Artificial Life Environment

A Framework For Artificial Life Simulations

Gerd Beuster

Artificial Intelligence Research Group, University Koblenz, Germany

Abstract

Artificial Life Environment (ALE) is a framework for artificial life simulations. Genetic algorithms, artificial neural networks and cellular automata provide the building blocks for the creation of artificial life simulations. ALE is becoming both a powerful research-, and educational-tool. ALE is still in alpha-stage.

1. Introduction

1.1 A Framework for Artificial Life Simulations

When developing software for artificial life simulations, most researchers write their own tools from scratch in a general purpose programming language. This approach as several disadvantages:

- Simulation software for artificial life is complex. It takes a lot of resources to develop a simulation software. We would prefer to spend our time and resources on the problem we are interested it, not on writing the software that we need to tackle the problem.
- Writing artificial life software is a tedious task. Every complex piece of software contains numerous bugs. Debugging is especially complicate in artificial life simulations, because usually we are dealing with non-deterministic processes. Bugs do not necessary result in complete failure, but may influence the result in subtle ways.
- Simulation software written for “personal use” is usually highly specialized. This makes it very hard for third parties to use the software. Verification and extension of the simulations is very complicate for people not directly involved in the creation of the software.
- Even if simulations from different people or groups are similar in their logical structure, it is hardly possible to combine and interchange components of the simulations, or to combine different simulations into a single one, because similar logical structures are usually not reflected in the structures of the programs. Similar simulations can and will be very different in their implementation.
- For a beginner, it is not easy to get into the field of artificial life, if one has to write one's own simulation tools from scratch.

1.2 Aims of ALE

ALE (Artificial Life Environment) is an attempt to overcome these problems. ALE provides *building blocks* for artificial life simulations suited for certain problems.

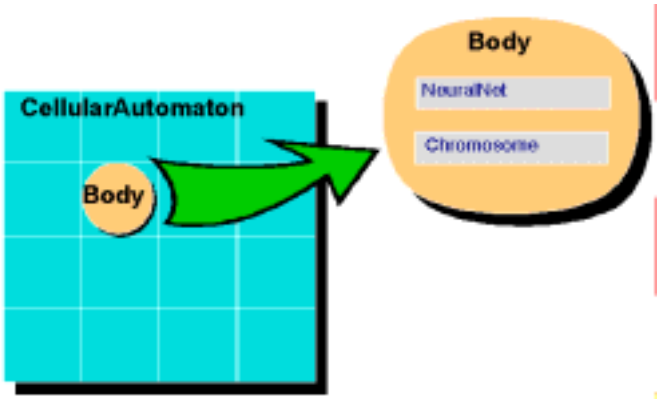

Figure 1: Components of ALE: CellularAutomaton, Body, NeuralNet, Chromosome

These building blocks can be recombined and changed in order to form artificial life simulations. By this approach, many of the problems stated in the last section can be solved or at least alleviated:

- A researcher who wants to set up a simulations no longer needs to start from scratch. Existing building blocks can be modified and combined to create the simulation the researcher is interested in. With ALE, she additionally gets a predefined set of analysis tools and a graphical user interface.
- The building-block-system forces a certain structure onto the simulation. Since the base structure is the same for all simulations written with ALE, it becomes more easy to exchange parts of simulations. When two researchers are working on a similar problem, they are able to share the simulations' components, and even whole simulations.
- People not familiar with the field of artificial life can start playing around with existing building block. This is made even more simple by ALE's graphical user interface. The GUI allows a novice user to set up simulations very quickly by changing existing simulations and creating new building-blocks. The new user gets a direct-feedback of how the different factors influence her simulation.
- By using a common, open sourced system, both the speed of development of the simulation tool and its reliability is improved.[6]

2 Simulation Building Blocks

ALE does not aim at providing a general purpose tool for artificial life simulations. There is already a high quality tool for general purpose artificial life simulations available - SWARM [4]. The goal of ALE is to provide a tool for certain classes of artificial life simulations. ALE focuses on BUGS-like [2] simulations. These are simulations with the following characteristics:

- There is some form of a spatial environment.
- This environment is populated by autonomous entities.
- The entities contain some form of genetic information.
- The entities have a decision-making unit, a “brain”.

We have populations of autonomous entities who are living in some environment. These entities are (partly) determined by their genes, and they have some form of a “brain” which tells them how to act. See figure 1. In the ALE-library, this structure is

reflected by the class hierarchy shown in figure 2.

These classes interact in the following way:

CellularAutomaton The environment is provided by class *CellularAutomaton*. The base class *CellularAutomaton* is a two dimensional cellular automaton with an asynchronous [9] update function. Class *CellularAutomaton* includes methods for adding entities to the cellular automaton, running the simulation, and keeping track of some statistical data. The cellular automaton runs the simulation by successively telling the entities on the cellular automaton to act.
Beside providing the simulation environment, class *CellularAutomaton* keeps track of some global data. Since the simulation is driven by the interaction of the individual entities, we have to decide how to keep track of data which is only observable on a higher aggregation level, for example the sizes of the populations, or the average fitnesses of the different entity-species. This task is taken care of by the cellular automaton.
The base class *CellularAutomaton* might not be appropriate for many simulations. Perhaps we need a 1-dimensional cellular automaton, or one with a synchronous update function, or one with six instead of eight neighboring cells. Because of the building-block-concept of ALE, we can simply exchange the base cellular automaton for a different model which implements the features needed for the specific simulation. No other part of the simulation has to be changed.

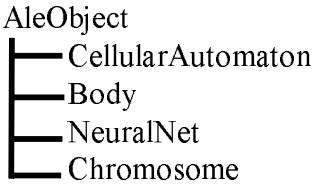

Figure 2:Hierarchy of base classes

Body Class *Body* is the class for the entities, the main actors of a simulation. Bodies populate the cells of the cellular automaton. Class *CellularAutomaton* successively calls the instances of class *Body* residing on it in order run the simulation. An entity[1] usually acts in the following way: It examines its environment and its internal states, and decided how to act (for example, it might decide to move) and how to change its internal states. How the entity acts exactly, and what it "sees" from its environment, depends solely on the actual implementation of the entity. The programmer is totally free in how she implements the entities for her simulation, but since all entities inherit from the same base class *Body*, it is possible to let entity from different origins interact within the same simulation, and it is also possible to exchange one kind of entities for another kind within a simulation.

Chromosome Although the programmer is free to implement the entities in whatever form she wants, there are two more class which can (and should!) be used when programming it. The first class is *Chromosome*. Every instance of *Body* contains an instance of *Chromosome* (or one of its subclasses). Class *Chromosome* is used to carry the genetic information of the *Body*. This is usually just a string of float or integer valued numbers and some operations on it. Again, how this genetic information is used by the entity, depends solely on the entity's implementation. For example, in some implementation of an entity, the genetic information might be ignored completely, in others (as we will see in a later example) it might determine the appearance, the physical condition or the behavior of the entity.

[1] We will use the term "entity" for instances of class *Body* and instances of classes who inherit from *Body*.

Basically, *Chromosome* contains the data and methods that we know from a genetic algorithm[5]: Gene-Information and operations for the reproduction of itself. In the base class, *Chromosome* provides float and integer valued genes with the manipulation operators mutation and one-point-crossover. A researcher who wanted to use different kinds of genetic operations, for example two-point-crossover, would write a subclass of *Chromosome* that implements two-point-crossover, and run the simulation with this class instead.

NeuralNet *NeuralNet* is the decision-making unit of the entities. The instances of this class act as follows: They get a sequence of float-valued numbers as inputs, and they return another sequence of float-valued numbers as output. Class *NeuralNet* is in the same position as *Chromosome*. Every instance of (a subclass of) *Body* contains a *NeuralNet*. How it is used, depends on the actual implementation of the entity. That is, the *Body* decides which kind of information it passes into the neural network, and it decides how the output of *NeuralNet* is interpreted. The name *NeuralNet* is basically kept for historical reasons. As a matter of fact, the decision making unit does not need to be a neural network. One can implement it in whatever technique one wants to. Figure 3 shows how the neural network works: The entity examines its Moore-neighborhood. A representation of its neighborhood is passed as an array of floats into the neural network. The neural network does some calculations, and returns an output value. The output value is interpreted by the entity as a direction to move to.

3 Application

Although the basic structure of all ALE-simulations is defined by the building blocks, it is possible to set up a wide variety of different simulations scenarios. The main part of this section covers a description of a BUGS-like [2] scenario. In the rest of the section, we will demonstrate the flexibility of ALE by some toy-examples from quite different areas.

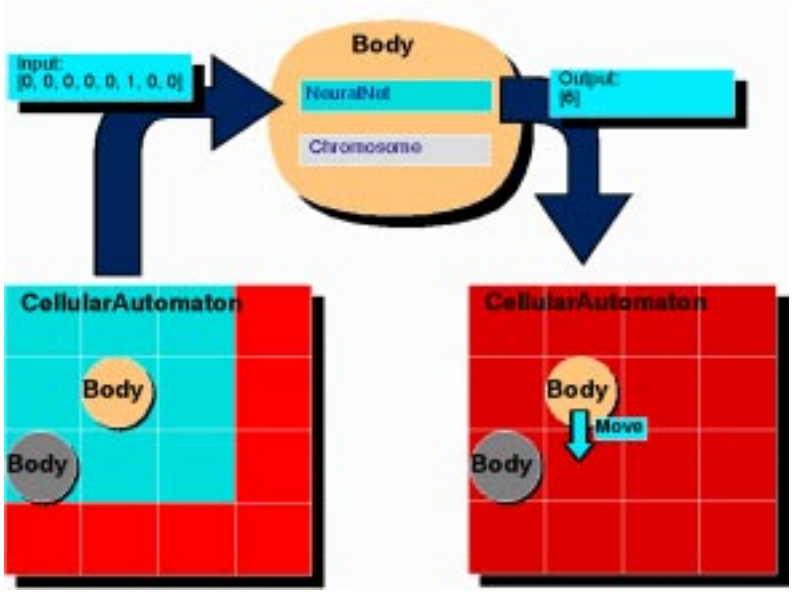

Figure 3: The entity examines its Moore-neighborhood. A representation of its neighborhood is passed as an array of float into the neural network. The neural network does some calculations, and returns an output value. The output value is interpreted by the entity as a direction to move to.

3.1 Predator/Prey Simulations

The scenario We want to simulate a world inhibited by predators and preys. The predators shall learn on a genetic level to hunt the preys, and the preys shall learn to evade the predators. A predator eats another entity by moving on the field occupied by the other entity.
Every entity has some amount of *energy*. Depending on the activity of the entity, its energy level raises or drops. When the energy level hits zero, the entity dies. When the energy level exceeds a threshold, it reproduces asexually. The energy household of an entity is defined by four variables:

1. The energy level at birth.
2. The energy consumption per move.
3. The energy gain when eating an entity.
4. The energy threshold for reproduction.

The possible actions of a predator or prey are to move to one of the cells surrounding the cell the entity is currently at. In order to determine their actions, the entities shall use their *NeuralNet* which itself is constructed according to the *Chromosome* of the entity.

Implementation

- Classes *Predator* and *Prey*

  Both the classes *Predator* and *Prey* inherit from base class *Body*. The only real difference between a predator and a prey is that preys can not eat other entities. For the prey, it is not possible to move to a cell that is inhibited by an other entity. Since they can not eat other entities, they must gain energy whenever they move. Therefore their energy consumption per move must be negative. One can think of this as a process where energy from the environment is taken by the entity, like in the real world plants facilitate sunlight.

  The energy consumption is defined by the first four genes of the *Chromosome*. When a new predator or prey is created, it checks these four first genes from its *Chromosome* and adjusts its energy-household according to these values. The genes are interpreted as shown in the following table 1.

Gene 0	Energy level at birth
Gene 1	Energy threshold for reproduction
Gene 2	Energy gain for eating entities
Gene 3	Energy usage per move

Table 1: Genes determining the energy of the entity

Predators and Preys have different appearances. This is necessary in order to give them a chance to distinguish between friends and foes. Every entity can see its Moore-neighborhood. For the eight cells surrounding the entity, it gets the information whether a predator, or a prey, or no entity is on the neighboring cell.

This information is passed into the neural network. The output of the neural network is interpreted as a move to one of the neighboring cells. See figure 3.

- *NeuralNetFeedForward*

  The base *NeuralNet* class is a virtual base class. It provides an interface for all neural network classes, but it does not implement any neural network itself. In order to get a useful neural network, we have to reimplement it in an inherited class. On of these inherited classes is *NeuralNetFeedForward*. *NeuralNetFeedForward* provides a fully connected feed-forward network with one hidden layer. The network does not do any online learning[7]. The weights of the network solely depend on the *Chromosome*, where every float-numbered gene on the chromosome-string is interpreted as the weight of a corresponding network node. This, of course, is not a very effective way for encoding a neural network, but we will use it in this example anyway. If we would want to use a more sophisticated network with our simulation, we could simply exchange *NeuralNetFeedForward* for a different implementation. Whenever the *Predator* resp. *Prey* has to act, it calls its *NeuralNetFeedForward* with the information of its surrounding cells represented as a string of numbers. These numbers are taken as the activation values of the input nodes of the neural network. The output values are calculated by the network, and are passed back to the body.

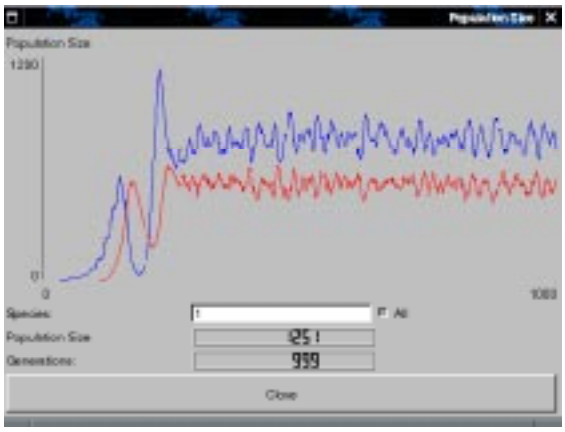

Figure 5: Evolving a stable eco-system by making the energy household of the entities subject to evolution. The two graphs show the population sizes of the predators and the preys. After an initial phase, the population sizes become very stable, with somewhat more preys than predators on the automaton.

- *Chromosome* and *CellularAutomaton*

  These classes do not have to be changed for our simulation. We use the standard two-dimensional cellular automaton with asynchronous updates, and the generic chromosome with one-point crossover.

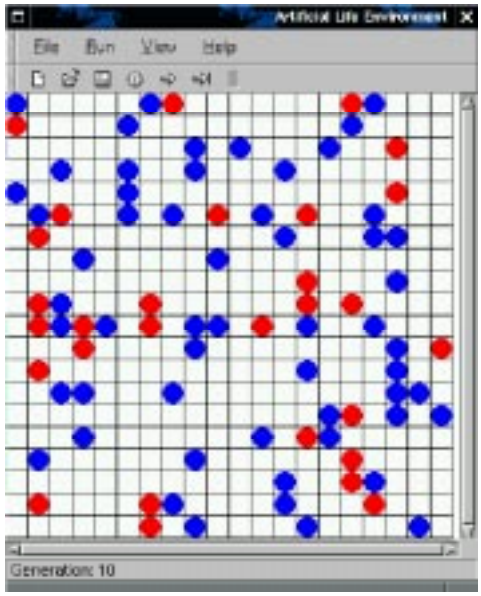

Figure 4: Screenshot of the running predator/prey-simulation. The predators are the darker entities, the preys are painted in a lighter color.

Running the Simulation Figure 4 shows a screenshot of the simulation in action. Since this simulation is just an example, we will not go into the details of the results of the simulation. It should be noted, though, that although the neural network encoding scheme is very poor, we can observe some improvement in the behavior of the entities. There is a second phenomenon to observe which is more interesting. A general problem

of low-scale simulation of ecological systems like this is that these systems usually are not stable. Within a view generations, one of the populations dies out and the whole ecological system breaks down. In this simulation, we found a surprising way to give these kinds of systems stability: When we make also the energy household of the entities (the first four genes of the entity as shown in table 1) subject to evolution, we get a stable systems! Figure 5 shows how the population sizes of the predator- and prey-populations reach a stable level.

3.2 More Examples

In the next examples, we will give some glimpse onto the flexibility of ALE. We will show how it can be used to construct simulations that are quite different to the BUGS-like scenarios shown before. Next, we will show how the traditional Game of Life as described by Conway [3] can be implemented in ALE.

3.2.1 Game of Life

Again, we create a new subclass of *Body*. We call it *Body2dLife*. These entities never move. They stay in their cell, and only change their appearance between two different states. One state indicates “living”, the other state indicates “dead”. The entities use their neural network to decide which state to enter: They examine their neighborhood and calculate how many cells are living around them. This number is fed into the network, and the network output is interpreted as the new state of the entity. So the rules of the game depend on the implementation of the network. In this example, we write a new neural network implementation called *NeuralNetConways*, which implements Conway's rules. To be honest, we do not use a real neural network for this. This “neural network” is simply a set of if-than-clauses in the form of “if the input value is two or three and the current state is `alive´, output is `alive´”. If we wanted to implement some other rules, we would simply exchange this “neural network” for a different one. Some pictures of the cellular automaton are shown in figure 6.

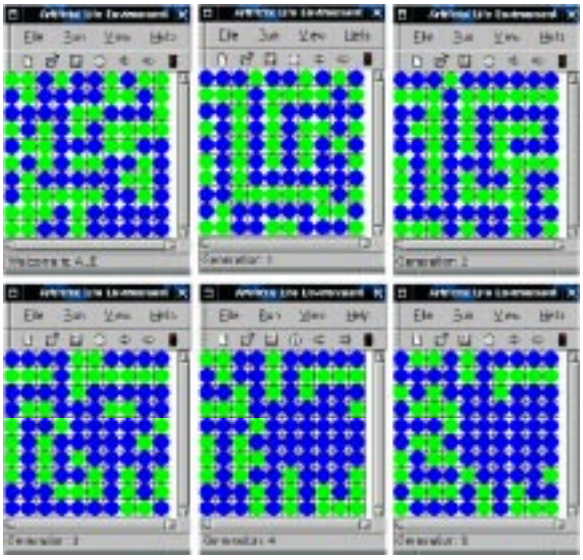

Figure 6:Some updates of the (asynchronous) cellular automaton running Conway's Game of Life.

There is one more big difference between this implementation of two dimensional Game of Life and Conway's original game: Conway uses synchronous updates, whereas ALE's *CellularAutomaton* always uses asynchronous updates. Overcoming this problem is easy. We reimplement the cellular automaton as the synchronous model, and plug this new cellular automaton into the simulation.

3.2.2 Genetic Algorithm

In the next example, we will implement a “traditional” genetic algorithm with ALE. A genetic algorithm runs through the following steps [5]:

1. Create a random population.
2. Choose the fittest entities for reproduction.
3. Create a new population by recombining and mutating the genes of the selected entities.
4. Unless an entity with the desired fitness has evolved or the maximal number of generations has been reached, go back to step 2.

In order to implement this algorithm in ALE, we first have to reimplement the *CellularAutomaton* as a one-dimensional model. In the first generation, the first row of the cellular automaton is filled with entities whose chromosomes are random. The evolutionary process is split up into two phases: In the first step, each entities chooses an other entity to compete against. The one with the higher fitness survives. In the second step, these surviving entities choose partners for reproduction and create a new population. In genetic-algorithm-terminology, this means that we are using tournament selection with elitism (the best entity always survives). Classes *Chromosome* and *NeuralNet* stay unchanged. *NeuralNet* is not used at all, and the values of *Chromosome* are simply interpreted as a possible solution to the problem posed to the genetic algorithm. Screenshots of the genetic algorithm running are shown in figure 7.

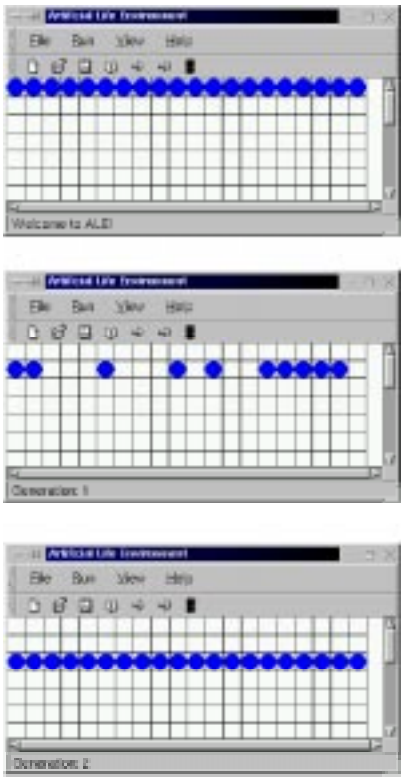

Figure 7: The two phases of the genetic algorithm: In the first phase, the fittest entities are selected. In the second phase, the new generation is created. The third phase is the first phase again.

4. Technics

Technically, ALE consists of a C++-library and a standalone program. The class library provides the base classes for the four main ALE building blocks: *CellularAutomaton*, *NeuralNet*, *Body*, and *Chromosome*. Additionally, it contains some inherited class of general use, e.g. the previously mentioned class *NeuralNetFeedForward*. The ALE library should compile on any system with a C++-compiler. It has been successfully compiled on Linux, Solaris, FreeBSD, and BeOS. The second way to use ALE is via the graphical user interface provided by the standalone program *kale*. *kale* allows the user to graphically interact with the components of ALE. One's own classes can be loaded dynamically into the program. *kale* makes use of the KDE desktop environment.
All parts of ALE are available as free software under the GNU Public License and can be downloaded from **http://www.uni-koblenz.de/gb/ale/**.
Please note that ALE is still in alpha-stage.

5 Conclusions

ALE is becoming a flexible tool for the development and examination of artificial life simulations. It can serve both as a research and an educational tool. For researchers, the building-block-concept of ALE makes it more easy to exchange program components and results. Beginners in the field of artificial life get a chance to get their hands on a flexible simulation tool. *kale* provides them with a graphical

user interface that allows them to get their hands directly onto an artificial life tool. The building-block-concept of ALE allows beginners to start their own work by changing and improving an existing system.

Reference

[1] Gerd Beuster. *Artificial life environment*. Master's thesis, University of Koblenz, 1999. http://www.uni-koblenz.de/~gb/ale/studienarbeit_ale.ps.gz.

[2] A. K. Dewdney. *Simulated evolution: wherein bugs learn to hunt bacteria*. Scientific American, May 1989.

[3] Martin Gardner. *Mathematical games - the fantastic combination of john conway's new solitaire game of „life“*. Scientific American, (223):120-123, October 1970.

[4] Nelson Minar, Roger Burkhart, Chris Langton, and Manor Askenazi. *The swarm simulation system: a toolkit for building multi-agent simulations*. http://www.swarm.org/archive/overview.ps.

[5] Margret Mitchel. *An introduction to genetic algorithms*. The MIT Press, Cambridge, Massachusetts, 1996.

[6] Eric S. Raymond, *The cathedral and the bazaar*. May 2000. http://www.tuxedo.org/esr/writings/cathedral-bazaar/cathedral-bazaar.html.

[7] Murray Smith. *Neural Networks For Statistical Modeling*. International Thomson Computer Press, London, Bonn, Boston, Johannesburg, 1996.

[8] G. Rozenberg V. Diekert. *The Book of Traces*. World Scientific Publ. Co., 1995.

[9] A. Muscholl V. Dikert. *Construction of asynchronous automata*, chapter In [8], pages 249-267. World Scientific Publ. Co., 1995.