Evolving automatons for distributed behavior arbitration

Pavel Petrovic, Department of Computer and Information Science, NTNU Trondheim, *ppetrovic@acm.org*

IDI/NTNU Technical Report 05/2005.
ISSN 1503-416X.

Abstract

Designing controllers for mobile robots is an advanced engineering task. We develop a method for designing the controllers automatically using artificial evolution. In contrast to similar approaches, our aim is to evolve controllers for more complex tasks involving multiple interactions and behaviors. We achieve this by providing a set of pre-designed simple low-level behaviors, which are either programmed manually or designed automatically. The output of our algorithm is a behavior arbitration mechanism consisting of a set of finite-state automata. Inspired by the incrementality of the Behavior-Based Robotics, we employ incremental evolution to decrease the size of the search space and to automatically design the robot controller in a bottom-up manner.

Introduction

In order to utilize ever advancing technological developments of materials, power supplies, computer technology, sensors, and actuators in the field of robotics, suitable controller architectures at a correspondingly advanced level need to be developed for these devices. Mobile robots could perform complicated tasks in unknown, nondeterministic, changing, noisy, and unpredictable environments, if methods for designing robot controllers could be developed. Successful completion of tasks will require the controllers to be adaptive and learning. Systematic research efforts to study possible methods for building such controllers are to be spent.

Designing a controller for a particular task for a mobile robot requires detailed knowledge of the robot hardware and software and an experienced engineer. We are seeking an alternative: automatic design of the controller. Our aim is to minimize the efforts and maximize the quality. One of the most popular methods for automatic design of robot controllers is Reinforcement Learning [22]. An alternative employed also by this work are Evolutionary Algorithms (EA) [17]. It has been demonstrated that EAs can be successfully used to design the controllers for trivial tasks such as wall-following, and obstacle-avoidance, or target tracking. The controllers are often based on arbitrarily-connected neural networks [2]. Adaptive behaviors on a different level could possibly be achieved by approaches [5] that evolve the neuron learning rules instead of the connection weights, which are changing dynamically during the task execution. Nevertheless, it appears that evolving more complex behaviors in a single evolutionary run is not plausible due to a complex search space, and becomes impossible without additional guidance of the EA. One way of self-guiding of an evolutionary algorithm is the use of co-evolution [7]. Co-evolution, however, applies well only to a narrow class of problems where two entities can be competing for the same resource. In addition, it suffers from possible cyclic loops in the relation of strategy dominance in effect causing stagnation of the algorithm instead of progression. Another possible guidance, adopted also in this work, is dividing the target task into more simple incremental steps [6]. This strategy is more general; however, it requires a scenario of incremental evolutionary steps. How to devise such incremental steps, and how to setup such incremental EAs is the focus of this and future work.

The complexity of interactions of a mobile robotic system implies a structured (non-monolithic) controller architecture. Traditional AI approaches based on centralized-processing and Sense-Plan-Act cycle have difficulties dealing with simultaneous robot activities that have different priorities. Even if such a controller would be successfully designed, its maintainability, modifiability, and performance would be compromised. Mobile robots must behave according to the outcome of several independent threads of reasoning. Therefore a modular architecture with reasoning modules executing in parallel is suitable. On the bottom side, these modules are as simple as direct reactive connections between sensors and actuators. On the top side, these modules might perform long-term reasoning inferences running on the background with low priority. These ideas are reflected well in various Behavior-Based (BB) and hybrid architectures [1]. One of the challenges of BB design is how the individual modules will be coordinated. This is referred to as action-selection problem or behavior arbitration; an overview is in [21].

In this work, we aim at evolutionary design of behavior arbitration for a controller of a mobile robot performing a non-trivial task, where simple reactive controller would not be sufficient. The controller has a particular BB architecture, while the arbitration is based on a set of finite-state automata. The design is the output from an incremental evolutionary algorithm. Further sections of this article discuss our controller architecture, evolutionary algorithm, simulation framework, incremental evolution method, example task setup, obtained results, further work, and conclusions.

Related work

Our efforts are to combine and integrate the ideas from several independent fields. The main area of interest is the field of Evolutionary Robotics [2, 5, 6, 8, 10]. Most of the work in the field is focusing on adaptive mobile robots with neural controllers. Inspiration from biology brings the laws of natural Darwinian evolution and motivates towards long-term evolution of individuals that successfully perform in their artificial life environment. In these cases, the focus is not on building systems that can be directly useful today or tomorrow, rather on the study of how the natural principles observed in the living species apply to the artificial robotic systems built by us. It is often not important what the agent will be able to do when the evolution completes, rather how the evolutionary process progresses, and how it interacts with the agent learning abilities. Aim is at answering the questions of how we could imitate the clever and very effective animal behavior that relies on imperfect and irregular patterns, and how to build artificial control systems that would have similar properties of the animal brains. Naturally, researchers with such motivations study controllers based on artificial neural networks, and explore various neural architectures. The systematic research efforts start and still progress with the early systems capable of obstacle avoidance, wall-following, target recognition and following, box pushing, simple autonomous flight, or survival of agents in artificial environment. It is sincere to point out that most (if not all) of the resulting evolved controllers perform a behavior that can easily be achieved by a manually programmed controller of small or moderate difficulty. However, the researchers in the field cull the argument stressing their interest in adaptive and self-organizing systems, and the early stages of the research field.

Another group of research efforts study the possible architectures for intelligent mobile robot controllers for robots performing real tasks in real environments. Traditional AI methods for building robotic controllers [15] typically rely on extensive centralized planning and sense-plan-act cycle. These approaches are limited mainly by their complexity, they are difficult to maintain, test, and debug. Others proposed modular incremental architectures of independent modules responsible for different behaviors that can be developed, tested, debugged, and even exchanged individually, and usually run in parallel when combined in a complex controller. The main research challenge in this area (called after the behaviors BB Robotics [1]) is how the behaviors can be coordinated, and how they should interact in order to share the agent resources. Substantial work and an overview have been done by Pirjanian [21]. There have been only few attempts to generate arbitration mechanisms automatically based on the required task or robot purpose. Their usefulness can be supported by the following reasons:

- 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.
- Mobile robots can be built for general purpose, and the arbitration 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 arbitration mechanism based on task description might be required. Having the option of automatic arbitration generation might save extensive amounts of work needed to hand-craft each arbitration mechanism.
- Manual arbitration 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. Automatic arbitration design might capture the undergoing characteristics of the robot interactions more reliably, efficiently and precisely.

Learning *action selection* (term sometimes used interchangeably with *behavior arbitration*) studies in his thesis Humphrys [8]. His focus is on the "communication" of agents that together form a controller; in fact, his agents are so simple that they correspond to the nodes in a recurrent neural network, with very few control actions generated by output nodes. Input nodes receive discrete sensing of an artificial ant moving on a rectangular grid. The topology and connection weights are evolved and the communication of the very few nodes in the network is studied on a standard artificial ant seeking food in a rectangular grid problem.

The most valuable inspiration for our work stems from the work of Lee et al. [10], where the more complex, high-level task is decomposed hierarchically and manually into several low-level simple tasks, which can further be decomposed to low-level tasks. The reactive controller consists of primitive behaviors at lowest level and behavior arbitrators 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. We see several possible improvements:

- avoiding the centralistic architecture,
- allowing for easier modification of task by introducing a new module without affecting substantially the existing controller hierarchy, and
- removing the limitation to purely reactive tasks.

The work has been continued on by the master thesis [18], 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 arbitrations: 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. Even though extensibility has been slightly improved from [10] and internal state was introduced, still only a limited subclass of finite-state automatons taking the arbitration role was supported. Reactive modules are not general, but have to follow with the simple unifying architecture only providing the activation level, and being based on the winner-take-all principle. Co-evolution that was used for increasing the difficulty of the evolutionary task could be applied due to the game character of the task, however it does not scale up to more general classes of tasks.

Controller architecture

A task of an autonomous robot usually consists of many interactions occurring, starting or ending at various time points. Between these events, the robot remains in some particular state. We consider state and event to be the crucial concepts for the controller architectures of the mobile robots. Many researchers advocated the use of neural network based controllers. However, the representation of states in the recurrent networks is still very poorly understood. Furthermore, responsiveness of neural networks to particular events is hard to asses and analyze, and the current learning mechanisms require thousands of interactions to learn even very simple behaviors. Controllers aiming at more complex tasks must have modular architectures, but it is yet unclear how this modularity can be efficiently achieved with neural networks, and what consequences it might have on their learning rules and algorithms. We suggest studying alternative architectures, which better reflect the concepts of robot state and environmental events.

Our controller architecture is strongly inspired by BB robotics. The large numbers of interactions, which can arbitrarily occur at any time of robot execution, mean that many different activities can be triggered at any time; many different sensory inputs and their various aspects have to be monitored continuously. Instead of having one centralized learning module, BB robotics suggests distributed control in multiple behavioral modules. Our controller also consists of several independent modules, which are running simultaneously. Many of the individual interactions have to be coordinated, and the modules might need to exchange relevant information about their states. The modules communicate by sending asynchronous messages, which can be either broadcasted or sent to a specific target module. In principle, each module can access the robot's low-level hardware (sensors and actuators), but a careful design approach leading to multiple abstract layer architectures should be taken. As a result, only very few modules (ideally one) should access each robot hardware resource and extract the relevant sensory information or synchronize the access to an actuator for all other modules. The competition for robot actuators is implemented using fixed or dynamic module priorities: the robot actuator executes the action requested by the module with the highest priority, or becomes idle, if no module requests an action. One of the main motivations for BB robotics is the incremental building of the robot and robot controller (bottom-up design). This implies the qualities labelled in the fields of software architecture as "modifiability" and "extensibility". Indeed, to add new functionality to a working controller, it is sufficient to add a new module, and integrate it into the controller by modifying the message interface for all relevant modules to respond to the new interactions. Figure 1 shows an example of a controller, where the robot is randomly exploring environment turning more likely towards more illuminated regions, while avoiding obstacles, until it enters a the line, which it starts to follow. The added dashed module introduces new functionality: the robot will start following the line only if it is located in an area, which is illuminated. To achieve this change, the whole existing controller can be preserved, but the line-following controller will need to introduce a new state *outside_illuminated_area*, which can be triggered by the context switching module. In this state, the line following module will not start following a line.

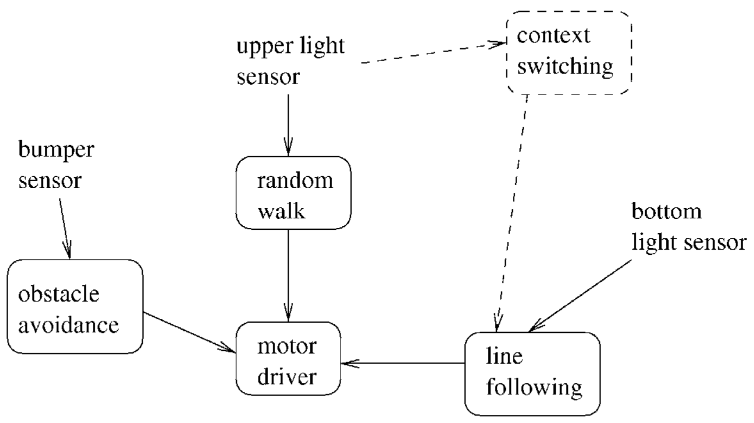
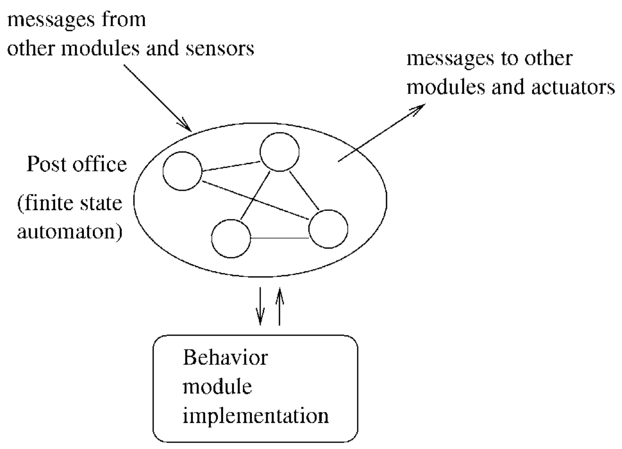


Figure 1. Extensibility of the controller.

Ideally, the message interfaces of the modules should be defined as simple and as general as possible so that each module can easily be integrated into a controller. In this way, modules can be reused across different controllers, which are built for different purposes and tasks. To extend this idea further, hardware modules, such as sensors and actuators can have certain level of intelligence and be accessed with unified interfaces so that they can be easily interchanged and configured. This idea is well adopted by LEGO robotics sets [9], and there exist other similar toolkits, such as Microbric [25], Parallax robotics [26], or Handyboard [24]. The idea of easy integration of intelligent sensors into larger networks is at focus for large players like Intel, or for as small groups as InnoC with their Simple Sensor Networks [27]. One could argue that the robot controller should be general and give the robot all the possible functionality by including all the possible behavior modules. However, the memory and CPU capacities of practical robotic systems are always limited, and thus, using separate programs (controllers) for each task, instead of insisting on one general-purpose controller, can be more feasible. To achieve generality, all the modules might be stored in a long-term memory, and only the relevant modules might be retrieved into an operational memory when a particular task is being solved. The controller thus contains a set of specific behavior modules with clearly defined message interfaces, and a coordination mechanism, which makes these modules talk together as required by a particular task.

Many previous approaches to the action-selection or behavior-arbitration problem are either centralized, for example [10] or too limiting, for example [13]. Our aim is to design a general architecture, which can cope with different design challenges in a systematic rather than an ad-hoc manner. In addition, this mechanism should be easy to integrate into the set of modules that are exchanging asynchronous messages. Each module has its own post-office module, which filters and translates the incoming messages into the messages recognized by the module itself, and which also monitors and possibly filters or modifies the messages being sent by the module. Since the state is a central concept in the controller, we chose to first study the post-offices that have the form of a finite-state automata. The state transitions are triggered by messages being received or sent by the module owning the post-office. Each state transition can result in a message being sent to the module or to other modules, see figure 2. Please see also figure 13 for examples of evolved FSAs and the representation of the transitions.



messages from other modules and sensors
messages to other modules and actuators
Post office
(finite state automaton)
Behavior module implementation

Figure 2. Single behavior module with its arbitrating post-office finite-state machines.

This coordination mechanism is distributed, and thus the standard advantages of the distributed systems: robustness, modularity, better communication throughput, better encapsulation and modifiability are gained for free. In particular, adding a new module to an existing controller requires only designing a new post-office for the added module and the minimum set of changes in post-offices of other modules, instead of inferring with all modules and modifying a centralized coordination mechanism.

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 in figure 3. The mouse explores the room in random movements, and whenever it smells or sees the cheese, it moves in its direction, grasps the food, and drags it back to its home.

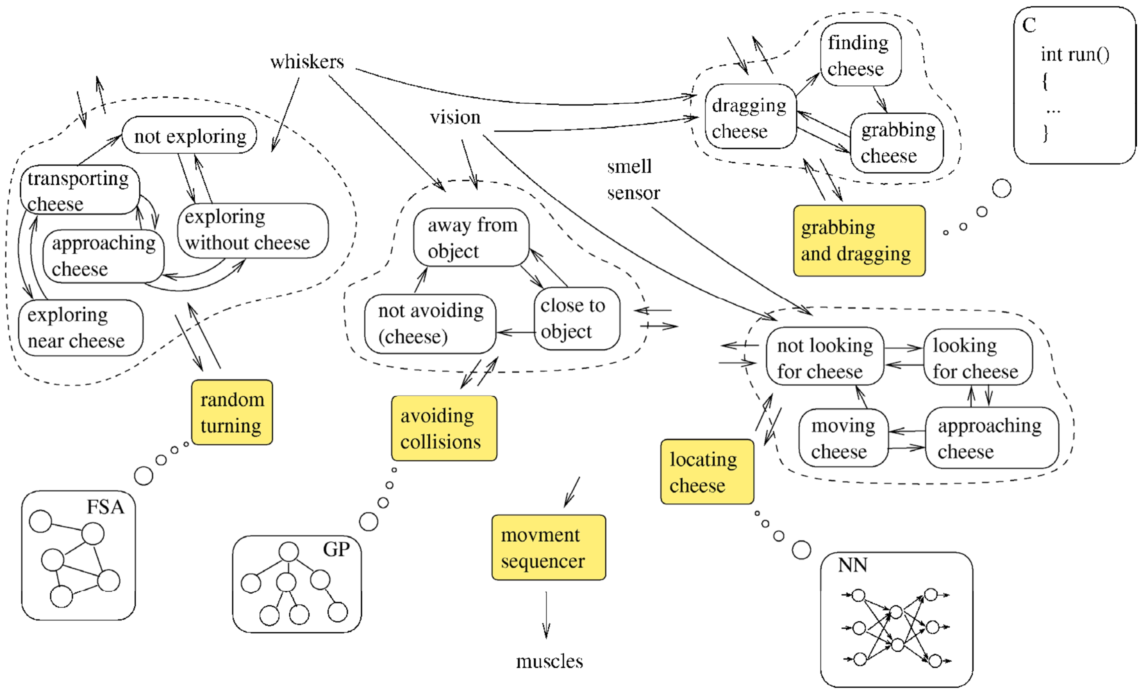


Figure 3. Example controller architecture for a mouse acquiring cheese task.

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. They simply react to a predefined message interface and produce status messages whenever their actions produce 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 machines[1], 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 the “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 3). The post-office modules are the communication interface competences with other competencies 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 post-office modules in our architecture are finite-state machines, but other languages or formalisms could be used in place, where the state transitions are triggered by incoming or outgoing messages, which may be transformed, or filtered. 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 finite-state automaton. 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 post-office module of the *random turning* competence follows with the events performed by other modules, and adjusts its state to represent the current sitation. Please refer to the section ‘Example task’ below for another specific example.

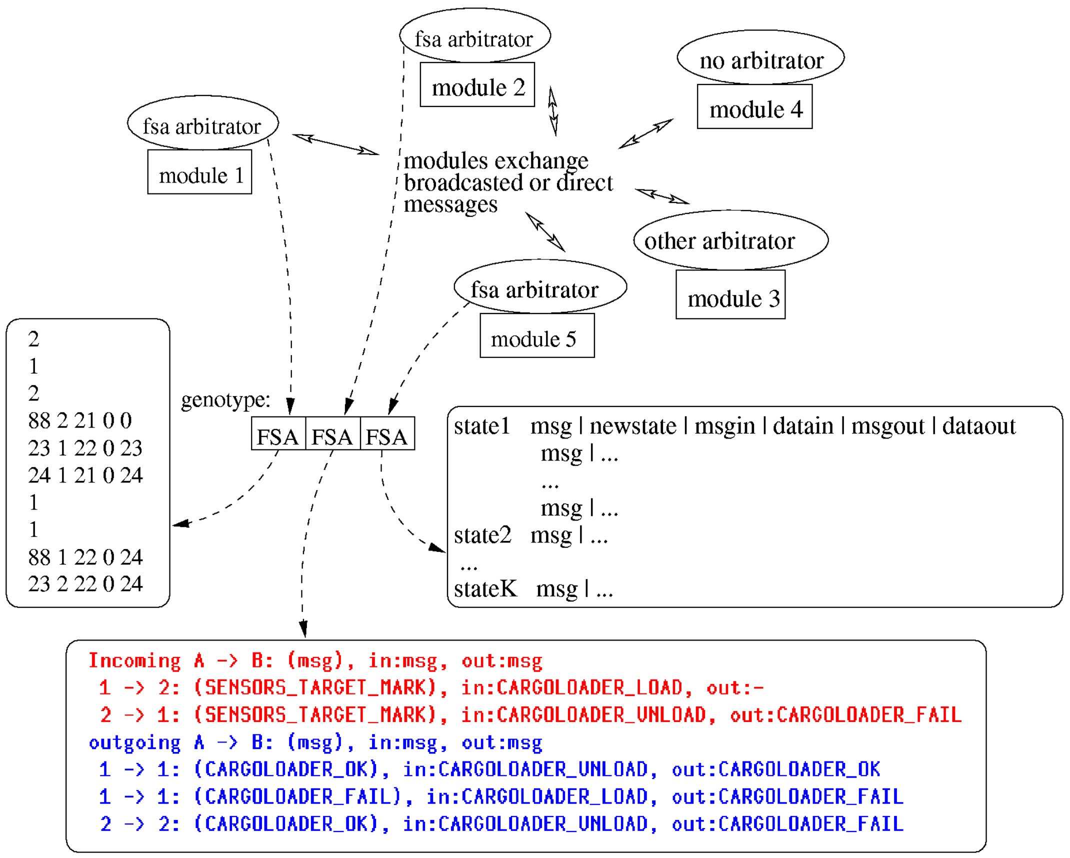
Evolutionary algorithm

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 automatons (FSAs). The genotype representation consists of blueprints of FSAs for the set of modules for which the FSAs are to be designed automatically (some modules might work without post-offices, other might use manually-designed post-offices, or some post-offices are held fixed because they are already evolved). An example of a genotype is in figure 4.

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 (please see figure 13 for examples, and the appendix for example of specification of the EA parameters including the specification of states, and transitions):

```
(msg_type, new_state, msg_to_send_out, [msg_arguments],
 msg_to_send_in, [msg_arguments])
```

[1] We use the terms finite-state machine and finite-state automaton interchangeably in this document.



```
2
1
2
88 2 21 0 0
23 1 22 0 23
24 1 21 0 24
1
1
88 1 22 0 24
23 2 22 0 24
```

```
Incoming A -> B: (msg), in:msg, out:msg
 1 -> 2: (SENSORS_TARGET_MARK), in:CARGOLOADER_LOAD, out:-
 2 -> 1: (SENSORS_TARGET_MARK), in:CARGOLOADER_UNLOAD, out:CARGOLOADER_FAIL
outgoing A -> B: (msg), in:msg, out:msg
 1 -> 1: (CARGOLOADER_OK), in:CARGOLOADER_UNLOAD, out:CARGOLOADER_OK
 1 -> 1: (CARGOLOADER_FAIL), in:CARGOLOADER_LOAD, out:CARGOLOADER_FAIL
 2 -> 2: (CARGOLOADER_OK), in:CARGOLOADER_UNLOAD, out:CARGOLOADER_FAIL
```

Figure 4. 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 as described above in the text), bottom oval shows symbolic representation as viewed by a viewer utility (used to analyze the evolved post-office modules); right oval shows the genotype structure for both incoming and outgoing messages for better explanation. Note that the outgoing messages in the viewed FSA are irrelevant when no other module is reacting to those particular types of messages.

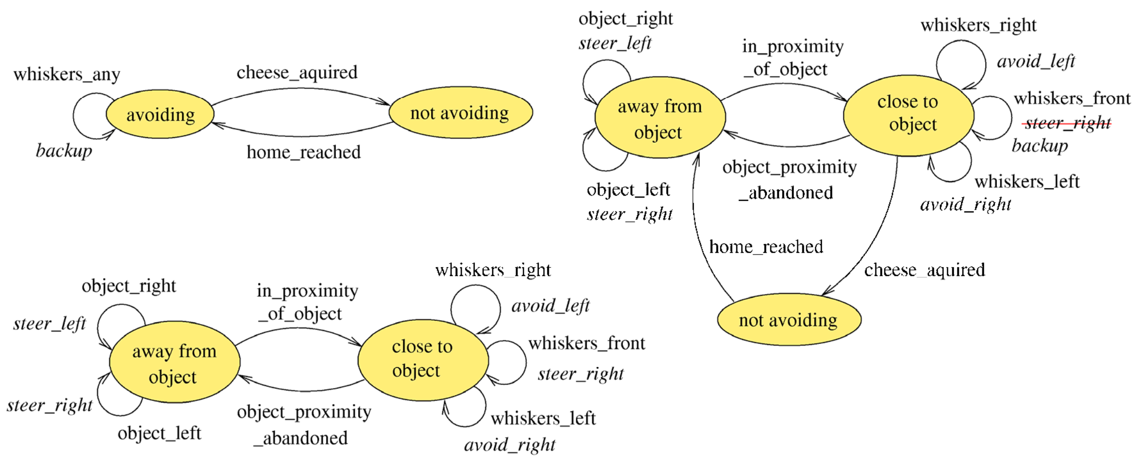


Figure 5. Example of crossover operator functionality (revisiting the mouse task). The two finite-state automata on the left are combined into a single resulting automaton (one of the two offsprings) on the right. The state "*not avoiding*" is inherited from the parent shown above left, together with both states of the parent shown below left. The states carry with them all their outgoing transitions from the parent to the offspring. The transition destinations are pointed to randomly chosen states of the part of the offspring inherited from the other parent, see text for details. The mutation is shown by striking line over the previous message label, where it was replaced by another message, see text for list of mutation types.

We use the standard genetic algorithm, 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 FSAs associated with each module. The remaining work is performed by the evolutionary algorithm.

The GA-*initialization operator* generates random FSAs that comply with the supplied specification. The *crossover operator* works on a single randomly selected FSA. It randomly divides states of the FSAs from both parents into two pairs of subsets, and creates two new FSAs by gluing the alternative parts together. A simple example is shown at figure 5, where two FSAs with partial functionality, each having 2 states, are combined by the crossover operator to form a new FSA that has three states. Later, the transition in the state labelled "*close to object"* is mutated: the message produced by the transition is changed from *steer_right* to *backup*.

The following paragraphs describe the crossover operator in detail.

Let the states of the first parent be $S = (S_1, \ldots, S_K)$, and the states of the second parent be $T = (T_1, \ldots, T_L)$. The operator randomly picks a set of states that will be inherited by the first offspring from the first parent, $O_{1fromS} \in S$, and a set of a possibly different cardinality, containing states that will be inherited by the second offspring from the second parent, $O_{2fromT} \in T$. The first offspring will then consist of states $O_{1fromS} + (T - O_{2fromT})$, and the second offspring will consist of states $O_{2fromT} + (S - O_{1fromS})$. The state transitions cannot be always preserved, because the number of states and their numbering changes. Figure 6 visualizes the crossover operator in a diagram.

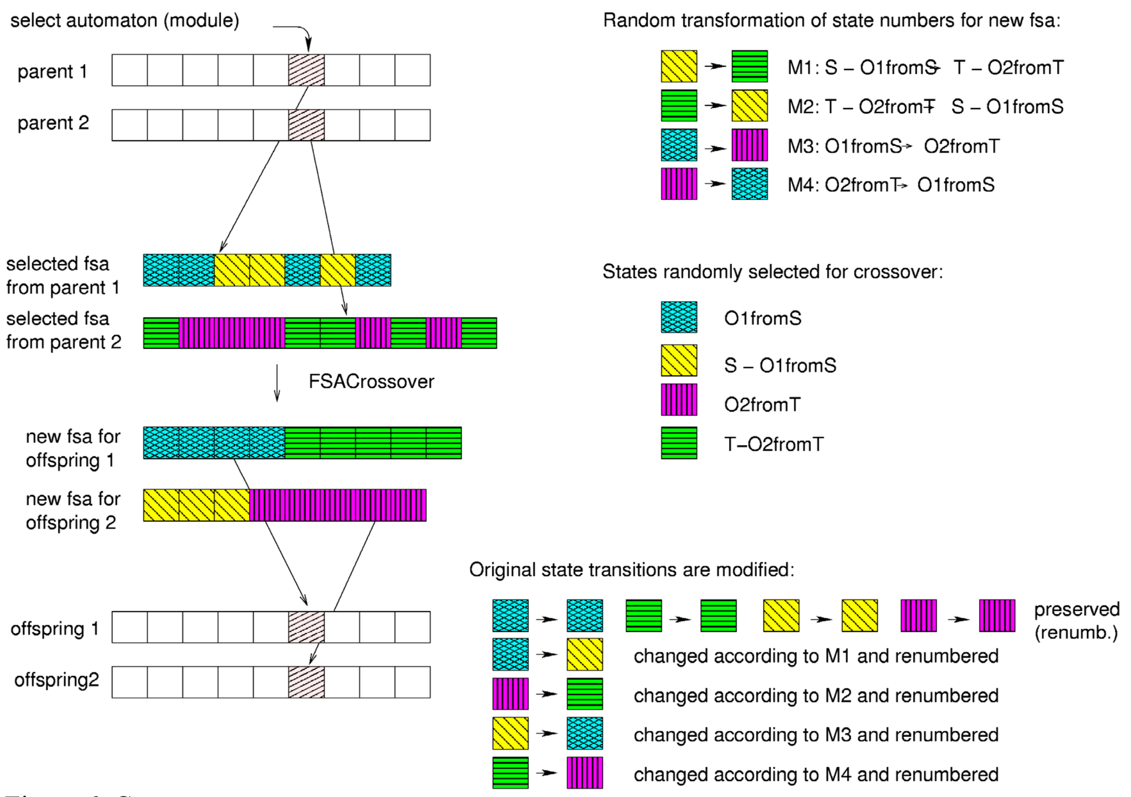

select automaton (module)
parent 1
parent 2
selected fsa from parent 1
selected fsa from parent 2
FSACrossover
new fsa for offspring 1
new fsa for offspring 2
offspring 1
offspring2
Random transformation of state numbers for new fsa:
M1: S – O1fromS → T – O2fromT
M2: T – O2fromT → S – O1fromS
M3: O1fromS → O2fromT
M4: O2fromT → O1fromS
States randomly selected for crossover:
O1fromS
S – O1fromS
O2fromT
T–O2fromT
Original state transitions are modified:
preserved (renumb.)
changed according to M1 and renumbered
changed according to M2 and renumbered
changed according to M3 and renumbered
changed according to M4 and renumbered

Figure 6. Crossover operator.

The following transitions are preserved: O_{1fromS} to O_{1fromS}, $S - O_{1fromS}$ to $S - O_{1fromS}$, O_{2fromT} to O_{2fromT}, $T - O_{2fromT}$ to $T - O_{2fromT}$. The states are renumbered according to the new state numbers.

Since the transitions leading to states that are now part of the other offspring would point nowhere, we randomly generate mappings between the exchanged states of the two offspring. Since the numbers of states in these four sets are different, bijection is not possible and we need all four mappings – one in each direction for both parts. Mapping M_1 of states in $S - O_{1fromS}$ to states in $T - O_{2fromT}$, mapping M_2 of states in $T - O_{2fromT}$ to $S - O_{1fromS}$, mapping M_3 of states in O_{1fromS} to O_{2fromT}, and mapping M_4 of states in O_{2fromT} to O_{1fromS}. The new numbers of states are taken into account when generating M_1, M_2, M_3, and M_4.

The transitions are then modified using the generated mappings, the mappings M_1 and M_4 are used to generate the first offspring, the other two to generate the second offspring:

Transitions that lead:

From $x \in O_{1fromS}$ to $y \in S - O_{1fromS}$ are changed to $M_1(y)$,
From $x \in O_{2fromT}$ to $y \in T - O_{2fromT}$ are changed to $M_2(y)$,
From $x \in S - O_{1fromS}$ to O_{1fromS} are changed to $M_3(y)$,
From $x \in T - O_{2fromT}$ to O_{2fromT} are changed to $M_4(y)$.

The above transformation is attempting to maximize the genetic information passed from the parents to offspring by conservative approach, where the transitions originally pointing to the same state will point to the same state also in the offspring.

The implementation of the crossover procedure:

1. choose the index of FSA to work on
2. randomly generate bits $O_{1fromS}[1..K]$ and $O_{2fromT}[1..L]$,
3. based on O_{1fromS} and O_{2fromT}, generate $S_{RENUM}[1..K]$ and $T_{RENUM}[1..L]$,
4. randomly generate $M_{13}[1..K]$, and $M_{24}[1..L]$ that represent M_1, M_2, M_3, and M_4,
5. form O_1 by copying states from O_{1fromS} and $T - O_{2fromT}$, and updating all transitions based on M_1, M_4, S_{RENUM}, T_{RENUM},
6. form O_2 by copying states from O_{2fromT} and $S - O_{1fromS}$, and updating all transitions based on M_1, M_3, S_{RENUM}, T_{RENUM}.

The *mutation operator* works upon a single FSA. One of the following operations is performed (the probabilities of the mutation types are parameters of the algorithm):

- a new random transition is created,
- random transition is deleted,
- 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,
- a random state is deleted as well as all its incident transitions,
- a random transition is modified: (one of its parts `new_state`, `message_type`, `msg_to_send_out`, `msg_to_send_in` is replaced by an allowed random value),
- a completely random individual is produced (this operator changes all FSAs),
- a random transaction is split in two and new state is created in the middle,
- the initial state number is changed.

In our experiments, we use roulette wheel and tournament selection schemes combined with steady-state or standard GA with elitism. Other parameters of the algorithm include (with these default values): *pcrossover* (0.3), *pmutation* (0.7), probabilities of all 8 mutation types that sum up to 1: *pnew_random_transition* (0.25). *pdelete_random_transition* (0.1), *pnew_state* (0.2), *prandom_state_deleted* (0.05), *prandom_transition_mutated* (0.25), *pnew_random_individual* (0.05*), psplit_transition* (0.05), *pchange_starting_state* (0.05); *population_size* (100), *number_of_generations* (60), *portion_of_population_to_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, detials in appendix.

Simulation framework

The aim is to design controllers for real robots. Our current testing hardware platform is the LEGO Robotics RCX equipped with 32KB RAM, up to 3 sensors and 3 motors, running programs built using GNU C compiler and binutils with LegOS [16]. Testing the performance of each individual in hardware would be completely infeasible, and therefore a simulator is essential. The objective function of our evolutionary algorithm evaluates individuals in simulation; each individual is started from several starting locations. We upload the final evolved controller on the real hardware to verify its real-world functionality.

Two standard approaches to simulation can be distinguished: in a discrete event simulation, the events can occur only at the start of a distinct unit of time during the simulation – events are not permitted to occur in between time units. The state of the system changes and is updated at discrete time events. A continuous simulation system is trying to model the exact behavior of the simulated system, its state, and the outcome of the simulation using mathematical formulas. Our approach falls more in the category of continuous simulation. The simulated system generates events at discrete time units (for example, start the motor A, determine the value of sensor 1), but its environment is continuous, and the events can occur at an arbitrary time (robot running against an obstacle, or over some pattern drawn on the floor). 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 simulated program, compute the current exact state of the simulated system by mathematical formulas, and determine the outcome of the interaction. The temporal granularity is thus limited only by the CPU or bus frequency. The simulated program is not interpreted by the simulator. It runs almost independently in the operating system of the simulating computer. However, 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 all the 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 precomputed 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.

None of the existing robot simulators satisfied our needs. We chose to implement our own simulator in language C, now available as open-source project [28]. The simulator is designed to cope with any program written in LegOS system, which controls an experimental robot with a compatible topology (figure 7 right). This allows us to use it both with our controller architecture, and with virtually any C-program that can control the robot. Small modifications would allow simulating robots with different topology.

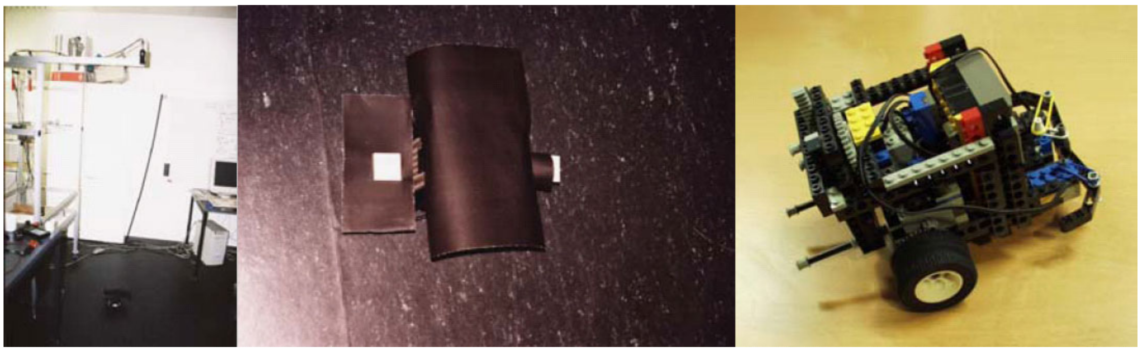


Figure 7. Camera setup for measuring the actual real-world outcome of the individual robot movements (left), the detail of the robot covered by black surface with 2 white marks detected by the calibrated software (centre), an experimental robot with high-lifting fork (right): its topology of the robot is compatible with a cylindrical shape with two independent motors propelling the wheels on the sides, one motor operating the high-lifting fork, front bumpers, and two light sensors pointing upwards and downwards.

There are several issues related to simulation. First of all, many researchers pointed out that simulating robotic systems accurately is almost impossible. Each sensor and motor part has somewhat different characteristics, and the outcome of each sensory or motor action depends on imperfect interactions with the real world. In order to achieve a comparable performance of the simulated and real robotic system, noise has to be applied both to sensory readings, and motor actions. In addition, the outcome of the motor action is hard to compute, and it appears to be more feasible to measure it and construct a table of basic motor actions and their outcomes as proposed by Miglino et.al. [14]. Figure 7 left shows a camera setup in the computer vision laboratory in Maersk institute in Odense that we used to measure the outcome of basic robotic actions in real-world. These values can be used to setup the simulator.

Another important issue is the execution speed of the simulated controller. By default, the controller runs at a real-time speed (1-to-1 ratio). Even though the CPU speed of the simulating hardware is higher than the CPU speed of RCX, the resulting behavior is compatible, since the modules of the controller are typically spending their time waiting for some event to occur to change their state in response. Obviously, the running speed on a fast simulating hardware can be increased. However, after some threshold, further speedup is impossible even though the CPU utilization remains about 0.0%. This threshold is reached when the frequency of events exceeds the response frequency of the controller. For instance, when the robot is crossing a line drawn on the floor, one of the modules must detect the line in order to turn the robot and make it follow the line. Once the controller misses the line, because the thread of the line follower module does not always get a time slice between the time the robot enters and leaves the line, the simulation speedup is too high – even though the controller still spends most of the time waiting for some event, and keeping CPU utilization very low[2]. Even though the accuracy of the simulator was somewhat compromised due to different CPU and OS architectures between the HW of the real robotic system and the simulating computer, and some of the delay constants used in the controller had to be adjusted for different speed-up ratios, we found the simulator accuracy satisfiable. More precise simulations could be achieved using a real-time operating system.

Incremental evolution

Having explained the tools, settings, and supporting methods, we reached the main focus of our work. Researchers observed in the past [6] 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 [6, 10] 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 (see [19] for an earlier overview of incremental evolution). 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 [12], 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 [6], where the gantry robot evolved forward movement first, followed by stages that required movement towards a large target, movement towards a small target (this would be again the change of environment), and distinguishing a triangle from square.

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 [13], and many later BB approaches.

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 Behavior

[2] Here it is interesting to note that upgrading from the old LinuxThreads to new pthreads library (NPTL) and utilizing the round-robin real-time scheduling in superuser mode allowed a speedup of more than one order of magnitude (100-500-times faster than real-time as contrasted to 10-times faster with older LinuxThreads).

Analysis and Training [22], 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.

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 [11], but automatic design of robot morphology is beyond the scope of this work. 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.

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.

Another important issue is how to transfer a population from the end of one incremental step to another step. 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 as an initial population in the next step. We therefore 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 from two or more previous incremental steps. In principle, our algorithm takes for each incremental step a full specification of blending, copying, and mutation ratios for all FSAs in the genome and all preceding incremental steps. In this way, one can design an evolutionary incremental process following a scenario with a topology of a complex oriented graph. However, in most of the situations, the incremental scenario does not have to be complicated and thus the number of the parameters to specify remains manageable.

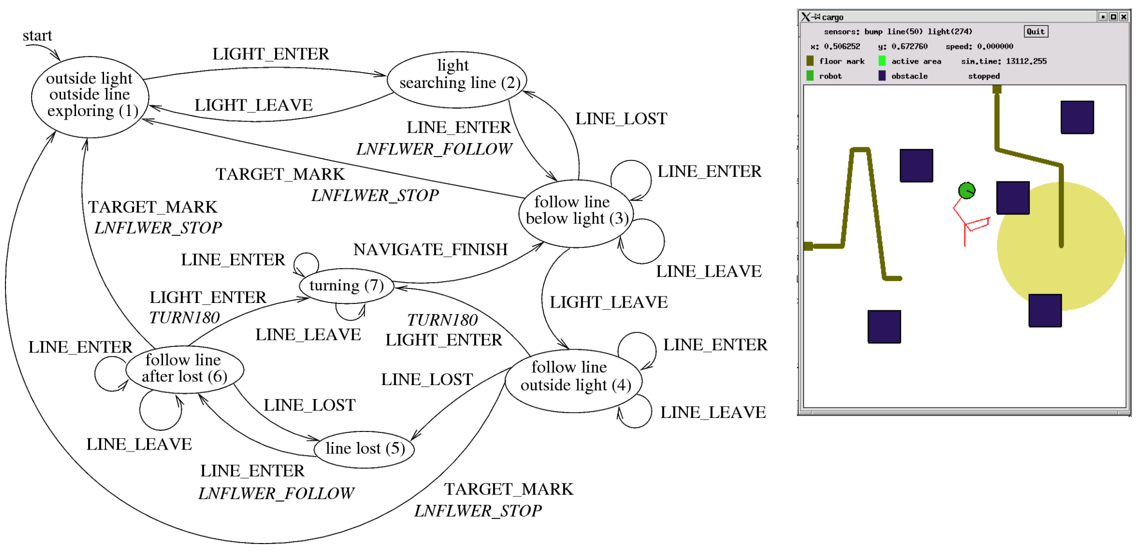
Distributed evolutionary computing

The simulated runs require extensive CPU resources, and thus we chose to utilize the idle CPU power of many machines around the campus. We have developed Q^2ADPZ, a specialized open-source system for distributed computing, which allows the user to submit jobs consisting of multiple tasks into a pool of idle CPUs that are running a daemon process (or Windows NT service) [4]. In addition, we make use of the computational cluster of our division [3], and finally, for the experiments running under real-time scheduling, we did setup a dedicated cluster of high-performance Linux machines [20]. A population of robot controllers is distributed over the computational cluster, where each node computes several fitness evaluations and submits the result over TCP/IP network and SQL database to the server running the main evolutionary algorithm. For our example task, we utilized more than 100 idle powerful 2.4GHz P4 desktop machines in the computer labs in the summer vacations period. The whole incremental scenario evolved the target behavior sucessfully in approximately 1 hour without any substantial attempts to optimize it – we were busy enough to see that the experiment delivered a positive result.

Example task

Our goal was to evaluate the proposed controller architecture and incremental evolution method and compare it to a manual design of the arbitration 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 7 right). 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 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). 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. Figure 8 right shows a screenshot of a simulator with an example environment.

The environment is defined by a configuration text file, which specifies the shape and size of the environment, robot, obstacle, floor drawings, loading and unloading stations as well as light position, and intensity. In addition, the simulator software allows defining simple control events based on (possibly periodic) time, robot heading, location, and fork and cargo positions.



sensors: bump line(50) light(274)
x: 0.506252 y: 0.672760 speed: 0.000000
🟫 floor mark 🟩 active area sim.time: 13112.255
🟩 robot 🟪 obstacle stopped

Figure 8. Robot executing the target task in a simulated environment (right), and an example of a manually designed fsa arbitrator for the line follower module (left). The illuminated area on the right is depicted by a large filled circle; dark squares represent obstacles; thick lines are drawn on the floor and detectable by robot; loading and unloading stations are marked by rectangles at the end of line, and the thin line shows the trajectory for an example run. The transitions between states on the left are labeled by messages that trigger them, and in *cursive* by messages they generate. The robot first travels randomly until it enters light (state 2). Then it looks for line and follows it in states 3 and 4, with recovery in states 5, 6, and 7. Whenever it reaches the loading station, it stops following the line and resets to random walk in state 1. Module *explore* queries the sensor module to see if the robot is moving towards or away from light and controls the turning of the robot in order to reach the light more quickly.

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.

Linefollower – 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.

Cargoloader – executes a procedure of loading and unloading cargo on demand: when the robot arrives to the cargo loading station, it has to turn 180°, 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 input and output message interface of all modules is shown in table 1. The arbitration mechanism, which is our focus, consists of FSA post offices attached to individual modules. Figure 8 left shows the hand-made FSA for the linefollower module. Other modules that use FSAs are *cargoloader*, *avoidance*, and *explore*.

We have successfully designed the arbitration using our incremental evolutionary algorithm. According to the Fitness Space guidelines proposed in [5], we attempted to keep the fitness function as implicit, internal, and behavioral as possible. In particular, in the later steps, we are only counting the number of correctly delivered cargo objects, whereas in the earlier steps, we measure the total distance traveled, the time the robot runs over the line, the quality of the line following (that is how much is the robot interacting with the motors and sensors while it follows the line), and how much time it spends colliding with obstacles. In addition, we favor FSAs with less states and transitions. Figure 9 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 4th and 5th incremental steps, the task and the environment were simplified, but the controller already contained all its functionality.

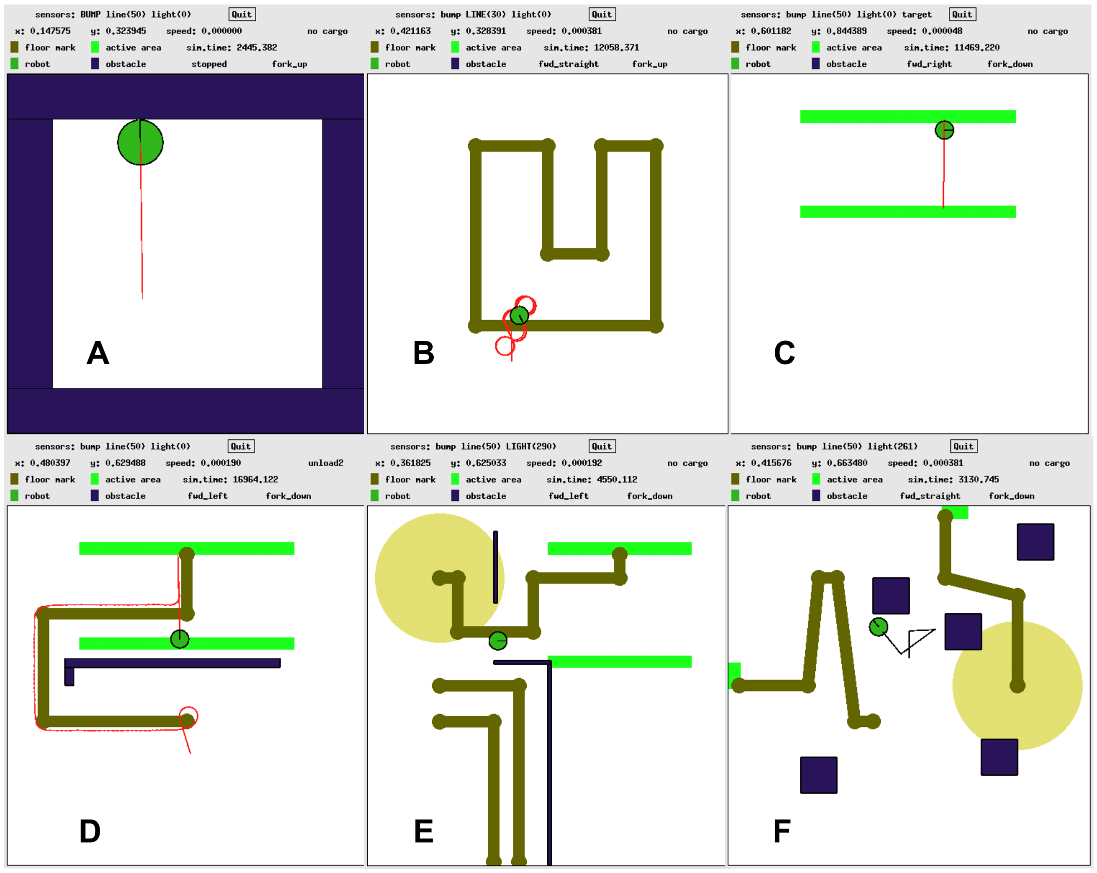
Module	**Recognized incoming messages**	**Generated outgoing messages**
Avoidance	AVOIDANCE_START SENSORS_BUMPERS_PRESSED SENSORS_BUMPERS_RELEASED	-
Sensors (Bumpertracker, Lighttracker, and Linetracker)	SENSORS_LIGHT_QUERY	SENSORS_BUMPERS_PRESSED
		SENSORS_BUMPERS_RELEASED
		SENSORS_LIGHT_ENTER
		SENSORS_LIGHT_LEAVE
		SENSORS_LIGHT_INCREASE
		SENSORS_LIGHT_DECREASE
		SENSORS_LIGHT_NOCHANGE
		SENSORS_LINE_ENTER
		SENSORS_LINE_LEAVE
		SENSORS_TARGET_MARK
Cargoloader	CARGOLOADER_LOAD CARGOLOADER_UNLOAD	CARGOLOADER_OK CARGOLOADER_FAIL
Explore	EXPLORE_START EXPLORE_STOP	-
Linefollower	LINEFOLOOWER_FOLLOW LINEFOLLOWER_STOP SENSORS_LINE_ENTER SENSORS_LINE_LEAVE	LINEFOLLOWER_LINELOST
Motordriver	MOTORDRIVER_POWER m, pwr MOTORDRIVER_NOPOWER m	-
Navigate	NAVIGATE_FORWARD t NAVIGATE_BACKWARD t NAVIGATE_RIGHT t NAVIGATE_LEFT t NAVIGATE_AROUNDLEFT t NAVIGATE_AROUNDRIGHT t NAVIGATE_TURNRND t NAVIGATE_STOP NAVIGATE_FAST NAVIGATE_SLOW	NAVIGATE_COMMAND_FINISHED NAVIGATE_COMMAND_INTERRUPTED NAVIGATE_BUSY
Beep	BEEP_BEEP x	
Console	CONSOLE_PRINT x	

Table 1. Message interfaces for behavioral modules defined by the module designer.

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:

$$m_n = \varphi \,.\, m_{n-1} + (best\_fitness_n - best\_fitness_{n-1})$$

where $best\_fitness_i$ is the fitness of the best individual in population i, φ is a constant (we used φ=0.2), and m_0 is initialized to 0.9 . $best\_fitness_0$.



```
sensors: BUMP line(50) light(0)   Quit
x: 0.147575  y: 0.323945  speed: 0.000000      no cargo
floor mark   active area   sim.time: 2445.382
robot        obstacle      stopped      fork_up

sensors: bump LINE(30) light(0)   Quit
x: 0.421163  y: 0.328391  speed: 0.000381      no cargo
floor mark   active area   sim.time: 12058.371
robot        obstacle      fwd_straight   fork_up

sensors: bump line(50) light(0) target   Quit
x: 0.601182  y: 0.844389  speed: 0.000048      no cargo
floor mark   active area   sim.time: 11469.220
robot        obstacle      fwd_right    fork_down

sensors: bump line(50) light(0)   Quit
x: 0.480397  y: 0.629488  speed: 0.000190      unload2
floor mark   active area   sim.time: 16964.122
robot        obstacle      fwd_left     fork_down

sensors: bump line(50) LIGHT(290)   Quit
x: 0.361825  y: 0.625033  speed: 0.000192      no cargo
floor mark   active area   sim.time: 4550.112
robot        obstacle      fwd_left     fork_down

sensors: bump line(50) light(261)   Quit
x: 0.415676  y: 0.663480  speed: 0.000381      no cargo
floor mark   active area   sim.time: 3130.745
robot        obstacle      fwd_straight   fork_down
```

Figure 9. Experimental environments for 6 incremental evolutionary steps of *creative* incremental scenario, A – F. 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 rewarding 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.

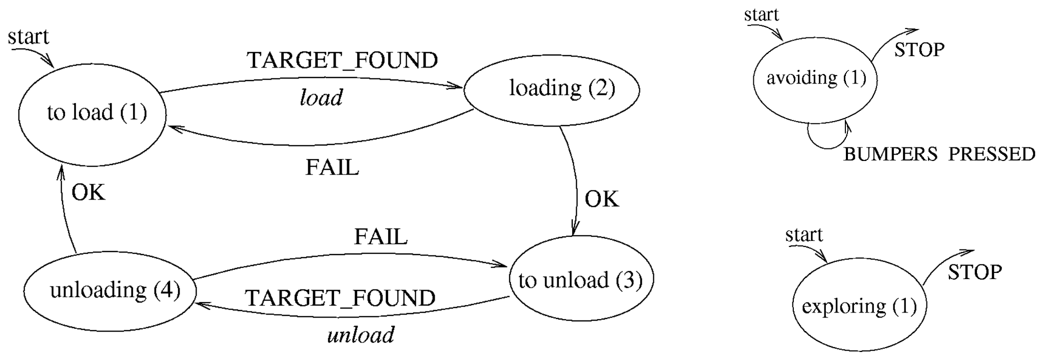


Figure 10. FSA arbitrators for modules cargoloader, avoidance, and explore. The avoidance FSA forwards to the module only BUMPERS_PRESSED message, while the explore FSA forwards to the module all messages.

Results

To verify the controller architecture and task solvability, we have first designed the post-office arbitrators manually. The most complex arbitrator was shown in the figure 8 (left), while the remaining three arbitrators were simpler and are shown in figure 10.

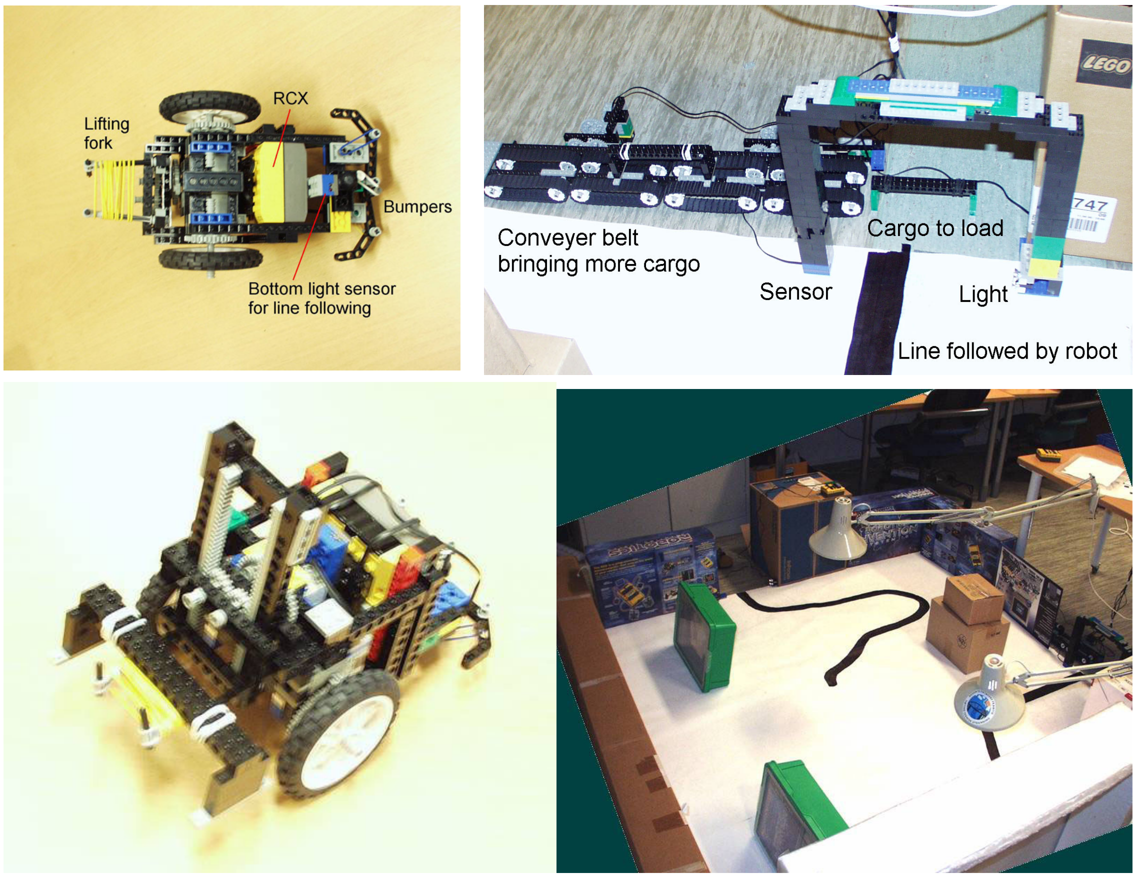

RCX
Lifting fork
Bumpers
Bottom light sensor for line following
Conveyer belt bringing more cargo
Cargo to load
Sensor
Light
Line followed by robot

Figure 11. Setup for the real robot: bottom view of the robot (top left), loading station (top right), robot carrying cargo (bottom left) and the environment with obstacles, lamps, loading and unloading stations. The cargo loading and unloading is handled automatically by two RCX modules that use the HiTechnic photo-sensor to detect the presence of the robot and send IR signal both to the robot and to the computer that switches the lights using the X10 lamp modules.

This controller performed well and resulted in reliable cargo delivery behavior. We have also tested the controller on the real robot. Figure 11 shows the real-world setup, robot and its environment. The transition to the real-world settings was straightforward, except of that 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. We experimented with a framework for obtaining a better correspondence as shown in figure 3. The robot with the diameter of 12 cm took ca. 430 seconds for completing the full task of a single loading-unloading sequence.

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 arbitrators 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.

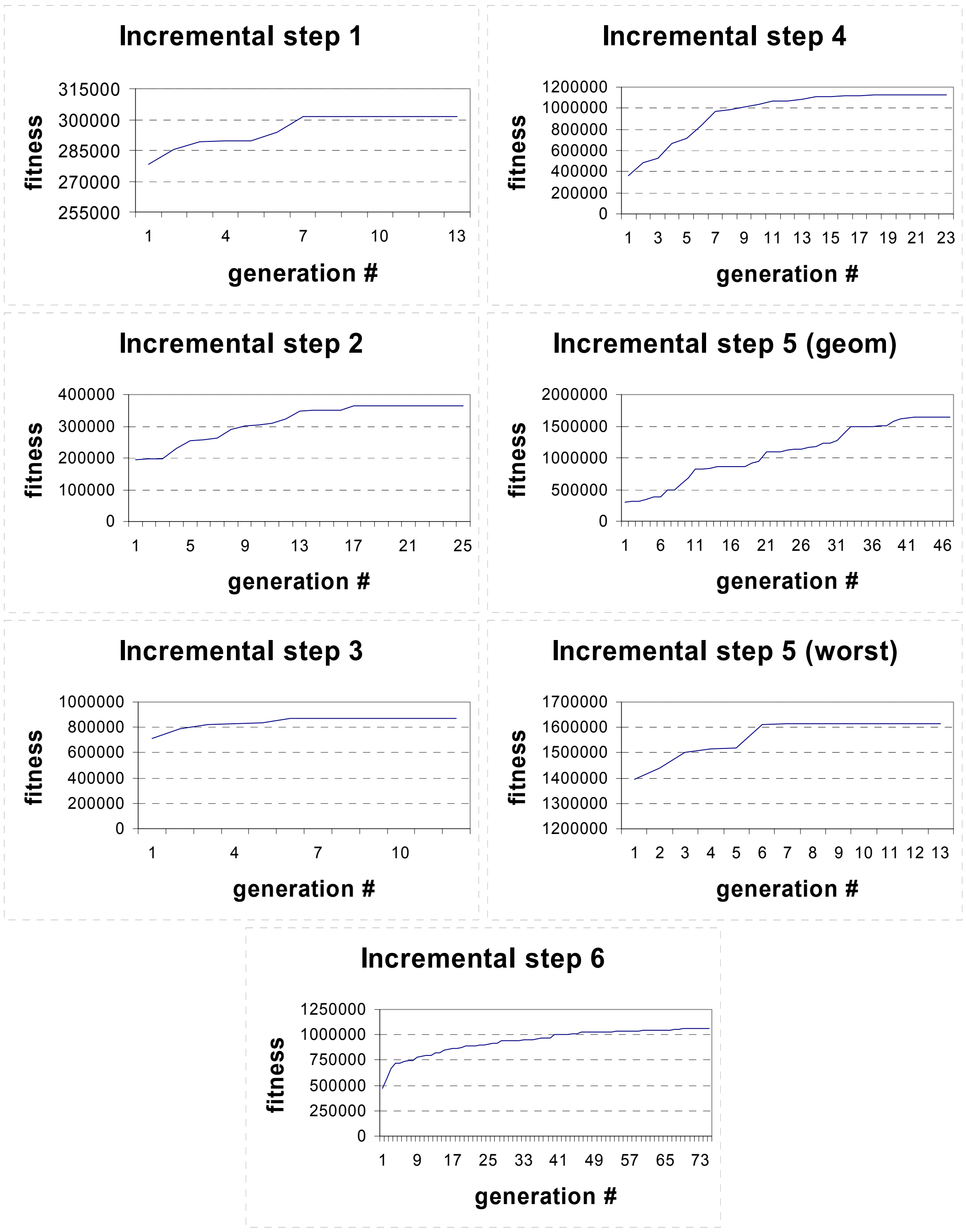


Figure 12. Best fitness for all incremental steps (average over 10 runs), creative incremental scenario. In the incremental step 5 (geom), the plotted fitness is an geometric mean of fitness achieved from the three different starting locations. This step was followed by 5 (worst), where the plotted fitness is the worst fitness achieved in the three runs from different starting locations.

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. All evaluated FSAs and the trajectories of best-fitness improving runs were saved to files and extensive logs were produced for further analysis.

Run \ Step	1	2	3	4	5	6	All
1	12 / 961	24 / 2850	9 / 813	15 / 3776	19 / 4561	9 / 1666	88 / 14627
2	9 / 740	22 / 2581	6 / 611	9 / 2477	58 / 12291	18 / 2924	122 / 21624
3	7 / 586	11 / 1483	7 / 709	12 / 3122	42 / 9353	42 / 6227	121 / 21480
4	5 / 450	16 / 1955	11 / 955	14 / 3618	52 / 11584	74 / 10761	172 / 29323
5	8 / 665	16 / 1953	6 / 614	12 / 3099	27 / 5998	9 / 1688	78 / 14017
6	7 / 562	15 / 1905	9 / 842	6 / 1852	24 / 5424	7 / 1378	68 / 11963
7	7 / 594	5 / 794	6 / 600	19 / 4570	23 / 5427	16 / 2599	76 / 14584
8	8 / 638	17 / 2048	7 / 683	22 / 5240	45 / 9867	46 / 6784	145 / 25260
9	11 / 855	17 / 2070	8 / 749	14 / 3495	56 / 12052	20 / 3218	126 / 22439
10	11 / 858	13 / 1663	6 / 610	18 / 4373	19 / 4526	25 / 3856	92 / 15886
Average	9 / 691	16 / 1930	8 / 719	14 / 3562	37 / 8108	27 / 4110	109 / 19120

Table 2. Generations/evaluations in each incremental step, 10 different simulated runs. The relation between the number of generations and the number of evaluations is not direct: the algorithm evaluates only new individuals, and among them only those that are not found in the cache.

Later experiments followed the *creative* scenario with 6 incremental steps shown in figure 9. Table 2 shows the number of evaluations used by each incremental step for 10 different runs. Figure 12 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 run 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 (so step 5 in table 2 refers to evaluations in both steps).

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 is only the final population of the directly preceding step. 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 third incremental step, where the line originally leading to the loading station has been 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.

```
1 -> 1: (AVOIDANCE_START), in:AVOIDANCE_START, out:-
1 -> 1: (SENSORS_BUMPERS_PRESSED), in:SENSORS_BUMPERS_PRESSED, out:-
1 -> 1: (SENSORS_BUMPERS_RELEASED), in:SENSORS_BUMPERS_RELEASED, out:-
```

```
1 -> 1: (SENSORS_LINE_ENTER), in:LNFLWER_FOLLOW, out:-
1 -> 2: (SENSORS_LINE_LEAVE), in:-, out:-
2 -> 2: (SENSORS_LINE_ENTER), in:SENSORS_LINE_LEAVE, out:-
2 -> 2: (SENSORS_LINE_LEAVE), in:SENSORS_LINE_ENTER, out:-
3 -> 1: (SENSORS_LINE_ENTER), in:LNFLWER_FOLLOW, out:-
```

```
1 -> 2: (SENSORS_TARGET_MARK), in:CARGOLOADER_LOAD, out:-
2 -> 1: (SENSORS_TARGET_MARK), in:CARGOLOADER_UNLOAD, out:-
```

```
1 -> 1: (SENSORS_LINE_ENTER), in:LNFLWER_FOLLOW, out:-
1 -> 2: (SENSORS_LINE_LEAVE), in:-, out:-
2 -> 2: (SENSORS_LINE_ENTER), in:SENSORS_LINE_LEAVE, out:-
2 -> 2: (SENSORS_LINE_LEAVE), in:SENSORS_LINE_ENTER, out:-
2 -> 5: (SENSORS_TARGET_MARK), in:LNFLWER_STOP, out:-
3 -> 3: (SENSORS_LINE_ENTER), in:SENSORS_LINE_LEAVE, out:-
3 -> 3: (SENSORS_LINE_LEAVE), in:SENSORS_LINE_ENTER, out:-
3 -> 1: (SENSORS_TARGET_MARK), in:-, out:-
4 -> 4: (SENSORS_LINE_ENTER), in:SENSORS_LINE_LEAVE, out:-
5 -> 3: (SENSORS_TARGET_MARK), in:-, out:-
```

```
1 -> 1: (SENSORS_LINE_ENTER), in:SENSORS_LINE_ENTER, out:-
1 -> 5: (SENSORS_TARGET_MARK), in:-, out:-
1 -> 3: (SENSORS_LIGHT_ENTER), in:LNFLWER_STOP, out:-
2 -> 2: (SENSORS_LINE_ENTER), in:SENSORS_LINE_LEAVE, out:-
2 -> 2: (SENSORS_LINE_LEAVE), in:SENSORS_LINE_ENTER, out:-
2 -> 5: (SENSORS_TARGET_MARK), in:LNFLWER_STOP, out:-
3 -> 2: (SENSORS_LINE_LEAVE), in:LNFLWER_FOLLOW, out:-
4 -> 4: (SENSORS_LINE_ENTER), in:SENSORS_LINE_LEAVE, out:-
4 -> 6: (SENSORS_LINE_LEAVE), in:SENSORS_LINE_ENTER, out:-
4 -> 5: (SENSORS_TARGET_MARK), in:LNFLWER_STOP, out:-
5 -> 5: (SENSORS_LIGHT_ENTER), in:-, out:-
5 -> 5: (SENSORS_TARGET_MARK), in:SENSORS_LINE_ENTER, out:-
6 -> 4: (SENSORS_LINE_LEAVE), in:SENSORS_LINE_LEAVE, out:-
```

```
1 -> 1: (SENSORS_LINE_ENTER), in:SENSORS_LINE_ENTER, out:-
1 -> 4: (SENSORS_TARGET_MARK), in:-, out:-
1 -> 5: (SENSORS_LIGHT_ENTER), in:LNFLWER_STOP, out:-
2 -> 2: (SENSORS_LINE_ENTER), in:SENSORS_LINE_LEAVE, out:-
2 -> 2: (SENSORS_LINE_LEAVE), in:SENSORS_LINE_ENTER, out:-
2 -> 4: (SENSORS_TARGET_MARK), in:LNFLWER_STOP, out:-
3 -> 3: (SENSORS_LINE_ENTER), in:SENSORS_LINE_LEAVE, out:-
3 -> 3: (SENSORS_LINE_LEAVE), in:SENSORS_LINE_ENTER, out:-
3 -> 4: (SENSORS_TARGET_MARK), in:LNFLWER_STOP, out:-
4 -> 4: (SENSORS_LINE_ENTER), in:SENSORS_LINE_ENTER, out:-
4 -> 2: (SENSORS_TARGET_MARK), in:-, out:-
4 -> 5: (SENSORS_LIGHT_ENTER), in:LNFLWER_STOP, out:-
5 -> 2: (SENSORS_LINE_LEAVE), in:LNFLWER_FOLLOW, out:-
5 -> 6: (SENSORS_LIGHT_LEAVE), in:SENSORS_LINE_ENTER, out:-
6 -> 4: (SENSORS_LINE_LEAVE), in:SENSORS_LINE_LEAVE, out:-
```

Transition tables describe transitions in format:

A → B: (m), in: m_1, out: m_2

Meaning that transition from state A to state B occurs, when message m arrives. Then message m_1 is sent to the module and message m_2 is broadcasted to other modules.

The relevant messages are:

AVOIDANCE_START

SENSORS_BUMPERS_PRESSED
SENSORS_BUMPERS_RELEASED

SENSORS_LINE_ENTER
SENSORS_LINE_LEAVE
SENSORS_TARGET_MARK

SENSORS_LIGHT_ENTER
SENSORS_LIGHT_LEAVE

LNFLWER_FOLLOW
LNFLWER_STOP

CARGOLOADER_LOAD
CARGOLOADER_UNLOAD

Figure 13. Finite-state machines evolved in each step.

We tried to evolve the target behavior with *sequential* incremental scenario without simplifying 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). Table 3 shows the number of evaluations performed in 10 different simulated runs with the *sequential* scenario. Figure 14 plots the best fitness.

Run \ Step	1	2	3	4	5	6	7	All
1	9 / 614	21 / 2966	39 / 8396	21 / 4298	43 / 8516	7 / 1072	7 / 1122	147 / 26984
2	5 / 393	42 / 5400	*48 / 6058*	*21 / 2849*	*65 / 8256*	7 / 1107	13 / 1903	201 / 25966
3	7 / 497	*44 / 2798*	*35 / 4850*	*29 / 3950*	**180 / 9592**	*20 / 1408*	*6 / 477*	321 / 23572
4	5 / 357	*40 / 2440*	*44 / 5826*	*44 / 5696*	**12 / 872**	*12 / 905*	*13 / 942*	170 / 17038
5	6 / 469	*43 / 2819*	*30 / 3701*	*76 / 9668*	**9 / 728**	*26 / 1666*	*71 / 4863*	206 / 23914
6	10 / 636	28 / 3736	38 / 7495	42 / 8254	*9 / 1368*	*6 / 992*	6 / 980	139 / 23461
7	10 / 641	55 / 6976	31 / 5828	24 / 4797	*28 / 3675*	9 / 1293	11 / 1626	168 / 24836
8	7 / 500	49 / 6356	62 / 11590	22 / 4696	*11 / 1685*	44 / 5539	19 / 2555	214 / 32921
9	5 / 393	42 / 5400	*48 / 6058*	*21 / 2849*	*65 / 8256*	7 / 1107	13 / 1903	253 / 25966
10	5 / 394	27 / 3164	47 / 9306	11 / 2648	15 / 2305	12 / 1747	26 / 3600	143 / 23164
Average	7 / 489	39 / 4206	42 / 6911	31 / 4971	44 / 4525	15 / 1684	19 / 1997	196 / 24782

Table 3. Evaluations in each incremental step for 10 different simulated runs (for the values in *cursive*, the population size was reduced from 200 to 100, or from 300 to 200; those in **bold** face ran with population 100 instead of 300). The parameters were varied for empirical testing.

To gain better understanding of underlying processes, we studied the contribution of the various mutation operators to the fitness improvements. The fitness of the offspring that was generated using each mutation operator was compared with the fitness of the parent, and the difference was stored. Figure 15 compares the relative contribution of the mutation operators in all incremental steps of the *sequential* scenario, that is with how large portion each operator contributed to the progress. Obviously, operator with higher rate has higher chances to contribute. On the contrary, figure 16 looks only at the individual performances of all single operators, and plots the ratio of the cases with positive and negative fitness change that resulted from operator application.

Discussion

Multi-agent control architectures are the focus of contemporary research. Our robot controller architecture is inspired by the principle of independent robot competencies performed, or being ready to be activated, in parallel. This principle is prevalent in Behavior-Based Robotics approaches. In our case, the behaviors communicate by sending addressed or broadcasted message signals possibly containing data. The implementation relies on shared memory locations between the individual threads of computation, and the messagepassing interface function calls achieve high efficiency (usually no data is copied; rather the functions with arguments refering to common memory space are called). Our framework for incremental evolution allows a general design of the incremental evolutionary scenario: the populations from several steps can be combined together in different ratios for each particular FSA. In addition, it is possible to specify a general "incremental" function, which specifies a termination criterion in each incremental step, i.e. a required condition for proceeding to the next incremental step. Incremental function can contain the following variables: *gennum*, the current generation number; *gennum-thisstep*, current generation of this step; *best-fitness*, the best fitness of the last generation; *avg-fitness*, average fitness of the last generation, *learning-momentum*, which is the current learning rate with history; *num-eval*, total number of evaluations so far; *num-eval-thisstep*, number of evaluations since the beginning of this step, *total-gennum*, total planned number of generations, *real-runtime*, real time since application start in seconds. These variables can be combined with numeric constants, binary operators and predicates.

Each incremental step has its own definition file for the environment, separate fitness function, structure of the controller, and describes the robot topology. Despite this generality, we concluded to several guidelines for designing the incremental scenarios:

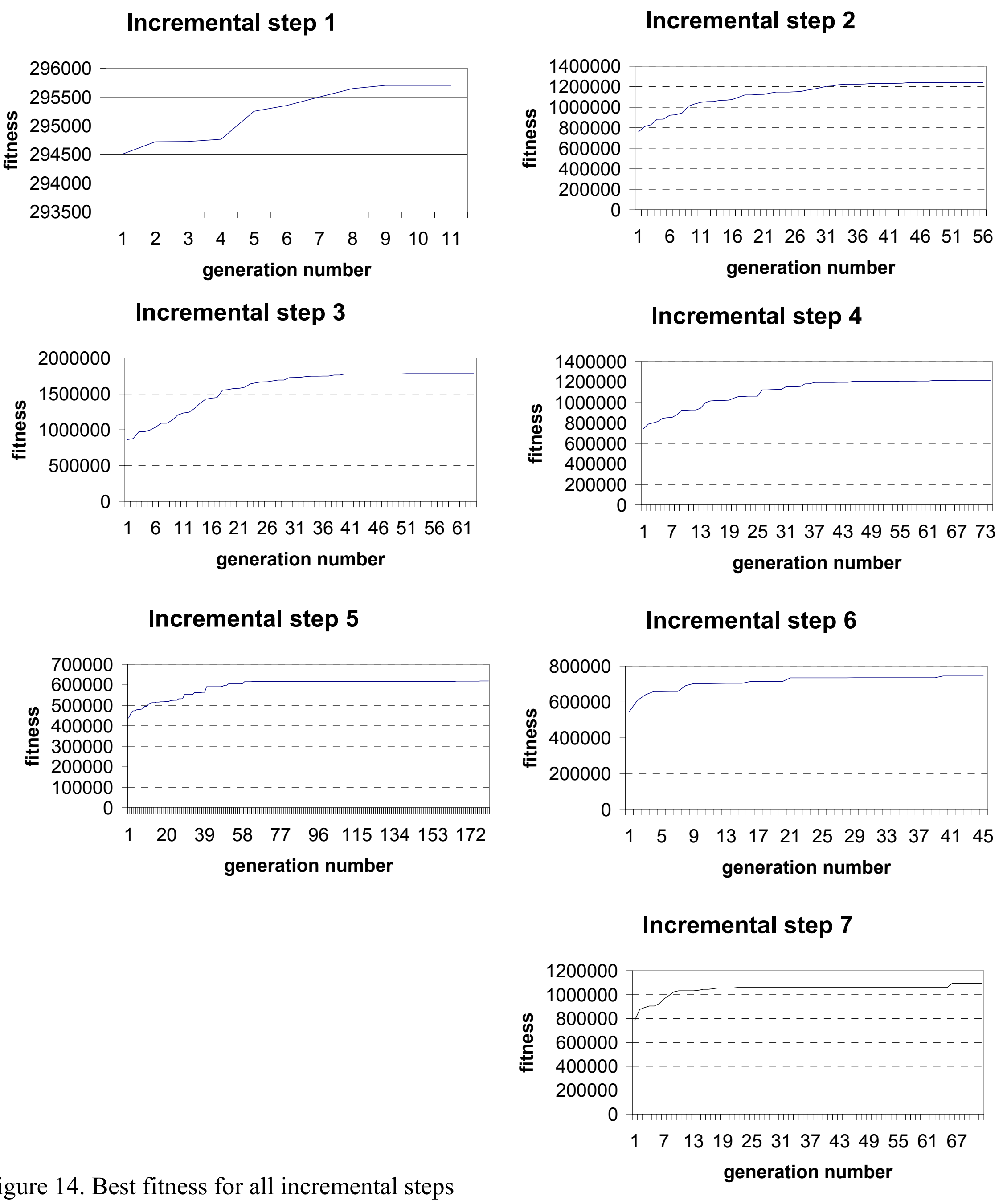


Figure 14. Best fitness for all incremental steps (average over 10 runs) with *sequential* scenario.

1. Individual incremental steps ought to be as focused as possible. If a certain competence required for the target behavior can be learned completely in some incremental step, it should not be the subject of further learning and improvement in other incremental steps, as it would only increase its complexity multiplicatively.

2. Each incremental step should focus on a relatively narrow and well-identifiable competence. Avoiding evolving of multiple competencies at the same time is essential. If some competence requires a cooperation of multiple competencies, this itself has to be identified, and formulated as a distinguished, well-defined competence. .

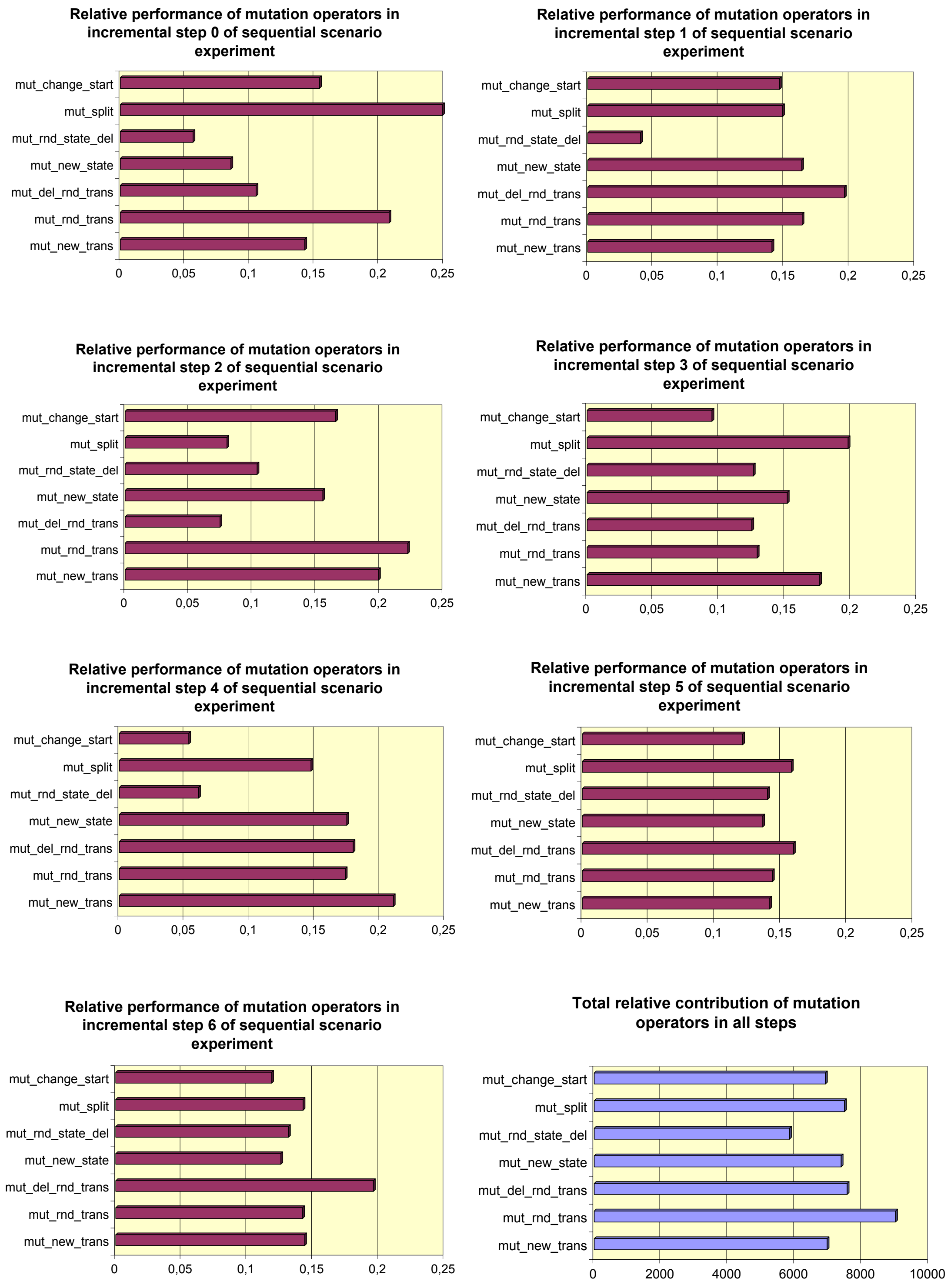
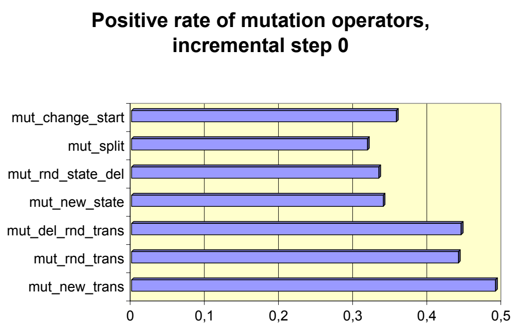
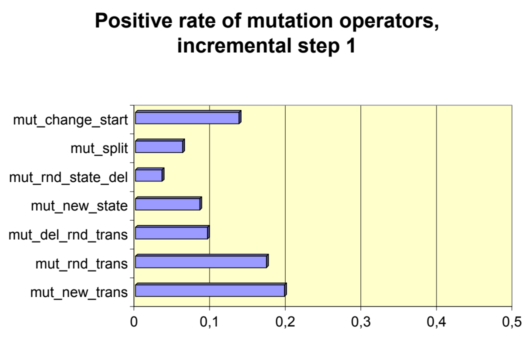
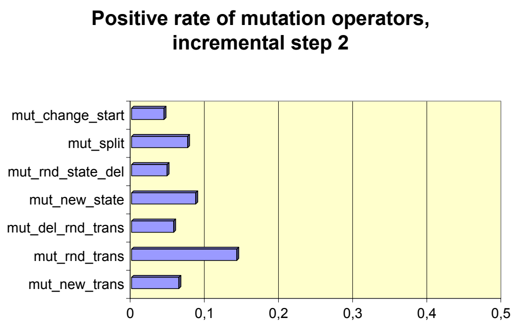
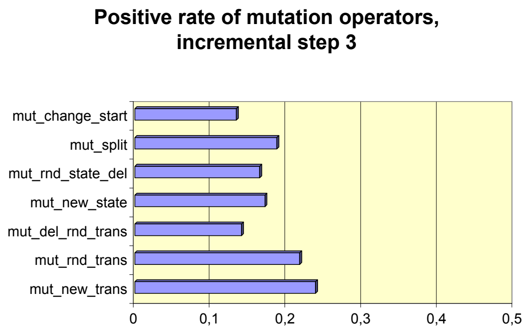


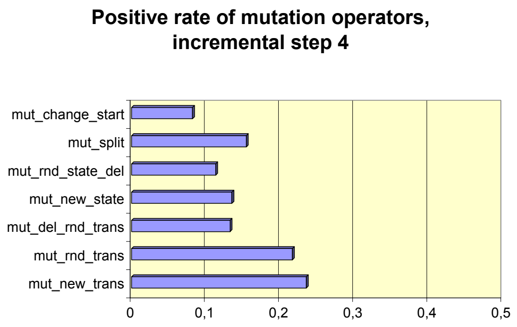
Figure 15. Contribution of mutation operators to evolutionary progress in all incremental steps of a single run of the *sequential* experiment. Due to different mutation rates, the absolute contribution of mutation operators was scaled by the number of operator applications: in total 3970, 3321, 3951, 13329, 2974, 21857, 17773 applications of *mut_change_start*, *mut_split*, *mut_rnd_state_del*, *mut_new_state*, *mut_del_rnd_trans*, and *mut_new_trans* operator resp. Low success rate of *rnd_state_del* in earlier steps is due to the lower and upper limits on number of states. Steps introducing higher complexity in task benefit from *mut_split*, when an extra state is smoothly added, extending one transition to two steps. The totals graph show that most of the work is done by simple mutation of a single transition *mut_rnd_trans*, however all operators contribute significantly in more than one incremental step.

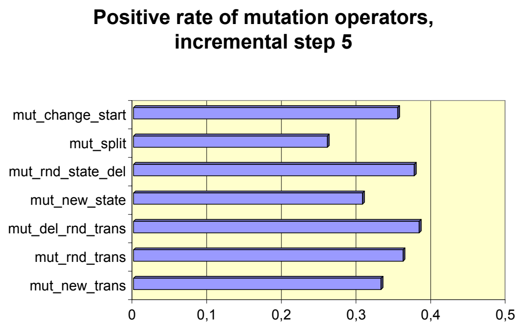












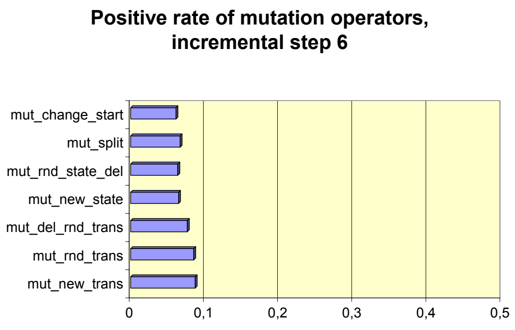
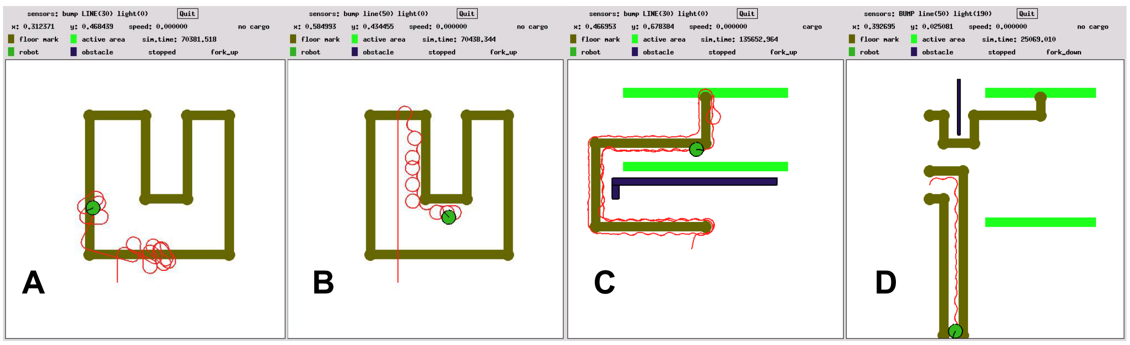


Figure 16. Positive rate of mutation operators expresses how often the operator generated a positive fitness increase. In the first step (step 0), the total number of evaluations was low, therefore high positive rate was achieved. In the fifth step, a very large potential for improvement existed. The deletion operators (*mut_del_state*, *mut_del_trans*) show lower rate as they often harm useful parts of automata.

3. Modifying the environment in order to create training situations for the robot is a very efficient method of devising simpler incremental steps. Modifying the objective function only, while keeping the same environment, is more challenging, and often leads to multitudes of false behaviors. Evolution tends to discover unexpected tricks due to the large number of possible interactions in a typical target environment. Figure 17 shows few samples of incorrect evolved behaviors.

4. The early apparent suggestive decomposition of the task is often not necessarily the most efficient way of task decomposition for incremental evolution. The average number of evaluations was significantly lower in the case of more elaborate task decomposition as shown in the results section of this work.


```
sensors: bump LINE(30) light(0)        Quit
x: 0.312371   y: 0.468439   speed: 0.000000        no cargo
🟫 floor mark  🟩 active area   sim.time: 70381.518
🟩 robot       🟦 obstacle        stopped        fork_up
```

```
sensors: bump line(50) light(0)        Quit
x: 0.584993   y: 0.434455   speed: 0.000000        no cargo
🟫 floor mark  🟩 active area   sim.time: 70438.344
🟩 robot       🟦 obstacle        stopped        fork_up
```

```
sensors: bump LINE(30) light(0)        Quit
x: 0.466953   y: 0.678384   speed: 0.000000        cargo
🟫 floor mark  🟩 active area   sim.time: 135652.964
🟩 robot       🟦 obstacle        stopped        fork_up
```

```
sensors: BUMP line(50) light(190)        Quit
x: 0.392695   y: 0.025081   speed: 0.000000        no cargo
🟫 floor mark  🟩 active area   sim.time: 25069.010
🟩 robot       🟦 obstacle        stopped        fork_down
```

Figure 17. Examples of evolved misbehaviors demonstrating richness of controllers with FSA arbitrators evloved 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 wished 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 earlier environment viewer version).

5. Special care has to be taken to prevent the individuals from gaining fitness by random coincidences without performing the required task. If the populations with several hundreds individuals evolve for hundreds of generations, even very unlikely events do happen and if the EA uses elitism, such faulty, but lucky behavior might completely push all promising individuals out of the population. Such events can, to a limited extent, be prevented by multiple starts from all starting locations. This, however, increases the total evaluation time for each individual. In our experiments, we used four different methods of calculating the individual fitness from the scores gained in all starting locations: arithmetic mean, geometric mean, worst fitness, and combination. The worst fitness has to be used, if we want to set a hard constraint and require that the individual performs the required behavior consistently and reliably. It should be used especially when the objective function permits a very high score in lucky situations – thus both mean and geometric mean thresholds could pass an individual that would solve only some of the runs, accidentally with a luckily high score (please note, that the geometric mean would not suffer from this problem if the unsuccessful runs would receive zero-fitness, that is, however, not the case, because the objective function typically rewards multiple aspects of the task, and thus it is almost never 0). In case of the combined option, we use the geometric mean for all runs from the same starting location, but the worst fitness among all those means for all starting locations. In other

words, the robot has to solve each starting location, but its performance does not have to be completely stable.

Many tasks (including ours) comprise a high degree of randomness. The same individual can gain fitness values that vary often more than 10%. This can lead into an illusory fitness improvement while the quality of the individuals does not change – or even decreases slightly, especially if elitism is in use: a small change introduced to an individual will result in a new individual, which by a lucky coincidence gains higher fitness and replaces the old individual in the population. This makes it also more difficult for newly found individuals, which introduce changes in a good direction, to steer the evolution away, as there are already many individuals who have the more lucky fitness for their real quality in the population, especially if the evolution stagnated for some time already. For this reason, it is recommended to evaluate all individuals over in each generation. Unfortunately, in our case, this would exceed out practical CPU limits, but it remains for future work to study how much this issue hinders the evolution of arbitrators.

Sometimes, one cannot avoid transferring certain evolved features to succeeding evolutionary steps. There is a high risk that the new incremental step will quickly move in its search away from the evolved features, which thus become easily forgotten. A remedy for this situation would be a continuous flow of individuals from previous incremental step, however it remains for the future work to evaluate this strategy.

It seems though that an insight of a human expert knowing the details of the robot hardware and software is still required in order to design a functional incremental decomposition. Future work should focus on eliminating these and making the process of creating the controller as automatic as possible.

Three advanced evolutionary techniques were applied in order to improve the evolutionary search process:

1. The population was automatically reinitialized each time it converged prematurely;

2. The states and transitions, which were never entered during the evaluation by the fitness function, could optionally be removed automatically. On one hand, this leads into much more readable and concise FSAs, on the other hand, it reduces the evolvability as the overhead genetic material present in the population becomes forbidden, and thus the population can get stuck in local extremes more easily. Removing unused transition also helps to win the useful code over bloat that can prevent the evolution from progressing at any reasonable pace.

3. Fitness cache implemented using SQL database helped to decrease the overall time required to reach the solution.

Conclusions and Future Work

We have designed a parallel, distributed behavior-based controller architecture where individual modules communicate by sending messages. Modules implement simple behaviors and are usually hand-coded, or optionally evolved. Messages are processed by post-offices attached to modules in order to filter relevant messages and adapt the behavior of the modules with independent behavioral responsibility to the purpose of the combined controller thus achieving efficient synergic behavior coordination. On example task of cargo delivery, we have demonstrated a successful design both manually and using automatic method based on incremental evolutionary algorithm that evaluates the individuals in simulation. The evolved individuals representing the arbitrators, i.e. the post-offices of all modules, take form of finite-state automatons.

The incremental method is based on dividing the target task into multiple tasks of increasing complexity (a partial order relation). Evolved populations with individuals that successfully perform more simple tasks are transformed and combined into initial populations for more complex tasks thus making it possible to evolve complex behaviors, which could not be automatically programmed otherwise.

The computational demand of the algorithm is high and requires utilization of distributed computational power. We have used three various systems for distributed computation to harness the idle CPU power of university student lab computers and cluster. Further directions of this work are several. A natural-language interface for specifying the target task would allow to program robot for complex task by giving description in human language. This would also require work on automatic partitioning of the target task into incremental steps, and an AI system that could achieve that by understanding the semantic descriptions of the basic robot capabilities. Automatic programming of multiple groups of robots is a straight extension of this work. Improvements in the simulation techniques, using real-time operating systems, simulating more realistic environments, as well as implementing the robot simulator and evolutionary algorithm directly on the robot hardware remain for further studies. Other representations in addition to FSA taking the role of behavior arbitrators should be investigated. The crispness of FSAs can be alleviated by making them more continuous: either by transitions that occur with certain probability, or by messages that would be probability vectors for all message types, or by staying in multiple states at the same time with distributed probability. This is our connection to the fields of Hidden Markov Models and neural networks, however it remains to be investigated whether such representations apply smoothly to the crisp robotic tasks constraint by the discrete robot body, actuators, and world interactions, which are probably modeled best by discrete states.

References

[1] Arkin R C (1998) *Behavior-Based Robotics*. MIT Press, Cambridge, MA.
[2] Beer R D, Gallagher J C (1992) *Evolving dynamical neural networks for adaptive behavior*. Adaptive Behavior, 1, 91-122.
[3] Cassens J, Constantinescu Z (2003) *It's Magic: SourceMage GNU/Linux as a High Performance Cluster OS*. (abstract). LinuxTag 2003 Conference, July 10-13, Karlsruhe.
[4] Constantinescu Z, Petrovic P (2002) *Q2ADPZ, An Open Source, Multi-platform System for Distributed Computing*, ACM Crossroads Student Magazine, 9, 2.
[5] Floreano D, Urzelai J (2000) *Evolutionary Robots with on-line self-organization and behavioral fitness*. Neural Networks, 13, 431-443.
[6] Harvey I (1995) *The Artificial Evolution of Adaptive Behaviours*. DPhil Thesis, University of Sussex, 1995.
[7] Hillis W D (1990) *Co-evolving parasites improve simulated evolution as an optimization procedure*. Physica D 42:228-234.
[8] Humphrys M (1997) Action Selection methods using Reinforcement Learning. PhD thesis, University of Cambridge.
[9] Lau K W, Tan H K, Erwin B T, Petrovic P (1999). *Creative Learning in School with LEGO Programmable Robotics Products*. Proceedings to Frontiers in Education'99.
[10] Lee W, Hallam J, Lund H H (1998) *Learning Complex Robot Behaviours by Evolutionary Computing with Task Decomposition*. In Lecture Notes in Computer Science, Volume 1545, p. 155.

[11] Lund H H (2001) *Co-evolving Control and Morphology with LEGO Robots*. In Hara and Pfeifer (eds.) Proceedings of Workshop on Morpho-functional Machines, Tokyo. Springer-Verlag.
[12] Lund H H, Miglino O (1998) *Evolving and Breeding Robots*. in Proceedings of First European Workshop on Evolutionary Robotics, Springer-Verlag.
[13] Maes P (1990) *How to do the right thing*. Connection Science Journal, Special Issue on Hybrid Systems, 1.
[14] Miglino O, Lund H H, Nolfi S (1995) *Evolving Mobile Robots in Simulated and Real Environments*, Artificial Life, 2, 4, 417-434.
[15] Nilsson, N J (1980) *Principles of Artificial Intelligence*. Palo Alto: Tioga Press.
[16] Noga M L (1999) *Designing the legos multitasking operating system*. Dr. Dobb's Journal, Miller Freeman Inc. November 1999.
[17] Nolfi S, Floreano D (2001) *Evolutionary Robotics. The Biology, Intelligence, and Technology of Self-organizing Machines.* MIT Press, Cambridge, MA.
[18] Ostergaard E H (2000) *Co-Evolving Complex Robot Behaviour*, Masters Thesis, University of Aarhus.
[19] Petrovic P (1999) *Overview of Incremental Evolution Approaches to Evolutionary Robotics*. In Proceedings to Norwegian Conference on Computer Science, 151-162.
[20] Petrovic P (2004) *Distributed System for Evolutionary Robotics Experiments*, IDI Technical report 05/04, Norwegian University of Science and Technology, Trondheim.
[21] Pirjanian P (1999) *Behavior Coordination Mechanisms – State-of-the-art*. Technical Report IRIS-99-375, Institute of Robotics and Intelligent Systems, School of Engineering, University of Southern California, October 1999.
[22] Sutton R S, Barto A G (1998) *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.
[23] Urzelai J, Floreano D, Dorigo M, Colombetti M (1998) *Incremental Robot Shaping*. Connection Science, 10, 341-360.
[24] Handyboard, http://handyboard.com/
[25] Microbric robotics set, http://www.microbric.com/
[26] Parallax robotics, http://www.parallax.com/html_pages/robotics/
[27] InnoC, Simple Sensor Networks, http://www.innoc.at/
[28] Eval homepage at NTNU, http://www.idi.ntnu.no/grupper/ai/eval/

Appendix A – Example project files

```
# sequential83_e.prj  - defines parameters of the evolutionary algorithm
# number of incremental steps (this many INCREMENTAL BLOCKS appear below AND
#  how many values are in each category marked INCR_VECTOR below)
7

# probability of the crossover (INCR_VECTOR - for each incremental step 
different
#  value in another row)
0.3
0.3
0.3
0.3
0.3
0.3
0.3

# probability of mutation (per individual) (INCR_VECTOR)
0.7
0.7
0.7
0.7
0.7
0.7
0.7

#### detailed mutation probabilities for all operators (INCR_VECTOR all)
#  MUT_NEW_RANDOM_TRANSITION
0.25
0.25
0.25
0.25
0.25
0.25
0.25
#  MUT_DELETE_RANDOM_TRANSITION
0.1
0.1
0.1
0.1
0.1
0.1
0.1
#  MUT_NEW_STATE
0.2
0.2
0.2
0.2
0.2
0.2
0.2
#  MUT_RANDOM_STATE_DELETED
0.05
0.05
0.05
0.05
0.05
0.05
0.05
#  MUT_RANDOM_TRANSITION_MUTATED
0.25
0.25
0.25
0.25
0.25
0.25
```

```
0.25
#  MUT_NEW_RANDOM_INDIVIDUAL
0.05
0.05
0.05
0.05
0.05
0.05
0.05
#  MUT_SPLIT_TRANSITION
0.05
0.05
0.05
0.05
0.05
0.05
0.05
#  MUT_CHANGE_STARTSTATE
0.05
0.05
0.05
0.05
0.05
0.05
0.05

# p_trim  (probability of trimming all states and transitions that were not
used) (INCR_VECTOR)
0.5
0.5
0.5
0.5
0.5
0.5
0.5

# p_newinlast (probability of creating new transition in the state which was
terminal) (INCR_VECTOR)
0.5
0.5
0.5
0.5
0.5
0.5
0.5

# number of individuals (population size)  (INCR_VECTOR)
100
200
300
300
300
200
200

# number of generations (total - all steps together)
600

# portion of population to replace (INCR_VECTOR)
1.0
1.0
1.0
1.0
1.0
1.0
1.0

# selection type (1 = RouletteWheel, 2 = Tournament)
```

```
1
1
1
1
1
1
1

# total number of modules in the controller (INCR_VECTOR)
10
10
10
10
10
10
10

# module message data in file
project/cargo/modules/alldata.dat

# module specification files for all modules - the modules must be listed
#  in the order of their module ids!  (INCR_VECTOR)
project/cargo/modules/navigate.mod
project/cargo/modules/cargoloader.mod
project/cargo/modules/motordriver.mod
project/cargo/modules/lnflwer.mod
project/cargo/modules/avoidance.mod
project/cargo/modules/explore.mod
project/cargo/modules/bumpertracker.mod
project/cargo/modules/linetracker.mod
project/cargo/modules/lighttracker.mod
project/cargo/modules/beep.mod
#
project/cargo/modules/navigate.mod
project/cargo/modules/cargoloader.mod
project/cargo/modules/motordriver.mod
project/cargo/modules/lnflwer.mod
project/cargo/modules/avoidance.mod
project/cargo/modules/explore.mod
project/cargo/modules/bumpertracker.mod
project/cargo/modules/linetracker.mod
project/cargo/modules/lighttracker.mod
project/cargo/modules/beep.mod
#
project/cargo/modules/navigate.mod
project/cargo/modules/cargoloader.mod
project/cargo/modules/motordriver.mod
project/cargo/modules/lnflwer.mod
project/cargo/modules/avoidance.mod
project/cargo/modules/explore.mod
project/cargo/modules/bumpertracker.mod
project/cargo/modules/linetracker.mod
project/cargo/modules/lighttracker.mod
project/cargo/modules/beep.mod
#
project/cargo/modules/navigate.mod
project/cargo/modules/cargoloader.mod
project/cargo/modules/motordriver.mod
project/cargo/modules/lnflwer.mod
project/cargo/modules/avoidance.mod
project/cargo/modules/explore.mod
project/cargo/modules/bumpertracker.mod
project/cargo/modules/linetracker.mod
project/cargo/modules/lighttracker.mod
project/cargo/modules/beep.mod
#
project/cargo/modules/navigate.mod
project/cargo/modules/cargoloader.mod
```

```
project/cargo/modules/motordriver.mod
project/cargo/modules/lnflwer.mod
project/cargo/modules/avoidance.mod
project/cargo/modules/explore.mod
project/cargo/modules/bumpertracker.mod
project/cargo/modules/linetracker.mod
project/cargo/modules/lighttracker.mod
project/cargo/modules/beep.mod
#
project/cargo/modules/navigate.mod
project/cargo/modules/cargoloader.mod
project/cargo/modules/motordriver.mod
project/cargo/modules/lnflwer.mod
project/cargo/modules/avoidance.mod
project/cargo/modules/explore.mod
project/cargo/modules/bumpertracker.mod
project/cargo/modules/linetracker.mod
project/cargo/modules/lighttracker.mod
project/cargo/modules/beep.mod
#
project/cargo/modules/navigate.mod
project/cargo/modules/cargoloader.mod
project/cargo/modules/motordriver.mod
project/cargo/modules/lnflwer.mod
project/cargo/modules/avoidance.mod
project/cargo/modules/explore.mod
project/cargo/modules/bumpertracker.mod
project/cargo/modules/linetracker.mod
project/cargo/modules/lighttracker.mod
project/cargo/modules/beep.mod

# number of fsas that are part of the genome (INCR_VECTOR)
#  (they will be saved to project/cargo/fsa/$project_id/$step_number)
1
1
1
2
2
2
2

# module ids for the fsas that are evolved (each on separate line)
(INCR_VECTOR)
5
#
4
#
4
#
2
4
#
2
4
#
2
4
#
2
4
#

# number of fsas that are fixed and loaded from file (INCR_VECTOR)
1
2
2
2
2
```

```
2
2

# module ids for the fsas that are fixed and loaded from file (one per line)
#  followed by file name (INCR_VECTOR) ; the extension .module_number will
#  be added to each file automatically
#---s0
6
project/cargo/fsa-handmade/fsa.dat
#---s1
5
project/cargo/fsa/sequential/1/fsa.dat.evolved
6
project/cargo/fsa-handmade/fsa.dat
#---s2
5
project/cargo/fsa/sequential/1/fsa.dat.evolved
6
project/cargo/fsa-handmade/fsa.dat
#---s3
5
project/cargo/fsa/sequential/1/fsa.dat.evolved
6
project/cargo/fsa-handmade/fsa.dat
#---s4
5
project/cargo/fsa/sequential/1/fsa.dat.evolved
6
project/cargo/fsa-handmade/fsa.dat
#---s5
5
project/cargo/fsa/sequential/1/fsa.dat.evolved
6
project/cargo/fsa-handmade/fsa.dat
#---s6
5
project/cargo/fsa/sequential/1/fsa.dat.evolved
6
project/cargo/fsa-handmade/fsa.dat

########### trigger messages specifications for all modules that are evolved
(INCR_VECTOR)

#---------step0
### trigger message specification block - module 5 - avoidance
5
# number of trigger messages
3
# list of the messages
SENSORS_BUMPERS_PRESSED
SENSORS_BUMPERS_RELEASED
AVOIDANCE_START
###

#---------step1
### trigger message specification block - module 4 - lnflwer
4
# number of trigger messages
2
# list of the messages
SENSORS_LINE_ENTER
SENSORS_LINE_LEAVE
###

#---------step2
### trigger message specification block - module 4 - lnflwer
4
# number of trigger messages
```

```
4
# list of the messages
SENSORS_LINE_ENTER
SENSORS_LINE_LEAVE
SENSORS_LIGHT_ENTER
SENSORS_LIGHT_LEAVE
###

#---------step3
### trigger message specification block - module 2 - CARGOLOADER
2
# number of trigger messages
1
# list of the messages
SENSORS_TARGET_MARK
###
### trigger message specification block - module 4 - lnflwer
4
# number of trigger messages
5
# list of the messages
SENSORS_LINE_ENTER
SENSORS_LINE_LEAVE
SENSORS_TARGET_MARK
SENSORS_LIGHT_ENTER
SENSORS_LIGHT_LEAVE
###

#---------step4
### trigger message specification block - module 2 - CARGOLOADER
2
# number of trigger messages
1
# list of the messages
SENSORS_TARGET_MARK
###
### trigger message specification block - module 4 - lnflwer
4
# number of trigger messages
5
# list of the messages
SENSORS_LINE_ENTER
SENSORS_LINE_LEAVE
SENSORS_TARGET_MARK
SENSORS_LIGHT_ENTER
SENSORS_LIGHT_LEAVE
###

#--------step5
### trigger message specification block - module 2 - CARGOLOADER
2
# number of trigger messages
1
# list of the messages
SENSORS_TARGET_MARK
###
### trigger message specification block - module 4 - lnflwer
4
# number of trigger messages
5
# list of the messages
SENSORS_LINE_ENTER
SENSORS_LINE_LEAVE
SENSORS_TARGET_MARK
SENSORS_LIGHT_ENTER
SENSORS_LIGHT_LEAVE
###
```

```
#--------step6
### trigger message specification block - module 2 - CARGOLOADER
2
# number of trigger messages
1
# list of the messages
SENSORS_TARGET_MARK
###
### trigger message specification block - module 4 - lnflwer
4
# number of trigger messages
5
# list of the messages
SENSORS_LINE_ENTER
SENSORS_LINE_LEAVE
SENSORS_TARGET_MARK
SENSORS_LIGHT_ENTER
SENSORS_LIGHT_LEAVE
###

########## trigger msgs end

# max. number of states for each evolved module (INCR_VECTOR)
4
#
5
#
5
#
4
6
#
4
6
#
4
8
#
4
10
#

# min. init. number of states for each evolved module (INCR_VECTOR)
1
#
1
#
1
#
1
1
#
1
1
#
1
1
#
1
1
#

# max. init. number of states for all modules evolved (INCR_VECTOR)
4
#
5
#
5
```

```
#
4
6
#
4
6
#
4
8
#
4
10

# max. count of transitions in one state (only 1 number for all modules)
(INCR_VECTOR)
5
5
5
5
5
7
8

# min. count of transitions in one state (only 1 number all modules)
(INCR_VECTOR)
1
1
1
1
1
1
1

# max. init # of transitions in one state (1 number) (INCR_VECTOR)
4
4
4
4
4
4
4

# min. init # of transitions in one state (1 number) (INCR_VECTOR)
2
2
2
2
2
2
2

# probability that the outgoing message is sent in the incoming fsa transition
0.1

# phi for the learning momemtum (used for increments) (INCR_VECTOR)
0.2
0.2
0.2
0.2
0.2
0.2
0.2

# name of the environment-description project file
project/cargo/cfg/sequential4.prj

# previously saved incremental steps filename table
# how many they are
```

```
#1
0
# filenames (these are indexed from (-1)...(-how_many)).
#project/cargo/population/for_sequential/1

# list of steps that are requested to be saved into file for possible
continuation of evolution
# how many they are
7
# which steps (one per line, steps are indexed starting with 0)
0
1
2
3
4
5
6
# and the filenames
project/cargo/population/sequential_/0
project/cargo/population/sequential_/1
project/cargo/population/sequential_/2
project/cargo/population/sequential_/3
project/cargo/population/sequential_/4
project/cargo/population/sequential_/5
project/cargo/population/sequential_/6

# prefix formula that can include variables (parenthesis treated as whitespace)
(INCR_VECTOR)
# gennum             ... current generation,
# gennum-thisstep    ... current generation of this step
# best-fitness       ... best fitness of the last generation
# avg-fitness        ... average fitness of the last generation
# learning-momentum  ... m_new = phi * m_old + (best_fitness -
last_best_fitness)
# num-eval           ... total number of evaluations so far
# num-eval-thisstep  ... number of evaluations since the beginning of this
step
# total-gennum       ... total planned number of generations defined above
# real-runtime       ... real time since application start in seconds
#
# numeric constants, and binary predicates &, |, ^, =, <, >, [, ], !
# and binary operators +, -, *, /
#  ([ means <=; ] means >=; ! means !=)

#skip first two steps, they are already computed
& ] best-fitness 295000 < learning-momentum 50
& ] best-fitness 900000 < learning-momentum 0.001
& ] best-fitness 900000 < learning-momentum 0.001
& ] best-fitness 800000 < learning-momentum 1
& ] best-fitness 400000 < learning-momentum 1
& ] best-fitness 400000 < learning-momentum 50
& ] best-fitness 700000 < learning-momentum 50

# population pass method from all previous steps:
# number of directly preceding steps (INCR_VECTOR)
0
0
1
1
1
1
1

# and their list (negative values are taken from file according to previous
#  steps filename table) (INCR_VECTOR)
#-1
1
2
```

```
3
4
5

# for all fsas that are present in k multiple steps, specify the ratios
#   for blending - p-portion of the original population will come from copied
#   individuals, t-portion of the new population will be born by copying and
mutating x-times
#   and the rest of the new population will be initialized randomly
(INCR_VECTOR)
#-------k-times: (where k is the number of fsas evolved in this step)
# mid
# p_1 t_1 x_1
# ...
# p_nsteps t_nsteps x_nsteps
#-------
# = mid - identifies fsa; p_i identifies portion of the whole (1.0)
#         new population, where the fsa for given mid will be generated by
copying
#         individuals from i-the of the directly preceding steps listed above
#         t_i represents the portion which will be copied and mutated x_i
times,
#         q identifies portion of the new pop. which will have the fsa for mid
#         initialized randomly.
#         copying fsas from previous steps and mutating x_i times
#         the sum of q and all p_i for all mids together should be less than 1.
#         probabilities for all preceding steps have to be listed
#         (even if this fsa doesn't occur in some of them - when the p_i
should be 0)
0
#
0
#
4
0.3 0.4 0.3 2
#0.5 1.0 0.0 0
0
#
2
0.0 0.1 0.0 1
4
0.4 0.2 0.3 2
0
#
2
0.5 0.5 0.3 2
4
0.5 0.5 0.3 2
0
#
2
0.5 0.5 0.3 2
4
0.5 0.5 0.3 2
0
#
2
0.5 0.5 0.3 2
4
0.5 0.5 0.3 2
0
#
# the list is terminated by extra row with 0
# note: in the first step, or step that has no preceding step since there are
no preceding
# steps, the population is generated just randomly and this structure should
be omitted
```

```
# number of starting positions for the objective function (INCR_VECTOR)
3
3
3
3
3
3
3

# number of runs from all starting locations (INCR_VECTOR)
1
4
4
4
2
1
2

# fitness function timeout [s]   (INCR_VECTOR)
150
150
150
150
200
250
400

# fitness checkpoints (if the fitness at given time is less than specified,
#  the individual is killed and its currently gained fitness is used)
(INCR_VECTOR)

# format:  simulation_time required_fitness (list should be ordered by time)
# 10.0 200
# or:
# p q
# t1 dt
# where p - portion of population that is taken into account to get average
progress
#           (0 - only best individual)
#       q - contstraint requirement multiplication quotient for measured
fitness progress
#       t0 - first checkpoint (0 = right from the start)
#       dt - checkpoint time interval (ms of simulated time)
# their number or -1 for automatic checkpoints
#-1
#0 0.05
#20000 5000

#-1
#0 0.05
#20000 5000

#-1
#0 0.05
#20000 5000

#-1
#0 0.05
#20000 5000

#-1
#0 0.05
#20000 5000

#-1
#0 0.05
#20000 5000
```

```
#-1
#0 0.05
#20000 5000

0
0
0
0
0
0
0

##### fitness function weights

# w_obstacle_time (INCR_VECTOR)
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0
-1.0

# w_below_light_time (INCR_VECTOR)
0.0
0.0
0.0
0.0
0.0
0.0
0.0

# w_following_line_time (INCR_VECTOR)
0.0
5.0
-5.0
#-5.0
0.1
0.0
0.0
0.0

# w_following_line_below_light_time (INCR_VECTOR)
0.0
0.0
10.0
10.0
1.0
0.1
0.0

# w_total_distance (INCR_VECTOR)
0.0
0.0
0.0
0.0
0.0
0.0
0.0

# w_robot_moving_changed (INCR_VECTOR)
0.0
100.0
0.0
3.0
0.0
0.0
0.0
```

```
# cnt w_script_count[cnt]  - cnt should match the number of scripts in the
simulation.prj  (INCR_VECTOR)
#--step0
0
#--step1
0
#--step2
0
#--step3
4
0.0
500000.0
0.0
0.0
#--step4
4
0.0
100000.0
0.0
0.0
#--step5
4
0.0
10000.0
0.0
100000.0
#--step6
4
0.0
0.0
0.0
100000.0

# cnt w_active_area_count[cnt];  (INCR_VECTOR)
0
0
0
0
0
0
0

# w_num_states  (INCR_VECTOR)
-10.0
-10.0
-10.0
-10.0
-10.0
-10.0
-10.0

# w_num_trans   (INCR_VECTOR)
-5.0
-5.0
-5.0
-5.0
-5.0
-5.0
-5.0

# w_offset (the score is lifted by this constant to avoid negative values)
(INCR_VECTOR)
300000.0
300000.0
300000.0
300000.0
300000.0
```

```
300000.0
300000.0

# w_suppress_score_before_load (0 or 1, no scores are accummulated before
cargo is loaded)  (INCR_VECTOR)
0
0
0
0
1
1
0

# combine_start_loc (0 - average, 1 - worst fitness, 2 - best fitness, 3 -
geom. mean, 4 - worst+geom)
4
4
4
4
4
4
4

#### end of fitness function weights

# name of the output log file for ea
log/ea_sequential.log

# name of the galib output log file with statistics
log/ga_sequential.log

# project_id for fsa and trajectory file locations
sequential

# project_id for cache
sequential_

# use_preserved_fitness - if nonzero, previous values in the database will
#  not be deleted on startup - use with caution, leave 0 if not sure
(INCR_VECTOR)
0
0
0
0
0
0
0

# commands to be executed after each incremental step on the slave if running
# in distributed mode (only 1 line per step, use 'none' if no commands)
(INCR_VECTOR)
scp cargo@search:current3/project/cargo/fsa/sequential_/0/*.evolved.*
/home/current3/project/cargo/fsa/sequential/0
scp cargo@search:current3/project/cargo/fsa/sequential_/1/*.evolved.*
/home/current3/project/cargo/fsa/sequential/1
scp cargo@search:current3/project/cargo/fsa/sequential_/2/*.evolved.*
/home/current3/project/cargo/fsa/sequential/2
scp cargo@search:current3/project/cargo/fsa/sequential_/3/*.evolved.*
/home/current3/project/cargo/fsa/sequential/3
scp cargo@search:current3/project/cargo/fsa/sequential_/4/*.evolved.*
/home/current3/project/cargo/fsa/sequential/4
scp cargo@search:current3/project/cargo/fsa/sequential_/5/*.evolved.*
/home/current3/project/cargo/fsa/sequential/5
scp cargo@search:current3/project/cargo/fsa/sequential_/6/*.evolved.*
/home/current3/project/cargo/fsa/sequential/6

# commands to be executed after each incremental step on the master if running
# in distributed mode (only 1 line per step, use 'none' if no commands)
(INCR_VECTOR)
scp /home/current3/project/cargo/fsa/sequential/0/*.evolved.*
cargo@search:current3/project/cargo/fsa/sequential_/0
scp /home/current3/project/cargo/fsa/sequential/1/*.evolved.*
cargo@search:current3/project/cargo/fsa/sequential_/1
scp /home/current3/project/cargo/fsa/sequential/2/*.evolved.*
cargo@search:current3/project/cargo/fsa/sequential_/2
scp /home/current3/project/cargo/fsa/sequential/3/*.evolved.*
cargo@search:current3/project/cargo/fsa/sequential_/3
scp /home/current3/project/cargo/fsa/sequential/4/*.evolved.*
cargo@search:current3/project/cargo/fsa/sequential_/4
scp /home/current3/project/cargo/fsa/sequential/5/*.evolved.*
cargo@search:current3/project/cargo/fsa/sequential_/5
scp /home/current3/project/cargo/fsa/sequential/6/*.evolved.*
cargo@search:current3/project/cargo/fsa/sequential_/6

# stop-file - when this file is detected after the generation is completed,
the application
#  terminates (and saves the population of the current incremental step, if
requested)
project/cargo/true.stop_computing

# skip-file - when this file is detected after the generation is completed,
the incremental
#  step is completed and the application proceeds with the next incremental
step (the incremental
#  step to skip must be added to the end of this file name!)
project/cargo/true.skip_step

# save_fsa_file
1

# use_benchmark (for distributed slaves)
0

# master_startup_delay (how long time master should wait for slaves to start
before submitting work, seconds)
150

# threshold for m_new for random reinitialization of population (INCR_VECTOR)
0=not used
0.5
0.5
0.5
0.5
0.5
0.5
0.5

# if the learning is stopped, reinitialize the following portion of population
randomly (INCR_VECTOR)
0.7
0.7
0.7
0.7
0.7
0.7
0.7

# whether robot_moving_changed applies only while following line:0/1
(INCR_VECTOR)
1
1
1
1
1
1
```

```
1

# w_robot_moving_changed_under_light (INCR_VECTOR)
0.0
0.0
100.0
100.0
1.0
0.0
0.0
```

```
# sequential.4.prj.3: description of environment
#

#### modules message data in file (used only in non-evolutionary mode)
project/cargo/modules/alldata.dat

#### number of modules in use
10

#### their list (mid and name)
1
navigate
2
cargoloader
3
motordriver
4
lnflwer
5
avoidance
6
explore
7
bumpertracker
8
linetracker
9
lighttracker
10
beep

#### list of the modules that use fsa (0-terminated)
2
4
5
6
0

### version of code to start
1

#### environment type: RECTANGLE

1

#### environment dimensions: width height

1.0 1.0

#### number of obstacles

5

#### obstacle description: type(RECTANGLE) x y width height (0,0 is a bottom
left corner)

1 0.2 0.2 0.1 0.1
1 0.4 0.7 0.1 0.1
1 0.6 0.6 0.1 0.1
1 0.8 0.85 0.1 0.1
1 0.7 0.25 0.1 0.1

#### number of floor marks

#4
2

#### another floor mark: type(LINE) x y width Nsegments value
```

```
####                     x1 y1
####                     ...
####                     xN yN
#### the coordinates of vertices of this polyline are
#### in the center of the line (i.e. there's a line around
#### these points in all directions up to a distance
#### width/2 - i.e. round corners)
####  -> this means that you should start/end the line in distance
####     width/2 from where it actually ends

2 0.4 0.4 0.05 5 30
0.35 0.4
0.3 0.8
0.25 0.8
0.22 0.5
0.03 0.5

2 0.8 0.5 0.05 3 30
0.8 0.75
0.6 0.8
0.6 0.97

#### floor marks description: type(RECTANGLE) x y width height value

# 1 0.5875 0.975 0.025 0.025 46
# 1 0.0 0.4875 0.025 0.025 46

#### number of light sources

2

#### light source description: type(POINTLIGHT) x y z intensity

1 0.8 0.5 1 1
1 0.4 0.4 1 1

#### environment light conductivity constant 0-1

0.3

#### number of active components

5

#### active areas description format (all conditions are in conjuction):
#
# type                 ; now always 1=conditional active area
# activation/delay     ; -1: one time only,
#                      ;  0: on each entrance,
#                      ;  d: on each entrance, if d [ms] passed since last
entrance
# x y width height     ; location of the robot in the environment to activate
# heading tolerance    ; robot heading to activate (-1: any, otherwise:
<heading-tolerance; heading+tolerance>)
# fork_state (r0)      ; fork state to activate (-1: any, 1, 2, 3, 4 for UP,
DOWN, MOVING_UP, MOVING_DOWN resp.)
# fork_position_min fork_poisition_max (r1)      ; fork must be <min;max> to
activate, <0.0;1.0> for any
# carrying_cargo (r2)  ; state of cargo (-1: any, 0, 1, 2, 3, 4 for NO, YES,
PUSHING, UNLOADING1, UNLOADING2)
# {reg min max}*       ; reg must be <min;max> 0-31 system registers, 32-255
user registers, min, max are 'doubles'
# -2 event_index       ; which script event to execute (refers to the list of
events below)

# loading station: signal to robot on entrance
1
-1
```

```
#0.58975 0.965 0.025 0.01
0.58975 0.965 0.075 0.01
#0.0 0.25
0.0 0.75
-1
0.0 1.0
0
-2 1

# loading station: cargo loading - switching lamps and active areas
1
-1
#0.58975 0.975 0.025 0.025
0.58975 0.975 0.075 0.025
#3.1415926536 0.25
3.1415926536 0.75
2
0.0 0.3
0
-2 2

# unloading station: signal to robot
1
-1
#0.025 0.48975 0.01 0.025
0.025 0.48975 0.01 0.075
#4.712389 0.25
4.712389 0.75
1
0.3 1.0
1
-2 1

# unloading station: start cargo unloading
1
-1
#0.0 0.48975 0.025 0.025
0.0 0.48975 0.025 0.075
#1.5707963 0.25
1.5707963 0.75
-1
0.0 1.0
1
-2 3

# unloading station: cargo unloaded - switching lamps and active areas
1
-1
#0.025 0.48975 0.01 0.125
0.025 0.48975 0.01 0.075
#1.5707963 0.25
1.5707963 0.75
2
0.0 0.3
4
-2 4

#### number of time events

1

#### time events description: type periodic [start count] time event_index
(refers to list of events below)
# following types are recognized
# 1 ... SCRIPT EVENT
# periodic is
#  either 0, then time is a global simulation time
#  or 1, then also the start and count arguments must be given. start is
global simulation time of
#       the first occurence, count is number of occurences and time is the
period

# in the beginning, turn on light 1 and off light 2

1 0 0 4

#### number of script events

4

#### script events description
#
# multiple lines for each script,
# each line contains a command, script terminated by command 0 (STOP)
# following commands are recognized:
# 10 light_ID ... (TURN LIGHT ON)
# 11 light_ID ... (TURN LIGHT OFF)
# 12 active_area_index  ... (reinitialize active area)
# 13 msg      ... (message msg received from IR port)
# 14 reg      ... set current register to reg (0-31 system registers, 32-255
user registers)
# 15 val      ... put value (double) val into current register

# light_ID start from 1

# script 1: send msg to the robot that it is close to cargo loading/unloading
station

13 84
0

# script 2: turn on light 2, turn off light 1, reinitialize active areas 3, 4,
5, set carrying_cargo = 2

10 2
11 1
12 3
12 4
12 5
14 2
15 2
0

# script 3: set carrying_cargo = 3

14 2
15 3
0

# script 4: turn on light 1, turn off light 2, reinitialize active areas 1, 2,
set carrying_cargo = 0

10 1
11 2
12 1
12 2
14 2
15 0
0

#### type of robot (round)

1

#### dimensions:
```

```
####  - radius

0.025

####  - fork size relative to radius

0.3

#### robot speed ratio (how the motor speed is translated to robot speed)
#### should be estimated by Henrik's method

0.00000150

#### fork relative ratio - moving upwards (per millisecond of simulated time)

0.0012

#### fork relative ratio - moving downwards (per millisecond of simulated time)

0.0014

#### initial location and heading: number of locations followed by x y heading

12
#for evolving avoidance
0.5 0.5 0.0
0.76 0.45 0.0
0.1 0.9 0.5
0.5 0.5 0.0
0.76 0.45 0.0
0.1 0.9 0.5
0.5 0.5 0.0
0.76 0.45 0.0
0.1 0.9 0.5
0.5 0.5 0.0
0.76 0.45 0.0
0.1 0.9 0.5
#0.5 0.5 0.0
#0.76 0.45 0.0
#0.76 0.45 0.0
#0.76 0.45 0.0
#0.76 0.45 0.0
#0.76 0.15 0.0

#0.95 0.7 2.4
#0.5 0.5 0.0
#0.1 0.1 1.2
#0.7 0.1 0.2
#0.91 0.75 1.0

#4.8

#### time slowdown constant (how many times slower should be the simulation
time than the system time)

200

#### trajectory file
####  (used only when run without evolution)

project/cargo/traj/sequential/3/t


#### fsa files for all fsas (these are not used when called by objective
function,
####  in that case the values specified in evolutionary config file are used)
####  list is terminated by 0.  each filename is automatically suffixed
with .x, where
```

```
####  x is the number of module

2
project/cargo/fsa-handmade/fsa.dat
4
project/cargo/fsa-handmade/fsa.dat
5
project/cargo/fsa-handmade/fsa.dat
6
project/cargo/fsa-handmade/fsa.dat
0

#### how often does the viewer refresh the visual output (in ms) zero means no
viewer, default: 200

#200
0

#### realtime (0/1): are we running realtime (1) or time-shared (0); realtime
must be run as root
1

#### save trajectory file (0/1)
0

#### NOISE LEVELS:

#### sensor1 gaussian noise level variance (0 = noise free)
0.0

#### sensor2 gaussian noise level variance (0 = noise free)
0.0

#### sensor3 gaussian noise level variance (0 = noise free)
0.0

#### robot speed gaussian noise level variance (0 = noise free)
0.0

#### robot heading gaussian noise level variance (0 = noise free)
0.0

#### robot heading change by obstacle gaussian noise level variance (0 = noise
free)
0.0
```