FREVO: A Tool for Evolving and Evaluating Self-organizing Systems

Anita Sobe, István Fehérvári, Wilfried Elmenreich
Alpen-Adria Universität Klagenfurt/Lakeside Labs
Institute of Networked and Embedded Systems
Austria
firstname.lastname@aau.at

Abstract

Typically, self-organizing systems comprise of a large number of individual agents whose behavior needs to be controlled by set parameters so that their interactions lead to the creation of the desired system. To be self-organizing, the system must mimic the evolutionary process. One way to do this is by use of an evolutionary algorithm. This mimics naturally-occurring genetic variation (mutation and recombination of genes). To fulfill this purpose, we have created a tool named FREVO (FRamework for EVOlutionary design), which separates the input needed into the following components: target problem evaluation, controller representation and the optimization method. FREVO provides well-defined interfaces for these components and supports a graphical user interface to simulate the evolutionary process. After obtaining the outcome for a simulation, it is possible to validate and evaluate the results within FREVO. FREVO has been successfully applied to various problems, from cooperative robotics to economics, pattern generation and wireless sensor networks. In this paper, we give an overview of the architecture of FREVO and introduce a case study involving smart grid networks.

1. Introduction

Over the past few decades, technological systems have become smaller, more powerful and are often networked to each other. This opens the possibility of creating large scale applications to meet a demand for solving large scale dynamic and complex problems. However, in order to control systems, a solution must be devised that has the appropriate scalability and flexibility to complete tasks. An appealing solution would be to engineer a self-organizing system for this purpose. Typically, self-organizing systems contain a large number of agents that interact with each other and their environment. Although individual agents generally exhibit simple behavior, as a large group, a global pattern usually emerges that is more complex than simply the cumulative sum of each agent's individual behavior.

However, it appears to be difficult to design a self-organizing system for a given task. Traditionally, systems are often build like a jigsaw puzzle; each system component has to fit in order to get a correct working system. When building complex systems, this approach is very difficult to maintain. A key issue is that the designer of a self-organizing system has to give up the "direct" control approach. Instead, the intended goal is to be achieved indirectly by defining the agents' rules. This is typically a very difficult task, since the global behavior of a system of interacting agents can hardly be predicted for a given set of local rules. In some cases, the emergent behavior is even counter-intuitive, where even experts have a strong tendency to falsely predict the effect of an agent's parameter in a complex system (see for example the case of the slime mold behavior described by Mitch Resnick [1]).

1.1. Evolving Neural Network Controllers

In our work, we concentrate on the evolution of neural controllers as often used as candidate representations for cooperative robots [2] or strategy development for game playing [3]. Since the design of neural networks is crucial itself, we use genetic algorithms to automatize the process of designing the control system and, thus, decreasing the necessary human interaction. In our previous work, we evolved neural network controllers for self-organizing robots [4], agent-based decision games [5], etc.

To solve such problems it is necessary to implement the controller, the neural network and the genetic algorithm with a simulation. The evaluation of this system would consist of hand-crafted parameter changes and numerous simulation restarts. This may be successful in achieving the goal for a single simulation, but a drawback is that it cannot be reused for a different simulation, nor can the results be compared if other optimization mechanisms or alternative neural network controllers should be evaluated.

There is a need for a generalization of design and implementation of such a system to reduce the set-up time for a problem and improve its evaluation possibilities. Therefore, we introduce FREVO (FRamework for EVOlutionary design), a framework that splits the design of a system into problem, representation, and optimization allowing for exchanging different parts seamlessly. Appropriate interfaces simplify the design as well as support for statistics and

graph generation help the evaluation of such self-organizing controllers.

1.2. State-of-the-Art

To the best of our knowledge only very specific frameworks exist. In domains such as multi-agent adaptive system design, machine learning and optimization methods are proven to be successful and have resulted in vast set of available software libraries and frameworks at the user's disposal.
In the past decade, experts in specific optimization and machine learning have made their code public. However, these packages focus only on one specific optimization method (e. g., genetic algorithms [6]) or on one target representation (e. g., neural networks [7]).
General-purpose libraries provide a larger set of methods and/or representations to the user [8] [9], but they either lack the possibilities for the user to easily exchange methods and representations within the same problem or they require deep programming knowledge or expertise in algorithms.

This paper introduces the architecture of FREVO in Section 2 and explain its main features and further shows FREVO's capabilities in a simple case study related to smart grids in Section 3.

2. FREVO Architecture

Our proposed framework FREVO, supports engineers in evolutionary design and evaluation of self-organizing systems. It provides the necessary tools to make an agent-based model of a desired self-organizing system and to search for the required local interaction rules using iterative heuristic search. In addition, FREVO also supports the evaluation of evolved solutions under predefined conditions. The key feature of FREVO is the component-wise separation of the key building blocks: the *problem*, *method*, *representation*, and *ranking*. This structure enables the components to be designed separately allowing the user to easily change and evaluate different configurations and methods or to add an interface to an external simulation tool. Each component can be developed and tested separately and reused for new projects. FREVO comes with a graphical user interface allowing the engineer to pick the particular components for a project (see Figure 1). The component concept also supports fast evaluation of different configurations, for example, in order to see which controller representation (e. g., neural network vs. finite state machines) works better for solving a given problem.

FREVO is written in Java and makes use of Java's object oriented model by defining interfaces to generic components. When a new type component needs to be added, the user is required to write the code for the component. In this task, the user is assisted by FREVO's generator tool and

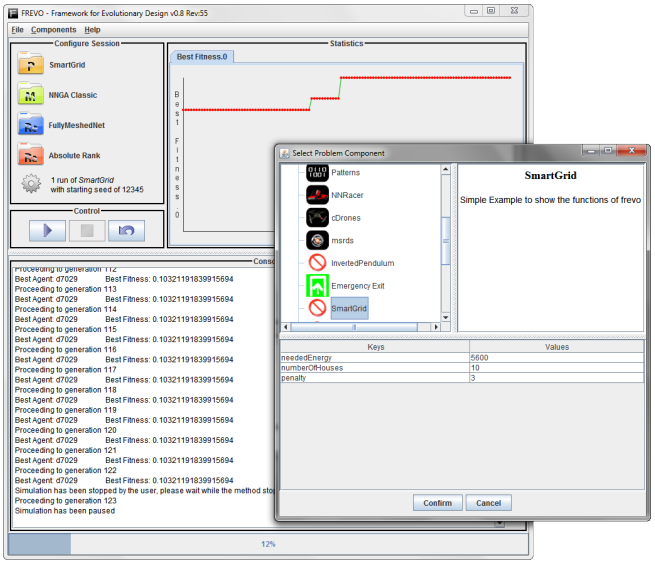

Figure 1. Overview of FREVO graphical interface while selecting a Problem component

guided by the methods required by Java's interfaces. FREVO is released as open source under GPL v3.0 (available at: `http://www.frevotool.tk`).

In the following sections, we define the four component types within FREVO. As depicted in Figure 2, the architecture introduces a waistline interface between the problem (top) and the other parts of FREVO. This eases modeling of a new problem, since only a few methods have to be provided: an interface for connecting the agent's I/O to the agent controllers, a method for evaluating the problem (typically by a simulation run) and a fitness value that is given as a feedback from an evaluation. A single composition of a problem, method, representation and ranking defines a so-called FREVO session containing details of all parameters set for the experiment. In order to store simulation data for later experiments and evaluations, FREVO saves individual sessions and results in a compact XML format that can be easily accessed, displayed and archived using tools available within the program.

2.1. Problem definition

In the problem definition the evaluation context of the agent's behavior has to be implemented. In other words, this component is responsible for the evaluation of the candidate representations.

Due to the simplified nature of interfaces within FREVO, implementing an own model should take little effort. It is also possible to connect FREVO with external simulation tools. There are two types of problem components: The first variant is to implement a subclass of `AbstractSingleProblem` (see Figure 3), connects a controller into a simulation and returns a fitness value as the

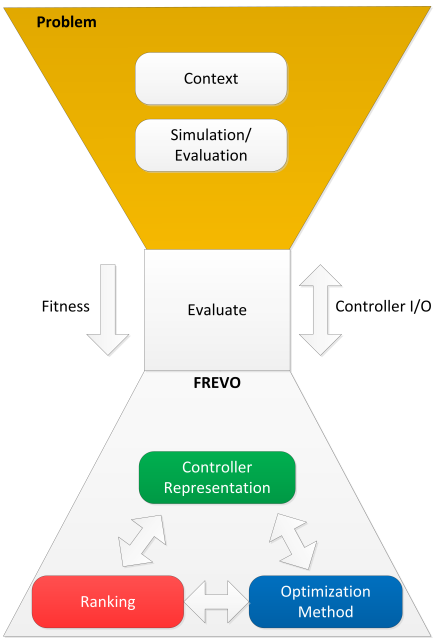

Figure 2. FREVO components (Problem (yellow), Optimization Method (blue), Representation (green) and Ranking (green))

```java
package utils;

import core.AbstractRepresentation;
import core.AbstractSingleProblem;

public class MyProblem extends AbstractSingleProblem {

    @Override
    protected double evaluateCandidate(AbstractRepresentation candidate) {
        // Your fitness function comes here
        return 0;
    }

    @Override
    public double getMaximumFitness() {
        // Returns the fitness value corresponding to the best possible solution
        return 0;
    }

}
```

Figure 3. Implementing a new problem class

result. To support multi-agent systems, there can be multiple instances of the controller in the simulation, but they are essentially playing "on the same team". An example for this problem type could be a group of robots in a cooperative search mission.

The other variant, implemented as a subclass from `AbstractMultiProblem`, evaluates multiple candidates relative to each other. An example therefore would be a simulation of two soccer teams playing each other. The result of such a simulation only gives a relative ranking and requires to run a tournament algorithm to get a ranking of a pool of candidates.

2.2. Candidate Representation

The candidate representation describes the structure of a possible solution, i. e., the implementation of an agent's controller. This implementation is agnostic to the selected problem. It contains generic structures such as artificial neural network (ANN) models that can be used for different problems, ranging from the control of a robot, to making economic decisions in market situations.

In order to support optimization algorithms, every candidate representation must extend the `AbstractRepresentation` class that enforces the required mutation and/or recombination functions along with the proper output format and range. Furthermore, these components can optionally support different output formats for later processing or network analysis such as Pajek [10].

The release version of FREVO comes with the following representations:

- *Fully-meshed net*: A time-discrete, recurrent ANN where each neuron is connected to every other neuron and itself. During evolution, the biases of each neuron, plus the connection weights are taken into account.
- *Three-layered net*: A similar ANN to the previous one, but with a feed-forward structure instead of the fully-meshed one.
- *NEAT net*: An ANN whose connectivity structure is also taken into account for selection during evolution. This representation was implemented based on the NEAT model described in [11]
- *Econets*: Neural networks that can learn from their previous outputs, based on the principles of ECONETS presented by Parisi et al. [12]
- *MealyFSM*: A Mealy Machine (a special case of a Finite State Machine) whose structure and transition probabilities are evolved.

2.3. Optimization Method

The optimization method is used to optimize a chosen candidate representation in order to maximize the fitness returned from the problem definition. Typically, an optimization method creates a pool of possible candidates from the solution representations, evaluates them using the problem definition and gradually obtains candidates with better performance. Currently, we have the following two optimization methods provided in the release version of FREVO:

- *NNGA*: An evolutionary algorithm that supports multiple population, different selection schemes (eg. roulette wheel selection) while trying to maximize population diversity as well [13].
- *GASpecies*: An evolutionary algorithm that sorts candidates into species based on their genotype (similar structures are put into the same species) to prevent

immature solution to die out too soon. This is particularly beneficial when representations might undergo big changes during evolution.

- *Spatially Structured Evolutionary Algorithm*: This algorithm arranges candidates in a two-dimensional map and performs genetic operations such as selection, mutation and recombination in a local context, i. e., in the respective Moore neighborhood of a candidate. This algorithm has a slower convergence rate, but better population diversity than a standard genetic algorithm.

2.4. Ranking

The ranking module evaluates all candidates and returns a ranking of them based on their fitness. In case of problems derived from AbstractMultiProblem, solutions can be only compared relative to each other, e. g., if a soccer playing strategy is evolved by repeatedly simulating a soccer match between two teams [4]. For this case, FREVO allows to choose between several *ranking mechanisms* defining how to pair the solutions in order to find a dependable ranking with a low number of comparisons (i.e. simulation runs). See [14] for a discussion on robust methods for this purpose.

3. Case Study

FREVO has been used for the evaluation of many case studies, involving evolutionary robotics [15], [16], [4], [17], pattern generation [18], and wireless sensor networks [19]. In this section, we show the capabilities of FREVO by way of a simple scenario. Let us consider a village consisting of a number of households connected to a smart energy grid.

Each household can dynamically buy energy units from the energy market at each hour of the day. The amount to buy depends on a predicted per hour consumption and a flexible part, where energy units can be bought if the price is currently low these units are stored in a battery. The total amount bought has to match a defined daily energy need.

Using FREVO, we have to define this scenario as a problem component. The provided component creator helps to set up the necessary class with the needed methods and the required XML configuration file. The design of the problem requires the definition of input values and output types for the neural network (which we do not have to implement). Additionally, we need a fitness function to train the neural network. We modeled the current price, the current predicted consumption, the money spent by the current household, the energy bought by the current household and the target daily energy need as inputs for the agent. The agent needs to output one value is between 0.0 and 1.0 representing the fraction of flexible energy units that should be bought within this hour. The overall energy acquisition for an hour is the sum of the number of flexible energy units (e_f) plus the fixed number of energy units (i.e., the predicted consumption e_{pc}).

$$toBuy = output \cdot (e_f) + e_{pc}$$

The fitness is calculated at the end of the day for all households. It includes a penalty if too few energy units are bought, i.e., the price for the difference increases the price spent by a factor of the penalty. The fitness increases if the highest number of energy units is bought (e_b) for the lowest price. Furthermore, it is better to buy only the needed energy. The expected outcome is that at each hour the predicted number of energy units is bought, and during the lowest price period additional units are bought until the energy threshold is reached.

$$fitness = e_b / moneySpent$$

For our scenario we select our problem together with an existing representation, optimization and ranking method. In the following, we will explore which combination of optimization method and candidate representation allows for the most stable results. As optimization method we make a session with NNGA and another session using the GASpecies method. The representation can be either a fully-meshed neural network or a three-layered neural network. We chose empirically to evolve 200 generations, because the fitness values stabilized. The smart grid problem is configured for 10 households, where each needs at least 5,600 energy units a day. For all other settings the default values of FREVO are chosen. One can store the configuration for later use in a session file.

To account for random effects, we repeat each evolution run 10 times with different random seeds and present the fitness development as box plots per generation (FREVO generates automatically the correct output for generating boxplots of the fitness with R). In all scenarios the highest evolved fitness is 0.1136. In the figures 4 and 5 one can see that the fitness variance of GASpecies is higher than of NNGA as shown in Figures 6 and 7. The reason is that GA Species does not reach in all runs the highest fitness. Although the combination of GASpecies and three-layered neural network seems to need less generations to reach a reasonable fitness, the combination with the fully-meshed network results in reaching the higher fitness if sufficient time is given for the evolutionary algorithm. The NNGA algorithm shows very stable results, where all runs reach the highest fitness within a few hundred generations.

It is easy to define different scenarios to be compared using FREVO. One can decide what optimization method or candidate representation results in the most stable and best performing simulation/system for a given problem. The next step is to validate and evaluate the resulting candidate representation. FREVO generates result files that can be loaded after evolving the candidates. The result file contains the setting and neural network weights of the population candidates of the last evolved generation, ranked by fitness.

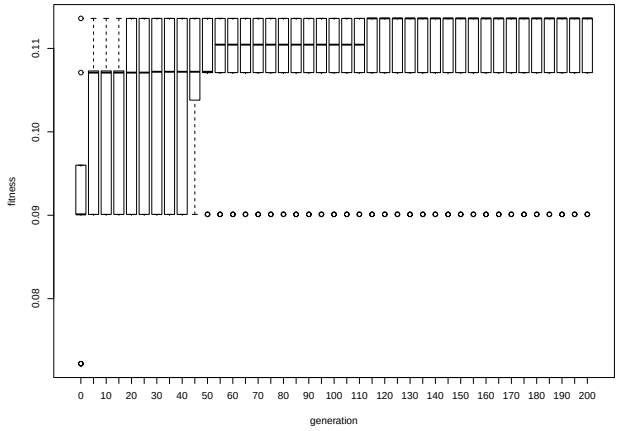

Figure 4. Fitness development vs. number of generations, optimization method: GASpecies, representation: Fully meshed neural network

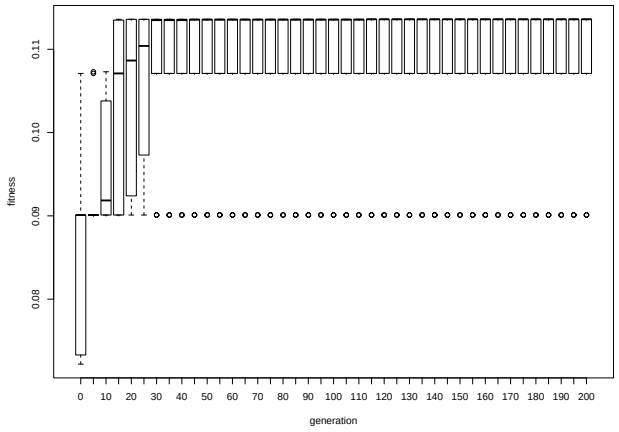

Figure 5. Fitness development vs. number of generations, optimization method: GASpecies, representation: Three-layered neural network

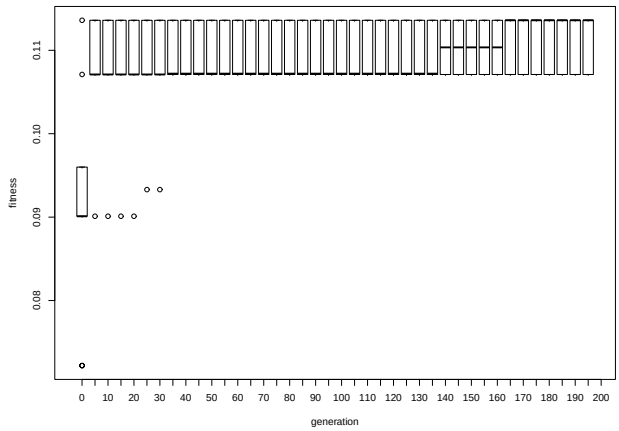

Figure 6. Fitness development vs. number of generations, optimization method: NNGA, representation: Fully meshed neural network

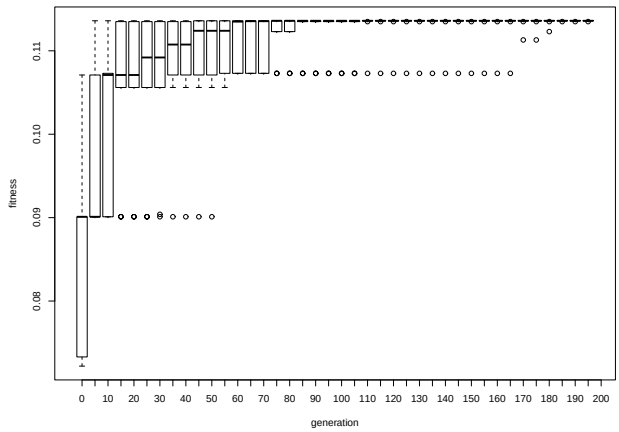

Figure 7. Fitness development vs. number of generations, optimization method: NNGA, representation: Three-layered neural network

One can also configure in the optimization settings that the results files are periodically written every n^{th} generation.

An overview is given in Figure 8, where on the left side the candidates are listed. By selecting one candidate, it can be replayed with the same settings as done during the evaluation. In order to test with different settings, the parameters can be adjusted in the window to the right. For example, the scalability of the evolved candidate can be evaluated by adjusting the number of houses.

An additional feature of FREVO is the possibility to implement graphical evaluations, such as done in [18] to validate the pattern generation function of an evolved cellular automaton. In the smart grid scenario it is sufficient to print out the number of bought energy units per household at each hour and to print the fitness to see whether the parameters lead to decreased fitness. In our scenario, the households buy the required energy units and then buy more when the price is at the lowest stage. The behavior of the buyers does not change if the number of houses is increased. However, the fitness decreases because currently it is dependent on totally bought energy units, and the totally spent money which increases if the number of houses increases. Normalizing the fitness by the number of households solves this issue and results in the comparable fitness in all cases. Thus, FREVO helps in validating and evaluating not only the resulting candidate representation, but also the problem implementation itself.

4. Conclusion

We introduced FREVO, a tool for designing and evaluating self-organizing systems. FREVO concentrates on evolutionary methods for agent controllers, as often applied in autonomous robots, but extends this principle to arbitrary

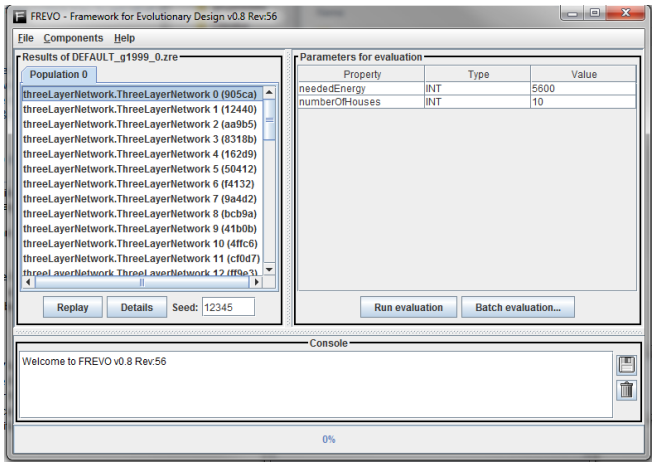

Figure 8. FREVO in evaluation mode. Left: list of candidates, right: configurable parameters

applications. With the principle of reusable, independent components, FREVO allows for easily exchanging different implementations of neural controllers, optimization methods and ranking methods. A simple example can be implemented with a small number of lines of code. The implementation effort is reduced to defining the context, the fitness function and define input and output of the agent. After evolving the agent controllers, the simulations with the resulting candidates can be replayed either with the same settings or with different parameters for evaluation purposes. As an example we implemented a system of households buying energy in a dynamical energy market. We showed the scalability of the evolved neural network by changing the number of households connected to the smart grid. As a future work we want to concentrate on further integrating different representation models, such as evolvable Finite State Machines and optimization methods and ranking methods for instances of AbstractMultiProblems.

Acknowledgment

This work was supported by the Austrian Science Fund (FFG) grant FFG 2305537 and the Lakeside Labs GmbH, Klagenfurt, Austria, and funding from the European Regional Development Fund and the Carinthian Economic Promotion Fund (KWF) under grant KWF 20214—21532—32604, and KWF 20214—22935—34445. We would like to thank Lizzie Dawes for proofreading the paper.

References

[1] M. Resnick, *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds (Complex Adaptive Systems)*. The MIT Press, 1997.

[2] A. Nelson, E. Grant, and T. Henderson, "Evolution of neural controllers for competitive game playing with teams of mobile robots," *Robotics and Autonomous Systems*, vol. 46, no. 3, pp. 135 – 150, 2004.

[3] M. Sipper, Y. Azaria, A. Hauptman, and Y. Shichel, "Designing an evolutionary strategizing machine for game playing and beyond," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 37, no. 4, pp. 583 –593, july 2007.

[4] I. Fehervari and W. Elmenreich, "Evolving neural network controllers for a team of self-organizing robots," *Journal of Robotics*, 2010.

[5] I. Fehervari and W. Elmenreich, "Towards evolving cooperative behavior with neural controllers," *IFIP Fourth International Workshop on on Self-Organizing Systems*, pp. 2–3, 2009.

[6] K. Meffert, "JGAP - Java genetic algorithms and genetic programming package." [Online]. Available: http://jgap.sf.net

[7] T. Schaul, J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, T. Rückstieß, and J. Schmidhuber, "PyBrain," *Journal of Machine Learning Research*, vol. 11, pp. 743–746, 2010.

[8] C. Igel, T. Glasmachers, and V. Heidrich-Meisner, "Shark," *Journal of Machine Learning Research*, vol. 9, pp. 993–996, 2008.

[9] T. Abeel, "Java machine learning library," 2009. [Online]. Available: http://mloss.org/software/view/136/

[10] A. M. Wouter de Nooy and V. Batagelj, *Exploratory Social Network Analysis with Pajek*. Cambridge University Press, 2002.

[11] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002. [Online]. Available: http://nn.cs.utexas.edu/?stanley:ec02

[12] D. Parisi and F. Cecconi, "Econets: Neural networks that learn in an environment," *Network*, vol. 1, no. 2, pp. 149–168, 1990.

[13] W. Elmenreich and G. Klingler, "Genetic evolution of a neural network for the autonomous control of a four-wheeled robot," in *Sixth Mexican International Conference on Artificial Intelligence (MICAI'07)*, Aguascalientes, Mexico, nov 2007.

[14] W. Elmenreich, T. Ibounig, and I. Fehervari, "Robustness versus performance in sorting and tournament algorithms," *Acta Polytecnica*, vol. 6, no. 5, pp. 7–18, 2009.

[15] I. Fehervari and W. Elmenreich, "Towards evolving cooperative behavior with neural controllers," in *IFIP Fourth International Workshop on Self-Organizing Systems*, 2009.

[16] I. Fehervari and W. Elmenreich, "Evolutionary methods in self-organizing system design," in *Proceedings of the 2009 International Conference on Genetic and Evolutionary Methods*, 2009, pp. 10–15.

[17] A. Pintér-Bartha, A. Sobe, and W. Elmenreich, "Towards the light – Comparing evolved neural network controllers and finite state machine controllers," in *Proceedings of the Tenth International Workshop on Intelligent Solutions in Embedded Systems*, Klagenfurt, Austria, jul 2012, pp. 83–87.

[18] W. Elmenreich and I. Fehervari, "Evolving self-organizing cellular automata based on neural network genotypes," in *Proceedings of the Fifth International Workshop on Self-Organizing Systems*, vol. LNCS 6557. Springer Verlag, 2011, pp. 16–25.

[19] C. Costanzo, V. Loscri, E. Natalizio, and T. Razafindralambo, "Nodes self-deployment for coverage maximization in mobile robot networks using an evolving neural network," *Computer Communications*, vol. 35, no. 9, pp. 1047–1055, 2012.