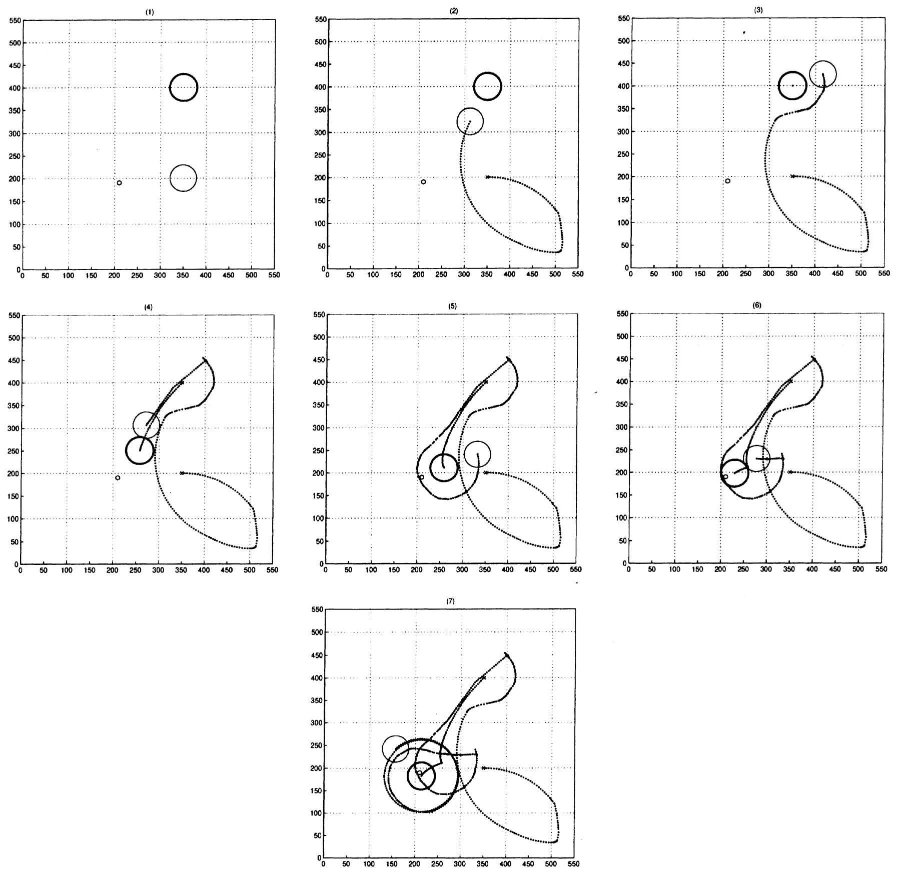

Fig. 12. The behavior sequence of the robot: (1) the initial conditions; (2) the robot wandered around the environment to look for the box; (3)–(7) the robot continuously performed the building block controller *push-box-toward-light* to achieve the task after it found the box.

block continuously after the robot had found the box, the overall task was then achieved successfully.

6.6. Evolving arbitrators vs. evolving monolithic controllers

In the above experiments, we have shown that by taking an explicitly distributed control architecture and then evolving behavior primitives and arbitrators, one can synthesize control systems to achieve complicated task without making too much effort. In order to evaluate the proposed approach more objectively, we had also used our GP system to evolve a monolithic controller for the same overall task and then compared the performance of the two different methods.

For the experiments of evolving monolithic controllers, we defined IRs, LDRs, BRs and the numerical thresholds as terminals. The fitness function and other experimental parameters were exactly the same as those in evolving arbitrators. A successful controller was expected to integrate different sensory information to achieve the target task.

Fig. 13 compares the fitness curves of the best individual fitness at each generation for the different strategies, in which values are averaged over 10 runs. The upper fitness curve represents the result of evolving monolithic controllers; it indicates that the task can not be accomplished by the method of evolving controller as a whole from the scratch. On the contrary, the lower fitness curves in Fig. 13, which are the results of evolving arbitrators for the *pushing-box-toward-light* task and the overall task, clearly show that our

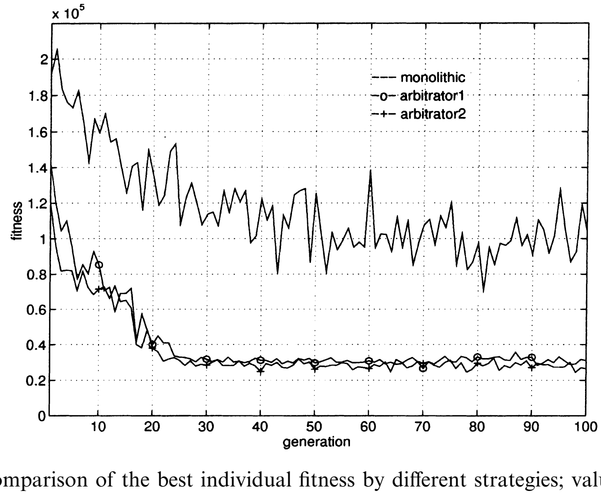

Fig. 13. The comparison of the best individual fitness by different strategies; values are averaged over 10 runs. In this figure, arbitrators1 and 2 are the results of evolving arbitrators for the *push-box-toward-light* task and the overall task, respectively.

approach can efficiently and consistently evolve control systems to achieve the target task (the typical robot behaviors have been illustrated in Figs. 11 and 12).

7. Conclusion

In this paper, we have indicated the challenges of extending the behavior-based approach to design control systems and have analyzed, from different points of view, the main difficulties in applying evolutionary techniques to synthesize robot controllers for complicated tasks. None of them are trivials, nor are they easy to achieve. In order to reduce the load of robot programmers, and to evolve distributed control systems efficiently, we proposed our approach at the intermediate level. Instead of programming all components for a behavior-based robot or evolving the overall control system as a whole, we suggest to perform task decomposition in which a behavior-based architecture is adopted (and a target task is decomposed to fit that architecture), and to evolve separate behavior primitives and arbitrators for coordination. This allows the robot control systems for more complicated skills to be evolved in an incremental way. As is indicated, the use of evolving arbitrators to coordinate lower-level controllers for complex tasks makes the job of defining fitness functions more straightforward and simple, and the tasks easier to achieve. Moreover, the resulting control systems can be explicitly distributed, understandable to the system designer, easy to maintain, and perform like a behavior-based system.

To assess the proposed approach, we have employed it to evolve control systems to achieve a moderately complex task, in which a robot has to explore a given environment to find a box without bumping the walls, and then push the box to a goal position indicated by a light source. Experimental results have shown that our approach can reliably evolve control systems for the target task. In addition, we have also conducted experiments to evolve a monolithic control system for the same task, to compare the performance of two different approaches. The results indicate that because of the difficulties described, the target cannot be achieved successfully by the latter approach.

From the point of view of employing the evolutionary technique to evolve a single control module, our GP system has some advantages, because of the genetic representation designed. It has the feature of operating variable-size genotype as other GP-based work, but does not have the disadvantage of the need of a large population size. In fact, we have shown that our GP system can evolve reliable and robust behavior controllers within a relatively small number of generations, by a relatively small number of population size (In [10,11], a population size of a few thousands was needed to evolve controllers for even simpler tasks.) Also, the logic representation means our system is computationally cheap, and its characteristic of low output sensitivity to input noise makes the result robust, subject to the unreliable sensors and motors in the real world. These features are all very important in evolving robot controllers.

Our work presented here points to some prospects of further research. One is to use the proposed approach to evolve control systems for various tasks and even far more difficult tasks to examine its generality. In particular, it will be worthwhile to extend our system to evolve controllers for sequential tasks which involve internal states in the control mechanisms. For this, it is necessary to introduce some memory components, such as flip-flops, into our circuit trees to participate the evolution. Another direction is to investigate how to integrate the system presented in this paper, with our previous work which explores the co-evolution of robot controllers and physical structures [24], to synthesize complete autonomous robots automatically.

References

[1] R.A. Brooks, A robot that walks: emergent behaviors from a carefully evolved network, Neural Computation 1 (2) (1989) 365–382.

[2] R. Pfeifer, C. Scheier, Sensory–motor coordination: the metaphor and beyond, Robotics and Autonomous Systems (1997).

[3] L. Steels, Building agents out of autonomous behavior systems, in: L. Steels, R. Brooks (Eds.), The Artificial Life Route to Artificial Intelligence, Erlbaum (Lawrence), London, 1993.

[4] R.A. Brooks, Challenges for complete creature architectures, in: From animals to animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior, 1991, pp. 434–443.

[5] R.A. Brooks, Artificial life and real robots, in: Proceedings of the First European Conference on Artificial Life, 1992, pp. 3–11.

[6] D. Cliff, I. Harvey, P. Husbands, Explorations in evolutionary robotics, Adaptive Behavior 2 (1) (1993) 73–110.

[7] M. Dorigo, M. Colombetti, Robot shaping: developing autonomous agents through learning, Artificial Intelligence 71 (2) (1994) 321–370.

[8] D. Floreano, F. Mondada, Evolution of homing and navigation in a real robot, IEEE Transactions on Systems, Man and Cybernetics 26 (3) (1996) 396–407.

[9] T. Gomi, A. Griffith, Evolutionary robotics – an Overview, in: Proceedings of IEEE International Conference on Evolutionary Computation, 1996, pp. 40–49.

[10] J.R. Koza, J.P. Rice, Automatic programming of robots using genetic programming, in: Proceedings of AAAI-92, 1992, pp. 194–201.

[11] C.W. Reynolds, Evolution of corrider following behavior in a noisy world, in: From animals to animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior, 1994, pp. 402–410.

[12] S.J. Ross, J.M. Daida, C.M. Doan, T.F. Bersano-Begey, J.J. McClain, Variations in evolution of subsumption architectures using genetic programming: the wall following robot revisited, in: Genetic Programming: Proceedings of the First Annual Conference, 1996.

[13] J.R. Koza, Genetic Programming: on the Programming of Computers by Means of Natural Selection, MIT Press, Cambridge, MA, 1992.

[14] S. Nolfi, Using emergent modularity to develop control system for mobile robots, Adaptive Behavior (1997).

[15] R.A. Brooks, A robust layered control system for a mobile robot, IEEE Journal of Robots and Automation RA-2 (1) (1986) 14–23.

[16] M. Colombetti, M. Dorigo, G. Borghi, Behavior analysis and training: a methodology for behavior engineering, IEEE Transactions on Systems, Man, and Cybernetics 26 (6) (1996) 365–380.

[17] L.-J Lin, Scaling up reinforcement learning for robot control, in: Proceedings of International Conference on Machine Learning, 1992, pp. 182–189.

[18] M. Mataric, Reward functions for accelerated learning, in: Proceedings of International Conference on Machine Learning, 1994, pp. 181–189.

[19] R.J. Firby, Task networks for controlling continuous processes, in: Proceedings of the Second International Conference on AI Planning Systems, 1994, pp. 49–54.

[20] E. Gat, Robot navigation by conditional sequencing, in: Proceedings of IEEE International Conference on Robotics and Automation, 1994, pp. 1293–1299.

[21] P. Agre, D. Chapman, Pengi: an implementation of a theory of activity, in: Proceedings of AAAI-87, 1987, pp. 268–272.

[22] S.J. Rosenschein, L.P. Kaelbling, A situated view of representation and control, Artificial Intelligence 73 (1995) 149–174.

[23] S.J. Rosenschein, L.P. Kaelbling, The synthesis of digital machines with provable epistemic properties, in: Proceedings of Conference on Theoretical Aspects of Reasoning about Knowledge, 1986, pp. 83–98.

[24] W.-P. Lee, J. Hallam, H.H. Lund, A hybrid GP/GA approach for co-evolving controllers and robot bodies to achieve fitness-specified tasks, in: Proceedings of IEEE International Conference on Evolutionary Computation, 1996, pp. 384–389.

[25] R. Tanese, Distributed genetic algorithms, in: Proceedings of the Third International Conference on Genetic Algorithms, 1989, pp. 434–439.

[26] F. Mondada, E. Franzi, P. Ienne, Mobile robot miniaturation: a tool for investigation in control algorithms, in: Proceedings of the Third International Symposium on Experimental Robotics, 1993.

[27] W.-P. Lee, J. Hallam, H.H. Lund, Applying GP to evolve behavior primitives and arbitrators for mobile robots, in: Proceedings of IEEE International Conference on Evolutionary Computation, 1997, pp. 501–506.