Compositional Skills Building

Lorenzo Riano[a,1,*], T. M. McGinnity[a]

[a]*Intelligent Systems Research Centre, University of Ulster, Londonderry, BT48 7JL, United Kingdom*

Abstract

We propose a robotics algorithm that is able to simultanously combine, adapt and create actions to solve a task. The actions are combined in a Finite State Automaton whose structure is determined by a novel evolutionary algorithm. The actions parameters, or new actions, are evolved alongside the FSA topology. Actions can be combined together in a hierarchical fashion. This approach relies on skills that the robot is already provided with, like grasping or motion planning. Therefore software reuse is an important advantage of our proposed approach. We conducted several experiments both in simulation and on a real mobile manipulator PR2 robot, where skills of increasing complexity are evolved. Our results show that i) an FSA generated in simulation can be directly applied to a real robot without modifications and ii) the evolved FSA is robust to the noise and the uncertainty coming from real-world sensors.

Keywords: Robotics, Finite State Automata, Evolutionary Algorithms, Neural Networks

1. Introduction

A close look at the literature in robotics reveals that robots are increasingly being provided with sophisticated skills. Many of these skills are coded in programs or routines freely available to researchers and engineers alike, so that they can build more sophisticated systems. However, in spite of the huge array of skills available, the creation of truly effective autonomous robotics systems still evade researchers.

Among other reasons, we believe this is due to the difficulty of creating a reliable complex system, even when such a system is composed of already available parts. Even if a single skill is based on a well-known and reliable technology, there are little or no tests of the same skill when it has to be integrated with other skills. Most of the time even this reliability cannot be proven, as the robot is required to operate in a world where a huge amount of information comes from noisy and deceptive sensors.

When faced with the task of designing a robot that solves a particular problem, a roboticist has to answer the following questions:

- Which skills are needed and how to combine them.
- How skills should be modified to work in cooperation with others.
- Which skills are not available and need to be created.

In this paper we propose to address these problems in a autonomous way and within the same framework. We rely on the assumption that many low and high level actions can be reliably performed by a robot using *ad-hoc* algorithms. These include for example object detection, motion planning and grasping. The role of our proposed algorithm is therefore to i) structure and organise the execution of available actions, ii) adapt these actions to solve a particular problem, iii) create new actions when necessary. This process can be carried on in a hierarchical fashion, and hence we use the term "compositional skills building". Given this compositional nature of skills into actions, we can interchangeably use the terms actions (created by our proposed algorithm) and skills (provided by an external routine).

In our proposed approach an action is performed by a Finite State Automaton (FSA, plural automata) [1], whose nodes represent skills that are externally provided to the robot (or previously created actions) and whose transitions are the outcomes of the actions. Each action can have a set of parameters. The FSA are instantiated by a an evolutionary process [2] that simultanously evolves the topology of the FSA and the parameters of the actions. Given the particular problem that we aim to solve we have been required to devise new evolutionary operators, as described in section 3.6 and section 3.7.

The proposed approach does not depend on a particular implementation of an action. Therefore if a new action is provided that performs better than an old one, the corresponding node in the evolved FSA can be replaced with the new action without impairing the functionalities of the FSA. Reuse of components thus becomes an important advantage of our proposed algorithm.

*Corresponding Author

Email addresses: l.riano@ulster.ac.uk (Lorenzo Riano), tm.mcginnity@ulster.ac.uk (T. M. McGinnity)

[1]Dr. Riano is supported by InvestNI and the Northern Ireland Integrated Development Fund under the Centre of Excellence in Intelligent Systems project.

We have conducted several experiments, both on a real robot and on a simulator, and we conducted extensive testing to prove that the obtained FSA can reliably drive the robot to solve the problem for which they were evolved. To illustrate the work we provide four videos showing experiments with the real robot, and pointers to the source code which implements the proposed approach.

This paper is organised as follow: in section 2 we review the related work and the main differences between our approach and others in literature; in section 3 we consider the techniques and the algorithm we developed; in section 4 we describe three of the experiments we conducted; the results are discussed in section 5; finally in section 6 we draw the conclusions and we consider plans for future work.

2. Motivation and Related Work

The idea of sequencing a robot's behaviour in several sub-actions has been explored several times in the past. In Brook's seminal work [3] a robot is controlled by several sub-modules called behaviours. The organisation and arbitration of these modules was handcrafted. Subsequently hybrid architectures have been proposed to provide the robot with higher-level skills [4]. Here a planning module played a key role in that this module's task was to choose which behavior should be activated at which time step. In the classic formulation of hybrid architectures the behaviours were rigid structures that could not be changed or adapted by the planner.

More recent planning-based works [5, 6] include the possibility to optimise the free parameters of two subsequent actions so that the overall execution of the plan is optimal. This optimisation happens only when two actions have to be performed together, therefore it applies only to a limited set of scenarios. Plans are constructed on-line by using knowledge extracted from the web. A plan is then executed by invoking elementary program units, which are analogous to the actions we define in section 3.1. The concept of elementary programs, or actions, is used also in [7] in the context of manipulation and geometric planning.

Our approach borrows ides from the robotics planning literature, in that we share with it some of the goals outlined in the previous section. An FSA can be seen as the structure that instantiates and carries on the execution of a plan. However, unlike a planning system, we do not require a detailed description of every action's pre-coditions and post-conditions to correctly generate a plan. In section 4 we show that a simple simulator suffices to generate FSA that are robust when applied to a real robot. A second major difference between our proposed approach and classic planning is that we allow actions to be adapted to a particular problem by varying their parameters. Moreover, as we show in section section 4.2, in our proposed approach adapting an action or creating a new one is perfomed in the same framework, while the same might not be as easy using classical planning approaches.

A different approach can be found in [8], where the authors propose a Reinforcement Learning algorithm that allows skills to be created and hierarchically combined. The same work has been applied to a robotic domain in [9]. Although this work provides an approach to skills building than could be considered more sophisticated than ours, it does not allow for the adaption of old skills nor it provides a convenient way to structure actions. Other approaches that involve sequencing actions to obtain a robot controller are described in [10, 11], although their scope is mainly to build low-level controllers.

Several approaches have been proposed to learn an FSA, many of which focus on approximating a grammar. Some of the most successful use an evolutionary algorithm approach, like [12, 13]. As our scope is different, we decided to adopt strategies similar to those adopted in neuroevolution [14]. The main problem when evolving the topology and weights of a neural network is using a crossover algorithm that does not destroy the information content of the offspring networks [15]. One of the main successful approaches has been NEAT [16]; here the evolutionary history of genes is recorded so that they can be later aligned before performing crossover. This approach has proven to be successful in a variety of tasks. Other more recent appraoches are presented in [17, 18].

Although evolving an FSA is similar to evolving a neural network, we see a fundamental difference in that in an FSA states that are directly linked are very likely to be strongly interdependent. Therefore preserving the connectivity between subsets of FSA while performing crossover has a stronger importance than in neural networks. This motivated us to propose the new evolutionary algorithm for the evolution of FSA presented in the next sections.

To summarise, this work is motivated by the following considerations:

- Several robotics skills are proved to reliably deliver good performance in a real-world scenario. Any complex action can and should make use of them whenever possible.
- Combining skills is often not sufficient to solve a task. Skills need to be adapted or newly created when needed.
- Although classical planning approaches allow for the composition and sequencing of actions to reach a goal, to the best of our knowledge no planning system proposed so far allows for skills to be adapted (with the limited exception of [5, 6], see the discussion above) or created during the planning process.
- Learning an FSA has so far focused on grammatical inference. Therefore we had to devise a new algorithm which is suitable for our scope.

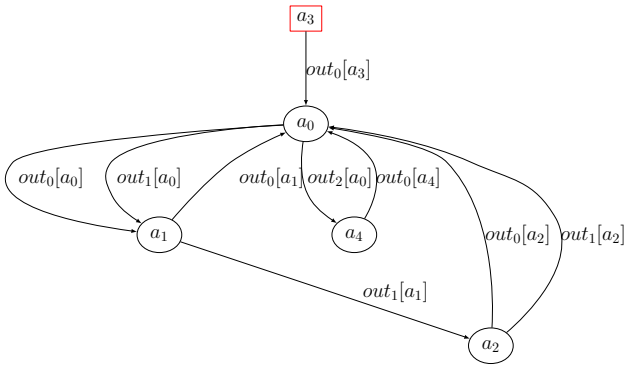

Figure 1: The schematic representation of an FSA. An FSA is represented as directed graph with parallel edges. Every node is associated to an action. The red squared node is the starting node. To the right of every edge is its label, i.e. an outcome of an action.

3. Techniques

3.1. Finite State Automaton

Given our particular application, our formulation of the FSA differs from the classic one adopted for example in [1]. We define an FSA as a quadruple (A, O, δ, s), where:

- A is a finite, non-empty set of actions.
- O is a finite set of outcomes. Each outcome j of state a_i is denoted by $out_j[a_i]$.
- $\delta : A \times O \to A$ is the transition function.
- s is the initial state.

In addition to the above, all the states have a (potentially empty) set of real-valued parameters and they share a common memory where they can read and write data. The role of the shared memory is to share information so as to enable passing of data between actions (more details are in section 3.8 and in section 4). An example of an FSA is given in Figure 1. The parameters are used to adapt an action to a particular problem.

The main difference with the model illustrated in [1] is the lack of the input alphabet, or inputs. This means the the transition from one action to another depends on the action outcome only. However the common memory replaces and enhances the input function. Although the main features and behaviour of our proposed model still closely resembles the classic FSA's one, the presence of a common memory makes it closer to a Turing Machine than to an automaton.

In the following we will always use the above definition of the FSA unless otherwise stated.

3.2. Evolutionary Algorithm

The evolutionary algorithm we used follows the general standard structure described for example in [2], and it is shown in Algorithm 1. The implementation made use of the library PyEvolve described in [19].

Algorithm 1 The evolutionary algorithm

```
Initialize a population of N individuals (section 3.5)
while Task is not solved do
    Evaluate each individual in the population according to the task (section 4)
    for i = 1 → N/2 do
        Select two individuals g1 and g2 from the population using Tournament Selection [2]
        With probability p_cross generate two children c1 and c2 using crossover (section 3.7)
        With probability p_mut mutate both c1 and c2 (section 3.6)
    end for
end while
```

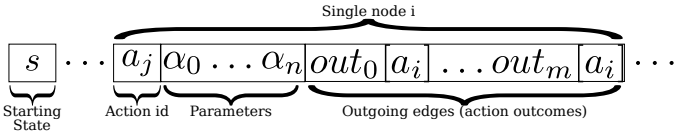

Figure 2: A linear representation of a genome. Every genome has a starting state, followed by a serie of elements each of them representing a node's associated action, parameters and outgoing edges.

3.3. Genome Representation

The genome is represented as a directed graph $G = (V, E)$ with parallel edges, where V is the set of the nodes and E is the set of edges. A node $v_i \in V$ is associated with a single action type $a_j \in A$, and it has a (possibly empty) list of real-valued parameters $0 \leq \alpha_i \leq 1$ (as in our model of FSA described in section 3.1). The meaning of the parameters is action-specific. As every action a_j has a fixed number of outcomes, every node will have a specific number of outgoing edges, each of them representing the specific outcome of an action. In addition to the nodes and edges, the genome encodes the FSA starting state. There is no restriction on the action type the node can be associated to, and several nodes can have the same action type.

Figure 2 shows a linear representation of a genome. In all our experiments we did not use this linear representation, but we resorted to a direct graphical one using the tool illustrated in [20]. In the following we will use the term genome and graph to refer to the same structure presented above.

Crossover and mutation happen with probabilities p_{mut} and p_{cross} respectively. Although the literature discussing the choice of these parameters is rich (see for example [2]), no optimal solution has been found so far. After extensive experiments we found that the parameters listed in Table 1 yield good results, although we strongly believe that the parameters choice does not have a strong influence on the overall evolutionary algorithm performance.

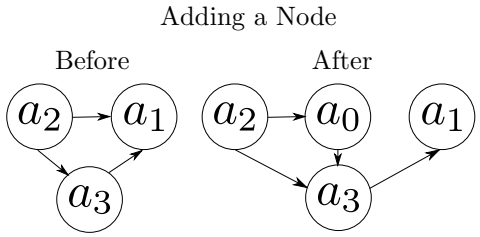

Figure 3: An example of adding a node to a graph. The new node is labeled α_0 and it is inserted between α_2 and α_1, breaking their joining edge. As α_0 has one outgoing edge this is randomly created, joining α_0 and α_3. All the nodes are reachable from the starting node α_2 so no node is pruned away (see section 3.4).

3.4. Genetic Operators

In our proposed evolutionary algorithm we developed both mutation and crossover operators. Each of these operators has to guarantee that every node in the graph has the same number of outgoing edges as the number of outcomes of the associated action. Moreover every node which is not reachable from the starting node will be removed from the graph. This is to ensure that the evolutionary search is not wasted in areas that do not contribute to the overall fitness function.

3.5. Initialization

During the initialization phase for each graph a fixed number S of nodes is created, each of them with a random parameters vector and a random associated action. Then for each possible outcome of the associated action a new edge to a randomly selected node is created. The graphs are then pruned of the unreachable nodes (see previous section). Therefore although the graphs start with the same number of nodes, their size changes before the evolutionary algorithm starts.

3.6. Mutation

Mutation happens both at the graph-level and at the node level. At the graph level one or more of the following mutations can happen:

- A new node is added to the graph with probability p_{mut}. The new node will have randomly initialized parameters and outgoing edges. To ensure that the new node will be reachable from the starting one, a random node with a least two incoming edges is selected from the graph and one of its incoming edges is re-routed to the new node. An example of this process is illustrated in Figure 3.
- A random node is removed from the graph with probability p_{del}. All the incoming edges are rerouted to randomly selected nodes.
- The starting node is changed. The new one is selected among the nodes which have at least one outgoing edge and not all the edges are self-loops.

At the node level the following mutations can happen:

- A random outgoing edge is re-routed to a different randomly selected node in the graph.
- Zero or more parameters are added a zero-mean normal distributed random number.
- The node's associated action changes. In this case the parameters and all the outgoing edges are randomly recreated. As this is a more drastic change in the overall graph topology, it happens with probability p^2_{mut}.

3.7. Crossover

Crossover is performed between two parents genomes, g_1 and g_2, to create two children graphs c_1 and c_2. As explained in section 2 crossover has to ensure that groups of nodes that are potentially working together will not be broken in the process. Finding groups of nodes that are closely coupled is a very hard problem[2], so we assume that nodes whose distance is small (as the path length in the graph) are more likely to be working together than nodes that are far away. This allows for sub-solutions to be developed and maintained by subgraphs and to be preserved over generations if they contribute positively to the fitness function.

Algorithm 2 The crossover algorithm

1: $num\_nodes \leftarrow random()$
2: $source\_g_1 \leftarrow random()$
3: $source\_g_2 \leftarrow random()$
4: $subgraph\_g_1 \leftarrow bfs(g_1, source\_g_1, num\_nodes)$
5: $subgraph\_g_2 \leftarrow bfs(g_2, source\_g_2, num\_nodes)$
6: $edges(g_1) \leftarrow edges(g_1) \setminus edges(subgraph\_g_1)$
7: $edges(g_2) \leftarrow edges(g_2) \setminus edges(subgraph\_g_2)$
8: **for** $i = 1 \rightarrow num\_nodes$ **do**
9: swap $v_{1,i} \in subgraph\_g_1$ with $v_{2,i} \in subgraph\_g_2$
10: **for** $e \in edges(subgraph\_g_2, v_{2,i} \in subgraph\_g_2$ **do**
11: $add\_edge(g_1, v_{1,i} \in subgraph\_g_1, head(e))$
12: **end for**
13: **for** $e \in edges(subgraph\_g_1, v_{1,i} \in subgraph\_g_1$ **do**
14: $add\_edge(g_2, v_{2,i} \in subgraph\_g_2, head(e))$
15: **end for**
16: **end for**
17: Fix missing outgoing edges in g_1 and g_2

The proposed crossover algorithm is shown in Algorithm 2, while Figure 4 shows a graphical representation of two child graphs created by combining two parents, g_1 and g_2. The first steps of the algorithm are to select a random number of nodes to swap (num_nodes) and an initial starting node for both graphs (lines $1-3$); in

[2]Even if it was simple, enforcing strict topological structures will limit the capability of evolutionary algorithms to find novel solutions or even a solution at all.

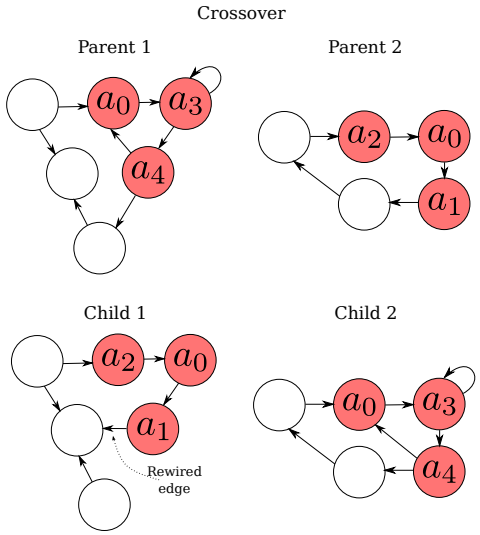

Figure 4: An illustration of the crossover algorithm. The two top graphs are the parents, while the two bottom graphs are the resulting children. See text for details.

Figure 4 these are nodes a_3 and a_0 respectively and 3 nodes have been selected for swapping. In lines $4-5$ we select the subgraphs that will be swapped. As described above we assume that close nodes are more likely to be closely working together than distant ones. For this reason we select a subgraph made by *num_nodes* nodes via Breadth First Search (BFS) starting from the previously selected starting nodes. In Figure 4 these are the nodes a_0, a_3, a_4 for graph g_1 and a_2, a_0, a_1 for g_2. Two subgraphs are created by using the extracted nodes and all the edges between them (lines $6-7$ in Algorithm 2 and the red nodes in Figure 4). In lines $6-7$ all the edges in the newly created subgraphs are removed from the original graphs g_1 and g_2. Finally, the nodes in both subgraphs are swapped (i.e. the associated action and parameters are swapped), and the edges are recreated (see the bottom part in Figure 4). An effect of this procedure is that some nodes (notably the ones belonging to the extracted subgraphs) will have missing outgoing edges. Therefore new edges are created for all the nodes whose outcomes are not matched by an outgoing edge. This is shown in the bottom left part of Figure 4, where the node a_1 had one of its edges randomly rewired.

3.8. Simulation

One of the main limitations of evolutionary algorithms in robotics is that they require a simulator to be effective. This is due to the fact that the same experiment has to be carried on hundreds if not thousands of times in order to reach a solution. This is not only extremely time consuming when performed with a real robot, but it could also be dangerous for the machine. To avoid this problem, evolutionary robotics is usually developed in simulation, then the result is applied to the real robot [21–24]. This creates gaps between a solution found in simulation and its applicability to a real robot. Although these gaps can be reduced by using a very accurate simulator, this increases the computational cost of evolutionary algorithms, to the point that using a simulator becomes as impractical as using a real robot.

In our approach this problem is avoided by concentrating on high-level actions and their effect, rather than the physics of the robot and its interactions with the environment. For example we deal with a grasping action in a high-level way: the result of *grasp(object_i)* is simply "*object_i is in the robot gripper*", without caring about the low-level details of the grasping itself. The main assumption behind this work is that several actions are already implemented, including high level ones like perception, grasping, navigation and localisation. If a better grasping algorithm is provided, the old one can be replaced with the new one without affecting the simulations or any previously evolved FSA. This is therefore an implementation of the "minimal simulator" approach described in [25]. Noise can be introduced by allowing actions to have a *failure* outcome based on pre-conditions not being met or even a random event.

All the actions described in section 4 have been implemented in the abstracted way described above. As mentioned in section 3.1 all the states share a common memory where they can read or write data. We used this memory as a high level representation for both the robot and the world state. Every action queries the common memory to find if it can be executed, and the result of its execution is written in the common memory.

Once the evolutionary algorithm has converged to a solution, only the meaningful parts of the FSA are retained to be used on the real robot. For example one could keep the resulting FSA topology and replace all the simulated actions with the real ones, or one could only use the parameters that our algorithm has found for one or more actions. Therefore the amount of information to retain from the result of the evolutionary algorithm depends on the task the robot has to solve and on the particular actions that are needed to be executed.

The above approach allows for fast simulations in the context of evolutionary algorithms while still obtaining an easy transfer of the solution evolved in simulation to the real robot, as we will show in section 4.

4. Experiments

We tested our proposed system in three different experiments, of which two have been applied to a real robot. The robotic platform we used is a mobile manipulator PR2 robot manufactured by Willow Garage[3]. It is two-armed with an omni-directional driving system. Each arm has 7 degrees of freedom. The torso has an

[3] `http://www.willowgarage.com`

Figure 5: The moble manipulator platform PR2. The robot has two $7DOF$ arms and an holonomic base. The pan-tilt head unit is equipped with two stereo cameras, one high resolution camera and one texture projector (the red light). A tilting laser and a fixed laser on the base are used for navigation and motion planning.

additional degree of freedom as it can move vertically. The PR2 has a variety of sensors, among them a tilting laser mounted in the upper body, two stereo cameras (with narrow and wide field of view) and a laser scanner mounted on the base which is used for mapping and navigation. Several skills that we relied on are developed by the ROS[4] community, and they include:

- Detecting and grasping unknown objects using $3D$ information [26].
- Planning and executing a collision-free trajectory with the $7DOF$ arms [27].
- Navigation and obstacle avoidance using an omni-directional base [28].

For all the experiments we used the same set of parameters summarised in Table 1. The code for the simulations is available on-line[5] where all the experiments described below are included.

4.1. Pouring Water

We first conducted a pilot experiment to test the capabilities of the proposed evolutionary algorithm. The robot is presented with a jug of water and a glass, and its task is to pour water into the glass. The set of actions it can perform are:

Table 1: Parameters used in the experiments.

Parameter	Value	Description
p_{mut}	0.1	Mutation probability, section 3.6.
p_{cross}	0.1	Crossover probability, section 3.7.
p_{add}	0.2	Probability of adding a node, section 3.6.
p_{del}	0.01	Probability of deleting a node, section 3.6.
N	300	Number of individuals, algorithm 1
S	50	Initial number of nodes for each graph, section 3.5

- **RecogniseObject**: The robot uses the stereo camera in front of it and finds the $3D$ location of both the jug of water and of the glass. The objects' locations are stored in the common memory. This action has only the outcome *success* as we assume the objects are visible and recognisable by the robot.
- **MoveToObject**: The robot approaches the jug of water with one arm. This action is required to grasp an object. The action can end with an outcome of *success* with probability 0.8 and with an outcome of *failure* with probability 0.2 or if the object's location is not in the common memory. The effect of this action is that the gripper position will be within $10cm$ of the jug. WHERE 0.2 AND 0.8 COME FROM
- **GraspObject**: The robot grasps the jug. The action has an outcome of *success* with probability 0.8 and an outcome of *failure* with probability 0.2 if either the objects have not been detected or the gripper has not approached them. The presence of the object is recorded in the common memory if the action succeeds.
- **MoveGripperToParam**: The robot moves the gripper to a x, y, z position in the space specified by the 3 action's parameters. The action can end with an outcome of *success* with probability 0.8 and with an outcome of *failure* with probability 0.2. The new position of the gripper is recorded in the common memory.
- **RotateGripperToParam**: The robot rotates the gripper in place to a ρ, ϕ, θ angle specified by the 3 action's parameters. The action has only the *success* outcome.
- **CheckSuccess**: The robot checks if the water is in the glass. This action fails if water has not been

[4] `http://www.ros.org`
[5] `https://github.com/lorenzoriano/Graph-Evolve`

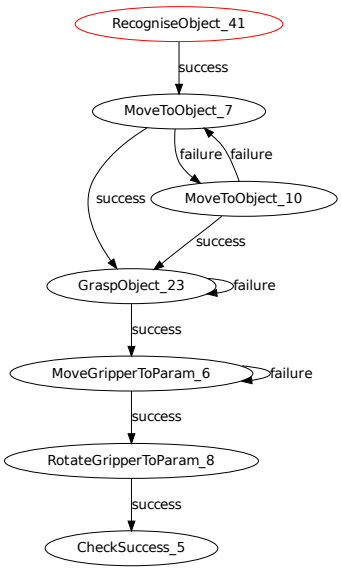

Figure 6: The FSA evolved to pour water.. The red circle marks the initial state. For clarity the parameters and the *timeout* outcomes are not shown. Every node is labelled with the action it is associated with. The additional number is to distinguish between several instances of the same action.

poured or if the robot rotated the gripper, with the jug of water, before having moved over the glass. The two outcomes *success* and *failure* terminate the state machine execution.

In addition to the above outcomes, all the states have a *timeout* condition if the FSA runs for more than 20 steps, to avoid infinite loops. We introduced a random *failure* condition for most of the actions to simulate the possibility of the actions not completing successfully (for example the motion planner might fail to find a solution in a given time slot). This is also to induce robustness in the evolved FSA.

Finding a solution to this problem is not trivial as there is only one topology that executes all the actions in the right order. The same is true for the 6 parameters that need to be evolved, as only one set of real values leads to a correct solution. We used the fitness function f in Equation (1), where $d(jug, glass)$ is the euclidean distance between the jug of water and the glass. This fitness function has been designed to promote intermediate solutions so that they can be combined with the crossover algorithm described in section 3.7. The goal of the evolutionary algorithm is to minimise the fitness.

$$f = \begin{cases} 0 & \text{if the FSA's outcome is success} \\ 20 & \text{if the jug is not in the gripper} \\ 2 \times d(jug, glass) & \text{if the FSA's outcome is timeout} \\ d(jug, glass) & \text{otherwise} \end{cases} \quad (1)$$

The evolutionary algorithm evolved the FSA with 7 states after 418 generations, as shown in Figure 6. The numbers after the state names are used to discriminate

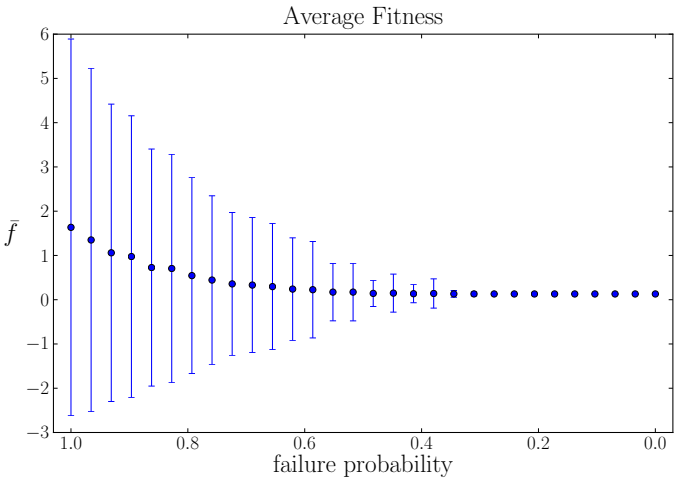

Figure 7: The fitness value obtained by FSA of Figure 6. The plot shows the mean value obtained after 5000 repetitions of the same experiment, while the vertical bars represent the standard deviation.

between nodes associated with the same action (e.g. "MoveToObject_7" and "MoveToObject_10"). We tested the FSA with failure probabilities for all the actions ranging from 0.0 to 1.0. Each test has been repeated 5000 times. Figure 7 shows the mean fitness value obtained for each value of the failure probabilities, together with the associated standard deviation. It can be observed that with failure probabilities less than 0.5 the FSA correctly solves the task almost all of the time, i.e. the outcome is *success* and the fitness given by Equation (1) is 0.

Unlike the following experiments, for safety reasons, we did not test this solution on the real robot.

4.2. Moving to Grasp

Many robotics applications require the robot to be able to manipulate objects. The approach we use to grasp an object [26] works only if the object is reachable by the robot. However during our experiments we found that an object is often hard to reach, even if it is close to the robot. This is due to physical constraints of the robot's arms that are not easy to analytically model. Several approaches have been proposed [29, 30] to deal with this problem, and the results prove that this is still a hard benchmark for robotics algorithms. We therefore decided to use our approach to find a general solution.

The main goal of this experiment is to show how our proposed approach can generate novel actions when the available ones are not sufficient to solve a problem. Our generic action is represented by a fully recursive neural network with fixed topology [31] whose weights are represented by the parameters of the associated node in the FSA. We did not employ any specific neuroevolution technique to evolve the network's topology and weights (see for example [14, 22]) as we decided to focus on the evolutionary algorithm we proposed.

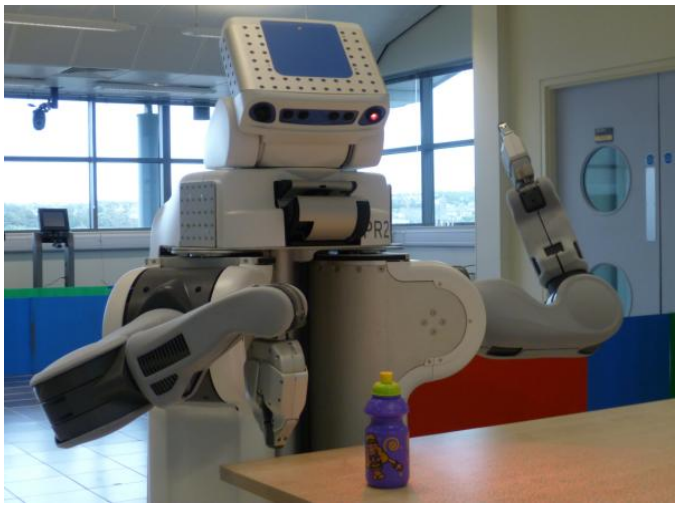

Figure 8: The PR2 while pushing an object. Several random object's positions were tested, and the robot had to use both arms.

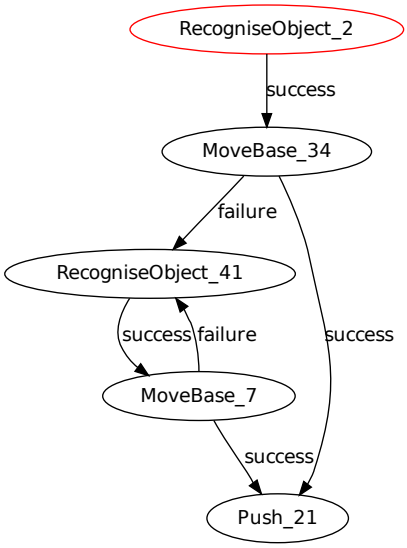

Figure 9: The FSA evolved to reach and object and push it.. The red circle marks the initial state. For clarity the parameters and the *timeout* outcomes are not shown. Every node is labelled with the action it is associated with. The additional number is to distinguish between several instances of the same action.

Representing a grasping scenario requires an accurate simulation of the environment and the robot. As this will slow down the evolutionary process, a fast computation of an action's outcome is required. In [32] the outcome of actions in a RoboCup scenario is learnt using neural networks, while in [33] the same is obtained using clustering and decision.

In this experiment we decided to use a Radial Basis Function neural network (RBFNN) [34] to classify whether an object is reachable or not. We collected training and validation data over a full day of experimentations, where the robot was moving randomly with respect to an object on a table (see Figure 8). To generate more random positions the robot had to try to push the object instead of grasping it. During our experiments we found that if the robot can push an object then it can grasp it as well. The opposite is not necessarily true, so pushing is a harder task than grasping. For each tentative push we recorded the position of the object in the robot's frame of reference, the robot's torso height and which arm was used to push the object, or if the object is unreachable. Overall we collected 872 data points: of these 500 points were used for training and 372 for validation. We then trained a RBFNN with 5 inputs, the x, y, z position of the object in the robot frame of reference, the angle θ between the robot and the object on the $x - y$ plane and the robot's torso height h. The RBFNN classifies the input into three classes, 0 if the object is unreachable, 1 if it is reachable with the left arm and 2 if it is reachable with the right arm. After training the network had a performance of 92% correct classifications over the validation data set. We have thus obtained a fast way to determine if an object is reachable from a given position: this can be used in a simulator without resorting to a computationally expensive analysis.

The next step was to generate an FSA to move the robot and push an object. The simulation environment includes a table with an object placed on top and a robot which starts within $2m$ of the table. Every time the FSA is evaluated the table position and dimensions, the object position and the robot position are randomly generated. The environment is $5m^2$ and it has no other obstacles apart from the table. The robot can use the following actions:

1. **RecogniseObject**: this is the same action as in section 4.1, with the difference that only one object will be detected.
2. **MoveBase**: this action is executed based on a fully recurrent neural network with 5 inputs, 3 hidden neurons and 5 outputs. The inputs are the object location in the robot's frame of reference as described above, and the outputs are the x, y, θ position and orientation of the robot base, the robot's torso height, and which arm to use. To represent the network's weights 38 parameters were necessary. This action ends with *failure* if the desired position, calculated by the neural network, collides with the table, or with *success* otherwise. The new position of the robot is registered in the common memory.
3. **Push**: this action makes use of the previously trained RBFNN to determine if it can push the object. It ends with either *success* if the RBFNN predicts that the object is reachable with the arm chosen by the **MoveBase** action, or with *failure* if the object is not reachable or if the network chose the wrong arm. The FSA will exit with the outcome of this action.

We used a simple fitness function that returns 1 if the FSA terminates with *success*, 0 otherwise. Given the large number of parameters required to be determined the evolutionary algorithm took 1633 generations to converge

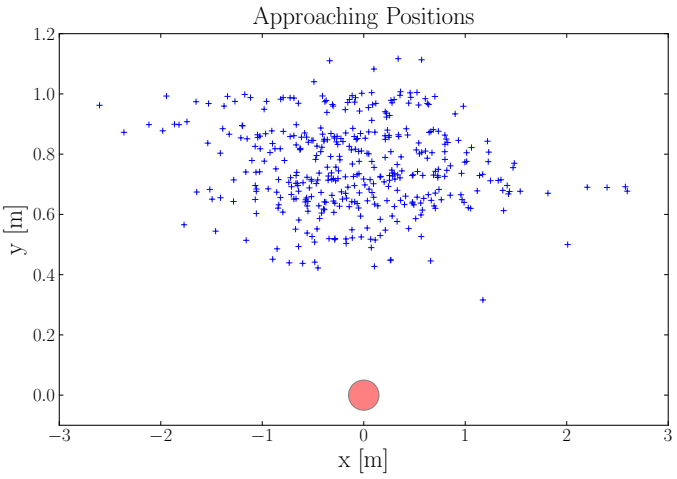

Figure 10: Testing the Moving to Grasp action. The crosses represent the locations of the object in the robot frame of reference, while the red circle represents the robot. This plot has been obtained by placing both the robot and the object in several random locations. 395 locations are represented.

to a solution. The resulting FSA is shown in Figure 9. The two actions **MoveBase_34** and **MoveBase_7** represent two different neural networks, or two different approaches to the same solution.

In order to test the evolved FSA on the real robot we fed the output of **MoveBase** to the navigation system described in [28]. The **Push** action has been substituted with the real push action we used to train the RBFNN. The FSA has been tested with the 395 random starting locations shown in Figure 10. All the positions are shown in the robot's frame of reference. As the robot needs to locate the object before attempting to reach it, we only allowed the object to be not more than 1.5 meters away from the robot and not behind it. The robot was able to reach and push the object 323 times out of the 395 tests, thus obtaining a success rate of about 82%. Most of the failures where due to the **MoveBase** action generating poses that were unreachable by the robot, given the safety constraints imposed by the navigation system.

Although we cannot claim that the collected training set exhaustivly represents all the possible object's locations the robot could face, the generalization capabilities of the RBFNN proved to be sufficient to interpolate new data points, as proved by the experiments with the real robot. Moreover, as the object location is expressed in the robot frame of reference, these points are rotational and translational invariant, i.e. a form of extrapolation is possible and it has been used by the evolutionary algorithm.

The video "moving_to_push.avi" shows one example where the robot first searched for the object, then adjusted both the torso height and its position, and finally pushed the object. The source code for the RBFNN and the fully recurrent neural network is available separately[6].

4.3. Stacking Objects

The final experiment's goal is to show how previously evolved actions can be used in a new evolutionary configuration and to study the interplay between parameters of different actions. We devised a scenario where a robot is facing two objects and it has to stack one over the other. There are no restrictions on which object the robot is allowed to take, as long as at the end of the trial one object is positioned over the other.

The FSA is allowed to use the following actions:

1. **RecogniseObject**: this is the same action a in section 4.1.
2. **GraspParam**: this action allows the robot to grasp one object. It has one parameter that, if greater than 0.5, the robot will grasp the first object, otherwise it will grasp the second. The behavior and the outcomes are similar to the **GraspObject** action in section 4.1, with the difference that no **MoveToObject** action is required to approach the object. It has a success probability of 0.7. If successful the new gripper position and the which object the robot is holding is stored in the common memory.
3. **MoveToReach**: this is the action evolved in the previous experiment. The new pose of the robot is stored in the common memory. This action has a success probability of 0.7.
4. **MoveGripperToStack**: this action has 4 parameters, the first one is used to decide which object to move the gripper over (in a similar way to the parameter of the **GraspParam** action), while the other 3 parameters represent the x, y, z displacement, in respect of the chosen object, used to calculate where to move the gripper. The new position of the gripper (and of the object, if it has been grasped) is stored in the common memory. The action has a *failure* outcome if i) random event with probability 0.3 or ii) the new gripper position collides with the table or an object or iii) the desired pose is not reachable, which is determined by the RBFNN described in the previous experiment. The other possible outcome is *success*.
5. **OpenGripper** this action opens the gripper and releases an object if it was being held. The object falls along the z axis and stops when it hits an obstacle. This action has only a *success* outcome.
6. **CheckSuccess**: this action checks, via a collision detection algorithm that has been implemented in the simulator, if the two objects are stacked. This action returns *success* if the objects are stacked, *failure otherwise*. The FSA execution terminates with this action.

[6] `https://github.com/lorenzoriano`

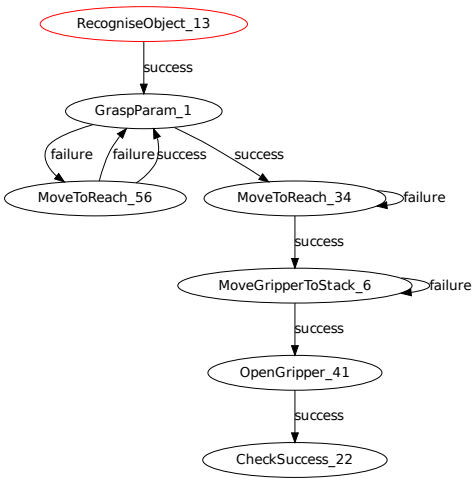

Figure 11: The FSA evolved to stack two objects in front of the robot.. The red circle marks the initial state. For clarity the parameters and the *timeout* outcomes are not shown. Every node is labelled with the action it is associated with. The additional number is to distinguish between several instances of the same action.

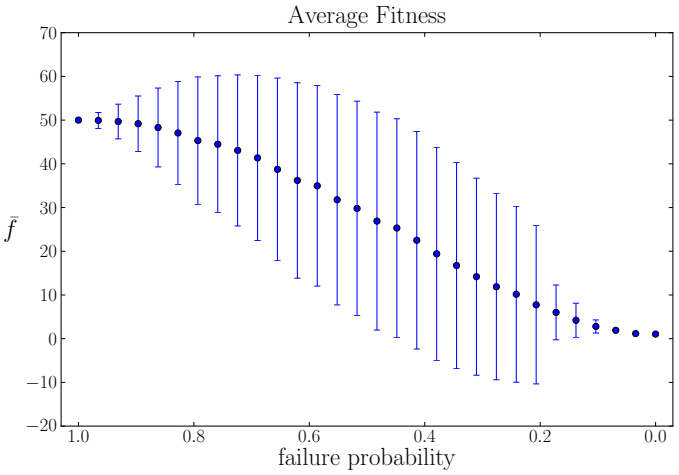

Figure 12: The fitness value obtained by stacking FSA of Figure 11. The plot shows the mean value obtained after 5000 repetitions of the same experiment, while the vertical bars represent the standard deviation.

For this experiment we lowered the number of possible transitions to 15 before a *timeout* exit outcome is produced. The fitness function we used is shown in Equation (2), where $d(o_1, o_2)$ is the distance between the two objects. Again this function rewards intermediate solutions to allow for crossover to work properly. The goal of the evolutionary algorithm is to minimise f.

$$f = \begin{cases} 0 & \text{if the FSA's outcome is success} \\ 50 & \text{if no object is not in the gripper} \\ 2 \times d(o_1, o_2) & \text{if the FSA's outcome is timeout} \\ d(o_1, o_2) & \text{otherwise} \end{cases} \tag{2}$$

Figure 11 shows a successful FSA which the evolutionary algorithm obtained after 603 generations. We then tested the FSA with 30 different probabilities of failure, with each test repeated 3000 times (see the same test in section 4.1). The results are shown in Figure 12. It can be observed that the evolved solution is sensitive to an high failure probability: if it is less than 0.3 (the value we used during the evolution) the FSA correctly solves the problem, but with an higher failure probability the success has a much larger variability. This is due to the high number of actions that could fail, compared to the experiment in section 4.1, and to the smallest number of tentatives allowed, 15 instead of 20. With an high level of failure the robot does not manage to grasp any object, hence the mean fitness distributed around 50 in the left part of Figure 11.

We then deployed the evolved FSA to the PR2 robot. The **GraspParam** actions uses the grasping algorithm in [26], while the **MoveGripperToStack** uses the motion planning library described in [27]. The **MoveToReach** is the same as the one we used on the real robot in section 4.2.

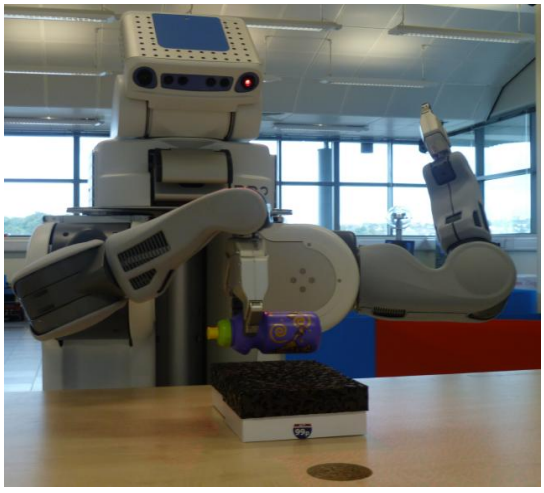

Figure 13: The PR2 while stacking a bottle over a box. The robot was controlled by the FSA in Figure 11.

We tested the FSA in 20 different scenarios, with objects of different shapes and positions (see Figure 13 for one example). The robot successfully executed the stacking action in all the scenarios. We provided three videos, "stacking_1", "stacking_2" and "stacking_3" that shows the robot dealing with different objects. Of particular interest is the third video that shows the robot placing a bottle inside a bag. This proves that the evolved FSA can deal with scenarios for which it was not evolved.

5. Discussion

As introduced in section 1 our proposed approach has three primary goals:

- **Autonomously build actions out of skills with which the robot is already provided**. In all the experiments our proposed algorithm solved the problems by evolving an FSA that made use of the actions we had provided. In the third experiment the evolutionary algorithm successfully re-used an action that had been evolved before.
- **Adapt an action to a specific problem**. In section 4.1 and section 4.3 the robot had to adapt the actions to solve the problem it was facing. In particular in the second experiment the parameters of two actions (*grasping* and *moving over*) had to be co-evolved in order to obtain a correct solution.
- **Create new actions when required**. In section 4.2 we showed that, when the parameters encode a generic calculator like a recurrent neural network, new actions can be created when the old ones are not sufficient. Moreover we proved that the resulting FSA is robust and able to deal with several real world scenarios.

In a real scenario an action fails for reasons that cannot be described by random events (however some sampling based algorithms can randomly fail, see for example [35]). This could potentially have made our simulations and results inaccurate. However, the main goal of introducing a *failure* outcome for many actions is to induce the evolved FSA to have fall-back strategies in case an action should fail. For example, planning an arm movement could fail given temporary noise in the obstacles' detection, or moving the robot base to a particular goal might fail in dynamical environments. During our experiments we found that simply retrying an action often solves the problem, if this is in principle solvable. Hence our decision to introduce random failure conditions.

One main test to verify the correctness and generality of an evolved FSA is then to test it under different levels of failures. Both the results in section 4.1 and section 4.3 show that the FSA are robust to noise. The fitness plot in Figure 12 exhibits a greater sensibility to failures than the plot in Figure 6. We believe this is due both to the increased number of degrees of freedom of the problem (the robot can move the base, together with the arms) and to the smaller number of transitions we allowed for the second task (15 instead of 20). However the FSA are robust in the range of failure probabilities we used during the evolutionary phase (0.3 in both experiments).

In all the experiments, the evolved FSA do not have an optimal structure. The FSA of Figure 6 repeats the action **MoveToObject** twice, while the FSA of Figure 11 repeats the action **MoveToReach** twice. The FSA in Figure 9 is a special case as, although it has two instances of **MoveBase**, these are different actions as the associated neural networks have different weights. However we tested a new FSA containing only either **MoveBase_34** or **MoveBase_7** and we did not observe a difference in the performance. We then concluded that both neural networks are functionally equivalent and the presence of both of them does not increase the FSA's robustness.

Observing solutions with always only two redundant nodes might seem a strange result. However, there are two forces driving the number of nodes in the FSA during evolution: the genome starts with a high number of nodes (50, as shown in Table 1), and too many nodes render the possibility of finishing with a *timeout* outcome very likely. Therefore a shorter FSA is more likely to have a greater fitness, hence the penalty for timeouts in equations (1) and (2). Our proposed algorithm therefore found that duplicating an action still achieves the highest fitness, while more than one duplication is likely to reduce it.

In the experiments above we provided the evolutionary algorithm the right set of actions to work with. This has been done to focus mainly on the main problems we aimed to solve, i.e. structure a set of actions to create a complex skill and adapt them to particular problems. Although this choice has certainly helped to obtain results in a shorter number of generations, we do not believe this has any influence over our results. Our proposed algorithms deals with genomes that initially contain 50 nodes, but whose size increases and decreases as the evolutionary process unfolds over time. For example in Figure 11 the action **MoveToReach_56** suggests that at one point during evolution there were at least 56 nodes in the FSA. Many of these nodes have been found to decrease the genome's fitness and therefore they have been evolved out. Therefore, as discussed above, the limited number of transitions an FSA is allowed during one execution acts as a force that pushes the evolutionary algorithm to look only for the nodes that positively contribute to the fitness. This in turn will automatically remove actions that do not contribute to solve a problem.

The same argument applies when the evolutionary algorithm has to choose between several actions, most of them being not useful to solve the task. This is what happens when evolving a neural network, like the **MoveBase** action in section 4.2. Most of the evolved networks represent actions that do not contribute to the overall fitness of the FSA, and are therefore discarded.

This means that the our proposed algorithm automatically choses the right skills to solve a particular task. This process can be however very long when considering a complex task that requires several different actions to be sequenced. Therefore scalability might become an issue. Our solution is to hierarchically structure actions, as we did in section 4.3, where the **MoveToReach** action had been previously evolved and then used as a skill in a new evolutionary process. This is a technique widely adopted in engineering and in evolutionary algorithms as well, sometimes being referred to as "scaffolding" [36].

The experiments in section 4.2 and section 4.3 and the attached videos show that bridging the gap between a simple simulation and the real robot is straightforward. This is due to our choice of working with high level actions that encapsulate many otherwise-computationally expensive details. For example a grasping action does not require the mechanics of the actual grasping or a suitable environment representation, as we rely on algorithms that already deal with these details. This renders the simulations fast and suitable for use in evolutionary algorithms. Probably the best effect of this choice is that **the resulting FSA is independent of the particular actions it uses**. This means that if a better grasping or object detection algorithm is provided, the old action can easily be replaced and updated without the need for the FSA to be changed. The results of our proposed algorithms are therefore general and appliable in several different scenarios (as proved also in section 4.3 and in the attached videos). Moreover, in situations where a simple simulator is not sufficient, like in section 4.2, we found that a RBFNN was able to reliably approximate the robot's behaviour and interaction with objects in the real world.

The main parameters controlling our proposed algorithm are listed in Table 1. We decided to use the particular type of evolutionary algorithm described in section 3.2. However we believe that the general approach does not depend on this particular choice, but it applies to different strategies too. We will conduct further experiments in this regard.

To improve the readability of this paper we decided to omit some minor implementation details, especially regarding the mechanics of the simulators we adopted. We refer the reader to the available source code for full reproducibility of the results.

6. Conclusions and Future Work

The main contributions of this work are:

- A framework that combines structuring, adapting and creating new actions to solve robotics problems.
- A new evolutionary algorithm to evolve both the topology and the parameters of FSA.

We performed experiments to prove that our proposed evolutionary algorithm has good performance in a variety of scenarios. Moreoever we proved that, although the evolutonary process is performed in simulation, the results have been straightforwardly applied to a real robot. We have included four videos to document the experiments and the source code for the evolutionary algorithm is available online.

We believe that hardly our proposed algorithm will produce an FSA's topology which performs better than an hadcrafted one. Although evolving the FSA's structure is fundamental to provide a working solution, it is not the main goal of this work, as discussed above. Therefore we are currently focusing on the generative aspect of the approach, where an FSA will become more similar to a neural network, whose nodes might represent actions, than to an automaton.

As this approach relies on an externally provided set of skills, we do not see it suitable to solve problems for which already well-estabilished algorithms yield good results. Actions that cannot be expressed by an FSA will not be evolved by our algorihtm as well. Examples include motion planning, object or in general pattern recognition and optimal control.

One of the major limitation of this approach is the necessity of writing a simluation environment to describe a task's environment, even if our approach only requires a non-detailed simulator. Our solution in section 4.2 has been to approximate the effect of the robot's actions with an RBFNN. We are currently investigating approaches to automatically learn the outcome of actions, so that they will not need complex simulations to be used.

7. References

[1] J. Hopcroft, R. Motwani, J. Ullman, Introduction to automata theory, languages, and computation, Addison-wesley Reading, MA, 1979.

[2] T. Bäck, Evolutionary algorithms in theory and practice, Oxford University Press New York, 1996.

[3] R. Brooks, Intelligence without representation, Artificial intelligence 47 (1-3) (1991) 139–159.

[4] R. C. Arkin, Behavior-based robotics, MIT Press, ISBN 0262011654, 1998.

[5] M. Beetz, D. Jain, L. Mösenlechner, M. Tenorth, Towards performing everyday manipulation activities, Robotics and Autonomous Systems 58 (9) (2010) 1085–1095, ISSN 09218890.

[6] F. Stulp, M. Beetz, Refining the execution of abstract actions with learned action models, Journal of Artificial Intelligence Research 32 (1) (2008) 487–523.

[7] L. Kaelbling, T. Lozano-Pérez, Hierarchical task and motion planning in the now, 2011.

[8] G. Konidaris, A. Barto, Skill discovery in continuous reinforcement learning domains using skill chaining, Advances in Neural Information Processing Systems 22 (2009) 1015–1023.

[9] G. Konidaris, S. Kuindersma, R. Grupen, A. Barto, Autonomous Skill Acquisition on a Mobile Manipulator, in: Proceedings of the Twenty-Fifth Conference on Artificial Intelligence, 2011.

[10] R. Burridge, A. Rizzi, D. Koditschek, Sequential composition of dynamically dexterous robot behaviors, The International Journal of Robotics Research 18 (6) (1999) 534.

[11] G. Neumann, W. Maass, J. Peters, Learning complex motions by sequencing simpler motion templates, in: Proceedings of the

26th Annual International Conference on Machine Learning, ACM, 753–760, 2009.
[12] J. Bongard, Active Coevolutionary Learning of Deterministic Finite Automata, Journal of Machine Learning Research 6 (2005) 1651–1678.
[13] S. M. Lucas, T. J. Reynolds, Learning deterministic finite automata with a smart state labeling evolutionary algorithm., IEEE transactions on pattern analysis and machine intelligence 27 (7) (2005) 1063–74, ISSN 0162-8828.
[14] D. Floreano, P. Dürr, C. Mattiussi, Neuroevolution: from architectures to learning, Evolutionary Intelligence 1 (1) (2008) 47–62, ISSN 1864-5909.
[15] N. Radcliffe, Genetic set recombination and its application to neural network topology optimisation, Neural Computing & Applications 1 (1) (1993) 67–90.
[16] K. O. Stanley, R. Miikkulainen, Evolving Neural Networks through Augmenting Topologies, Evolutionary Computation 10 (2) (2002) 99–127.
[17] C. Mattiussi, Analog Genetic Encoding for the Evolution of Circuits and Networks, IEEE Transactions on Evolutionary Computation 4193 (5) (2007) 671–607.
[18] F. Gomez, R. Miikkulainen, Accelerated Neural Evolution through Cooperatively Coevolved Synapses, Journal of Machine Learning Research 9 (2008) 937–965.
[19] C. S. Perone, Pyevolve: a Python open-source framework for genetic algorithms, SIGEVOlution 4 (1) (2009) 12–20.
[20] H. Aric, D. A. Schult, P. J. Swart, Exploring network structure, dynamics, and function using NetworkX, in: Proceedings of the 7th Python in Science Conference (SciPy2008), 11–15, 2008.
[21] I. Harvey, E. Di Paolo, R. Wood, M. Quinn, E. Tuci, Evolutionary robotics: a new scientific tool for studying cognition., Artificial life 11 (1-2) (2005) 79–98, ISSN 1064-5462.
[22] N. Kohl, K. Stanley, R. Miikkulainen, M. Samples, R, Evolving a real-world vehicle warning system, Proceedings of the Genetic and Evolutionary Computation Conference .
[23] A. L. Nelson, G. J. Barlow, L. Doitsidis, Fitness functions in evolutionary robotics: A survey and analysis, Robotics and Autonomous Systems 57 (4) (2009) 345–370, ISSN 0921-8890.
[24] J. Urzelai, D. Floreano, Evolution of adaptive synapses: Robots with fast adaptive behavior in new environments, Evolutionary Computation 9 (4) (2001) 495–524.
[25] N. Jakobi, Half-baked, ad-hoc and noisy: Minimal simulations for evolutionary robotics, in: Fourth European Conference on Artificial Life, The MIT Press, 348–357, 1997.
[26] K. Hsiao, S. Chitta, M. Ciocarlie, E. Jones, Contact-reactive grasping of objects with partial shape information, in: Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on, IEEE, 1228–1235, 2010.
[27] OMPL, The Open Motion Planning Library (OMPL), URL `http://ompl.kavrakilab.org/`, 2010.
[28] E. Marder-Eppstein, E. Berger, T. Foote, B. P. Gerkey, K. Konolige, The Office Marathon: Robust Navigation in an Indoor Office Environment, in: International Conference on Robotics and Automation, 2010.
[29] F. Stulp, A. Fedrizzi, M. Beetz, Action-related place-based mobile manipulation, 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems (2009) 3115–3120.
[30] D. Berenson, J. Kuffner, H. Choset, An optimization approach to planning for mobile manipulation, 2008 IEEE International Conference on Robotics and Automation (2008) 1187–1192.
[31] R. D. Beer, J. C. Gallagher, Evolving Dynamical Neural Networks for Adaptive Behavior, Adaptive Behavior 1 (1) (1992) 91–122.
[32] S. Buck, M. Riedmiller, Learning situation dependent success rates of actions in a RoboCup scenario, Lecture Notes in Computer Science (2000) 809–809.
[33] M. Schmill, T. Oates, P. Cohen, Learning planning operators in real-world, partially observable environments, in: Proceedings Fifth International Conference on Artificial Planning and Scheduling, 246–253, 2000.
[34] T. Poggio, F. Girosi, Networks for approximation and learning, Proceedings of the IEEE 78 (9) (1990) 1481–1497.
[35] S. LaValle, J. Kuffner Jr, Rapidly-Exploring Random Trees: Progress and Prospects, in: Algorithmic and computational robotics: new directions: the fourth Workshop on the Algorithmic Foundations of Robotics, AK Peters, Ltd., 293, 2001.
[36] J. Bongard, Behavior chaining: Incremental behavior integration for evolutionary robotics, Artificial Life 11 (2008) 64.