A Completely Evolvable Genotype-Phenotype Mapping for Evolutionary Robotics

Lukas König
Institute AIFB
Karlsruhe Institute of Technology
76128 Karlsruhe, Germany
lukas.koenig@kit.edu

Hartmut Schmeck
Institute AIFB
Karlsruhe Institute of Technology
76128 Karlsruhe, Germany
hartmut.schmeck@kit.edu

Abstract

To achieve a desired global behavior for a swarm of robots where each robot has a local view and operating range in the environment is a well-known and challenging problem. Evolutionary Robotics is a self-adaptation approach which has been shown to effectively find robot controllers for behaviors which are hard to implement by hand. There, evolvability is highly dependent on controller representation during evolution. It is known that using a genotypic controller representation which also encodes parts of the genotype-phenotype mapping (GPM) can lead to a meta-adaptation of the evolutionary operators to the search space structure, thus improving evolvability. We enhance this idea using a fully flexible GPM which is represented in the same way as the behavioral controllers are, and, therefore, can be completely evolved along with the behavior. The approach is based on finite state machines and extends an existing framework for decentralized evolution of robot behavior in swarms of mobile robots. Experiments indicate that the evolvable GPM outperforms both the extensively improved operators of the existing framework and a standard operator for the new real-valued genotypes with fixed GPM.

1. Introduction

Programming robots by hand is often challenging and can be unfeasible particularly for swarms of robots where predicting the emerging global behavior is hard [2]. The potentials of collective behavior, however, are great [1], and applications for swarms of robots gain importance reaching from delivering scenarios in store-houses to accessing dangerous or unapproachable areas by adapting to previously unknown conditions. Evolutionary Robotics (ER) is a self-adaptation technique which arose in the last two decades from the well-established field of evolutionary computation. It is known to be effective in developing robot behaviors in various scenarios such as exploitation of collective behavior, but also in single robots performing onboard evolution in simulation, or in many other scenarios [3], [15]. Evolution, based on the Darwinian principle of a survival of the fittest, is capable of finding controllers which outperform human solutions in terms of effectiveness in solving the task, simplicity of the controller, and generalizability of the learnt behavior [16].

Impact of the genotype-phenotype mapping (GPM) on Evolvability. Evolution can be done in solution space (phenotypic space) alone, i. e., mutation and recombination are applied directly to behavioral programs of robots. However, utilizing a genotypic representation different from the solution space by performing search in the genotypic and evaluation in the phenotypic space can improve performance of evolution. There, the GPM can be fixed or itself subject to evolution [15].

(1) Fixed GPM: In classical Genetic Programming (which the main ideas for the original framework were borrowed from [11]), there is no distinction between genotype and phenotype (cf. [14]). In [9], bitstrings are used to genetically encode phenotypes which are C programs, using a repair mechanism as part of the GPM to make every string valid; this leads to a significant improvement in terms of quality of the found solutions compared to an approach with phenotypes only. The authors explain the improvement by the hard syntactical constraints on mutation and recombination when working on the phenotypic space. Using a genotypic representation makes it possible to alter the genotypes arbitrarily, and map them onto the nearest legal phenotype using the repair mechanism. Thus, originally infeasible gaps in the search space get replaced by neutral plateaus which have been shown to improve evolvability [10].

The evolutionary operators used in our earlier work [11]–[13] are working on a space of finite state machines (FSMs) which also have hard syntactical constraints. Therefore, an improvement can be expected by using a genotypic representation. In this paper, genotypes are sequences of integers which encode FSMs. During translation, a script is created providing a repair mechanism. Experiments with this fixed representation show an improvement in terms of mean fitness and complexity of evolved behaviors which is even larger than expected.

(2) Evolvable GPM: In ER, due to the underlying dynamical fitness landscape, even more sophisticated GPMs are suggested, to make complex behaviors evolvable [15]. Encodings are supposed to have (a) *expressive power* (i. e., many different phenotypical characteristics should be encodable, e. g., parts of the GPM itself), (b) *compactness* (i. e., the length of the genotype should not directly reflect the complexity of the phenotype, e. g., by introducing repeated structures), and (c) *evolvability* (i. e., the evolutionary operators should generally be able to produce improvements).

(a) Expressive power: In the presented approach, the complete GPM is part of the genome, leading to a high expressive power. While parts of the GPM have successfully been evolved earlier, to our best knowledge evolution of the complete GPM has not been done before (although it has been suggested, e. g., in [9]). The required flexibility of the mapping is achieved by letting the translation process which performs the GPM be part of the genotype and subject to the evolutionary operators itself. This means that the interpretation of the behavioral part of the genotype as well as the interpretation of the translation part changes during evolution.
    In nature, the GPM has evolved along with the other properties of living beings and there are still some mechanisms active that change the semantics of genes, i. e., the GPM [4]. An evolvable GPM allows for evolution to learn structural properties of the search space, which depend on the learnt behavior, and to use this knowledge for improving evolvability during a run. Fig. 1 shows an example of adaptation of the mutation step size when searching the minimum of a Schaffer 2 function in a one-dimensional real-valued search space. Due to the flexibility of the GPM, operators can be adapted indirectly by changing the meaning of a part of the genotype, thus changing the impact of the operator when working on that part. More complex behaviors are expected to be evolvable, although needing possibly more running time for GPM adaptation. Experiment time has, therefore, been increased from 80, 000 simulation cycles in [12] to 300, 000.
(b) Compactness: The GPM is performed by FSMs which can have loops, thus being able to create repeated structures which allow for a compact genotypical representation.
(c) Evolvability: this is a property which is hard to identify in a system. The suggestions made in [15] have been respected for the initial settings of the runs. As the GPM changes during evolution, this cannot be guaranteed for later states of the runs, but as argued before an improvement in evolvability is expected to occur during the runs.

The experimental results imply that an improvement in terms of complexity of the behavior has occurred (Sec. 5).

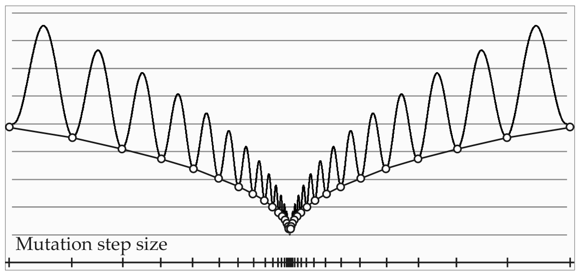

Figure 1. Possible adaptation of mutation step size when searching the minimum in a one-dimensional real-valued search space given by a Schaffer 2 function: $f(x) = x^{0.5}\left(\sin^2\left(50 \cdot x^{0.2}\right) + 1\right)$. When approaching the optimum at $x = 0$, an individual's mutation step size should decrease.

Controller and GPM Representation. A general idea of the presented model is to use the same representation for robot controllers and for the GPM to make it possible to evolve both simultaneously using the same set of evolutionary operators. We use a FSM model called *Moore Automaton for Robot Behavior (MARB)* for the representation of robot controllers (based on our preliminary work in [11]–[13]). A very similar model called *Moore Automaton for Protected Translation (MAPT)*, differing only in sensor and actuator spaces, is used for the representation of GPMs.

The evolutionary operators studied here do not involve recombination. Diversity is established by mutation only. Also, the *memory genome* which is proposed in [12], introducing a decentralized elitism for the framework, is not used. The operators are kept as simple as possible to allow for a study of the effects of the evolvable GPM and the new fixed GPM on performance. In future studies, a recombination operator and the memory genome are planned to be combined with the new GPM.

The reality gap. A well-known problem in evolutionary robotics is the transfer of simulated results into real-world applications (cf. [6], [7]). This problem, often referred to as the *reality gap*, arises from several real-world factors (unknown changes in the environment, unpredictable locomotive and sensory differences between robots, mechanical and software failures, etc.) which are hard to simulate. We face the reality gap by combining simulation and reality as suggested in [16]. The evolutionary parameters and operators are adjusted in a rather simple simulation which does not support a detailed physics engine. However, it is sufficient to study the mechanisms of evolution. Actual evolution is done online on real robots. Experiments in this paper are performed in simulation only because a real robot platform is currently not available, but experiments on real robots have been done in [11] and are planned to be done in the future.

Decentralized Online Evolution. The proposed evolutionary framework is completely decentralized and can be implemented in simulation as well as on real robot platforms. The framework is designed to work onboard of robots (simulated or real), accomplishing an evolutionary algorithm without any central control. In that sense it is an application of the "embodied evolution" proposed in [17]. Due to the decentralization, the framework scales well to large swarms of robots and can be easily implemented for different simulation and robot platforms. Furthermore, the behavior is evolved *online* which means that during a run, currently evolved behavior is evaluated by observing its performance on the given task. This requirement is given by tasks where robots have to adapt quickly to a new situation and learn how to deal with a novel objective, e. g., when swarms of robots explore new and unknown areas where challenges may change constantly.

The Jasmine-IIIp Robot. The framework is defined in a general way and is applicable on different robot platforms. Up to now, it is implemented and tested on the Jasmine IIIp robot which is also simulated. The Jasmine IIIp series is a swarm of micro-robots sized $26 \times 26 \times 26$ mm^3. It can process simple motoric commands like driving forwards and backwards and turning left and right. Each robot has seven infra-red sensors (as depicted in Fig. 2) returning values from 0 to 255 in order to measure distances to obstacles (cf. www.swarmrobot.org). In simulation, the return value of a sensor is calculated by the function $d(x) = \lfloor 255 \cdot 51^{(R-(x \cdot A \cdot B))/150} \rfloor$, where x is the distance from the middle of the robot to the closest object in range of the sensor (in millimeters), and R is the distance of the sensor to the middle of the robot; A is 1 if the obstacle is a wall, and 2 if it is a robot (walls reflect infra-red light better than robots); B is 1 for sensors 2 to 7, and 0.75 for sensor 1 due to its greater detection radius. In one simulation step, a robot moves 4 mm straight forward (*Move*-command) or turns left or right by an angle of 10 degrees (*Turn*-command; mapped on real world dimensions). A crash with an obstacle (i. e., a wall or another robot) is simulated by placing the robot at a random free place within a 4 mm radius from the last position before the collision, and turning it by a random angle (if this is possible without a new collision).

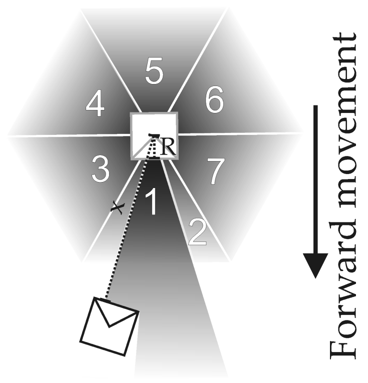

Figure 2. Placement of infra-red sensors around a (simulated) Jasmine-IIIp robot. Sensors $2-7$ are using an infrared light source with an opening angle of 60 degrees to detect obstacles in every direction of vision. Sensor 1 has an angle of 20 degrees to allow detection of more distant obstacles in the front.

$\star$ is used as a "do not care" symbol which can be empty or the index *trans*.

Genes are sequences of numbers; a robot's genome consists of one behavioral and one translator gene. A formal definition of the genotypic space is given in Sec. 3.

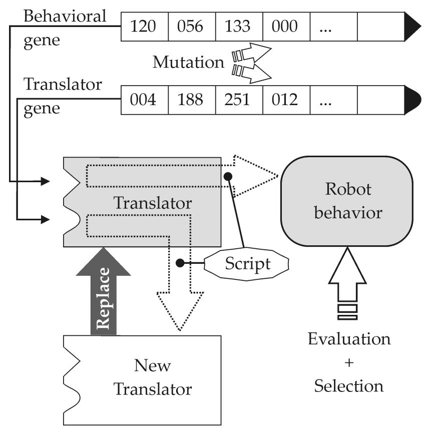

Figure 3. Translation process from genetic sequences to behavioral and translator automata. After a mutation of one of the sequences, the according automaton gets replaced by the translation of the new sequence.

2. Automaton Model

In this section, the behavioral and evolutionary model are described. Parts of the model have also been described in [11]–[13], however, all parts involving the translator automaton, which is a central component in the GPM, are new. To keep notations consistent with earlier work, symbols concerning behavioral automata will be used without index while symbols concerning translator automata will be assigned the index *trans*. For the purpose of a simpler notation,

Moore Automata for Robot Behavior (MARB) and for Protected Translation (MAPT). Essentially, the same model based on Moore Automata is used for the description of robot behavior and for the description of the translation process from genotypes to phenotypes (behavioral or translator automata), i. e., the GPM, cf. Fig. 3. Every robot carries its own translator, i. e., a GPM, which can be altered during evolution. Using one single automaton model makes it possible to apply the same evolutionary operators to both types of automata. The space of behavioral automata is called *MARB*, the space of translator automata *MAPT*; where it does not cause any confusions, the same abbreviation is used for elements of these spaces, i. e., actual automata. See Fig. 5 (a) for an example MARB and Fig. 5 (b) for an example MAPT.

For both types of automata at each state of the automaton an *operation* is executed. This is a movement operation in the behavioral case, and an operation for the construction of a new (behavioral or translator) automaton in the translator case. The transition function, which determines the next state to enter, is defined based on values from *virtual* or *real sensors*. In the case of behavioral automata, the accessible sensors are 7 *real* sensors which are placed on a robot and indicate distances to objects. In the case of translator automata they are 5 *virtual* sensors, two of which represent reading heads on the genetic sequence and the other three serve as registers, each capable to store one byte value. Therefore, the only difference between MARBs (navigating a robot through an environment) and MAPTs (producing new automata by traversing a genetic sequence) are the different operation and sensor spaces they work on.

Preliminaries. A (real or virtual) sensor is represented by a sensor variable which can take a byte value. Let $B = \{0, ..., 255\}$ and $B_+ = \{1, ..., 255\}$ be the set of all and only the positive byte values, respectively. A sensor variable is denoted by the letter h with a numeric index. For $i \leq j \in \mathbb{N}$ and a set of sensor variables $\{h_i, h_{i+1}, ..., h_j\}$ let $(v_i, v_{i+1}, ..., v_j) \in B^{j-i+1}$ be a tuple of the actual corresponding values of the sensors.

In the case of *behavioral automata*, the set $H = \{h_1, ..., h_7\}$ defines the sensor variables. For $1 \leq i \leq 7$, h_i is associated to the real sensor labeled with i in Fig. 2. At a specific time step in an environment, the tuple of current values of the sensors is $V = (v_1, ..., v_7) \in B^7$.

In the case of *translator automata*, the sensor variables are defined by the set $H_{trans} = \{h_{99}, ..., h_{103}\}$. For $99 \leq j \leq 103$, h_j is associated to the virtual sensor labeled with h_j in Fig. 4. At a specific position during the traversal of a genetic sequence, the tuple of actual values of the sensors is $V_{trans} = (v_{99}, ..., v_{103}) \in B^5$. There, h_{100} represents the value at the current position of the reading head, h_{99} the next value (which is the value right of the position of the reading head or zero if the reading head is on the rightmost position); $h_{101}, ..., h_{103}$ represent three registers which can be fed by byte values to store information about the previous translation course.

Note that only one of the sensor sets H and H_{trans} is used at the same time. The sensors $h_8, ..., h_{98}$ are never used; this gap is kept clear to make it possible to add more sensors.

Two randomized functions are assumed: $rand(X)$ takes a finite set X as a parameter and returns a random element out of it based on uniform distribution; $rand_{gaussian}()$ returns a random real number by standard normal distribution (with a mean of 0 and a standard deviation of 1).

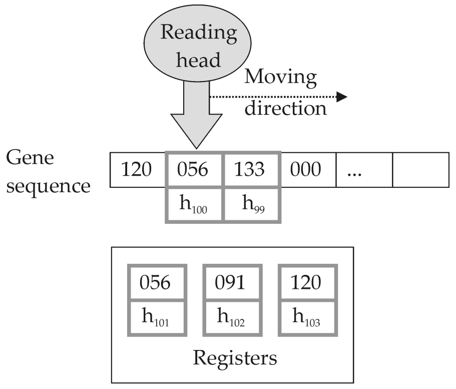

Figure 4. Virtual sensors of a translator; two of them return the value of the current and next gene sequence symbol, the other three serve as registers.

Automaton Definition. A (deterministic) Moore Automaton as defined in [5] is a 6-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$. Q is a set of states, Σ is an input alphabet, Δ is an output alphabet, $\delta : Q \times \Sigma \to Q$ is a transition function, $\lambda : Q \to \Delta$ is an output function, and $q_0 \in Q$ is the initial state.

The execution of a Moore Automaton begins at the initial state q_0; for every state $q \in Q$, an output $\lambda(q)$ is written on some output stream, and a unique following state $\delta(q, v)$ is determined by the transition function based on which symbol $v \in \Sigma$ is currently read on some input stream. In the case of behavioral automata, the input stream is given by the movement through an environment and the retrieved sensor values. For translator automata, the input stream is given by the virtual sensor values resulting from the traversal of a genetic sequence from left to right, performing a state change for each symbol. The output at each state causes a movement action for behavioral automata and a construction command for translator automata.

In the model presented here, the flexible parts of the automaton which are subject to evolution are: the set of states Q including q_0, the output function λ, and the transition function δ. The output alphabet is fixed and defined to be a set of operations $\Delta = Op$ or $\Delta = Op_{trans}$ as defined below for behavioral or translator automata, respectively. The input alphabet is also fixed and defined to consist of all possible combinations of sensor values $\Sigma = V$ or $\Sigma = V_{trans}$ for behavioral or translator automata, respectively. The states are identified by positive byte values, i. e., $1, ..., 255$.

A Moore Automaton is depicted graphically by drawing one circle per state $q \in Q$ labeled with the state's name q and the output $\lambda(q)$ (the *name* is put at the top of the circle, the *command* and *parameter* at the bottom and the *additional parameter* in brackets in the middle); an incoming arrow marks the initial state. For any two states q, q' with $\delta(q, v) = q'$ for some $v \in \Sigma$, an arrow (representing a *transition* or an *edge*) is drawn from the circle for q to the circle for q'. The arrows are labeled with conditions which belong to the transitions (see below); cf. Fig. 5.

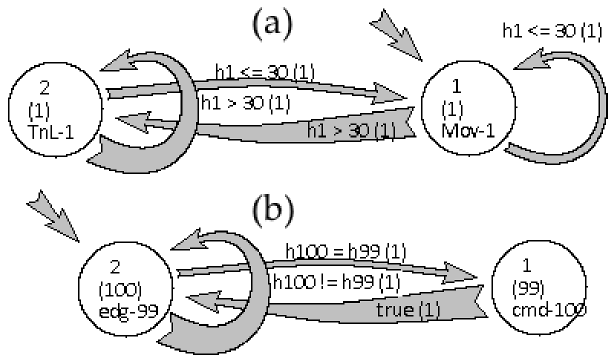

Figure 5. (a) Example MARB performing a simple collision avoidance behavior (move forward: "$Move(1)$" as long as no obstacle is ahead: "$h_1 \leq 30$", turn left: "$TurnLeft(1)$" if an obstacle is ahead: "$h_1 > 30$"). (b) Example MAPT performing a simple translation (create edges from the state named by next symbol to the one named by current symbol: "$edg(99, 100)$" as long as next symbol is different from current: "$h_{99} \neq h_{100}$", replace current state's command by current symbol in sequence: "$cmd(100, 99)$" if current and next symbol are equal: "$h_{99} = h_{100}$"). Arrow thickness denotes likelihood for condition to get *true*.

Conditions. To avoid defining for every state $q \in Q$ and every input symbol $v \in \Sigma$ a separate outgoing transition (which would result in $|V| = 256^7$ or $|V_{trans}| = 256^5$ transitions per state), we define a set of conditions to cluster the input alphabet. A transition is then taken if the corresponding condition evaluates to *true* under the current sensor values.

For *behavioral automata*, the set C of *conditions* over the sensor variables H is the set defined by:

$$c ::= true \mid false \mid z_1 \triangleleft z_2 \mid (c_1 \circ c_2),$$

with $z_1, z_2 \in B_+ \cup H$, $\triangleleft \in \{<, >, \leq, \geq, =, \neq, \approx, \not\approx\}$, $\circ \in \{AND, OR\}$, c_1, c_2 conditions.

For *translator automata*, the condition set C_{trans} over H_{trans} is defined accordingly.

true and *false* are called *atomic constants*, $z_1 \triangleleft z_2$ is called an *atomic comparison*. Therefore, a condition can be an arbitrary combination of atomic comparisons and atomic constants, connected by *AND* and *OR*. For $a, b \in B$, it is defined: $a \approx b$ iff $|a - b| \leq 5$ and $a \not\approx b$ iff $\neg a \approx b$.

Example conditions are:
$true, false, h_1 < h_2, 20 > h_7 \in C$;
$(h_{100} \approx h_{99}\ OR\ h_{102} \not\approx 120) \in C_{trans}$.

A function $E : C \times V \rightarrow \{true, false\}$ evaluates a condition $c \in C$ to *true* or *false* in the obvious way, inserting the current sensor values $v \in V$ for the sensor variables.

Note that for technical reasons, the constant 0 must not be part of a condition; however, every atomic comparison containing 0 can be expressed as an equivalent condition without 0, e. g., $\forall v : E(h_1 > 0, v) = E(h_1 \geq 1, v)$.

As conditions and transitions are subject to evolution, two cases of inconsistency can occur and have to be considered. (1) If for a state none of the outgoing transitions have a condition that evaluates to *true*, there is an implicit default transition to the initial state. Fig. 5 shows two automata with two states and a complete definition of transitions. However, all transitions pointing to the initial state 1 could be deleted, since they are implicitly defined. (2) If, on the other hand, more than one condition evaluates to *true*, the first of the *true* transitions (in order of insertion during construction of the automaton) is chosen.

Operations. For both types of automata the operations which are executed at any state have the form of a 3-tuple (*Command*, *Parameter*, *additional Parameter*) of bytes: $(A, X, Y) \in B \times B \times B$. The command A is specified by the sets Cmd and Cmd_{trans} in the following (which are internally mapped into B). Parameter and additional parameter may be ignored for some commands. The operation (A, X, Y) is also written as $A(X, Y)$ (or $A(X)$, $A()$ if parameter and / or additional parameter are ignored).

Operations for behavioral automata. Behavioral commands are defined by the set

$$Cmd = \{Move, TurnLeft, TurnRight, Stop, Idle\}.$$

All possible behavioral operations are defined by the set $Op = Cmd \times B \times B$, which mean for an operation $A(X)$ (for all behavioral operations the additional parameter Y is ignored): drive forward for at most $X\,mm$, turn left for at most $X\,degrees$, turn right for at most $X\,degrees$, stop performing the current operation, or keep performing the current operation, respectively. For $Stop$ and $Idle$, the parameter X is also ignored. A behavioral automaton's operation can be, e. g., $Move(10) \in Op$.

Operations for translator automata. A translator is supposed to construct a complete automaton which consists of nodes, edges, state labels, and conditions. To achieve a desired universality in the GPM, there should exist for each (behavioral or translator) automaton a at least one combination of translator t and genetic sequence g, such that the translation of g by t yields a, i. e., $dec(t, g) = a$ or $dec_{trans}(t, g) = a$ (in the notation introduced in Sec. 3; in the presented model it even holds that independently of the sequence g there exists a translator t with $dec_{\star}(t, g) = a$). The language of conditions, however, is context-free and cannot be expressed by a regular language which means that it is impossible for a pure finite-state model to create all syntactically correct conditions. Moreover, it is desired that every operation executed by a translator has a well-defined effect on the outcoming automaton and that this effect is not trivial for most operations. E. g., if an operation for inserting an edge between the states $a, b \in B_+$ would be defined to work only if the states a and b already exist in the automaton, this operation would have no effect for most combinations of a and b in most automata of usual size.

To solve both these problems, a script language has been introduced which provides access to a stack memory for the construction of conditions; additionally, for the interpretation process a repair mechanism is performed on invalid operations to make their effect valid, and, in many cases, non-trivial. The translator then only has to create a valid script out of a genetic sequence and the rest is done by executing of the script. Translator operations will, therefore, be based on the script operations defined in the following.

A *script* is a sequence Op_{scr}^* of script instructions to be executed sequentially. A script instruction $A(X, Y)$ belongs to a set of operations $Op_{scr} = Cmd_{scr} \times B \times B$ with

$$Cmd_{scr} = \{nod, cmd, par, add, TH, TC, TO, edg\}.$$

The commands nod, cmd, par, and add are used to insert a state and change its command, parameter, and additional parameter, respectively. TH, TC, and TO are commands for building a condition in the internal memory by inserting sensor variables (H), atomic constants (C) and operators (O) in a postfix manner. The command edg inserts a transition between two states using the currently constructed condition. If edg is invoked while the current condition is not yet finished, the condition is finished by standard values before it is used. If the parameter of an instruction is out of range for that instruction, the script interpreter maps it on the adequate range via a modulo operation (this procedure is assumed to be already done in the following). Standard values replace invalid values; they are generated by a standard value generator based on several counters for the different value spaces guaranteeing a deterministic, but diverse value distribution. Fig. 6 shows the process of translation by using a script.

The script instructions are interpreted as follows:

- $nod(X)$: Insert state X (if state X already exists do nothing; if X is the first inserted state, declare it initial state) and use a standard command, parameter, and additional parameter for the operation.
- $cmd(X, Y)$: If state X does not exist, execute $nod(X)$. Change the *command* of state X to Y.
- $par(X, Y)$: If state X does not exist, execute $nod(X)$. Change the *parameter* of state X to Y.
- $add(X, Y)$: If state X does not exist, execute $nod(X)$. Change the *additional parameter* of state X to Y.
- $TH(X)$: Add X to condition as next postfix-symbol if a sensor variable is syntactically correct here, otherwise execute $TO(X)$.
- $TC(X)$: Add X to condition if an atomic constant (*true* or $false$) is correct here, otherwise execute $TO(X)$.
- $TO(X)$: Add X to condition if an operator (<, >, ...) is correct here, otherwise execute randomly $TH(X)$ or $TC(X)$ by uniform distribution.
- $edg(X, Y)$: If state X does not exist: $nod(X)$. If state Y does not exist: $nod(Y)$. If the current condition *cond* is unfinished: finish *cond* by using standard values. Insert a transition from state X to state Y labeled with *cond*.

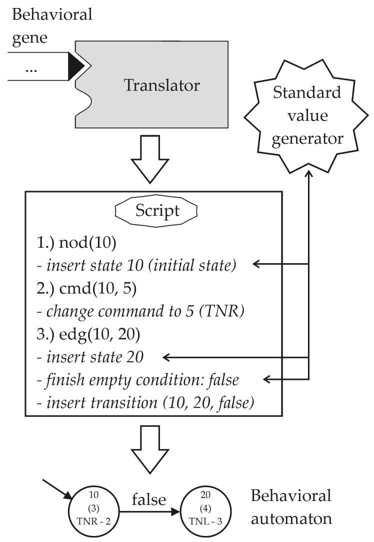

Figure 6. Example translation of a behavioral gene into a behavioral automaton.

The set of commands for translator automata is a superset of the script commands and defined to be

$$Cmd_{trans} = Cmd_{scr} \cup \{RC, RV, NXT\}.$$

The set of translator operations is accordingly defined to be $Op_{trans} = Cmd_{trans} \times B \times B$. For a translator a script operation $A(X, Y) \in Op_{scr} \subset Op_{trans}$ has the meaning: insert $A(X, Y)$ into the script.

The register operations $RC(X, Y)$ and $RV(X, Y)$ mean: store in the register sensor X the *constant* value Y or the *variable* value from sensor Y, respectively. If X or Y is out of range, a modulo transformation is performed to set it into the range of the register sensors $(h_{101}, ..., h_{103})$.

The operation $NXT()$ does not perform any action. The automaton simply moves on to the next state while the reading head moves one step further to the right.

This model is capable of generating any possible automaton from the spaces MARB or MAPT. The term *Moore Automaton for Protected Translation* is dedicated to the usage of a script which protects the GPM from syntactical failures during translation.

3. Genotypic and Phenotypic Spaces

In this section, the genotypic space (search space) and the two phenotypic spaces (solution spaces; one for behaviors and one for translators) are defined. The genotypic space consists of sequences of numbers which are mapped on behavioral or translator automata in the respective phenotypic spaces.

Genotypic space. As mentioned before, a *gene* or *(genetic) sequence* is a flexible-size array of numbers from the *genotypic space*

$$\mathcal{G} = B^*.$$

Position i in a gene g is denoted by $g[i]$, $|g|$ is the length of g (i. e., the number of bytes). A genome G^r of a robot r consist of two genes, namely the behavioral gene g^r and the translator gene g^r_{trans}, and is defined by the tuple

$$G^r = (g^r, g^r_{trans}) \in \mathcal{G} \times \mathcal{G}.$$

Phenotypic spaces. Two different kinds of phenotypic spaces are distinguished: (1) The *space of (behavioral) phenotypes $\mathcal{P} = MARB$* is the space of all Moore Automata for Robot Behavior. Its elements are called *behaviors* and are executed on robots as behavioral programs being, therefore, directly subject to selection. (2) The *space of translator phenotypes $\mathcal{P}_{trans} = MAPT$* is the space of all Moore Automata for Protected Translation. The elements are called *translators*; their function is to translate behavioral genes as well as translator genes into robot behaviors or other translators, respectively. They are evaluated only indirectly as they influence *evolvability* of the robot behavior during a run causing better or worse long-term performance (there is one exception to that statement as empty translators are sorted out directly after mutation; see Sec. 4).

Please be aware that the definitions of genotypic and phenotypic spaces used here differ from those provided in [11]–[13] where behavioral automata were positioned in the genotypic space.

There exist two mappings (decodings) from the genotypic space to the two phenotypic spaces:

$$dec : \mathcal{G} \times \mathcal{P}_{trans} \rightarrow \mathcal{P}, \quad (1)$$
$$dec_{trans} : \mathcal{G} \times \mathcal{P}_{trans} \rightarrow \mathcal{P}_{trans}. \quad (2)$$

The mapping dec denotes the creation of a robot behavior out of a gene using an existing translator. The mapping dec_{trans} denotes the creation of a new translator out of a gene using an existing translator. Therefore, both decodings do not only depend on the gene which is being decoded, but also on the translator. Different translators can produce different outcomes for the same gene.

Partial completeness of $\mathcal{P}_{trans}$. It even holds that any behavioral (or translator) automaton can be produced by dec (or dec_{trans}) out of any gene g depending only on the decoding translator (and $|g|$ has to exceed some value k depending on the output automaton, as a translator performs exactly $|g|$ state changes for a translation):

$$\forall g \in \mathcal{G}, |g| \geq k, \forall b \in \mathcal{P}\ \exists t \in \mathcal{P}_{trans} : dec(g, t) = b, \quad (1)$$
$$\forall g \in \mathcal{G}, |g| \geq k, \forall t' \in \mathcal{P}_{trans}\ \exists t \in \mathcal{P}_{trans} : dec_{trans}(g, t) = t'. \quad (2)$$

This can be proven easily by constructing for each behavior b (or translator t) a translator which simply ignores the input and creates as a constant output the behavior b (or the translator t). This property is called the *partial completeness* of the space of translator phenotypes $\mathcal{P}_{trans}$.

Universal translators. A direct implication of the above completeness statement is that for a fixed translator there may exist behaviors (or translators) which cannot be created, regardless of which gene is being decoded (e. g., for the aforementioned constant translators, most behaviors cannot be created). Translators, therefore, have more impact on the outcome of decodings than the decoded genes have.

There is, however, a class of translators which can produce every possible behavior (or translator) depending on the decoded gene. A *universal translator u* is a translator which meets one of the following requirements:

$$\forall b \in \mathcal{P}\ \exists g \in \mathcal{G} : dec(g, u) = b, \quad (1)$$
$$\forall t' \in \mathcal{P}_{trans}\ \exists g \in \mathcal{G} : dec_{trans}(g, u) = t'. \quad (2)$$

As in the beginning every point of the phenotypic spaces should be reachable, a universal translator is used as initial translator in all experiments. For runs with evolvable translator, however, it is possible that at a certain time step not every phenotype can be reached anymore (although always every genotype can).

4. Evolutionary Operators

Mutation, *selection* and *fitness calculation* are described in this section. For simplicity reasons, there is no *recombination operator* applied. Also, no *memory genome* is used (cf. [12]). These two operators will be tested in future experiments.

Mutation Every t^{mut} (t^{mut}_{trans}.) simulation cycles, mutation is applied to all robots' behavioral (translator) genes. Where real-valued operations are used on the byte-valued genomes, a rounding operation is assumed to be used afterwards.

New mutation operator M: The following standard mutation operator for real-valued genomes is used: by a small probability ϵ (ϵ_{trans}), a value

$$R = rand_{gaussian} \cdot d, \qquad (R_{trans} = rand_{gaussian} \cdot d_{trans},)$$

is added to every byte of the gene. I. e., $R_\star$ is a real-valued random variable with normal distribution and a standard deviation of $d_\star$, being rounded to integer and put in range afterwards. Additionally, with probability $\epsilon_\star \cdot f^+_\star(|g|)$ a new byte with (a newly drawn) value $R_\star$ is appended to the end of the sequence; with probability $\epsilon_\star \cdot f^-_\star(|g|)$, the rightmost byte is deleted. $f^+_\star$ and $f^-_\star$ provide for short genes a higher probability to get longer than for long genes (see Tab. 1).

As a mutation takes effect first when the according gene is translated into a new automaton, after the mutation of a *behavioral gene*, the gene is translated by the current translator of the robot into a new behavior.

As well, after the mutation of a *translator gene*, the current translator t translates the new gene g'_{trans} and gets replaced by the new automaton $t' = dec_{trans}(g'_{trans}, t)$, cf. Fig. 3. However, this may lead to an unstable state, because a translation of g'_{trans} by t' can lead again to a new translator and so on causing an unpredictably long chain of mutations. Therefore, the replacement $t := dec_{trans}(g'_{trans}, t)$ is not done once, but until a stable state is reached and the translator does not change anymore (to deal with possible cycles, a maximum number of 100 translations is not exceeded). Of course, this process of mutating always to a stable state is quite artificial and lacks a natural counterpart. However, leaving it out would mean that a mutation could possibly cause effects which appear unpredictably far in the future. This would make it impossible to evaluate the automaton appropriately in the rather short time before it gets subject to selection.

At this point, there is also one single evaluation on the translator level applied: translators tend to degenerate to empty automata by mutation; as an empty translator always produces empty output (i. e., particularly non-moving behavior) which cannot be desired by any fitness function, it seems plausible to prohibit such mutations. In that case the mutation is undone and the translator remains unchanged.

Old mutation operator M_{old}: To provide comparisons to earlier approaches, experiments were also made with the mutation operator M_{old} provided in [12], [13]. Here, the best parameter setup described in the papers is used.

Reproduction and Selection. Selection is based on an operator similar to tournament selection in classical evolutionary computation. Due to the decentralized approach, however, selection cannot be defined as a population-based operator. In [11], two robots produce offspring when they come spatially close to each other. Child behavior is then produced by cloning the parent with the best fitness. Since reproduction in that approach occurs unpredictably, it is difficult to control reproduction rate and selection pressure.

In simulation, we use a "semi-decentralized" approach which allows for more control of reproduction rate and selection pressure while still being close to a completely decentralized method [12]. Thereby, a clock triggered by the simulation environment is used to synchronize reproduction. The robots no longer reproduce when they meet, but all robots reproduce simultaneously according to this global clock; every robot mates with one (or more, 7 in the conducted experiments, see Sec. 5) robot(s) it is spatially closest to at that moment, selecting always the best genome as its child genome. Note that in runs with evolvable GPM, additionally the translator of the chosen parent has to be cloned as the interpretation of the genome always depends on the decoding automaton.

In a completely decentralized version of this, a robot could collect genomes of other robots, performing selection onboard when a defined number of parents is reached.

Fitness Calculation. Fitness calculation also must be done in a decentralized way which leads to a fitness estimation, cf. [8]. Fitness is calculated for a *gate passing behavior* which is the more complex of the two behaviors described in [12]. The robots are supposed to avoid collisions and at the same time pass a gate in the middle of the field as often as possible (cf., e. g., Fig. 7). As these are two contrary requirements (passing the gate means approaching walls and, therefore, risking collisions), the behavior has an inherent complexity.

Every robot starts with a zero fitness in the beginning to which a *fitness snapshot* (see below) is added every t^{fit} simulation cycles. Every t^{evap} cycles the fitness is divided by 2. This *evaporation* is meant to diminish the influence of old behaviors. After mutation, the fitness is not changed (as small changes in the behavior are expected). After reproduction, the fitness of the selected parent is taken. The snapshot is calculated by the following algorithm:

```
$snapshot$ := $-3 \cdot$ #(collisions since last snapshot);
**if** gate has been passed since last snapshot **then**
    $snapshot$ += 10; /* Gate bonus. */
**end if**
**if** current operation is not a $Move(X)$ operation **then**
    $snapshot$ -= 1; /* Punishment for not moving. */
**end if**
```

5. Experimental results

Setup. All runs proceeded on a rectangular field sized 1440×980 mm^2 (cf. Fig. 7) with a gate in the middle (190 mm). 30 robots started at uniformly random positions and angles with empty behavioral automata and a universal translator u. The runs lasted for 300,000 simulation cycles (about 55 minutes in a real-world scenario). The following three blocks of runs have been performed for which Tab. 1 shows the detailed parameter setups:

1) Flexible GPM with mutation M. The parameter ϵ_{trans} varied in 50 equal steps from 0.01 ‰ to 2.00 ‰. For each ϵ_{trans}, 10 runs have been performed.
2) Fixed GPM with mutation M. ϵ was tested in preliminary studies with fixed GPM (50 different values, 10 runs each); set here constantly to best found value of 5 ‰.
3) Fixed GPM with mutation M_{old}. 500 equal runs with the best setup described in [12] have been performed.

For reproduction, each robot produced 1 child with the 7 robots closest to itself (i. e., 8 parents for 1 child) as this

Table 1. Parameter setup.

	Flexible GPM	Fixed GPM, M	Fixed GPM, M_{old}
t^{mut}	100 cyc	100 cyc	100 cyc
t^{mut}_{trans}	1000 cyc	-	-
ϵ	5‰	5‰	-
ϵ_{trans}	0.01, ..., 2‰	-	-
d	2	2	-
d_{trans}	1	-	-
$f^{+}(\lvert g \rvert)$	$20 - \lvert g \rvert/30$	$20 - \lvert g \rvert/30$	-
$f^{-}(\lvert g \rvert)$	$1 + \lvert g \rvert/30$	$1 + \lvert g \rvert/30$	-
$f^{+}_{trans}(\lvert g \rvert)$	10	-	-
$f^{-}_{trans}(\lvert g \rvert)$	10	-	-
Parents/Childr.	8/1	8/1	8/1
t^{fit}	50 cyc	50 cyc	50 cyc
t^{evap}	300 cyc	300 cyc	300 cyc

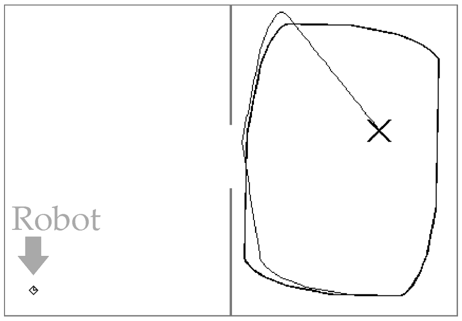

Figure 7. Trajectory of an evolved collision avoiding robot without gate passing (group 1). A robot is drawn to scale.

setting showed the best performance in [12].

Resulting Behavioral Automata. The evolved automata can be roughly separated to three groups (ascending in collaboration and population fitness):

1) individual collision avoidance without gate passing,
2) "selfish" gate passing, and
3) "altruistic" gate passing with collision avoidance.

Fig. 7 shows the trajectory of an automaton from group 1), executed on a single robot in an empty field; the "X" marks the starting point. The robot is able to avoid collisions with walls, but it cannot pass the gate. Behaviors from this group are the least complex as avoiding collisions can be done by a very simple two-state automaton. Individual fitness and population fitness are lower than in groups 2) and 3), since the gate passing bonus is not (or rarely) achieved.

Automata from Group 2) are more complex as they involve detecting and passing the gate (very robustly in many cases). However, these automata are only capable of exploiting the gate bonus for a small number of robots at once (one or two mostly) which pass the gate constantly and, therefore, have a high fitness. The population fitness is lower than in group 3), since the other robots cannot pass the gate at the same time. In some cases, also a collision avoidance is part of the behavior. Fig. 8 shows the trajectories of two robots with automata from group 2), finding the gate and passing it constantly.

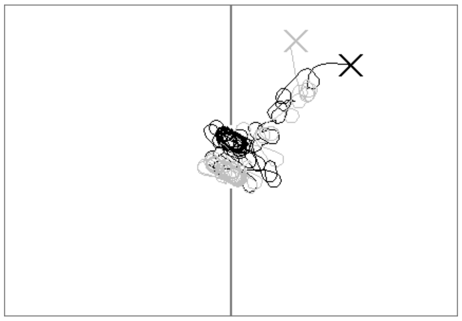

Figure 8. Trajectories of two evolved robots finding the gate and passing it constantly together (group 2).

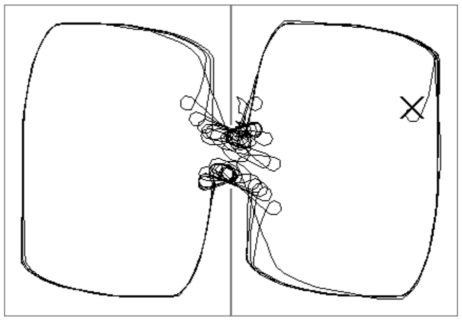

Figure 9. Trajectory of an evolved robot following walls and thus passing the gate by still avoiding collisions (group 3). Turning right is done by turning left for about 270 degrees in little circles which makes the according automaton surprisingly simple (cf. Fig. 10).

Group 3) involves the most adapted resulting automata. Here, the whole population is capable of exploiting the gate bonus leading to a high population fitness. An example behavior from group 3) is shown in Fig. 9 where a *wall following* automaton has been evolved, meaning every robot is driving parallel and close to the wall, following its course. Therefore, all robots can drive in a line and pass the gate twice during a circulation of the field. The depicted trajectory is additionally interesting as it is produced by the very simple automaton shown in Fig. 10 (tautological parts of conditions have been removed). The automaton does not even have a state with a *TurnRight*-operation, but it emulates turning right by turning left by a large amount until the robot faces in the desired direction.

As not all populations could be observed separately, it was found evident to define populations to belong to groups 2) or 3) if the number of gate passings exceeded a certain constant. By setting this constant to 0.8 gate passings per 100 simulation cycles (which seems to match best the observed runs where one of the gate passing behaviors occurred), 45% of the runs with mutation M (fixed or flexible) belonged to these groups, but only 29% of the runs with mutation M_{old}

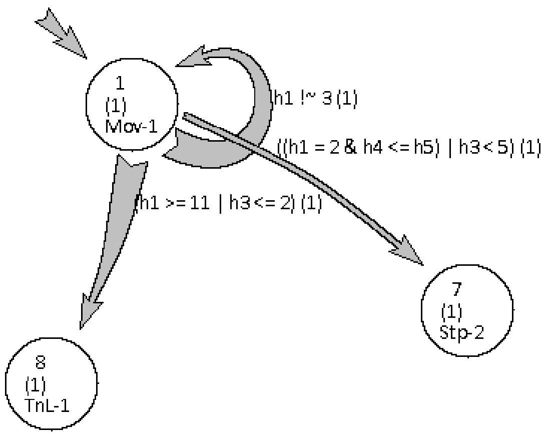

Figure 10. Surprisingly simple automaton for wall following without any *TurnRight*-operation (cf. trajectory in Fig. 9).

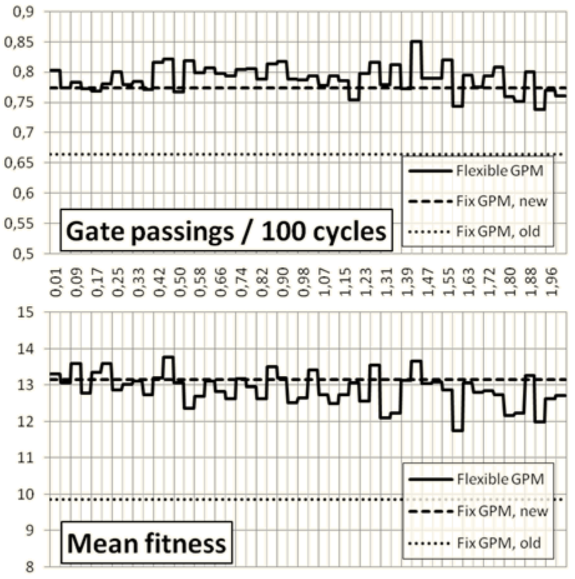

Figure 11. Mean gate passings per 100 cycles and mean fitness during runs with fixed GPM, and the 50 different ϵ_{trans}-settings (in per mill) with flexible GPM.

did. A relation between groups 2) and 3) is not provided as these groups fade into each other and are hard to distinguish even by observation. All observed populations belonged to one of the three groups as collision avoidance occurred in every run not in one of the two gate passing groups.

Overall Results. Fig. 11 shows the mean gate passings per 100 simulation cycles and the mean fitness during the runs for 1), the 50 different settings with flexible GPM (solid lines), 2), the best runs ($\epsilon = 5‰$) with fixed GPM and mutation M (dashed lines), and 3), the runs with fixed GPM and mutation M_{old} (dotted lines). The X-axis is labeled with the values for ϵ_{trans} in per mill (every other value is omitted).

Apparently, all new runs outperform the runs with old mutation in terms of the number of gate passings, indicating behavior complexity, and of mean fitness. This confirms the hypothesis enunciated in Sec. 1, that a GPM can improve performance of evolution based on FSMs.

As the figure shows, the number of gate passings is greater with flexible GPM than with fixed GPM for most values of ϵ_{trans} (on average 0.79 vs. 0.77 per 100 cycles). E. g., for ϵ_{trans} between 0.42 and 1.15, this is true in all but one case. This indicates that more complex behaviors evolved with flexible GPM than with fixed GPM which was assumed an effect of the evolvable GPM in Sec. 1. However, due to variance, the difference in performance is too small to be significant here, and a more precise study of the mutation probability ϵ_{trans} has to be done in the future. Nevertheless, we conjecture that the results confirm the assumed effect, as over a wide range of ϵ_{trans}-values the flexible GPM performs best. The optimum of these values has to be found, and with a further adaptation of parameters, we expect the flexible GPM to more clearly outperform the fixed version.

Another indication supports this conjecture: mean fitness is *lower* for flexible GPM for most values than for fixed GPM (on average 12.86 vs. 13.16). This can be explained by a higher diversity in the populations, due to two concurrent mutation operators, which leads to a higher probability for disappearance of already learnt good behavior. In runs with low mean fitness, the counted number of gate passings must have been achieved during rather short periods of good behavior while the rest of the time behavior and fitness were bad. Counterintuitively, low mean fitness combined with many gate passings during a run can, therefore, account for a high complexity of behavior which got lost again later. For future studies, experiments with the memory genome are planned, since the problem of loosing learnt good behaviors can be reduced significantly with this strategy [12].

6. Conclusion

In this paper, a new approach for encoding the genotype-phenotype mapping during evolutionary runs has been presented. Here, the GPM is encoded as part of the genotype which makes it flexible and *completely evolvable*. The introduced genotypic representation (both with fixed and flexible GPM) has been shown to improve performance of decentralized evolution compared to former approaches without an explicit distinction between genotype and phenotype on the controller level.

The flexible GPM performs better than the fixed GPM in terms of complexity of the evolved behaviors over a wide range of the tested parameters. Although the statistical significance of this result is not clear so far, there are sound indications that within this range an optimum can be found where the flexible GPM significantly outperforms the fixed GPM. For future studies, techniques like the memory genome [12] for reducing the loss of already learnt good behaviors will be used to verify this conjecture.

Furthermore, the influence of standard recombination operators will be analyzed, and the approach will be studied using more complex target behaviors. Experiments with real robots are also planned for the future.

References

[1] E. Bonabeau, G. Theraulaz, and M. Dorigo. *From Natural to Artificial Systems (Santa Fe Institute Studies in the Sciences of Complexity).* Oxford University Press, 1999.

[2] V. Braitenberg. *Vehicles: Experiments in Synthetic Psychology*. Cambridge, MA: MIT Press, 1984.

[3] D. Floreano, P. Husbands, and S. Nolfi. *Evolutionary Robotics in Springer Handbook of Robotics*. Springer, 2008.

[4] D. J. Futuyma. *Evolutionary Biology*. Sinauer Associates Inc., 1998.

[5] E. J. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[6] N. Jakobi. Evolutionary robotics and the radical envelope-of-noise hypothesis. *Adapt. Behav.*, 6(2):325–368, 1997.

[7] N. Jakobi, P. Husbands, and I. Harvey. Noise and the reality gap: The use of simulation in evolutionary robotics. In *Advances in Artificial Life: Proc. 3rd European Conference on Artificial Life*, pages 704–720. Springer-Verlag, 1995.

[8] Y. Jin. A comprehensive survey of fitness approximation in evolutionary computation. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 9(1):3–12, 2005.

[9] R. E. Keller and W. Banzhaf. Genetic programming using genotype-phenotype mapping from linear genomes into linear phenotypes. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 116–122. MIT Press, 1996.

[10] M. Kimura. *The Neutral Theory of Molecular Evolution*. Cambridge University Press, 1985.

[11] L. König, K. Jebens, S. Kernbach, and P. Levi. Stability of online and onboard evolving of adaptive collective behavior. In *Springer Tracts in Advanced Robotics*, 2008.

[12] L. König, S. Mostaghim, and H. Schmeck. Online and onboard evolution of robotic behaivior using finite state machines. In *8th International Conference on Autonomous Agents and Multiagent Systems*, 2009.

[13] L. König and H. Schmeck. Evolving collision avoidance on autonomous robots. In M. Hinchey, A. Pagnoni, F. Rammig, and H. Schmeck, editors, *Biologically Inspired Collaborative Computing*, 2008.

[14] J. R. Koza. *Genetic Programming – On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[15] S. Nolfi and D. Floreano. *Evolutionary Robotics. The Biology, Intelligence, and Technology of Self-Organizing Machines.* The MIT Press, Cambridge, Massachusetts, 2001.

[16] J. Walker, S. Garrett, and M. Wilson. Evolving controllers for real robots – a survey of the literature. *Adaptive Behavior*, 11:179–203, 2003.

[17] R. Watson, S. Ficici, and J. Pollack. Embodied evolution: Distributing an evolutionary algorithm in a population of robots. In *Robotics and Autonomous Systems*, pages 1–18, 2002.