Chapter X

Evolving Behavior Coordination for Mobile Robots using Distributed Finite-State Automata

Pavel Petrovič
Department of Applied Informatics, Comenius University
Slovakia

1. Introduction

This chapter describes a finite-state automata approach to Evolutionary Robotics (ER) (Petrovic, 2007). Most of the efforts in the field of ER, see (Nolfi and Floreano, 2000) for very representative examples concentrate around controllers with various neural architectures, mainly due to their high plasticity and adaptability. Despite these obvious advantages, the resulting controllers usually have monolithic form with information represented numerically across the whole networks, where it is difficult to understand and explain why particular actions are taken in particular situations, enumerate the possible types of behaviors the controllers may produce, and achieve modularity required in more complex tasks. These features may be needed when the field reaches the level of real-world applications, for example to automatically verify the safety. In addition, many robotics researchers have demonstrated and reasoned that modular controller architectures with simultaneously executing behavioral modules – often called Behavior-Based (BB) architectures, are favorable as compared to centralized and top-down architectures. In our quarter of the ER community, the ultimate goal of the efforts is to find useful methods for automatic programming of mobile robots performing non-trivial tasks. By non-trivial, we mean tasks for which manual controller design by an engineer is difficult, or applications where the engineer is not available at the time of controller design or adaptation. Modular neural networks are an interesting field of interest, however, the research results did not satisfy us in their ability to compensate for their shortcomings. We suggest to use a BB architecture and apply Evolutionary Computation (EC) to learn/design the coordination mechanism of the behaviors. Although it would be possible to use EC to evolve both the behaviors and the coordination, evolving simple behaviors have been addressed by many previous works, and thus this time, we concentrate at the coordination mechanisms of a set of pre-programmed (pre-evolved) behaviors. When searching for a suitable formalism for representing the coordination mechanisms, we found that the robot behavior can be veritably modelled by finite-state automata (FSA). We also found that EC have been previously successfully applied to automatic programming of state automata, a detailed overview is in the section 3. As explained in section 4, we have designed a set of experiments, which indicate that for tasks that share properties with BB coordination

mechanisms, the finite state representations outperform more conventional GP-tree representations. With this background, we have performed experiments with evolving behavior coordination based on FSA for less trivial tasks for mobile robots and we have shown feasibility of this approach as described in detail in section 5.

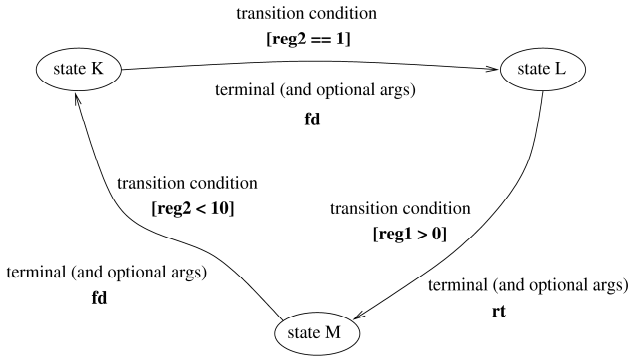

Fig. 1. FSA representation. Transition conditions correspond to GP-tree non-terminals, however each transition also has an associated terminal. The state transitions can point to the originating state, and multiple transitions between the same two nodes, or with the same transition conditions are allowed (only one is executed when transition occurs).

2. Background and Motivation

There have been only few attempts to generate coordination mechanisms automatically based on the required task or robot purpose. For example, (Koza, 1992) evolves a robot controller based on Subsumption Architecture (SA) for wall-following task. He writes: "*The fact that it is possible to evolve a SA to solve a particular problem suggests that this approach to decomposing problems may be useful in building up solutions to difficult problems by aggregating task achieving behaviors until the problem is solved.*" Evolution of SA has been recently addressed by a thesis of (Togelius, 2003). The usefulness of these approaches can be supported by the following reasons:

• They are *inovating*: Automatic method might explore unforeseen solutions that would otherwise be omitted by standard engineering approaches used in manual or semi-automatic design performed by a human.

• They can provide robot controllers with *higher flexibility*: Mobile robots can be built for general purpose, and the coordination mechanism for different achievable tasks might have to be different, either for reasons of critical limits on their efficiency, the bounds on the controller capacity, or conflicting roles in different tasks. In such a case, generating the coordination mechanism based on task description might be required. Having the option of automatic coordination generation might save extensive amounts of work needed to hand-craft each coordination mechanism.

• They can *cope with complexity*: Manual coordination design might suffer from the lack of understanding of the real detailed interactions of the robot with its environment. These interactions might be difficult to describe analytically due to their complexity. Automatic coordination design might capture the undergoing characteristics of the robot interactions more reliably, efficiently and precisely.

A valuable inspiration for our work stems from the work of (Lee et al., 1998), where the more complex, high-level task is decomposed hierarchically and manually into several low-level simple tasks, which can further be decomposed to lower-level tasks. The reactive controller consists of primitive behaviors at lowest level and behavior coordination at higher levels, both with the same architecture of interconnected logic-gate circuit networks. The evolution proceeds from the lowest level tasks up the hierarchy to the target complex task. The work has been continued on by the master thesis (Ostergaard, 2000), where a football player has been evolved with the use of co-evolution. The qualitative change is in the ability to work with internal state (as contrasted to purely-reactive controllers), and more complex architecture allowing two types of coordination: 1) a sequence of several states, and 2) selecting the module with the highest activation value; where the activation value is exponentially decreasing over time, and reset to maximum on request of the module.

3. FSA as Representation for Evolutionary Algorithms

FSA or FSM (we use these terms interchangeably) have been used as genotype representation in various works, although this representation lies on the outskirts of the evolutionary algorithms research and applications. Let us review the approaches first.

Evolutionary Programming, (Fogel, 1962, 1993) is a distinguished evolutionary approach that originally used FSA as the genotype representation. EP does not utilize recombination operators, and relies on mutations. The original EP works addressed the tasks of prediction, identification and control.

(Chellapilla and Czarnecki, 1999) introduce modular FSM subclass with restricted topology – in particular, the FSM are partitioned into several encapsulated sub-parts (sub-FSM), which can be entered exclusively through their starting states. The authors use modular FSM to evolve controllers for the artificial ant problem that was previously successfully solved by evolving binary-string encoded FSA in (Jefferson et al., 1992). They provide evidence that modular FSM perform better on this task than non-modular FSM, and they also provide evidence that direct encoding with structural mutations of non-modular FSM perform better than binary-string encoding used in (Jefferson et al., 1992). This idea of modular FSM has been adopted also by (Acras and Vergilio, 2004), who develop a universal framework for modular EP experiments and demonstrate its use on two examples.

(Angeline and Pollack, 1993) are experimenting with automatic modular emergence of FSA. They suggest to freeze and release parts of the FSA so that the frozen (or "compressed") parts cannot be affected by the evolutionary operators. The compression occurs randomly and due to the natural selection process, it is expected that those individuals where the compression occurs for the correctly evolved sub-modules will perform better and thus compression process interacts with the evolutionary process in mutually beneficial manner. Indeed, the authors document on the artificial ant problem that the runs with compression performed better than equivalent runs without compression. They reason: "*An explanation for these results is that the freezing process identifies and protects components that are important to the viability of the offspring. Subsequent mutations are forced to alter only less crucial components in the representation.*" (Lucas, 2003) is evolving finite-state transducers (FST), which are FSA that generate outputs, in particular, map strings in the input domain with strings in the output domain. FST for transforming 4-chain to 8-chain image chain codes were evolved in

this work, while three different fitness measures for comparing generated strings were used: strict equality, hamming distance and edit distance.

An interesting piece of work by (Frey and Leugering, 2001) considers FSM as controllers for several 2D benchmark functions and the artificial ant problem. In their representation, the whole transition function is represented as a single strongly typed GP-tree – i.e. a branching expression with conditions in the nodes that direct execution either to the left or to the right sub-tree, and finally arriving to a set of leaves that list the legal transition pairs (old state, new state).

In his PhD thesis, (Hsiao, 1997) is using evolved FSA to generate input sequences for digital circuits with the purpose of their verification, and fault detection. The author achieves best fault detection rate on various circuits (as compared to other approaches), except of those that require specific and often long sequences for fault activation.

(Horihan and Lu, 2004) are evolving FSM to accept words generated by a regular grammar. They use an incremental approach, where they first evolve FSM for simpler grammars, and gradually progress to more complex grammars. They use the term genetic inference to refer to their approach of generating such solution.

(Clelland and Newlands, 1994) are using EP with probabilistic FSA (PFSA) in order to identify regularities in the input data. The PFSA is a FSA, where the transitions are associated with probabilities as measured on input sequences. The EP is responsible for generating the topology of the FSA – number of states and how they are interconnected, and the transitions in PFSA are labeled according to their "fire rate". This combination can be applied for rapid understanding of an internal structure of sequences.

(Ashlock et al., 2002) are evolving FSM to classify DNA primers as good and bad in simulated DNA amplification process. They evolve machines with 64 states in 600 generations. They used the weighted count of correct/incorrect classifications as their fitness function. However, they sum the classifications made in each state of FSM throughout its whole run. They argue that if only the classifications made in the final state were taken into account, the performance was poor. In addition, this allows the machine to produce weighted classification – how good/bad the classified primer is. The best of 100 resulting FSM had success rate of classification of about 70%. Hybridization, i.e. seeding $1/6^{th}$ of a population of an extra evolutionary run with the best individuals from 100 previous evolutionary runs improved the result to about 77%. This work was continued in (Ashlock et al., 2004), where the FSM approach was compared to more conventional Interpolated Markov Models (IMMs), which outperformed FSM significantly.

In an inspiring study from AI Center of the Naval Research Laboratory, (Spears and Gordon, 2000) analyze evolution of navigational strategies for the game of competition for resources. The strategies are represented as FSM. Agent moves on a 2D grid while capturing the free cells. Another agent with a fixed, but stochastic strategy is capturing cells at the same time, and the game is over when there are no more cells to capture. Agents cannot leave their own territory. Authors find that the task is vulnerable to cyclic behavior that is ubiquitous in FSM, and therefore implement particular run-time checking to detect and avoid cycles. They experiment with the possibility to disable and again re-enable states (as contrasted to permanent and complete state deletion). They also compare evolution of machines with fixed number of states and evolution of machines, where the number of states changes throughout the evolutionary run. They discover that in the case of varying number, the machines utilize the lately-added states to lesser extent, as well as that deleting

states is too dramatic for performance, and thus suggest merging or splitting states instead of deleting and creating states. Due to the stochastic algorithm of the opponent agent in the game of competition for resources, the fitness function must evaluate each individual in many different games (G). Authors disagree with others who claim that keeping G low can be well compensated by higher number of generations and conclude that it results in unacceptable sampling error. The authors therefore evaluate all the individuals on fewer games (500), and if the individual should outperform the previous best individual, they re-evaluate it on many more (10000) games. Some further FSM-relevant references can be found for example in the EP sections in the GECCO and CEC conferences.

4. Analysis of FSA vs. GP-tree Representations

Each robot performing some activity is always in some state while it reactively responds with immediate actions or it proceeds to other states also as a response to environmental percepts – thus the activity of a robotic agent can be modeled by a state diagram accurately. We believe that state-diagram formalisms can in fact steer controllers themselves and be the back-bone of their internal architecture. Secondly, we believe that the state automata are easier to understand, analyze, and verify than other representations, for example neural networks. Thirdly, we believe that state automata are more amenable to incremental construction of the controller, because adding new functionality involves adding new states and transitions, and making relatively small changes to the previous states and transitions.

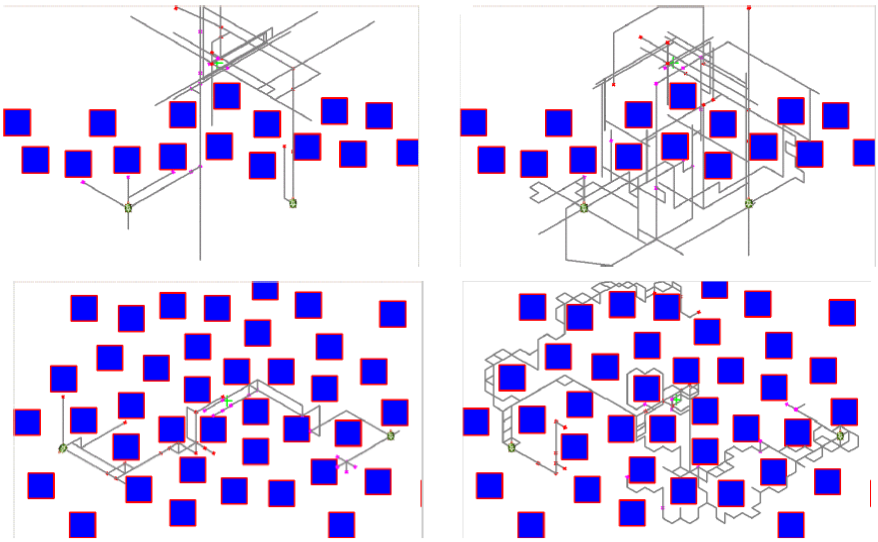

Fig. 2. The best trajectories from all generations of one evolutionary run, two starting locations. GP-tree on the left, FSA on the right. The topology of the trajectories follows the topology of the representation: FSA are better at encoding loops and strategies, GP-trees are better at searching and extending the trajectories similar to the iterative deepening algorithm.

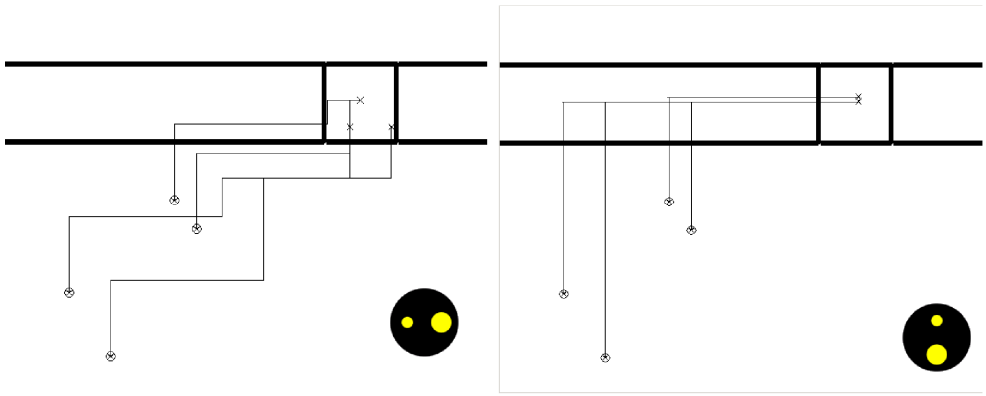

Fig. 3. Trajectories for the evolved individuals in *selected* runs for both representations. The four starting locations are marked by a small cross in a circle and the final positions are marked by a small cross. Individuals in both representations utilize the light sensor, however, the FSA solution is cleaner – moving straight in the middle between the two horizontal lines, turning right and finding the target square. The GP-tree representation is approaching the target in a stair-like movement and faces difficulties to align with the target location correctly when the sensory experiences in the final segments of the trajectories vary. (The robot was moved to the bottom right just to disclose the trajectories.)

In order to better understand the character of the FSA representation, we have devised a series of tasks and compared the performance of GP with program-tree representation against FSA representation, see Figure 1. In our implementation of GP-tree representation, the evolved programs are binary trees, with terminals (actions) in the leaves and non-terminals (control structures *if*, *while*, *repeat*, *seq*) that utilize binary relations in the nodes. Both terminals and non-terminals can require arguments, which come in form of registers that can be mapped to sensor inputs or internal state variables. In our implementation of the FSA representation, the automaton consists of N states, which are connected by ordered set of state-transitions associated both with a transition condition and an action. The first transition with a satisfying condition is performed and an associated action is executed. For details of the formalisms and experiments, please refer to (Petrovic, 2007). In this section, we review the most interesting points.

We have experimented with two simple 2D robot navigation tasks and several symbolic sequence tasks as follows: In the first task, robot was to navigate as close as possible to a specific target location (the distance determined the fitness of the solution), while avoiding collisions with obstacles (collisions subtracted from the fitness), see Figure 2. A binary sensor indicated whether the robot is heading roughly in the direction of the target. Even though there was only slight difference in the performance of both representations, inspecting the actual strategies reveals that GP program trees are better at incrementally extending the trajectories segment by segment in an iterative search process, while the FSA solutions are effective at encoding strategies. For instance, the solutions evolved in earlier generations shown at bottom right of Figure 2 avoid the obstacles always from the left-hand side. That brings the robot quite far from the target, but still closer than the starting location,

however, it is a strategy that can be further modified to a correct solution. FSA are better at encoding loops, because the state transitions allow easy creation of arbitraty loop structures, while the sequential depth-first exectution of the program trees is very prohibitive. The topology of the trajectories evolved in the consecutive generations reflects the topology of the representation itself: notice the tree structure of the trajectories of the GP program trees and the loop structures of the FSA. Similar effects were found in results from a docking task, where a robot was placed in a bottom-left quadrant of a 2D environment and was required to park into a square. While the GP program trees achieved slightly higher target accuracy on average, the best evolved FSA solutions have a cleaner strategy, see Figure 3, and Figure 4 right.

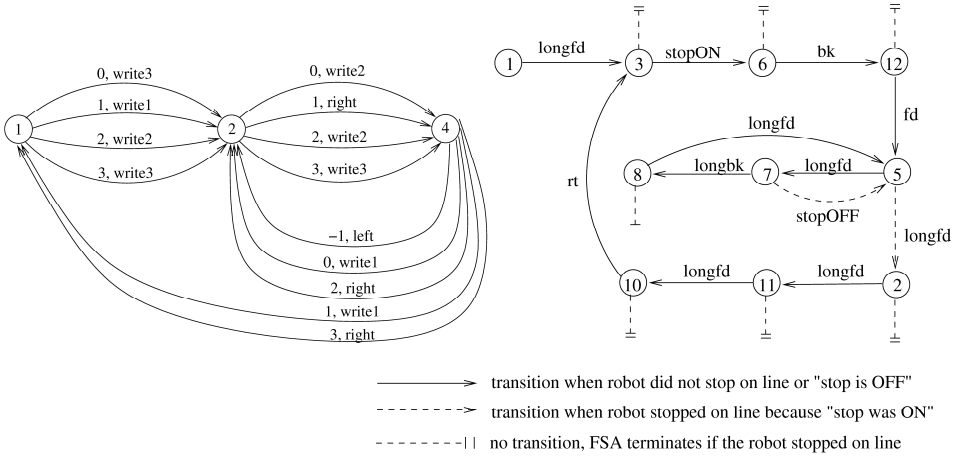

Fig. 4. *On the left*: The best evolved FSA in the switch task with three symbols. *On the right*: FSA that evolved in the docking task. The robot first moves forward in the loop of states 5-7-8 until it arrives to line, then it moves forward three times (states 2-11-10), turns right, and proceeds again until it stops at line (the left line of the target square). Next, it moves into the square (states 2-11-10 again), turns right facing now down, and finally the FSA terminates when the robot attempts to move back in the state 6 when it encounters a line (top line of the target square as the robot is facing down).

In the symbolic task $(abcd)^n$, a tape machine was expected to replace a specified part of the input/output tape with repeating pattern of symbols *abcd*. We were surprised to find that GP program trees evolved solutions much faster, see figure 5. The following objective function was used:

$$\text{fitness} = 1 - s \cdot q_s - \sum_{i=1}^{n_{starts}} \left(e_i / (l_i \cdot n_{starts}) - r_i \cdot q_r \right) \quad (1)$$

Where e_i is the number of incorrect symbols in the output word (including extra placed or missing symbols) in the i^{th} input word, l_i is the number of symbols in the input word, n_{starts} is the number of random input words presented to the program, r_i is the number of execution

steps, s is the size of the genotype, q_s and q_r are weight constants. Within 2000 generations, 74% of FSA runs evolved a correct solution, while 92% of GP-tree runs succeeded. The remaining 8% of GP-tree runs placed only 2 symbols incorrectly, while the incorrect FSA erred on 7-14 symbols. We found that GP performed so well due to its strong *while* non-terminal and the fixed structure of the loop required.

In the second symbolic task (labelled *switch*), the performance of FSA appeared to clearly outperform the GP program trees. In this task, the tape machine is to replace zeros in the input string with a closest non-zero symbol found on the tape in the direction to the left. For example, the sequence

```
10004030000200013004000000003000020
```

should be transformed into:

```
11114433333222213334444444443333322
```

with the same objective function as in the previous task. In this case, the evolution of GP program trees was much slower than FSA, see Figure 6.

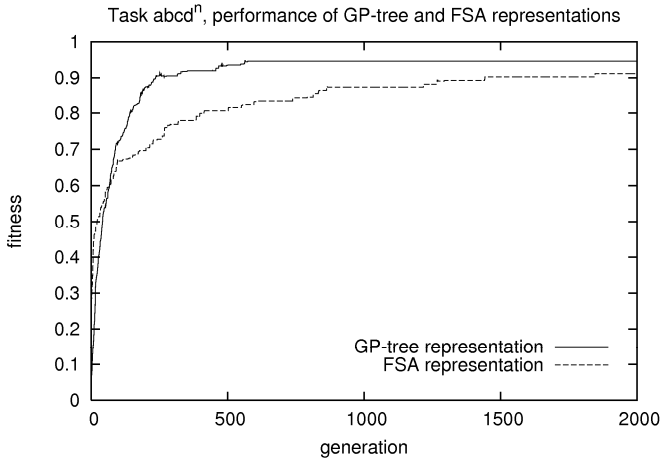

Fig. 5. Performance of the FSA and GP-tree representations on the task $(abcd)^n$. Average of 25 (GP-tree) and 23 (FSA) runs. The programs were presented with input word containing 32 ones, and had 150 execution steps for writing the output word. Population size 300, prob. crossover 0.6, prob. mutation 0.7, brooding crossover (number of non-strict broods 3, 30% training samples used for brooding), combining crossover (GP-trees): 0.25, 15 strict-elite individuals, tournament seletion (size 4, probability 0.8), max. GP-tree depth: 12, max. number of FSA states/transitions per state: 22/10, FSA shuffle mutation: 0.4.

The task was a difficult one, and only 3 out of 10 FSA runs found a solution, therefore we ran experiments with a simpler version of the same task containing only three types non-zero symbols. Within 2000 generations, all runs with the FSA representation evolved a correct solution, while no runs with the GP-tree representation found a correct solution, see Figure 7 and Figure 4 left for the smallest evolved FSA after prunning redundant transitions and state. The task can be interpreted from the robotic point of view as follows: the robot is always in one of the four (or three) states – it performs an activity of writing a specific

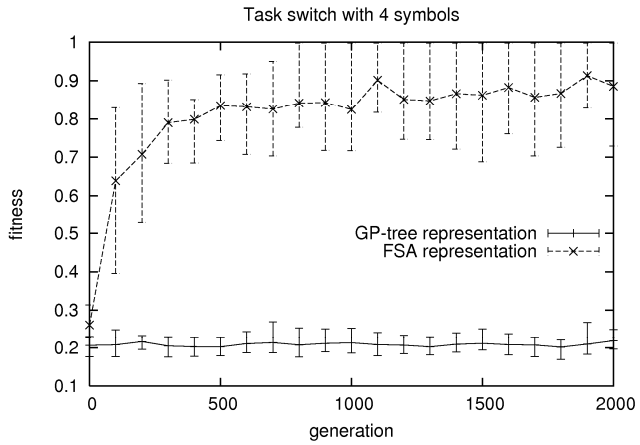

Fig. 6. Average of the best fitness from 14 (GP) or 10 (FSA) runs of the switch task with four symbols. Population size 300, probability of crossover 0.5, crossover brooding of size 3, with 30% test cases used to evaluate brooding individuals, probability of mutation 0.9, 15 elites, each individual evaluated on 10 random strings, uniformly random input word length 10-60, maximum 10 continuous 0-symbols, maximum number of GP-tree or FSA execution steps 300, FSA: pshuffle=0.4, number of states 1-15, number of transitions 1-15, pcross_combine=0.25, maximum tree depth=15, The number of evaluations is proportional to the generation number. The error bars show the range of fitness progress in all runs. Tournament selection (FSA), fitness-proportionate selection (GP-trees).

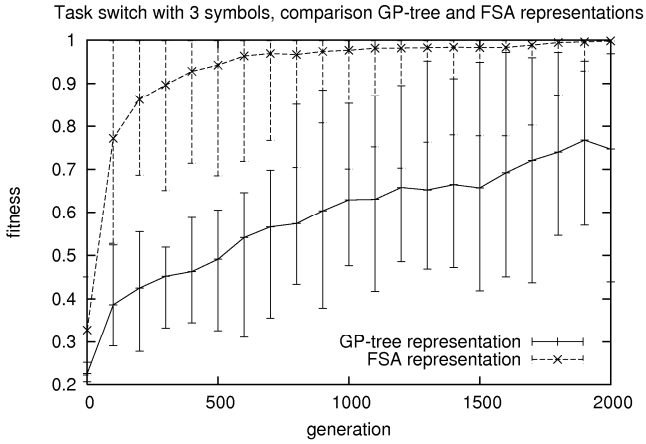

Fig. 7. Average of the best fitness from 15 (GP) or 34 (FSA) runs of the switch task with three symbols, tournament selection, and other parameters as in the four-symbol experiment. The final fitness of all FSA runs outperforms the final fitness of the best GP-tree run.

symbol and navigating right on the tape. When a specific event in the environment occurs, it switches to a different state, performing a different activity (writing a different symbol). Whatever the current activity, it is capable to switch to any other possible state in a reactive manner to provide a required response. This is exactly the kind of interaction that is typical for robotic tasks, and behavior coordination, where the states of the behavioral modules switch according to the task and environmental context.

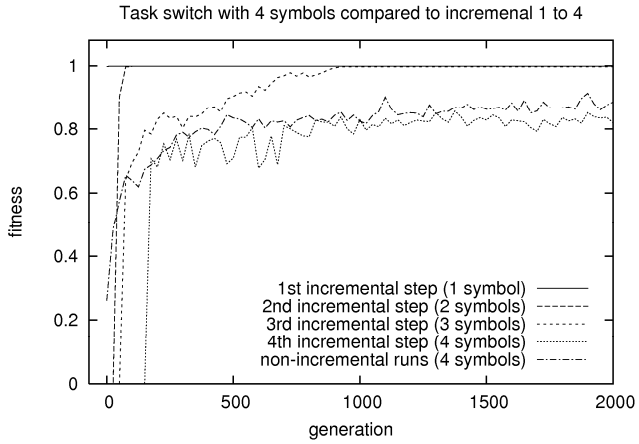

Fig. 8. Average of the best fitness from 11 (incremental) and 10 (non-incremental) runs of a 4-symbol switch task, FSA representation and tournament selection, parameters as in fig. 7.

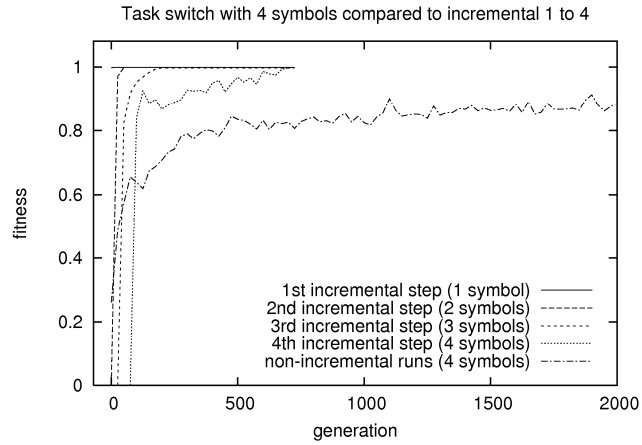

Fig. 9. Average of the best fitness from 10 runs of the four-symbol switch task with FSA representation and tournament selection. Individuals were frozen at the end of each step. Other parameters remained the same.

In further experiments with the switch task and FSA representation, we were interested to learn whether the evolution can find the solution faster, if supported by an incremental framework. When searching for a solution to the four-symbol task, we have first required the evolutionary algorithm to find a solution to the three-symbol version task and then introduced the input words containing the fourth symbol. A new action allowing to write symbol 4 was also added in the second incremental step. As shown at Figure 8, the performance of the evolutionary algorithm deteriorated in the case of the extra incremental step. Why is it that the incremental evolution failed in this case? At the first glance, the simplified three-symbol version of the task when preceding the fourth-symbol task should have made it easier to solve the task, because it prevented the fourh symbol from appearing and thus did not require the state machine to react correctly to all the transitions from the states when writing the fourth symbol while evolving the other states. Finding a correct sub-solution to a three-symbol task should have been much faster, and that solution could have been easily extended to the four-symbol task as it is a subset. However, requiring the evolution of four-symbol solution to pass through a correct and complete three-symbol solution is a big burden! There are many possible paths from a random solution to a complete four-symbol solution, which do not pass through complete three-symbol solution. Requiring the evolution to pass through a particular point in the search space is a strong bias, which we call *incremental bias*. In cases when the advantages of the incremental setup outweigh the disadvantage of the incremental bias, the incremental evolution improves evolvability and convergence rate. Its application must be judged with respect to this general tradeoff principle.

We have attempted to construct a different scenario, which would be beneficial. Dividing the task to four steps with 1, 2, 3, and 4 symbols consecutively would have the same outcome as described above. However, freezing the best evolved automaton at the end of each incremental step, and letting the evolution in the further steps only to add new states and transitions simplifies the task greatly thus reaching the correct solution using the incremental scenario much faster as shown at Figure 9. Even higher speed-up was achieved when the terminal set in the later incremental steps was restricted, see (Petrovic, 2007).

The preliminary study covered in this section suggests that:

1) FSA share the structure with robotic tasks. They may be suitable for their modelling and implementation, especially when used to automatically program behavior coordination in BB controllers.

2) FSA and GP program-trees are two different representations, which both outperform each other on different tasks and are complementary in their features.

3) The incremental evolution can be beneficial for the performance of evolutionary algorithm only if its advantages outweigh the disadvantages resulting from the incremental bias of the evolutionary search.

5. Evolving Behavior Coordination

The goal of this work is to design controllers for mobile robots automatically by means of artificial evolution. We take the assumption that the hardware details of sensors and actuators are quite specific and the low level interactions of the controller with the robot hardware can be implemented efficiently and without much effort manually, before the target task is known: the behavior modules can be written in any available language or

formalism manually. However, they can even be evolved automatically, if suitable. The part of the controller that we aim to design automatically here is the behavior coordination mechanism, in particular, a set of finite-state automata (FSA).

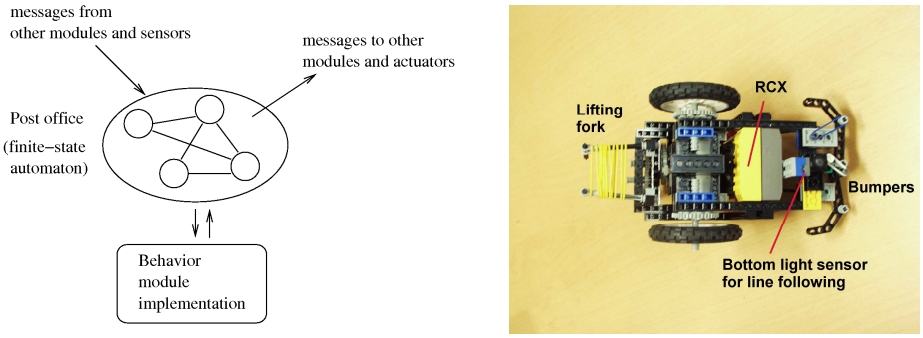

Fig.10. *On the left*: Single behavior module with its coordinating post-office finite-state automaton. *On the right*: bottom view of the robot. The upper light-sensor is installed on top side of the robot, and the lifting fork is operated by an extra motor and covered with rubber bands to increase friction.

5.1 Controller Architecture and Evolutionary Algorithm

Let us explain the controller architecture on a simplified artificial example of a mouse robot acquiring a piece of cheese from a room and bringing it back to its mouse hole. The controller is shown at Figure 11. The mouse explores the room in random movements, and whenever it smells or sees the cheese, it moves in the direction of the smell, grasps the food, and drags it back to its home.

The core of the controller is formed by several modules, which are implementations of simple competencies (grayed boxes). These competences alone do not give the robot any purpose or any intelligent behavior yet. While performing their specific activity, they simply react to a predefined message interface and produce status messages whenever their actions or the incoming message of interest produce or indicate a significant outcome. For example, the "locating cheese" competence receives inputs from the vision and smelling sensor and produces a desired direction of movement, if the cheese is detected. Whenever the cheese is detected, it reports the event by an outgoing message. The competencies might be provided by the robot builders, or programmed in any programming language. Alternately, they can be hand-designed or evolved finite-state automata, GP program trees, or neural networks. The architecture does not limit their internal architecture. Most of the competencies have their own thread of execution. The competencies might be understood as an "operating system" of the robot that provides higher-level interface for controlling the low-level robot hardware. The intelligence and a particular purpose of the controller are encoded in a set of post-office modules, at most one post-office for each competence (post-offices are encircled by dashed boundaries at Figure 11). The post-office modules (Figure 10 left), are the communication interfaces of competences with their peers and the remaining parts of the controller: sensors, and actuators. All messages received and sent by a particular competence module pass through its post-office module. The postoffice modules in our

architecture are finite-state machines, but other languages or formalisms could be used in place, as long as it is capable of filtering/transforming the incoming and outgoing messages of the module as needed for the specific robot task. Transitions can optionally result in generating new messages. In this way, the functionality of the competence module is turned on, or off, or regulated in a more advanced way, depending on the current state of the task, environment, and the robot performance represented by the state of the post-office FSA. The post-office simply filters or modifies the messages so that the competence module takes actions that are suitable in a particular situation. For example, the random turning competence will be activated only while the robot is exploring the room and searching for the cheese, or when it accidentally dropped and lost the cheese on its way back. The associated post-office module follows with the events performed by other modules that are relevant, and adjusts its state to represent the current situation. Please refer to the section 5.3 below for another specific example.

We use the standard Genetic Algorithm (inplemented using the GALib from MIT), with our specific initialization, crossover, and mutation operators. In the first stage, the designer prepares individual modules. For each module, he or she specifies the module message interface: the messages the module accepts and the messages it generates. In the second stage, the designer selects the modules for the controller and specifies lists of messages that can trigger incoming and outgoing transitions of the FSA associated with each module. The remaining work is performed by the evolutionary algorithm.

The genotype representation consists of blueprints of FSA for the set of modules for which the FSA are to be designed automatically (some modules might work without post-offices, other might use manually-designed post-offices, or some are held fixed because they are already evolved). An example of a genotype is in Figure 12. The number of states and the number of transitions in each state vary (within specified boundaries). Transitions are triggered by messages (incoming or outgoing) and have the following format:

(*msg type*, *new state*,*msg to send out*, [*msg arguments*], *msg to send in*, [*msg arguments*])

The GA-initialization operator generates random FSA that comply with the supplied specification. The crossover operator works on a single randomly selected FSA. It randomly divides states of the FSA from both parents into two pairs of subsets, and creates two new FSA by gluing the alternative parts together. It is designed so as to preserve as much information from both parents as possible, i.e. preserving and redirecting the state transitions consistently. For details, please refer to (Petrovic, 2007).

The mutation operator works upon a single FSA. One of these operations is performed (probabilities of mutations are parameters of the algorithm): 1) a random transition is created, 2) random transition is deleted, 3) a new state is created (with minimum incoming and outgoing random transitions); in addition, one new transition leading to this state from another state is randomly generated, 4) a random state is deleted as well as all its incident transitions, a random transition is modified: (one of its parts *new state*, *msg type*, *msg to send out*, *msg to send in* is replaced by an allowed random value), 5) a completely random individual is produced (this operator changes all FSA), 6) a random transaction is split in two and new state is created in the middle, 7) the initial state number is changed.

In our experiments, we use the roulette wheel and the tournament selection schemes combined with steady-state or standard GA with elitism. Other parameters of the algorithm include (with these default values): $p_{crossover}$ (0.3), $p_{mutation}$ (0.7), probabilities of mutations: $p_{new\_random\ \ transition}$ (0.25), $p_{delete\_random\_transition}$ (0.1), p_{new_state} (0.2), $p_{random\_state\_deleted}$ (0.05),

$p_{random\_transition\_mutated}$ (0.25), $p_{new\_random\_individual}$ (0.05), $p_{split\_transition}$ (0.05), $p_{change\_starting\_state}$ (0.05); *population_size* (100), *gen number* (60), $p_{population\ replace}$ (0.2), *number of modules in the controller* (10), specification of the message interfaces and trigger messages for all modules, initial and boundary values for *number of states and transitions*, *number of starting locations* for the robot for each evaluation, *timeout* for the robot evaluation run, specification of the *fitness function parameters*, input, output, and log file locations, please find more details in (Petrovic, 2007).

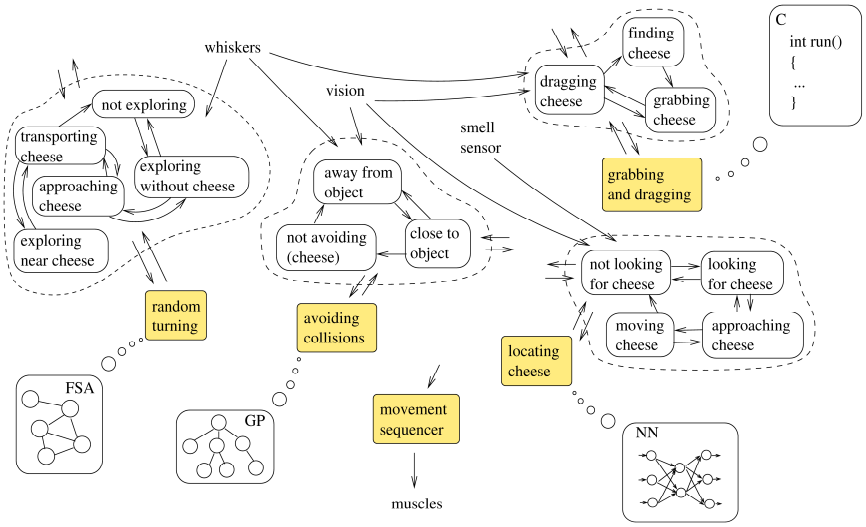

Fig.11. Example controller architecture for the task of a mouse acquiring cheese.

5.2 Incrementality

Researchers observed in the past (Harvey, 1995) that evolving robot behavior is a hard challenge for any EA. The fitness landscape tends to be rough, and evaluation of each individual typically takes a long time. Trying to evolve more complex behaviors is often too difficult. Some groups, such as (Harvey, 1995, Lee et al., 1998) advocated the use of incremental evolution, where the complexity of the target task is decreased by decomposing it to several simpler tasks, which are easy enough to solve by an evolutionary algorithm. We have identified five different ways, in which an evolutionary robotic algorithm can be incremental:

Environment (where is the robot performing?): the earlier incremental steps can be run in a simplified environment, where the frequency and characteristics of percepts of all kinds can be adjusted to make it easier for the robot to perform the task. For instance, the number of obstacles or distance to the target can be reduced, the environment can be made more deterministic, the noise can be suppressed, landmarks can be made more visible, etc. An example of this type of incrementality is (Lund and Miglino, 1998), where box-shaped obstacles were replaced by more difficult U-shaped obstacles after the avoidance behavior for the former was evolved.

Task (what is the robot doing?): the earlier incremental steps can require only part of the target task to be completed, or the robot might be trained to perform an independent simple task, where it learns skills that will be needed to successfully perform in the following tasks.

An example of this type of incrementality is (Harvey, 1995), where the gantry robot evolved forward movement first, followed by stages that required movement towards a large target, movement towards a small target, and distinguishing a triangle from square. For another example, we might first require a football playing robot to approach the ball, later we could require also to approach it from the right direction.

Controller (how is the robot doing it?): the architecture of the controller changes. For example, the final controller might contain many interacting modules, but the individual interactions can be evolved in independent steps, where only the relevant modules are enabled. In the later steps, the behavior might be further tuned to integrate with other modules of the controller. This type of evolutionary incrementality occurs seldom in the literature, but an example could be a finite-state machine-controlled robot negotiating a maze. The controller can be extended with a mapping module that is able to learn the maze topology, however, the output of the module has to be properly integrated with the output of the FSA. Non-evolutionary controller incrementality can certainly be seen in the Subsumption architecture and its flavors (Maes, 1990), and many later BB approaches. The task for the robot might require a complex controller, for example one with an internal state. Evolution can start with a simple controller that is sufficient for initial task and the controller can be extended later during the evolution. The change can be either quantitative, i.e. incrementing the number of nodes in a neural network, or qualitative, i.e. introducing a new set of primitives for a GP-evolved program.

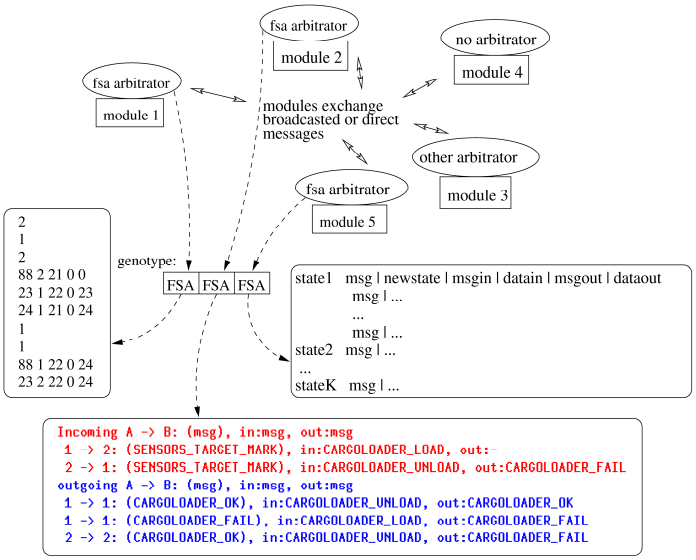

Fig.12. Controller architecture and genotype representation: left oval shows actual numeric genotype representation (it is a vector of numbers containing the number of states in FSA, number of incoming and outgoing transitions in each state, and detailed transition specifications), bottom oval shows a symbolic representation as viewed by a viewer utility (used to analyze the evolved post-offices), right oval shows the genotype structure for both incoming and outgoing messages for better explanation.

Robot sensors/actuators (with what. . . ?) the dimensionality of the search space might be reduced by disabling some of the robot sensors and actuators before they are needed for the task evolved in each particular step. An example of this can be seen in Incremental Robot Shaping of (Urzelai et al., 1998), where the Khepera robot had first evolved the abilities of navigation, obstacle avoidance, and battery recharge, before a gripper was attached to it and the robot had to evolve an additional behavior of collecting objects and releasing them outside of the arena. A subset of robot's equipment can be used in the early steps and more specific sensors and actuators added later.
Robot morphology (what form does the robot have?): the shape and size of the robot can be adjusted to make its performance better and reshaped according to final design in the later incremental steps. This kind of incrementality is also seldom seen in the literature. On the other hand, there are examples where the robot morphology itself is evolved (Lund, 2001). An example of morphological incrementality would be a vacuum-cleaning robot with the shape of an elliptical cylinder that needs to turn in proper direction to pass through narrow passages. It could be simplified to a circular cylinder to evolve basic navigation strategies and later updated to its final shape to achieve the proper target behavior. Another example is evolving the particular target shape layer by layer, if the physics allow.
From the implementation point of view, incrementality can be achieved by modifying the simulated environment, the objective function, the genotype representation and the corresponding controller implementation, and the configuration of the simulated robot. When evolving the target task in multiple steps, we do not necessarily require that the steps form a linear sequence. The behavior can be partitioned into simpler behaviors in various way, and the evolution progress will follow an incremental evolution scenario graph.
Another important issue is how to transfer a population from the end of one incremental step to another step. When entering a new step, already learned parts of the genotype can either remain frozen or continue to evolve. Sub-parts of the problem can be either independent, for example individual modules that can be tested separately, or depend to some extent on other parts — thus creating a dependence hierarchy (i.e. arranged in a tree, or in a graph as explained above). In case of dependencies, one can incrementally evolve the behaviors from the bottom of the hierarchy, adding upper layers and parallelize the algorithm later. This is a general framework. In order to find plausible solutions, EAs require that the initial population randomly samples the search space. However, the population at the end of one step is typically converged to a very narrow area, and thus it cannot be used directly as an initial population in the next step. We therefore propose to generate a new initial population from several ingredients:

- some portion of the original population containing the best individuals is copied,
- another part is filled with copied individuals that are mutated several times, and
- the remaining individuals are randomly generated.

However, it is also possible to blend the populations of two or more previous incremental steps. In principle, we propose that the evolutionary incremental algorithm will take for each incremental step a full specification of blending, copying, and mutation ratios for all genome sub-parts (such as finite-state automata) and all preceding incremental steps, see (Petrovic, 2007) for formal specification of the process.
For example, let us examine an incremental scenario with six incremental steps. The target genome in the last incremental step consists of three finite-state automata each corresponding to one of the behavioral modules that are subject to evolution, see Figure 13.

In addition, there may be other modules in the controller, which are already designed and which are not evolved in the particular step – these are shown filled at the figure. In our example, the second incremental step continues to evolve the automaton corresponding to the first module, and begins with evolving the automaton corresponding to module II. The third and fifth steps are completely independent, and evolve the automata for module III, and for modules I and III, respectively. The fourth and the sixth steps merge populations from multiple steps. Whereas the fourth step simply combines the automata for the module III coming from step 3, and automata for the modules I and II from the second step into common controller, the sixth step blends the populations from the fourth and fifth steps for both modules I and III, and further evolves the automata for the module II from the fourth step.

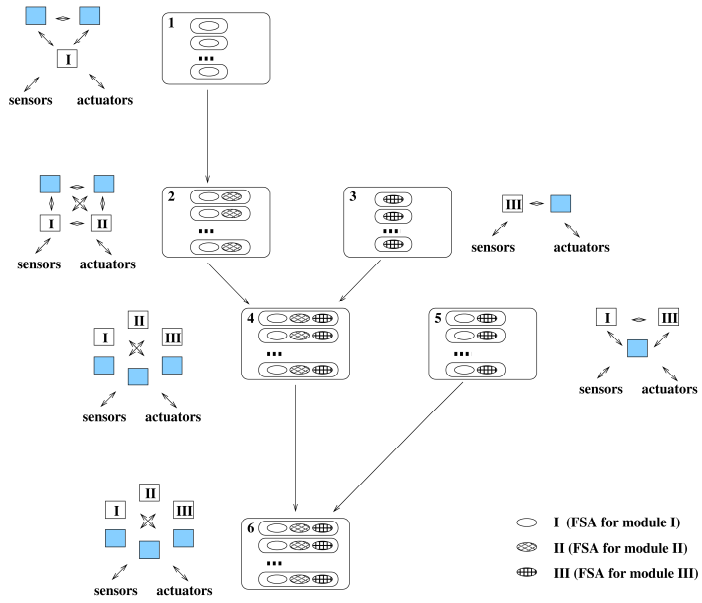

Fig.13. Population mixing in an incremental scenario.

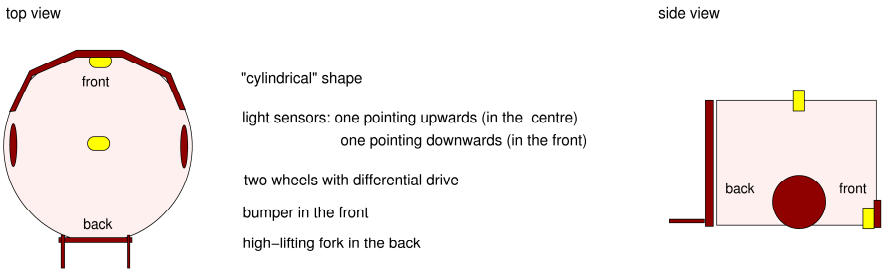

Fig.14. Simulated robot topology and features.

5.3 Experiment

Our experiment has the goal to evaluate the controller architecture and the incremental evolution method proposed in the previous chapters and compare them to a manual design of the coordination mechanism in the controller. The RCX hardware platform offers high flexibility and a multitude of possible configurations for laboratory robotics experiments. We tested the implementation of our controller architecture on a high lifting fork robot built around a single RCX (Figure 10 right, Figure 14). The task for the robot is to locate a loading station, where the cargo has to be loaded, and then locate an unloading station to unload the

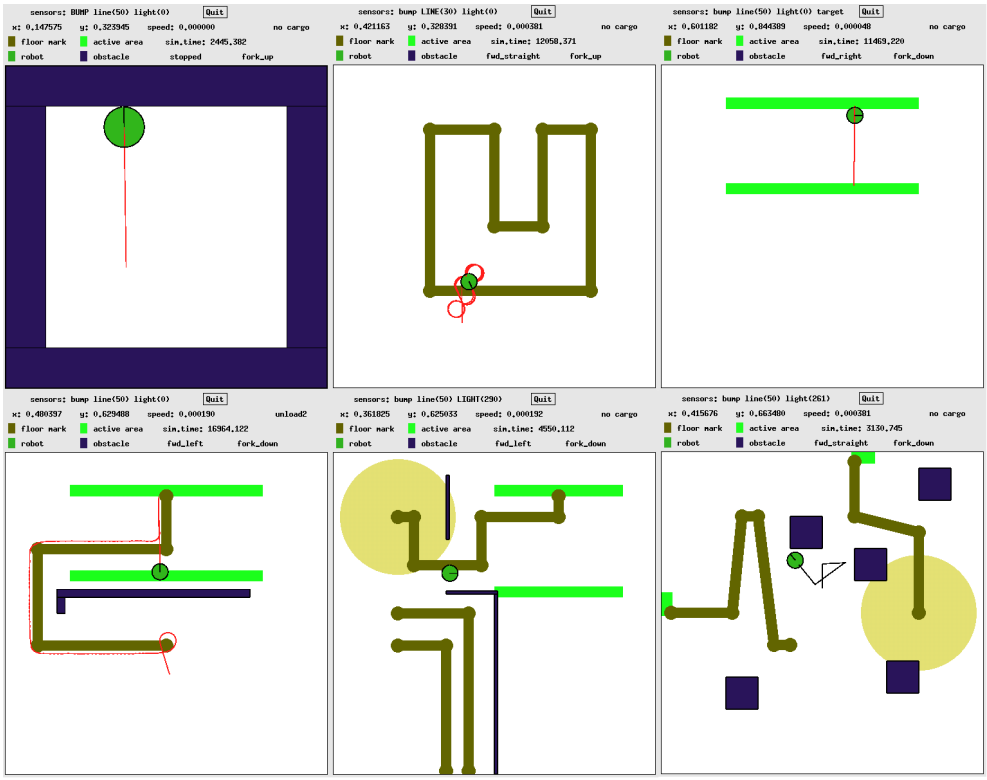

Fig.15. Experimental environments for 6 incremental evolutionary steps of creative incremental scenario, from left A – C, and D – F in the first and the second rows. A: *avoidance* — the robot is penalized for time it spends along the wall; B: *line following* — the robot is rewarded for the time it successfully follows the line; it must have contact with the line and should be moving forward; C: *cargo-loading* — robot is rewarded for loading and unloading cargo in an open area without lines or obstacles; D: *cargo-loading after line following* — follow-up of B and C, the robot is rewarded for loading and unloading cargo, but it has to successfully follow line to get to the open loading/unloading area; E: starting *line-following under light* — robot learns to start following the line that is under the light (it is started from different locations in order to make sure it is sensitive to light and not, for instance, to number of lines it needs to cross); F: *final task* — robot is rewarded for successfully loading and delivering the cargo, it uses the avoidance learned in A and behavior E.

cargo, and repeat this sequence until the program is turned off. The robot exists in a closed rectangular arena with obstacles to be avoided. Both loading and unloading stations lie at the end of a line drawn on the floor. The start of the correct line to be followed at each moment is illuminated from above by light (an adjustable office lamp), Figure 15F. The light source located over a segment of the line leading to the loading station is automatically turned off when the robot loads the cargo, and it is turned on when the robot unloads the cargo at the correct location. The reverse is true for the light located over the line leading to the unloading station. Other lines might exist in the environment as well. We chose this task for four reasons: 1) compared to other evolutionary robotics experiments, it is a difficult task; 2) it is modular, in terms of the same behavior (finding and following line) being repeated two times, but with a different ending (either loading or unloading cargo), and therefore has a potential for reuse of the same code or parts of the controller; 3) it can be implemented both in real hardware and in our simulator (for the implementation of the switching lights, we used two standard office electric bulb lamps controlled by two X10 lamp modules and one X10 PC interface connected to a serial port of a computer that received an IR message from RCX brick when the robot loaded the cargo, which was in turn detected by an infrared emitter/detector from HiTechnic); 4) the task consists of multiple interactions, and behaviors, and thus is suitable for incremental evolution.

Our robot comes with a set of preprogrammed behavioral modules. We have coded them directly in the language C:

Sensors — translates the numeric sensory readings into events, such as robot passed over or left a line, entered or left an illuminated area, received an IR message from a cargo station, bounced into or avoided an obstacle.

Motor driver — accepts commands to power or idle the motors. The messages come asynchronously from various modules and with various priorities. The purpose of this module is to maintain the output according to the currently highest priority request, and fall back to lower priorities as needed. All motor control commands in this controller are by convention going through the motor driver.

Navigate — is a service module, which provides higher-level navigational commands — such as move forward, backward, turn left, as contrasted with low-level motor signals that adjust wheel velocities.

Avoidance — monitors the obstacle events, and avoids the obstacles when encountered.

Line-follower — follows the line, or stops following it when requested.

Explorer — navigates the robot to randomly explore the environment. It turns towards illuminated locations with higher probability.

Cargo-loader — executes a procedure of loading and unloading cargo on demand: when the robot arrives to the cargo loading station, it has to turn 180º, since the lifting fork is on the other side than the bumpers, then it moves the fork down, approaches the cargo, lifts it up, and leaves the station; at the unloading station, the robot turns, approaches the target cargo location, moves the fork down, backs up, and lifts the fork up again.

Console and *Beep* — are debugging purpose modules, which display a message on the LCD, and play sounds.

The coordination mechanism, which is our focus, consists of FSA post offices attached to individual modules. Figure 17 shows the hand-made FSA for the line-follower module. Other modules that use FSA are cargo-loader, avoidance, and explore. Figure 15 shows the six incremental steps and their respective environments for our main incremental scenario

(we refer to it as creative). Throughout the whole experiment, the robot morphology and the set of sensors and actuators remained unchanged. In the first three incremental steps, the task, the environment, and the controller were simplified. In the fourth and the fifth incremental steps, the task and the environment were simplified, but the controller already contained all its functionality. Evolution progressed to the next incremental step when an individual with a satisfactory fitness was found and the improvement ratio fell below a certain value, i.e. the evolution stopped generating better fit individuals. The improvement ratio m_n in generation n was computed using the following formula:

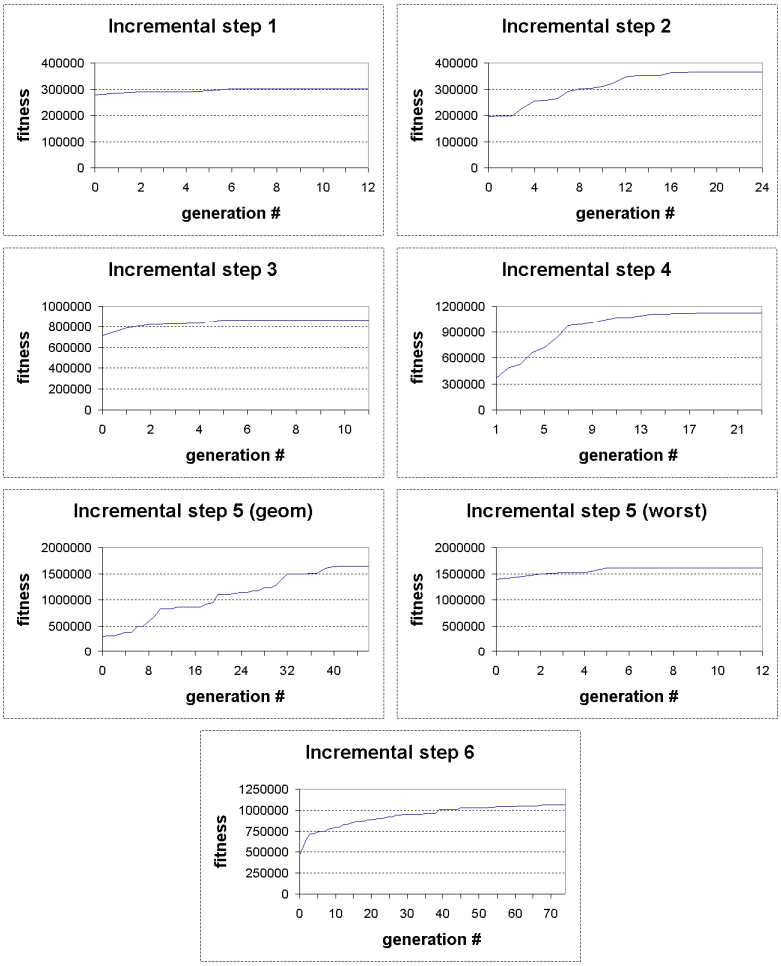

Fig.16. Best fitness for all incremental steps (averaged 10 runs), creative incremental scenario. In the incremental step 5 (geom), the assigned fitness is geometric mean of fitness achieved from the three different starting locations. This step was followed by 5 (worst), where the assigned fitness is the worst fitness achieved in the three starting locations.

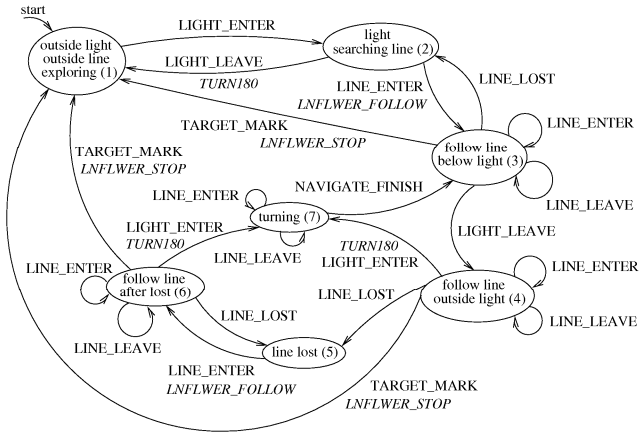

Fig.17. Example of a manually designed FSA coordination for the linefollower module.

$$m_n = \varphi \cdot m_{n-1} + (\text{best fitness}_n - \text{best fitness}_{n-1}) \quad (2)$$

where $best\ fitness_i$ is the fitness of the best individual in population i, φ is a constant (we used $\varphi = 0.2$), and m_0 is initialized to $0.9 \cdot best\ fitness_0$.

5.4 Simulation

Simulations in ER are more complex than usual, because we simulate the behavior of a robot in its environment as contrasted to a simulation of some process that is an integral part of the simulated environment. The behavior of robot depends not only on the physical processes in the environment, but also on its own actions generated by its program – i.e. there are two simultaneous systems to be simulated. The simulation of the robot controller can be performed through direct emulation of the robot hardware/OS on the simulating hardware and OS. None of the existing simulators satisfied our needs, we chose to implement our own directly in language C, to be compatible with the compiling environment of the HW platform.

Our *lazy simulation* approach (inspired by lazy evaluation in functional programming languages) updates the state of the simulated system only when the robot controller interacts with the robot hardware. At that time instant, we suspend the emulated program, compute the current exact state of the simulated system (environment and robot) by mathematical formulas, and determine the outcome of the interaction that triggered the update. The temporal granularity is thus limited only by the CPU or bus frequency of the simulating machine. The emulated program is not interpreted by the simulator. It runs almost independently within the operating system of the simulating computer. Instead of accessing the robot hardware, it accesses the simulator that is waiting in the background. For example, the robot controller program might be computing without interacting with the robot hardware for some time, during which the robot crosses several lines on the floor, triggers switching of the light by entering an active area, passes below a light, and bounces

to a wall, where it remains blocked for a while. At that point in time, the robot controller wants to read a value of its light sensor, for instance, and only at that point in time the simulator becomes active and computes the whole sequence of the previous events that occurred, and the current location and situation of the robot and the environment. Finally, the required value of the sensor reading is determined and returned to the program, which resumes its execution. To achieve better performance, the simulator pre-computes expected events before resuming the simulated program. The pre-computed information helps to test quickly whether the state of the robot or environment has changed since the last "interrupt", without processing all the data structures of the simulator. We call this approach lazy simulation because it uses maximum possible abstraction from the environment and performs simulation computation only when very necessary.

5.5 Results

To verify the controller architecture and task suitability, we have first designed the post-office coordination manually. This controller performed well and resulted in reliable cargo delivery behavior. We have also tested the controller on the real robot. The transition to the real-world settings was straightforward, except of the calibration of the sensors and timing of motoric actions. Still, in this experiment, the exact quantitative dimensions and distances played minor role, for the performance of the controller (except, perhaps, for the line-follower module), and therefore the distances and timings in the realistic actions did not need to correspond to the simulated one with 100% accuracy, and actual tuning of the timing could be performed separately for the simulated and realistic runs.

In the evolutionary experiments, we have tried to see if the evolutionary algorithm described above could evolve the target task by automatically designing all four FSA modules in a single evolutionary run. We ran the program for 20 times with a population of 200 individuals and 200 generations, and with a fitness function rewarding line-following, cargo-loading and unloading, distance traveling, and penalizing obstacles. However, none of the runs evolved the target behavior. To save computational effort, we have stored all previously evaluated genotypes with their fitness to the database. The objective function first checks if the genotype has already been evaluated and starts the simulator only in case of a new genotype. Furthermore, during the simulated run, we measure the fitness obtained by the best (or average of several best) individuals, and later, we automatically stop all individuals that achieved less than q% of the best measured fitness (q = 5%) in one of the periodically occurring checkpoints, if this was feasible. All evaluated FSA and the trajectories of best-fitness improving runs were saved to files and extensive logs were produced for further analysis.

Later experiments followed the creative scenario with 6 incremental steps shown at Figure 15. The general fitness function used in this task had a form

$$\begin{aligned} f = a &+ w_{\text{obstacle time}} \cdot t_{\text{obstacle}} + w_{\text{below light time}} \cdot t_{\text{below light}} + w_{\text{following line time}} \cdot t_{\text{following line}} + \\ &+ w_{\text{following below light time}} \cdot t_{\text{follow below light}} + w_{\text{total distance}} \cdot d_{\text{total}} + \\ &+ w_{\text{robot moving changed}} \cdot n_{\text{moving changed}} + w_{\text{num states}} \cdot n_{\text{states}} + w_{\text{num\_trans}} \cdot n_{\text{trans}} + \\ &+ \sum_{i=1}^{\text{num\_scripts}} w_{\text{script\_count}}(i) \cdot n_{\text{script\_started}}(i) + \sum_{i=1}^{\text{num\_active\_areas}} w_{\text{active\_area\_count}}(i) \cdot n_{\text{area\_started}}(i) \end{aligned} \quad (3)$$

Where a is an offset value, $w_{atribute}$ are weight constants of specific self-explanatory attributes. In addition, if $w_{supress\_score\_before\_load}$ is set to 1, no scores are accummulated before cargo is first time loaded successfully. Typical values of the weight constants were established empirically (Petrovic, 2007). In various steps, many of the constants were 0, i.e. the actual fitness function was much simpler. Figure 16 plots the best fitness average from 10 different runs. Each evaluation took into account the worst fitness for the three different starting locations and robot orientations, except of step 5, where we had to use the geometric mean of fitness from all three runs. This ensured that the behavior evolved in step 4 was not lost as the successful individuals from step 4 at least performed well when started close to the line that was leading to the loading station. Using the worst fitness resulted in loosing the behavior learned in step 4 before the sensitivity to light was evolved. On the other hand, this step was repeated with the same settings, except for the use of worst fitness instead of geometric mean, before proceeding to step 6, in order to eliminate the cheating individuals from the population. In order to obtain a better evaluation of our approach, we compared the runs against an alternative scenario (which we in fact designed first, and we refer to it as sequential) of incremental steps: The robot is rewarded in different incremental steps for:
1) avoiding obstacles, 2) following a line, 3) following a line under light (while being penalized for following line outside light), 4) loading cargo, 5) loading cargo, and for following a line under light after it has loaded cargo, 6) loading and unloading cargo (one time unloading is sufficient), 7) for loading and unloading cargo (multiple deliveries are required). This sequential scenario corresponds to the sequence of skills as the robot needs them when completing the target task, being thus a kind of straight-forward sequential decomposition. Contrary to the creative scenario, here the input material in each step consists only of the individuals from the final population of the directly preceding step.

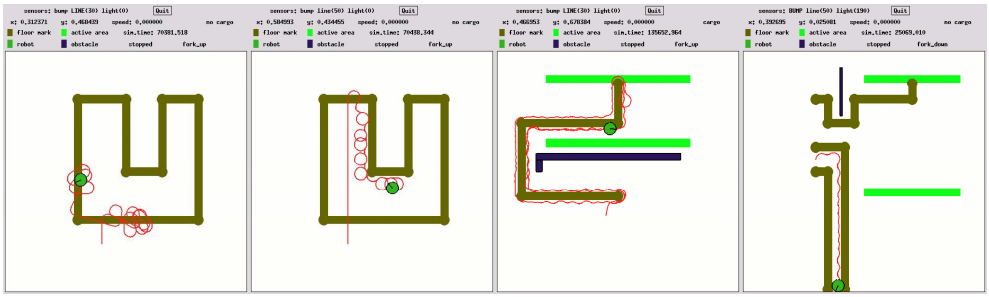

Fig.19. Examples of evolved misbehaviors demonstrating richness of controllers with FSA modules evolved for a set of predefined competence modules. In A and B, the robot is trained to follow the line. In A, when the line is encountered, it starts chaotic cyclic movements around the line, sometimes crossing a larger distance in a straight movement. In B, the robot starts following the line after it meets it for the second time, but instead of smooth following, it follows by the line by systematic looping – although actually achieving the desired behavior at certain quality level. In C and D, the robot needs to follow the line and then periodically transport cargo between the loading and unloading locations placed opposite to one another. In C, the robot loads the cargo once, but fails to stop following the line. In D, the robot fails to follow the correct (third from the bottom) line, which starts under top-mounted light (not shown in this version of viewer).

Another important difference is that the environments in all steps of sequential scenario were the same as in the final task, with the exception of the third incremental step, where the line originally leading to the loading station was changed to a loop, being illuminated by light along full its length; for this purpose we also removed one of the obstacles and introduced an additional light source. We tried to evolve the target behavior with sequential incremental scenario without this simplification of the environment, however, even after spending several weeks of efforts and years of computational time, and exhausting the parametric space of the configuration options, and various fitness functions, the correct controller functionality was never produced. In particular, it appears to be too difficult to evolve sensitivity to light, while not loosing the proper line-following behavior, if the line leaves the light and follows to the loading station, where it is non-trivial to turn and return back under the light to gain fitness. If the robots were rewarded for spending time under the light, they evolved all the possible tricks of pretending the line-following behavior, while moving in various loops, but forgetting the proper line-following behavior at the same time. Once the line-following behavior has been lost, it was very difficult for the evolution to reclaim it later again in the successor incremental steps, which required it. Modifying the environment in the third incremental step was sufficient, and the target behavior evolved in 11 of 15 runs, each run taking about 12 hours on a pool of 60 computational nodes (2 GHz PCs). Fitness plots and tables are in (Petrovic, 2007).

6. Future Work and Conclusions

We study the problem of automatic design of behavior coordination mechanism for behavior-based controllers of mobile robots by the means of evolutionary algorithms. While other researchers often investigate controller architectures inspired by neural networks, we focus on distributed coordination mechanisms based on finite-state automata. Plain evolutionary algorithms are not capable of overcoming the complexity of this design problem and therefore must be supported by additional framework. We use the method of incremental evolution, i.e. partitioning of the evolutionary task into a structure of simplified tasks of gradually increasing complexity from various viewpoints.

- We experimentally confirm that incremental evolution is a possible way of overcoming the complexity of evolutionary task in the field of Evolutionary Robotics, describe and experiment with various flavors of incrementality.
- We design and implement a new universal distributed coordination mechanism and controller architecture consisting of behavioral modules, message passing, and coordination modules associated with the behavior modules.
- We show how our coordination mechanism can be automatically designed using evolutionary algorithm with the help of incremental evolution on a non-trivial mobile robot task evolved in simulation and verified on a real robot. That means, we confirm that the behavior coordination mechanism in a behavior-based controller for a mobile robot can be automatically designed using evolutionary computation.
- In the creative scenario, various skills are learned independently in modified environments and merged together in later steps.
- In the sequential scenario, the same target environment is used, and the behavior is built in a sequential manner – from shorter sequences of actions to more complex sequences.

• The sequential scenario fails to evolve the target behavior due to a high complexity, but succeeds if the environment is modified in one of the steps.
• The creative scenario succeed to evolve the target behavior in significantly shorter time (t-test: 2.5786).
• We show that finite-state automata, the genotype representation used in our coordination architecture, outperforms GP-tree programs on tasks with structural similarity to behavior coordination problem.
• We show that incremental evolution can both improve and decrease the evolvability; the overall effect of its use can be both positive and negative and thus the use of incremental evolution requires understanding and preliminary analysis of the problem-specific search space.

One of the important strengths of the incremental approach to evolution of controllers stems from the incremental property of the behavior-based approach to building controllers.

When a functional controller is extended with a new functionality, this is typically done by adding one or more behavior modules. The coordination of the original controller needs to be adjusted to the new task, while preserving the original functionality.

In case of distributed coordination system, this practically means modifying the previous coordination to fit the new situation, and designing the new coordination for the new module(s). It is important to realize that this is the same kind of design step as was repeated several times while designing the original controller that is being extended. Thus, our methodology for design is open-ended: the controller is never fixed-finished, rather allows for future modifications that have same characteristics, issues, and progress as the original design process.

In the future work, we would like to relax the discreetness of the state automata by combining them with probabilistic approaches, as well as to compare the approach to other methods of automatic design of BB coordination.

Simulations of mobile robotic experiments certainly form a class of hard simulation problems. The number of interactions of the simulated system (a robot) with its environment is typically extremely high, since a mobile robot must continuously scan its environment using most of its sensors. Each such sensing is a separate event, and the density of the sensor events is typically bounded only by the speed of the robot hardware — sensors and CPU. In a multithreaded system, where multiple behaviors compete for the robot CPU resources, the simulation of the robotic system becomes challenging, in the sense that even very slight differences in timing might lead to quite different robot behavior and performance. Building accurate simulating environment on a different platform is difficult and unlikely. Therefore, the controllers designed in the simulation need to be robust enough in order to cope with the transition to real robots. Extra adjustment efforts might be needed during the transition process.

6. References

Acras, R.; Vergilio, S. (2004). Splinter: A Generic Framework for Evolving Modular Finite State Machines, *Proceedings of SBIA 2004*, pp. 356-365, Springer-Verlag

Angeline, P.; Pollack, J. (1993). Evolutionary Module Acquisition, *Proceedings of the Second Annual Conference on Evolutionary Programming*, pp. 154-163

Ashlock, D.; Wittrock, A.; Wen, T. (2002). Training Finite State Machines to Improve PCR Primer Design, *Proceedings of the CEC'02*, pp.13-18

Ashlock, D.; Emrich, S.; Bryden, K.; Corns, S.; Wen, T.; Schnable P. (2004). A Comparison of Evolved Finite State Classifiers and Interpolated Markov Models for Improving PCR Primer Design, *Proceedings of the 2004 IEEE Symposium on CIBCB*, pp. 190-197

Chellapilla, K.; Czarnecki, D. (1999). A Preliminary Investigation into Evolving Modular Finite State Machines. *Proceedings of the CEC'99, vol. 2*, pp. 1349--1356

Clelland, C.; Newlands, D. (1994). PFSA Modelling of Behavioural Sequences by Evolutionary Programming, *Proceedings to Complex'94*, pp. 165-172, IOS Press

Fogel, L. (1962). Autonomous Automata. *Industrial Research*, 4, 2, 14-19.

Fogel, L. (1993). Evolving Behaviors in the Iterated Prisoners Dilemma. *Evolutionary Computation*,1, 1, 77-97.

Frey, C.; Leugering, G. (2001). Evolving Strategies for Global Optimization – A Finite State Machine Approach. *Proceedings of the GECCO'2001,* pp. 27-33, Morgan Kaufmann.

Harvey, I. (1995). *The Artificial Evolution of Adaptive Behaviors*. PhD thesis, University of Sussex.

Horihan, J.; Lu, Y. (2004). Improving FSM Evolution with Progressive Fitness Functions. *Proceedings of GLSVLSI'04,* pp. 123-126

Hsiao, M.; (1997). *Sequential Circuit Test Generation using Genetic Techniques*. PhD thesis, University of Illinois at Urbana-Champaign.

Jefferson, D.; Collins, R.; Cooper, C.; Dyer, M.; Flowers, M.; Korf, R.; Taylor, C.; Wang, A. (1992). Evolution as a Theme in Artificial Life: The Genesys/Tracker System. *Artificial Life II,* pp. 549-578.

Koza, J. (1992). Evolution of Subsumption Using Genetic Programming, *Artificial Life I*,pp.110-119.

Lee, W.; Hallam, J.; Lund, H. (1998). Learning Complex Robot Behaviours by Evolutionary Computing with Task Decomposition. *LNCS 1545*, pp. 155.

Lucas, S. (2003). Evolving Finite State Transducers: Some Initial Explorations, *EuroGP'03*, pp.130-141.

Lund, H. (2001). Co-Evolving Control and Morphology with LEGO Robots. *Proceedings of Workshop on Morpho-functional Machines*, Springer-Verlag.

Lund, H.; Miglino, O. (1998). Evolving and Breeding Robots. *Proceedings to EvoRobot'98*.

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

Maes, P. (1990). How to do the Right Thing. *Connection Science Journal*, 1, 1990, Special Issue on Hybrid Systems.

Ostergaard, E. (2000). *Co-evolving Complex Robot Behavior.* Master thesis, Uni. of Aarhus.

Petrovic, P. (2007). *Incremental Evolutionary Methods for Automatic Programming of Robot Controllers,* http://urn.ub.uu.se/resolve?urn=urn:nbn:no:ntnu:diva-1748, NTNU-trykk.

Spears, W.; Gordon, D. (2000). Evolving Finite-State Machine Strategies for Protecting Resources, *Proceedings of the 12th International Symposium on Foundations of Intelligent Systems*, pp. 166-175, Springer-Verlag.

Togelius, J. (2003). *Evolution of the layers in a subsumption architecture robot controller.*Master Thesis, University of Sussex at Brighton.

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