UNIVERSITY OF DORTMUND
DEPARTMENT OF COMPUTER SCIENCE

Diplomarbeit / Diploma Thesis

Object-Oriented Ontogenetic Programming: Breeding Computer Programs that Work Like Multicellular Creatures

Peter Schmutter*
20/06/2002

*E-mail: email@Schmutter.de
Homepage: http://www.Schmutter.de

Diplomarbeit am
Fachbereich Informatik
der Universität Dortmund

Betreuer / Examiners:
Prof. Dr. Wolfgang Banzhaf
Dr. Peter Dittrich

To my parents

Abstract

As the research field called *Genetic Programming* has shown during the last decade, it is possible not only to write computer programs by hand but also to let the computer itself develop programs that solve given problems. This is achieved by simulating natural evolution on the computer for "breeding" programs that are well adapted to a specific problem environment. The use of mechanisms found in nature can lead to solutions to complex problems that by far outperform any man-made approaches. The reasons are that complex problems often are difficult to solve analytically and many other possible approaches are not accessible to the human way of thinking. The use of the mechanisms of evolution based on genetic variation and "survival of the fittest" is only one example. Another example are Artificial Neural Networks that imitate clusters of nervous cells and their interactions for solving difficult problems (inspired among others by the human brain).

The here presented work explores a different and new approach to adopting problem solving methods found in nature. It uses the natural cell control mechanism called *Gene Regulation* that according to modern molecular genetics is the basis of the cooperation between and differentiation into all the different cells in living creatures. The most astonishing example of self-organization between simple units that cooperate to solve complex problems is not the interaction between nervous cells on the basis of mutual electrical activation through explicit and directed connections. It is the interaction between all kinds of cells in a living creature which is based on the *diffusion* of messages in the form of produced substances. This interaction is much more powerful and flexible than the neural interaction because of many reasons. The main reason is, that a cell in this context is not only a simple unit which can have different levels of activation, but it is a complex system with many behavioural possibilities. The communication between the cells not only bases on different activation intensities but on many different message types which (also depending on their intensity) can have very sophisticated effects on the behaviour of a cell.

This new programming and control paradigm has been combined with genetic programming for breeding "multicellular" programs (which probably is the only feasible way of producing them). The system that implements this combination can not only be used to create programs with a new modular structure which has several advantages. It also is a great tool for developing systems of cooperating autonomous units like *Amorphous Computers* and *Multiagent Systems*.

Keywords:

Object-Oriented Ontogenetic Programming, Genetic Programming, Multicellular Programming, Swarm-Programming, Evolution of Distributed Intelligence, Gene Regulation, Embryology, Amorphous Computing, Multiagent Systems.

Acknowledgments

Thanks to Wolfgang Banzhaf for initially interesting me in genetic programming and ontogeny and then letting me do what I thought was right. Thanks to Peter Dittrich for drawing my attention to amorphous computing, for some good discussions and for always answering my e-mailed questions very quickly. Thanks to Christian Lasarczyk for always being of practical help when I needed it and thanks to Martin Villwock for finding several errors in the text concerning good English. Thanks to all the people that work at the chair of systems analysis for being such a nice community to be in. Last but not least thanks to all the developers that made Linux become such a great environment and thanks to Google for providing the currently best access point to the fantastic information base internet.

Contents

Chapter 1

Introduction

1.1 What do we know and what do we believe?

How often have you been absolutely sure that you are right and it later showed that you were wrong? Never? You cannot remember? Then there are three possibilities: 1. You are never absolutely sure. 2. You have a bad memory. 3. You are lying. You remember that it happened some times? Very good. And how do you explain that you still can be absolutely sure of things? Because everything speaks in favour of them? Was that not the case when you were sure of something that later showed wrong? If you have an argument with another person and both of you are sure to be right even though your opinions are contrary: What makes you be sure that you and not the other one is right? Because his arguments make much less sense? They might make less sense for you, because they do not fit into the other opinions and knowledge that you have. But for the person opposite it might be exactly the contrary. Your arguments do not fit as well into his outlook and experience as his arguments do. So for you everything (that is everything you know) speaks in favour of your views, but for somebody else nothing might speak in favour of these views. And this not only can happen with difficult questions for example about religious beliefs, but also with very simple experiences. People often experience the same situation very differently depending on their background. This is one of the main reasons why there are so many misunderstandings in human communication.

Even if everybody has the same opinion, that does not mean that this opinion is absolutely right. There was a time when everybody thought that the earth was flat. And even if you deny that the people at that time were real scientists, you can hardly deny that Isaac Newton and his colleagues were real scientists. But they all were proven wrong more than 200 years later by Albert Einstein. Because of these experiences that are typical for the history of science and because of logical problems with the verification of scientific laws, Karl Popper developed his now predominating view that it is not possible to verify a hypothesis about nature, but only to falsify it[1]. So every theory is only temporarily right as long as it has not been falsified[2]. No scientific "knowledge" is absolutely sure, because the only thing we can say is that it has not yet been falsified. But we can never be sure that it will never be falsified.

[1] He first published these thoughts in [Popper, 1935].

[2] This does not apply for theories about formal systems without any empirical component. Those theories can be verified. But that does not necessarily mean that the system is helpful. The conditions for a system of theories to be helpful are examined later in this section.

Already René Descartes realized[3] that the only thing that we can be absolutely sure of is that we (as a thinking subject) exist: "I think, so I exist."

Of course there are differences between a new scientific theory which is very specific and general knowledge that can be used every day. The latter has a much higher probability for being correct (i.e. for not getting falsified), because it has already withstood falsification very often and for a long time. But that does not mean that it can never be falsified. So theoretically, there is no real "knowledge" but everything is beliefs with different probabilities. Practically however, we can and do use the term "knowledge" for beliefs that have a very high probability for not getting falsified (e.g. that an object which has more mass than the displaced air falls to the ground when released) and we act as if we were sure that they are correct. As life is finite, there is no problem with this, because it is very improbable that we will witness an apple flying away when we let it fall. But still, only that it has never happened does not mean that it is totally impossible.

If there are different theories that all have not yet been falsified, in which one should we believe? Well, think again about the reason why you might be of the opinion that your arguments are better than the ones of another person. They make more sense in combination with the other ideas that form your outlook on the world and your knowledge. So we believe in those theories, that make the most sense for us, which means that they do not get in conflict with other knowledge, they help us understand things better or they can be used for practical purposes like prediction or production. Those theories are good theories. Because outlooks and knowledge of humans can be very different, the opinion which of some conflicting theories is the best one does not have to be undivided.

So the basis for the in the following presented knowledge and theories is:

1. The world is made of more or less probably correct beliefs.
2. A good theory is one that does not produce conflicts, serves well for explanation and/or can be used for practical purposes.

1.2 Nature is Wonderful

If you have ever tried to construct robots, you know how difficult it is to master all the technical problems to at least make them a little flexible, reliable and intelligent. We can construct robots that are stronger, faster, bigger and more exact than we are, but we cannot even get anywhere near the flexibility and efficiency of a very "simple" little insect. We cannot construct any robot that can do everything that this insect does, even if we could make it as big as we wanted. We do not even know exactly how this little creature achieves this perfection, because its workings are extremely complex even though it is so small compared to us. But that does not mean that we have no idea.

Biology has made great steps during the last decades. The rapidly growing knowledge of genetics has made it possible to take some cells from a mature sheep and give it an identical twin brother by letting them develop. According to medical scientists, it seems to be only a matter of years until we do not need artificial hearts or kidney donations any more because we can breed any spare parts for the human body without incompatibility problems. It is an open

[3] See [Descartes, 1996] for the English translation of his book "Meditationes de prima philosophia" published in 1641.

question if this is a good development: Immortality approaches. But it is really astonishing what is possible already now. The combination of genetics and microbiology makes it possible to use viruses as little helpers for the sake of human sanity. They can give the doctor a good helping hand by being able to give the human body cells more power in combatting a disease[4]. How does this wonderful cooperation work?

1.2.1 The code of life: DNA

For a cell to have more power here means to be able to execute functions that could not be executed before. How is it possible that cells get new abilities? How does a cell get its abilities? These questions are easy to answer if you know what DNA is. To put it very simply, it is a long chain of interconnected molecules that is found in every cell and contains all the information about what the cell could possibly do. This chain is like a long slice of paper with a text written on it. Every word[5] (which in the case of DNA is called a *gene*) of the text has a special meaning. The meaning of most of the genes is a description of how to construct the molecules of a specific substance. These substances are called *proteins*. The language in which the text is written only contains four different letters, but as one word can be very long[6], there is a vast number of different words which all describe the structure of different proteins. The whole text is called *genome*[7]. So we have a genome which consists of many genes which each code for a specific protein. And this genome (which is made of DNA) is contained in every cell.

1.2.2 The working unit: the Cell

What is at first astonishing is the fact, that even though the body of higher animals and humans consists of thousands of different cell types with very different functions, shapes and sizes, they all contain the same genome and have all developed out of one single cell. Why and how did they change their shape, size, location and function? And why are they different even though they contain the same genome? Does the genome not describe what the cell does after all? Yes, it does. But it describes, what the cell could possibly do. What it actually does and how it looks depends on which of the genes are read and which of the proteins described by them are being produced. And that depends on which genes are activated. So genes can

[4] This is one possible way for *gene therapy*. See for example [Pschyrembel, 1998] at this headword.

[5] Do not confuse this with a "word" in the scientific literature about genetics. As technical term it has a different meaning. Here it is only a visualization of the technical term "gene", like the text on the slice of paper is just a visualization of the chain of molecules.

[6] According to [International Human Genome Sequencing Consortium, 2001] the longest currently known gene in human cells contains 80,780 coding base pairs, which corresponds to a word of 80,780 letters.

[7] If you know any genetics you will probably have noticed that this is an extreme simplification of this complex subject. Of course, the genome of nearly all creatures is divided in many "chapters" on separate slices of paper called *chromosomes*. Its real structure is not one chain of molecules but two chains (except in some viruses) with a complex folding. And in some viruses it is made of RNA instead of DNA. To be even more exact, not every gene necessarily codes for a protein. If it is very short, the product would be called a polypeptide or even oligopeptide instead. But all this is information that is not necessary for understanding the principles of Genetic Programming and Object-Oriented Ontogenetic Programming. As it is impossible to be totally accurate because even the deepest biological expert knowledge is only a model and with that a simplification of reality, accuracy can not be the first goal. The goal must be to give as much information as is necessary for understanding the principles that will be explained here. And to give no information that is not necessary, because it unnecessarily complicates matters and makes understanding more difficult.

be active or inactive and the function of the cell depends on which genes are active and which are not.

You remember that a virus can give human cells more power? It achieves this goal by inserting specific genes that are needed for that power into the cell genome. Viruses reproduce by inserting their genome into a cell genome[8] and letting the cell produce new viruses. So the doctor only[9] has to replace most of the virus genome with the genes that are needed in the human body cells and the virus then takes the part of inserting it into the cells.

1.2.3 Division of labour and growth: Multicellularity

When a creature grows from a single egg-cell, the cells get more and more by dividing, which means making an exact copy of themselves, and sometimes they change their type because different genes get activated. But what activates or inactivates genes?

Not every gene codes for a protein. There are also some genes that have a quite different function. They can activate or inactivate other genes. And whether they do it or not depends on the substances that they find in the cell. Such a *regulator gene* might for example activate two coding genes (called *structural genes*) only if the concentration of a substance x in the cell exceeds a certain value. This regulation of gene activity is called *gene regulation*. To make it even more interesting, this substance x is very often a protein which can be produced either by the cell itself or by other cells in its neighbourhood. And many proteins can leave the cell where they are produced, *diffuse* to other cells and get into them. So it is possible that a protein produced by one cell activates a gene of another cell which therefore produces another protein which in turn inactivates the gene of the first cell which allowed the production of the first protein. You see that this mechanism enables very complex interactions between different genes as well of one cell as of different cells. And because the activation of genes also depends on other substances that are unequally distributed in the growing cell cluster of a developing creature, the cells differentiate to different types and start their intricate interactions[10].

And the fantastic or even unbelievable thing about these interactions and the growth and differentiation is: everything makes a lot of sense. The resulting cell cluster is a perfectly functioning creature where all the cells work together to make this creature live. The cooperation of cells in the body of living beings is one of the most amazing if not the most amazing example of small units that can each only execute a limited function but work together to achieve a higher goal. And it is very improbable[11] that the units know what they do, want to cooperate, have a higher goal in mind or have been told by somebody to do what they do for achieving the goal[12]. Instead, this cooperation simply emerges on the basis of the information contained in the genome. So you could say: The genome is the somebody who tells the cells

[8] Sometimes it is only inserted into the cell nucleus, which is the place where the production of proteins from genes takes place.

[9] Of course it is not as easy as that. There are many problems like for example the immune system, that recognizes the virus as enemy and tries to destroy it. For more information see [Pschyrembel, 1998] about gene therapy.

[10] For a much more thorough but still understandable explanation of the above described mechanisms see [Hirsch-Kauffmann and Schweiger, 1996] (my favorite biology book). Unfortunately, I do not know if there exists an English translation of it.

[11] Because that would not fit into the other theories of current biology. See section 1.1.

[12] As Jesper Hoffmeyer writes in [Hoffmeyer, 1997]: "We surely should not take it for granted that our different body parts love each other, or that they have any intentions as to maintaining us. As the American biologist Leo Buss has shown, we should rather ask ourselves how it can be, that the cells and tissues of our body do in fact co-operate in creating us."

what to do. But how could the genome be constructed so that this hypercomplex interaction results in just this perfect functioning body? (And it even does under very different environmental conditions and with many variations of the genetic code. This means that it is very reliable and insensitive to most errors.)

1.3 How did nature get to this Perfection?

For a long time, people looked at this perfection[13] and thought: "This can only have been constructed by an all-knowing and all-powerful god." This is a theory. And it has not yet been falsified. But is it also a good theory?

If you remember, a good theory should meet at least two conditions: It should be of help in explaining matters and it should not produce conflicts with the rest of the knowledge. Some theories even should be usable practically.

The theory of god as creator is not the kind of theory that we should be able to use practically. It is a theory that makes it possible to explain the perfection and origin of nature. And that, it really does. It is an easy explanation and an answer to many difficult questions. But does it also meet the second condition? In the old times when you did not go to the doctor but to the medicine man and he helped you by talking to the spirits, it really was a good theory, because the life and knowledge of the people at that time was full of ghosts, witches and other supernatural powers. It was a natural thing to believe in god or even in many different gods. But as the scientific knowledge increased and more and more things that could before only be explained by supernatural powers were found a scientific explanation for, it got increasingly difficult to believe in such a power. The other explanations were better theories, because they could be used practically whereas it was very dangerous to rely on supernatural powers.

Of course the existence and influence of supernatural powers could never be falsified because one could always argue that they were not in the mood for helping when they did not. But nevertheless, the scientific explanations were better, because they were reliable. So with the scientific revolution the supernatural powers were taken away more and more of their influence and power on people's lives. And at some point, it got so difficult to believe in these ruling classes for the people whose knowledge now consisted mainly of scientific explanations, that they tried to find another explanation even for the wonder of creation.

Charles Darwin[14] was the first to find such an explanation[15]. And this explanation is really a wonderful theory for those humans whose knowledge is made of scientific and natural explanations. Why is it "wonderful"? Because it not only meets all three conditions for a good theory[16], but it is also nearly as easy as the explanation with the supernatural creator. This theory is the *theory of evolution*.

On the basis of many different zoological facts, Darwin showed that the emergence of creatures is a historical process of evolution. But his most important achievement was, that

[13] And they had only a tiny little fraction of the knowledge about its perfect workings that we have now. They mainly saw the result.

[14] In his revolutionary book [Darwin, 1859].

[15] There were others before that also constructed theories for the development of the different lifeforms, like for example Jean de Lamarck, but their theories were not as good as the one of Charles Darwin because they did get in conflict with later findings.

[16] As you will see in section 1.5.2, it also (in contrast to the theory with god as creator) meets the third condition: It can be used practically!

he found a great causal explanation for this evolution. His explanation rests upon two main principles: 1. hereditary *variation* and 2. *selection* of the better adapted. Animals normally produce more offspring than would be necessary for the survival of the species. As life is dangerous for example because of predators and difficult because there is competition for food, water, space and partners, the individuals that are better adapted to solving these problems have a higher probability to survive and produce offspring (selection). The offspring is always a little different to its parents (variation), but it nevertheless inherits most of their attributes. So it is very probable that the parents pass their good qualities on to their offspring. Like that, good properties (for survival and reproduction) spread easier in the population than bad ones, because individuals with good attributes produce more offspring which tend to have the same good attributes.

Darwin did not know anything about genes and he did not know how the variation might work. Now, we have a lot of detailed knowledge about that. We differentiate between two main types of variation: 1. *mutation* and 2. *recombination*. Put very simply, mutation makes random variations to the genetic code[17]. For example, a letter of a "word" of the genetic text could be deleted or a random new one inserted at a random position. This changes the structure described by the gene. If the gene is activated, the cell will therefore produce another protein[18]. And this might change the power that this cell has in cooperating with the other cells and it might change an attribute of the whole individual. While mutation can happen all the time, recombination is only possible if two individuals get together and do something that combines their genomes. In the case of higher animals this is called *to have sex*[19]. Recombination can cause huge variations, but it does not create any new genes like mutation[20]. It simply takes the code of the mother and that of the father and produces a mix of the genes of both. The most prominent type of recombination is called *crossover*. Somehow[21] crossover manages to produce a mix that preserves all important attributes. The great majority of children have all the genes that are necessary for their complex development to a fully grown creature that looks and functions nearly like its parents.

Also bacteria have sex. But their recombination is of a quite different type which is made possible by a process called *conjugation*. Two bacteria build up a hose-shaped connection through which one of them gives a piece of its genome to the other. This sounds a bit as if it was the same thing as when we produce offspring. But it is not. The difference is, that in our case the two genomes exchange parts of their information, whereas in conjugation one is the donor and the other the recipient[22].

[17] In nature, there are many different forms of mutation. But most of them have the here described effects.

[18] This is often not the case, one possible reason for it being that the variation happened in a region of the gene, that does not have any influence on the structure of the produced protein. Such regions are very frequent in the genomes of higher animals and are called *introns*. As already stated the longest human gene contains 80,780 coding base pairs. But the real length of this gene (including all introns) is 4,800,000 base pairs. It is not yet known, if these introns have any function except reducing the probability that a mutation changes the function of the gene. Introns are not even the only non-coding DNA. About 70% of the human genome is extragenetic DNA which also does not have any coding function.

[19] This is not a technical term, but it is nevertheless important. At least for the animals (including us).

[20] This is true for most cases. But Seymour Benzer showed in [Benzer, 1957] that recombination is also possible within genes.

[21] How this works is mostly known and called *homology*. In section 3.2.2, this trick will be shortly explained and it will be shown that it might be introduced into the described system for enhancing the breeding of computer programs. For more information refer to [Hirsch-Kauffmann and Schweiger, 1996].

[22] Though this dissimilar mechanism can have the same evolutionary effects as crossover, it is nevertheless important in this context: While most of the previous approaches for breeding computer programs have used crossover, the system introduced in section 3.2 uses conjugation.

As we have seen, recombination provides new combinations of attributes and mutation creates new attributes. In combination with selection, which favours attributes that are good for survival and reproduction, the average genome in the population of creatures gets better and better in solving the problems of life. As one problem of life is the competition between the different individuals for resources, it is good for an individual to adapt to a new environmental niche. In the course of 3,750,000,000 years, this led from the simplest known organisms to the many different perfectly adapted species that now live on earth[23].

1.4 Should we try to copy nature?

This is a work on new developments in computer science, so we should start to look at this part of science. At first glance, computers do not have a lot to do with nature. They are purely artificial, very technical and very mathematical. Nearly everything about them is man-made whereas nature is not. But the theories that we have about nature are also made by us. So the image that we have of nature is also artificial and we will never be able to know anything about nature which is not our construction. You have to pay attention to the following important distinction: Our image of nature is purely artificial and we have constructed it so that it is easy for us to understand and meets at least two of the three conditions for a good theory. But nature itself (i.e. the something that seems to exist outside of us and about which we get information through our sensory system: eyes, ears, nose, sense of touch, etc.) is not made by us. This distinction between what we can know of nature and nature itself has been stressed by important philosophers like Plato[24], Kant[25], Schopenhauer[26] and many more. As we only have this reality, which consists of images that are made by us for bringing an order into the sensations that we have, it is of no importance[27] if there exists anything that is absolute and cannot be falsified and of what kind it may be[28].

So all theories about nature are man-made and they are quite explicit if they are scientific. Computer languages are very powerful in what you can describe by them. It seems as if it should theoretically be possible to formulate every theory about nature in a computer language and simulate nature on the computer. If that simulation behaves like nature, the theory seems to be a good theory because it (together with the other used theories) can produce the behaviour that it is supposed to describe. This seems to be a way of testing hypotheses and getting ideas for new ones. In fact, this is done very often. The part of computer science that tries to simulate biological processes is called *Artificial Life* (*ALife*).

But is it really possible to learn something about nature by trying to simulate it? Well,

[23]For more details on the evolution of species see also [Hirsch-Kauffmann and Schweiger, 1996] or [Maynard Smith and Szathmáry, 1995].

[24]In his allegory of the cave in the seventh book of [Plato, 1901] (originally called "Politeia").

[25]Kant distinguished in [Kant, 1781] between the thing-in-itself ("das Ding an sich") and the image of the thing that we get through our senses. He says that the only assertion that we can make about the thing-in-itself is that there must be something, that effects on our senses.

[26]He begins his major work [Schopenhauer, 1819] with the sentence: "Die Welt ist meine Vorstellung." ("The world is my representation.")

[27]Except of course if the idea that there is nothing absolute makes you be confused and unhappy. The goal of philosophy is not to confuse people but to build a base for an outlook on the world that makes sense for you. Therefore there are many different opinions in philosophy and they all can be good theories if they meet at least two of the three conditions mentioned in section 1.1. The here presented ideas are only one possible view which seems to go well with the development of scientific knowledge.

[28]This is the view of "constructivists" like Humberto R. Maturana and Francisco J. Varela.

let us see what we are looking for: We want theories that: 1. do not get in conflict with each other and 2. help us understand what we experience and if possible 3. can be used for practical purposes. Does simulation help in testing any of these conditions? On the answer to the second condition we have to decide ourselves. So for this condition, simulation is not of any help. But simulation is a good tool for finding inconsistencies[29]. This means that the first condition could be a reason for using simulation. The question if it can also help in finding theories that can be used practically is a really difficult one to answer. Rodney Brooks writes in [Brooks, 1992]:

> Previously we have been very careful to avoid using simulations for two fundamental reasons: 1. Without regular validation on real robots there is a great danger that much effort will go into solving problems that simply do not come up in the real world with a physical robot. 2. There is a real danger (in fact, a near certainty) that programs which work well on simulated robots will completely fail on real robots.

This is a statement which shows that it is very problematic to conclude knowledge about reality from a simulation. That does not mean that simulation can not at all reach the goal to produce theories that can be practically used. But these problems show that for a theory which is supposed to be usable in real life it is not enough to show its usability in a simulation. It also has to be tested in reality. So simulations can only give a hint about the practical utility in the real world.

This is different if you are looking for theories that are to be used for solving problems on a computer, because in this case the computer itself is the reality. This means that the program run is not a simulation any more but a test in reality. For the development of such problem solving programs we could try to use theories that we have about nature. We know that nature solves very difficult problems. So why not learn from nature and use the same principles for solving computational problems? This approach is also very widely used in computer science. The functioning of the human brain inspires the field of *Artificial Neural Networks* (*ANN*). Natural evolution and genetics inspire the field of *Evolutionary Computation* (*EC*). At first glance we have the same problem that we had in Artificial Life, only inverted: It is not sure that the theories that work for nature also work on the computer. But like in Artificial Life, this can be tested by using them in reality which in this case is the computer program. And as the many works in the two mentioned fields have shown, it is quite a good idea to try using principles from nature to solve computational problems. Also, these problem solvers not only work in a scientific environment and for easy problems, but there are real-life computational problems that they can already solve better than any other approach[30].

This does not mean that there are no disadvantages. Most nature-inspired problem solvers produce solutions that are very difficult to understand for humans. But as long as the solution is only to be used practically and not for theoretical purposes, this is not a big problem. A bigger problem for evolutionary problem solvers is that they normally take a long time for finding a good solution. Natural evolution took almost 3,750,000,000 years for developing all the complex solutions that we now find on earth. But we cannot wait so long for our program to finish. The good news is: The computational problems that we have to solve

[29] See for example [Elman et al., 1996].

[30] See for example the possible applications of Genetic Programming (one branch of Evolutionary Computation) presented in [Banzhaf et al., 1998].

are far from being as complex and difficult as the problems that natural creatures solve. It seems that artificial evolution has other advantages that make it faster, but at least some of these advantages also result from the fact that the problems are much simpler. For example, artificial evolution does not have to wait for about 22 years before performing each crossover operation. The generations are much shorter. Therefore artificial evolution can test more solutions in less time. But bacteria also do not take as much time for a generation. This means that they can evolve quite quickly compared to us which shows in the fact that they easily get resistant against antibiotics. And most (if not all) solutions produced by Evolutionary Computation until now are not even as complex as bacteria. The more complex the problems will get, the more time it will take to evaluate the solutions[31]. Of course there are other differences that might still make artificial evolution faster than the natural example[32]. It is an open question if we will ever be able to produce solutions as complex as human beings. But it is also an open question if we need such complex solutions.

1.5 What has been done until now

As mentioned in section 1.4, there are two major research areas in computer science that are strongly inspired by natural evolution and genetics: Artificial Life and Evolutionary Computation. In both areas there exists a body of literature that is already so big that it is impossible to present or even know every idea that has been published until now. This is not something unusual. In nearly all areas of research you have the problem of finding a good compromise between having broad and interdisciplinary knowledge (which is good for getting new ideas) and having complete knowledge about a special subfield (which makes a good expert). This introduction about recent work in Artificial Life and Evolutionary Computation tries to give you an overview with concentration on approaches that are more closely related to the ideas presented in this thesis. In the section about Artificial Life these will mainly be publications about modelling cells and especially multicellularity, as this is the best natural example for the organization of cooperation between many small parts to reach a higher goal. The section about Evolutionary Computation will present some approaches that either use such a cooperation for building problem solutions or evolve solutions that exhibit a cooperative behaviour themselves.

1.5.1 Simulating natural processes: Artificial Life

Simulation and modelling of biological processes has a long history in computer science. But until 1987, there was no common name for this field and it was therefore very difficult to find all the literature that had the same objective. Because of these problems, Christopher G. Langton organized the "First workshop on Artificial Life" which gave a name to this branch of science and made scientists of very different areas get aware of the many similarities in their work and visions.

Like there are many different areas of research in biology, there are also many different approaches to Artificial Life. If you have read everything up to this point, you know at least two very important processes of life on earth: Evolution and Gene Regulation. The

[31] Until now, the evaluation time depends mainly on the *type* of problems. But with higher complexity of the problems, also the complexity of the solutions will have to increase and this will take much execution time. The OOOP approach is an example for this development.

[32] For example a higher mutation rate.

latter is one of the key conditions for the development of individual creatures from a single egg cell to the fully grown individual. This development is called *ontogeny*[33]. As we have seen in sections 1.2.3 and 1.3, the working of these two developmental processes (evolution as the development of a species and ontogeny as the development of an individual member of the species) is very different. Even though some developmental stages of an individual have similarities with evolutionary stages of its species[34], the functioning of the development is of a totally different kind. While the driving forces for evolution come from outside (variation and selection are mainly an effect of the environment), ontogeny is mostly an effect of the internal information and interactions, only modified by the environment[35].

Phylogeny

Evolution is also called *phylogeny*[36]. This process is the basis for the whole branch called Evolutionary Computation. Without Darwin's seminal ideas, we would not be able to breed problem solutions. So this is probably by far the most simulated natural mechanism. And the good results of using its workings practically make Evolution be an even better theory. But there are also researchers in Artificial Life that want to find out more about natural phylogeny. Larry Bull has for example examined the initial conditions for the emergence of multicellularity in [Bull, 1997] by comparing the fitness of unicellular and simple multicellular simulated organisms under different conditions.

Multicellularity

A wide and frequently covered area of research is *morphogenesis* or *pattern formation*. The main question in these areas is: What are the conditions for the development of forms found in nature? As shapes, structure and patterns in nature are usually built of and by many cells that have grown into this form, the researchers working on these questions also have to model multicellularity.

One very popular approach to doing this is to use *Cellular Automata* (*CA*). A cellular automaton is a mathematical model of cells and their interactions originally conceived by Stanislas Ulam and John von Neumann and presented by the latter in [von Neumann, 1966]. Von Neumann wrote:

> The question we hope to answer ... is: what are the basic principles which underlie the organization of these elementary parts in living organisms?

The model is very far away from the natural example, but it is a good basis for studying complex interacting systems in general. Moshe Sipper describes cellular automata in [Sipper, 1997] as

[33] In contrast to the term *embryology*, this does not only denote the maturation process, but the whole lifetime development of the body including ageing and death.

[34] This lead Ernst Haeckel in 1866 to formulate the basic biogenetic rule: "The Ontogeny of an organism is a recapitulation of the Phylogeny." This does of course not mean that an embryo could survive in an environment in which its grand-grand-grand-... parents lived, but the rule nevertheless seems to help in explaining some peculiarities of embryological development. For more information refer to [Hirsch-Kauffmann and Schweiger, 1996].

[35] From another point of view this can of course be seen differently, but in this context it helps in understanding the different approaches needed for modeling Evolution and Ontogeny on a computer.

[36] This term is especially used in a context with Ontogeny.

> ... dynamical systems in which space and time are discrete. A cellular automaton consists of an array of cells, each of which can be in one of a finite number of possible states, updated synchronously in discrete time steps, according to a local, identical interaction rule. The state of a cell at the next time step is determined by the current states of a surrounding neighborhood of cells.

Hans Meinhardt for example uses differential equations in [Meinhardt, 1995] for developing many different patterns with striking similarity to patterns found on sea shells and according to [Pfeifer et al., 2000] these can easily be reformulated to cellular automata rules.

Another approach to simulating cells is to construct an *Artificial Chemistry*. This means that there are rules which determine the reaction of different substances getting into contact and that the substances can *diffuse*[37] into the local environment[38]. Chikara Furusawa and Kunihiko Kaneko use this method in [Furusawa and Kaneko, 1997] for studying cell differentiation. In [Furusawa and Kaneko, 1998], they simulate multicellular pattern formation with an artificial chemistry and in [Kaneko and Yomo, 2000], Kunihiko Kaneko and Tetsuya Yomo find that differentiation on the basis of the modeled interactions can lead to discrete cell types that are not mixed by recombination. Also Cefn Hoile and Richard Tateson use such a reaction-diffusion system in [Hoile and Tateson, 2000], but with the difference that they use an artificial neural network for producing the reaction output. Hiroaki Kitano and his coworkers present in [Kitano et al., 1997] their Virtual Biology Laboratories, in which they try to simulate all the biochemical processes happening in cells as detailed as possible.

Gene Regulation

There are also different approaches to modelling gene expression networks which work on the basis of the gene regulation mechanisms introduced in section 1.2.3. Most works concentrate on the interactions without modelling diffusion and spatial dimensions or growth. This means that these simulations only model the processes in single cells. Shoudan Liang et al. use boolean networks in [Liang et al., 1998] for inducing gene regulation models in a single cell from observed data. Sean Luke et al. use gene regulation abstracted from the natural example of Drosophila[39] for evolving deterministic finite state automata [Luke et al., 1999]. Thorsten Reil examines in [Reil, 1999] the dynamics of gene expression networks on the basis of template matching in a DNA-like sequence and draws interesting conclusions for artificial and biological ontogeny. Jan T. Kim also introduces a formalism for modelling gene regulation networks which is quite close to the natural example [Kim, 2001]. But they all only model the interactions in one single cell.

Peter Eggenberger uses the same principles of template matching in a DNA-like sequence to model gene regulation networks, but he expands this by adding a spatial dimension to it and by simulating multicellular growth processes. He uses this approach both for pattern formation in [Eggenberger and Dravid, 1999] and for morphogenesis of 3D shapes in [Eggenberger, 1997].

[37] This means spread out.
[38] For a much more thorough description of artificial chemistries see [Dittrich et al., 2001].
[39] Drosophila is a fly which is one of the most intensively studied species.

Swarm Behaviour

In their important book [Maynard Smith and Szathmáry, 1995] John Maynard Smith and Eörs Szathmáry compare a termite-hill with a well-designed house:

> Although the mound resembles a human building in having features ensuring the comfort of its inhabitants, it differs in that not one of its builders had a picture of the completed structure before building started. The structure emerged from the rule-governed behaviour of tens of thousands of interacting workers. In this, the mound resembles a human body rather than a human building. The body is built by the rule-governed actions of millions of cells. Nowhere is there anything resembling a blueprint of the body. At most, the genome is a set of instructions for making a body: it is not a description of a body.
>
> The resemblance between the development of an insect colony and of an organism has led to the concept of a 'superorganism'. The analogy has some value.

So according to John Maynard Smith, swarms of ants, termites or bees can in many respects be compared to multicellular organisms. Jesper Hoffmeyer also compares the body of multicellular creatures with animal swarms, only inversely. He writes in [Hoffmeyer, 1997]:

> The body can be understood as a swarm of cells and tissues which, unlike the swarms of bees or ants, stick relatively firmly together.

There has been quite a lot of work on simulating swarm behaviour by modelling mobile *reactive agents*[40] that interact in a simulated environment. Jean-Louis Deneubourg and Simon Goss found an explanation for the fact that ants find the shortest path between food sources and the nest and prefer nearby located food by simulating their interactive behaviour [Deneubourg and Goss, 1989]. The results have even been used to solve computational problems such as the Traveling Salesman Problem [Colorni et al., 1992]. Craig Reynolds simulated flocking birds [Reynolds, 1987] and Maja Mataric has done a lot of work on constructing socially behaving robots [Mataric, 1995; Mataric, 1997]. More complex agent-based models with agents that have internal states are widely used for all sorts of simulations including economic models, traffic simulations, ecological simulations and simulation games such as SimCity [Hiebeler, 1994]. But the behavioural rules for the individuals are nearly always formulated manually. There are very few attempts to finding good rules by letting them evolve automatically. We will get back to this in the next section.

1.5.2 Breeding problem solutions: Evolutionary Computation

The use of evolutionary principles for breeding solutions to computational problems instead of solving them analytically already started in the late 1950s with works of R. M. Friedberg [Friedberg, 1958]. But the application and invention of approaches that are still used today started in the 1960s.

[40] This is a term that is used in the field of Multiagent Systems, a branch of Artificial Intelligence which will be introduced in section 4.1.2, for describing computational units that do not have an internal memory but simply independently react depending on their environmental input and some internal rules.

Evolutionary Strategies and Evolutionary Programming

The first two important methods were *Evolutionary Programming* (*EP*), where finite state automata are being evolved, invented by Lawrence Fogel [Fogel et al., 1965] and *Evolutionary Strategies* (*ES*), which are numerical optimizations on the basis of evolutionary principles pioneered by Ingo Rechenberg and Hans-Paul Schwefel [Schwefel, 1965; Rechenberg, 1970]. Evolutionary strategies are together with the next development—*Genetic Algorithms*—the best known and until now the most widely used *Evolutionary Algorithms*[41] (*EA*).

Genetic Algorithms

Genetic algorithms (*GA*) were invented by John Holland in the early 1970s [Holland, 1975]. The main difference to evolutionary programming and evolutionary strategies is that the variations are not performed directly on the problem solution. There is a (often binary) representation of the solution which is varied and mapped onto the real solution by some function. The real solution then is tested on the problem to get the fitness of the individual. The used function is called the *genotype-phenotype mapping* because the representation corresponds to the natural genome (the *genotype*) and the real problem solution corresponds to the living being (the *phenotype*) which develops on the basis of the genome. This difference to Evolutionary Strategies is very important because on the one hand, it makes the representation be shorter and more powerful in expressing different solutions, but on the other hand, the differentiation between genotype and phenotype usually makes the search for a good solution less powerful. This happens because depending on the mapping function a small variation of the genotype can lead to a big variation in phenotype and the inverse. In evolutionary strategies, the size of the variational steps is reduced with the solution getting better. This makes it possible to approach the optimum quickly and then find it quite exactly without jumping around it with too big steps. This normally is not possible in genetic algorithms. But not all problem solutions can be represented in real numbers. For many problems it is much easier to use genetic algorithms. William E. Hart et al. examine the possibilities and impacts of developmental mechanisms (including not only the genotype-phenotype mapping but also learning which denotes an adaptive quality that remains in the problem solution) in genetic algorithms in [Hart et al., 1995].

An example for the use of genetic algorithms with cellular automata is described by Moshe Sipper in [Sipper, 1997]. He uses the genetic algorithm technique for evolving good reaction rules for the automaton cells and calls this method *Cellular Programming*. This is a first approach to breeding behavioural rules for many small units in order to enable them to cooperate for reaching a higher goal. But it is very limited in its usability as well because of the simplicity of cellular automata (uses regularly placed, synchronously updated, reliable units without internal memory and with quite restricted communication) as because of the limitations of the genetic algorithm (compared to Genetic Programming which is introduced later in this section).

Peter Eggenberger used genetic algorithms together with gene regulation processes for evolving artificial neural networks for given problems [Eggenberger, 1996]. Jens C. Astor and Christoph Adami independently describe a similar combination in [Astor and Adami, 1998].

[41] This term comprises all algorithms for evolutionary computation described in this section and is often used as a synonym of EC.

Genetic Programming

Genetic Programming (*GP*) is the most recent important new model for evolutionary computation. Even though already Alan Turing thought about breeding computer programs in 1950 [Turing, 1950], it was not really done until John Koza published his important book [Koza, 1992]. Since then, the genetic programming community grows rapidly. The main advantages of GP are the following:

1. GP does not impose any fixed length of the solution. Computer programs evolved by GP have variable length, so that the algorithm can find the size necessary for a solution itself. In principle, the maximal length can be extended up to the hardware limits.

2. GP does not require as much knowledge about the problem and the possible solutions (*domain knowledge*) as do GAs. This holds firstly because of the advantage mentioned above and secondly because it is not necessary in GP to decide on a genotype representation and a genotype-phenotype mapping function. These decisions are crucial for the power of GAs in solving a problem and they usually require a lot of domain knowledge.

3. As computer programs are very powerful in describing computations and computational behaviour, so is GP. Using GP, you can theoretically evolve any series of actions a computer can possibly do, provided that you give the GP algorithm a set of commands to choose from that can describe all possible actions. This set of commands is called the *function set*. It is normally quite easy to put together a powerful function set and to add or remove functions from it if they are (not) needed.

Of course, genetic programming also has several disadvantages (if not, why would there still be people using something else):

1. The number of possible programs that can be constructed by the algorithm (called the *search space*) is immense. This is one of the main reasons why people thought that it would be impossible to find programs that are good solutions to a given problem. But this problem is not as big as it seems, because there are also an enormous number of ways to construct a solution. You can construct already a nearly infinite number of solutions by simply adding commands that do not have any influence on the result. This shows that it is very improbable that GP finds the best solution, but it is nevertheless able to find a good solution. This does not only hold for GP but for all evolutionary algorithms. Many of ES and most of EP, GA and GP are *heuristic* algorithms, where it cannot be granted that they find a solution of predefined quality (e.g. the best), but which normally find very good solutions.

2. Depending on the programming language that is evolved by the GP algorithm, the process of finding a good solution can take a very long time. GP that uses machine code is usually very fast[42]. But if a high level language is used that must be compiled and creates big programs (e.g. C++), it can get very slow. On the other hand, high level languages have many advantages: You have a much bigger choice of specialized

[42] See [Banzhaf et al., 1998] for getting a good overview on this type of GP and genetic programming in general.

commands for the function set which can make the search space smaller[43]. It is much easier to start with already known good solutions for optimizing them. And the resulting programs can be easier to understand for humans. Moreover, as will be shown in section 2.1.2, high level languages provide functions and methods that make it much easier to evolve complex programs (e.g. by using classes and objects).

3. Even though the variations in GP are usually performed directly on the problem solution (the program code), it has the same disadvantage compared to ES as have GAs. A very small variation (e.g. changing one command) can have huge effects on the quality of the solution. In GP this problem is normally even more serious than in GA. There is a high probability that even a very small variation has a disastrous effect on fitness or that a variation has no effect at all. This can be described graphically by saying that the *fitness landscape*[44] is very rugged. It is a very serious problem, because the main reason for evolutionary algorithms being better than random search is that better solutions are selected and better solutions have a higher probability of producing better offspring (which means that variations of good solutions are usually better than variations of bad solutions). Fortunately, this still holds for GP, but the rugged nature of the fitness landscape makes it much more difficult to find the optimum.

There is a vast body of literature on genetic programming, but there are very few approaches until now that have tried to use it for optimizing the cooperation of many small entities. On the one hand, this is astonishing because the organization of cooperation in such a swarm of independently acting units is a very hard problem which cannot easily be solved analytically. So why not try evolution on it? But on the other hand, because it is such a hard problem, it could easily be too complex for current EC techniques. We will see later, that the here described new communication and control paradigm is perhaps a solution that makes the problem get in reach for evolutionary computation even for complex environments and behaviours. But we will first look at some other publications that have taken steps towards evolving cooperation with genetic programming.

Thomas Haynes et al., who come from *Multiagent Systems*[45] (*MAS*) research, use strongly typed genetic programming[46] in [Haynes et al., 1995] for evolving behavioural strategies for the agents in a multiagent domain. As testing domain they use the predator-prey pursuit problem, which is well known in the MAS research community. Their agents do not communicate and they all follow the same evolved rules. Sean Luke and Lee Spector use a very different and quite interesting approach for evolving teams of agents that follow different rules [Luke and Spector, 1996]. They use *Automatically Defined Functions*[47] and *Automatically Defined Macros*[48] for

[43] This holds, because if you have more specialized commands and you also have domain knowledge, you can choose only to include those commands into the function set that make sense for the particular problem. Moreover, as specialized functions are combinations of multiple low level functions, you could compare this question with the situation that you have one piece of paper and you have the choice between randomly printing characters on it or randomly printing words. Even though the set of words is much larger, the number of possible resulting texts is much greater if you use the characters.

[44] The fitness landscape is a metaphor for the dependency of the fitness on the location of the individual in the search space. If there were only two variable parameters in the EA, the fitness landscape would be the curve resulting from putting the two parameter values on the x and y axes and the corresponding fitness value on the z axis.

[45] Multiagent systems are explained in section 4.1.2.

[46] This is a special variant of GP for higher languages.

[47] A modularization approach for genetic programming presented by John Koza in [Koza, 1994]. An evolved program is divided into several separate functions and a main procedure which can use these functions.

[48] An extended version of automatically defined functions introduced by Lee Spector in [Spector, 1996].

evolving a whole team of different agents as one GP individual. Every function or macro incorporates the behavioural rules for a different agent.

The disadvantages of all the mentioned approaches are that the resulting behavioural strategies are not very flexible in adapting to dynamic environments, there is no or very limited communication between the acting units, and the resulting agents are either homogeneous [Haynes et al., 1995] or heterogeneous [Luke and Spector, 1996; Sipper, 1997], but cannot change their strategies while cooperating, for example if more agents for a special function are needed. This does not allow for evolving high level multiagent systems[49] like the ones that have until now only handcrafted behaviours.

As far as we know, nature uses a different approach. As we have seen in sections 1.2.1 through 1.2.3, all cells contain the same control program, the genome. But nevertheless, there exists a very complex interaction between the cells, which differentiate to many different types and perform very different functions. The basis of this—gene regulation—seems to be a solution to the problem how to evolve a team of many heterogeneous agents. Moreover, this mechanism implies a totally new kind of communication which is powerful even though it is very simple concerning the message structure. The use of gene regulation has many advantages, but we will come back to that later.

This is not the first work that stresses the power and importance of gene regulation for evolutionary computation. Sean Luke, Shugo Hamahashi and Hiroaki Kitano wrote in [Luke et al., 1999]:

> While evolutionary computation has drawn some inspiration from modern genetics, by and large the inspiration for this field has been aging Mendelian ideas ... Even Melanie Mitchell's highly-regarded GA text [Mitchell, 1996] introduces biological genes thus: "A chromosome can be conceptually divided into genes—functional blocks of DNA, each of which encodes a particular protein. Very roughly, one can think of a gene as encoding a trait, such as eye color. The different possible 'settings' for a trait (e.g., blue, brown, hazel) are called alleles." This is not how it actually works ... Genes form a complex network of interrelationships and regulation which, when seeded with initial chemical concentrations distributed throughout cells, results in miniature chemical machines ... In the rest of this paper, we begin by describing our work in modelling an interesting area of developmental genetics, gene regulation, which we think has particular utility to many new domains in evolutionary computation.

Also Hassan Masum and Franz Oppacher saw the power of gene regulation for evolutionary computation. They wrote in [Masum and Oppacher, 2001]:

> We focus particularly on regulatory networks and embryogenesis, as these provide a mechanism that is key in developmental biology and genetics but has not been substantively used to date in EC.

Seemingly, these works were not enough to make many researchers take notice of the powerful mechanisms that are yet unexplored. The above cited researchers saw the power for EC, but as they mainly work in Artificial Life research, they seemingly did not continue with the idea and did not yet produce systems capable of using that power for real applications. As we will

[49] Compare [Weiss, 1999].

see in section 2.1.1, my ideas for Object-Oriented Ontogenetic Programming and the use of gene regulation for evolutionary computation also developed out of studies in Artificial Life that modeled these processes and out of the resulting question why nobody uses this powerful mechanism for solving practical problems until now. I hope, that this work at last can make more people recognize the possibilities of gene regulation mechanisms for automatic problem solving and adaptive systems design and that it makes a whole new area of research begin to grow: *Evolution of Distributed Intelligence*[50] (*EDI*).

[50]This will be explained in more detail in chapter 4.

Chapter 2

The Multicellular Program

2.1 The Object-Oriented Ontogenetic Programming Paradigm

We want the computer to be our tool for breeding programs that control the behaviour and communication of many autonomous units which work together for solving problems that could not be solved by single units or with a centralized system. In this chapter, a new communication and control paradigm will be introduced, that seems to have the power for enabling very complex cooperative interactions of elaborate autonomous units but still is suited for automatic evolution of the unit control programs. This paradigm is called *Object-Oriented Ontogenetic Programming* (*OOOP*), because on the one hand it uses object-oriented programming techniques but on the other hand it expands these techniques with a form of communication and cooperation between the objects that is inspired by the interactions between cells in natural ontogeny (i.e. growth and differentiation processes in multicellular creatures). Before we will see how OOOP actually works and what are its pros and cons, we will take a short look at the evolution of its basic ideas.

2.1.1 How and why did the idea for the new paradigm evolve?

In the beginning, there was a lecture about genetic programming and a seminar about artificial life held by Wolfgang Banzhaf. I studied his great book [Banzhaf et al., 1998] and wrote a paper about the works of Peter Eggenberger on artificial embryology [Schmutter, 2000a]. Before, I had learned about gene regulation in genetics lectures for my subsidiary subject medicine. In the works of Peter Eggenberger, I read about how he simulated that mechanism on the computer. But I asked myself, why this was not used for problem solving. It looked as if it was a very powerful idea. In the lecture about GP, I got the idea to improve the existing modularization techniques for genetic programming. Then I thought about doing this by combining ontogeny like the one modeled by Peter Eggenberger with GA and GP for developing programs out of DNA-like representations with ontogenetic growth processes instead of a simple genotype-phenotype mapping. I thought that this would perhaps enable using tricks from nature like homology for improving crossover performance and that it might enable the evolution of more complex programs in general. I also thought that it would

perhaps make the *schema theorem*[1] apply for this approach to GP because it would be possible to develop a variable length program out of a fixed length representation.

My first idea was that the cells could grow in one dimension only and then develop into different parts of the program by gene regulation. The one-dimensionality would naturally induce an order of execution. But then, I had the idea that an object-oriented approach would be much smarter. The cells would be objects that could grow in a three-dimensional space and build much more sophisticated interactions. I named this "Object-oriented Genetic Program Development". Unfortunately, I then confused the objects with genes and planned to use as representation a set of arrays each of which corresponded to a gene and described one class. I thought that it would be a good idea to use these classes as predefined building blocks and let crossover only mix classes and mutation only vary the code in theses classes. When I thought about what I would do with the resulting objects, I noticed that I had confused something and started a new definition.

Now, objects were cells again and there was only one class for all the interacting objects. The genes were represented through member functions of the class. The activation of a gene corresponded to the call to a member function. This was an interesting new way of modularizing programs: There were modules that described different functions (the genes), but there also was a different level of modularization where all the modules consisted of the same program code (the cells). But as in the natural example, their function was not homogeneous because of different called functions depending on different concentration of messages at their location in the three-dimensional space. Also, the objects were able to have different internal states like different concentrations needed for activation of genes or temporal memory used by the evolved code. This definition was already the birth of OOOP, even though I did not yet see the really interesting field of applications that opened up. Those ideas evolved much later when I read about amorphous computing [Abelson et al., 1999]. The application to mobile entities even did not come to my mind until I read about the work of David Evans [Evans, 2000]. And that was after I had already finished implementing the *Object-Oriented Ontogenetic Programming System* (*OOOPS*) described in section 3.2. Now I have already introduced some key features of OOOP, but how does it really work?

2.1.2 How does it work?

Object-Oriented Programming is a great way of structuring programs and it has many other strengths, too. This is the reason why most computer programming of complex systems now is done with the object-oriented approach and most of the recent programming languages (e.g. C++, Java, Visual Basic) are object-oriented languages. In these languages, *objects* are units with internal functions and memory[2] of which some functions can be called from other objects or from the main program and usually build the only interface to the object's abilities and internal state. The structure of these objects is based on templates called *classes*. It is only possible to create an object by instantiating it from a class, but from one class you can instantiate as many equal objects as you want. The functions defined in a class can only be

[1] This is a theorem that John Holland formulated in [Holland, 1975]. Wolfgang Banzhaf et al. describe it in [Banzhaf et al., 1998] as follows: "Essentially, the schema theorem for fixed length genetic algorithms states that good schemata (partial building blocks that tend to assist in solving the problem) will tend to multiply exponentially in the population as the genetic search progresses and will thereby be combined into good overall solutions with other such schemata. Thus, it is argued, fixed length genetic algorithms will devote most of their search to areas of the search space that contain promising partial solutions to the problem at hand."

[2] To be more exact: they have internal variables called attributes.

Nature	OOOP
Genome = Description of all possible cell functions	**Class** = Description of all possible object functions
Gene = Description of one cell function	**Member function** = Description of one object function
Cell = Instance of working unit described by genome	**Object** = Instance of working unit described by class
Individual = Multiple cells with same genome, interacting by proteins	**Program** = Multiple objects of same class, interacting by messages
Cells behave differently depending on **active genes**	Objects behave differently depending on **executed functions**
A gene is activated if the concentration of **proteins** at the cell's position meets the requirements	A function is executed if the concentration of **messages** at the object's position meets the requirements

Table 2.1: Some analogies between nature and OOOP.

used and the variables can only store something in an object of the class. So the object is the working unit and the class is the information base that defines the structure and the possible functions of the working unit. This is the same relationship as that between the genome and the cell in nature. The genome is the informational basis but cannot execute any function itself. On the basis of its information has to be constructed a cell that then is the working unit.

To extend the analogy, the whole program should be made of many objects which are all instantiations of the same class like the body of an individual creature is made of many cells that all contain (and base on) the same genome. This already is a big difference to traditional object-oriented programming because normally the objects take on different functions in the program by being based on different classes. But as in nature, the objects in the *multicellular program*, which are all instantiations of the same "genome" class, can also take over different roles in the program by activating different "gene" functions. As a class contains several *member functions* which hold the executable code, these are great counterparts to the genes. A member function is activated (which means that it is executed) if its activation conditions are met. As we have seen in section 1.2.3, the activation of genes depends on the concentration of proteins at the cell's position. In analogy to that, the execution of a member function in OOOP depends on the concentration of messages at the object's position. The big advantage of this approach to classical object-oriented design is that all objects can adaptively change their roles while interacting and a very complex cooperation of the objects can emerge or grow automatically without having to be planned beforehand. Only the genome (that is: the class) has to provide all the possibilities and preconditions so that the useful cooperation can arise[3].

[3]This of course is a condition which is very difficult to fulfill. How should a genome be constructed so that a specific useful interaction results? Well, do you remember the question that we asked about the natural example: "How did nature get to this Perfection?" The answer to that question will also be the answer to the problem of constructing good OOOP genomes: We will evolve (or better "breed", because we determine the direction of development) them. How this is done exactly will be described in chapter 3.

Advantages of OOOP
Robust and fault tolerant
Adaptive and dynamic
Modular
Powerful interactions / communication
Evolvable
Emergent differentiation
Combines advantages of homogeneous and heterogeneous teams
For centralized and distributed problem solving
Close to nature
Open and easily extendable

Table 2.2: Advantages of object-oriented ontogenetic programming.

There are three important preconditions, that distinguish OOOP decisively from traditional object-oriented programming: Firstly, there is a topology which means that the objects have specified locations in a *virtual space*[4] (which does not have to be three-dimensional), secondly, all the objects are instantiations of the the same class which contains all possible functions, and thirdly, the interaction between the objects does not rely on calls to specific functions but on the production of "broadcast" messages that diffuse into the space and fade with growing distance to the producing object. As in nature, the behaviour of an object depends on the concentrations of messages at its location. It seems to be a good idea to design an *Object-Oriented Ontogenetic Programming Language* (*OOOPL*) that offers these mechanisms directly[5], but this was not the goal of the here presented research. Instead, the mechanisms were modeled using the possibilities of current object-oriented languages.

2.1.3 Advantages

Though we have already seen some advantages of the object-oriented ontogenetic programming approach, we will look at them and several others in more detail now. It really has to be stressed what a great problem solving ability the use of gene regulation in computer science can provide. OOOP is a natural and promising way to use these opportunities. Table 2.2 lists several benefits of object-oriented ontogenetic programming. These benefits will be explained in the following.

Robust and fault tolerant

The interactions in a gene regulation network have a strong implicit robustness as well against disturbances in the communication path as against faults or breakdown of individual acting members. In his work about the dynamics of gene expression [Reil, 1999] Torsten Reil finds:

[4] I also call this space *cell space* because it is the space in which the cells of the multicellular program are located.

[5] This language could also considerably improve the memory consumption of multicellular programs for example by allowing to dynamically instantiate member functions only if their activation conditions are met. This would be an even better analogy to nature and (more importantly) would save much memory as the objects would only reserve memory for their activated member functions. On the other hand, it could also slow down execution, because of the resulting continuous instantiation and memory allocation processes.

> Furthermore, the experiments suggest that a high degree of robustness is possible without selection. This is manifested in two ways: a) transient disturbances of gene expression (carried out manually in the model) do in the majority of cases not result in altered expression patterns, and b) loss of gene function (achieved by manually knocking out genes) does in most cases not affect the overall pattern of gene expression ... Traditionally, such robustness is thought to have arisen from evolution [Gilbert, 1994]; it was shown here that this need not be the case.

More generally, Cefn Hoile and Richard Tateson compare the interactions between cells in a multicellular creature with traditionally designed artificial systems [Hoile and Tateson, 2000]:

> Multi-cellular organisms have a number of features which are desirable in artificial systems. They can adapt to changing circumstances, through learning or evolution. "Faults" arising from internal errors or injury can often be rectified. Irredeemable faults are seldom the cause of catastrophic failure because the redundancy of the system allows most or all functions to continue.
>
> This contrasts markedly with most current artificial systems. Here the human designer takes on the burden of solving all these problems. The designer must gather a large amount of information about the operating conditions and specification of the system to be designed. Having described the problem, the designer must provide an explicit, practical and comprehensible solution.
>
> This design process usually results in a system which only operates under previously anticipated conditions, and which employs a centralized, hierarchical, non-redundant control system. Such control systems are applauded as transparent and efficient. However, many solutions are unavailable to human designers not because they are poor solutions in terms of performance in the system of interest but merely because they are not comprehensible by humans.

These are already good reasons for trying to use the organizational principles of natural cell clusters for artificial systems. Hoile and Tateson also mentioned already a second (more general) advantage of OOOP: a great power of adaptation.

Adaptive and dynamic

The ability of systems designed with object-oriented ontogenetic programming to adapt to dynamic environments and changing problems is one of the main features that this new paradigm promises to provide. All practically usable approaches to introducing development into automatic systems design (especially evolutionary computation) until now have focused on an extension of the genotype–phenotype mapping. They enrich this mapping by introducing developmental principles from nature or at least more complex mapping functions[6]. But the organizational strengths of ontogenetic processes are only used for developing a "mature" solution before using it. They do not remain in the final developed system. So the solutions produced by these approaches do not have the adaptive qualities of multicellularity. In contrast, the processes providing these qualities are an essential part of the solutions that base

[6]Examples for these approaches are [Banzhaf, 1994; Gruau, 1994; Keller and Banzhaf, 1996; Koza et al., 1996; Eggenberger, 1996; Astor and Adami, 1998; Ferreira, 2001].

on OOOP. Previous developmental approaches do not provide a real analogy to nature because in nature there is no "mature" solution (creature) which does not depend on ontogenetic processes any more. If the varied representation of an evolutionary algorithm is the genotype (the genome) and the executed solution is the phenotype (the living creature) this means that "usage of a solution" in nature denotes the life of a creature. But ontogenetic (and embryonic) development in nature happens within the lifespan of a create and not before the beginning of its being[7]. There is one work that recognizes this difference and introduces many exciting ideas for automatic programming. This work has also had some influence on the choice of name for OOOP[8], even though the used processes are very different[9]. It has been invented by Lee Spector and Kilian Stoffel in 1996 [Spector and Stoffel, 1996] and in my opinion deserves much more attention than it apparently has received. The important similarity between ontogenetic programming and OOOP is that in both cases the ontogenetic development is a constituting property of the produced solution. This attribute is in my opinion the heart of "ontogenetic" programming. Ontogenetic programs have a dynamic execution structure which interacts with the environment and allows growth and adaptation. Lee Spector and Kilian Stoffel describe this important difference to other developmental approaches as follows:

> These sorts of morphological mechanisms ... are simplifications of biological ontogeny ... they operate only prior to runtime; in contrast, biological ontogeny continues throughout the life span of an individual.

About the use of the real ontogeny for computer programs, they write:

> ... generally, developmental control of an arbitrary feature f of an individual is likely to be useful in two cases: 1) when there is an adaptive advantage to having an f value at maturity that differs significantly from the f values of new individuals, or 2) when there is an adaptive advantage to having different f values at different stages of an individual's life. The second case applies to features of computer programs for many domains.

Modular

Object-oriented ontogenetic programming is also a new modularization technique. Even if you do not want to create a system of autonomous interacting units, OOOP might enhance the power of GP in finding good building blocks for reuse in the created program and provide a new way of enabling the evolution of more complex programs. There exist already several modularization techniques in genetic programming that try to reach this goal. Automatically defined functions and automatically defined macros have already been mentioned. Justinian P.

[7] This of course depends on when you determine the beginning of existence of a creature. But in any case, the developmental processes continue during the whole lifespan, which is a big difference to the previously used genotype–phenotype mapping, which only happens before the solution is executed.

[8] The presented approach is called *Ontogenetic Programming*.

[9] This does not mean that it is less interesting. It is in fact extremely interesting because it is theoretically even more powerful than object-oriented ontogenetic programming and even more open. The simple idea is to introduce operators for program self-modification into the GP function set. But the presented approach also has several disadvantages that OOOP does not have. For example it is so strongly abstracted from natural ontogeny that the power of gene regulation networks can not give any hint on the power of this approach. Moreover, it does not model multicellularity, so it is not as well suited to being used for multiagent systems as OOOP is.

Rosca and Dana H. Ballard describe a different technique for discovering reusable subroutines in [Rosca and Ballard, 1994]. All these recent modularization techniques have enhanced the power of GP, but they also have different limitations and problems. In a later paper about an extension of the ontogenetic programming idea [Spector, 2001] Lee Spector writes:

> The more traditional approaches to the evolution of programs with subroutines (automatically defined functions ...) and macros (automatically defined macros ...) require that one specify in advance how many such modules are to be used and how many arguments they will receive. Syntax restrictions are also imposed by the definitions of the modules, complicating the expression of genetic operators and in some cases limiting the possibilities of crossover. A more refined approach using *architecture altering operations* eliminates the need for prespecification of the numbers of modules and arguments, but it does so at the expense of additional complexity [Koza et al., 1999].

OOOP does not have these problems, because the modules do not have to be prespecified (genes can be added to or removed from a genome without any problems, which in fact happens all the time) and the approach is very easy to understand and use if one knows the object-oriented programming concepts. The only "complexity" that the approach implies is that not all code of an individual is evolvable code. You have to use string search operators to find the appropriate positions for varying the program code. The power of using OOOP for modularization has not yet been tested, but as nature successfully uses a similar approach, it is probable that object-oriented ontogenetic programming will also enhance the modularization capabilities of GP. Easy and flexible exchange of evolved modules between individuals is surely provided by the approach. One more step towards the goal of fast emergence of good building blocks is the introduction of a second level of evolution by also computing fitness values for single genes, not only for whole individuals and favouring fitter genes for conjugation. This will be described in more detail in sections 3.2.1 and 3.2.2.

Powerful interactions / communication

The gene regulation mechanism and diffusion of messages provide a basis for very complex interactions and interdependencies. We have seen already in section 1.2.3 that these mechanisms enable the elaborate interactions between cells and between genes in one cell. Torsten Reil shows in [Reil, 1999] that already the gene regulation in one cell can produce extremely complex interactions. Even though the messages in OOOP are very simple (the only information they contain is a message type and the intensity of the message), they suffice for creating an extremely powerful communication. In a multiagent environment many problems can only be solved by heavily communicating units. Also for modularization of a single problem solving program, a powerful communication between the subcomponents is often necessary. As Mitchell A. Potter and Kenneth A. De Jong put it in [Potter and De Jong, 2000]:

> ... many problems can only be decomposed into subcomponents exhibiting complex interdependencies.

But a sophisticated communication usually makes high demands both on development and on (transmission-) hardware. This is not the case for the *OOOP communication paradigm*[10]. You

[10] As described, it bases on the diffusion of messages and on gene regulation. We have seen how this works in sections 1.2.3 and 2.1.2.

do not have to develop a complex communication protocol or a specific information exchange language and you do not need a very strong transmission hardware, because the messages are so simple that they can perhaps even get coded in analog attributes of the transmitted medium (e.g. wavelength and amplitude for immobile[11] units communicating via radio). The presented paradigm therefore is a promising approach to communication in environments with autonomous interacting units or for creating solutions with a modular program architecture.

Evolvable

Potter and De Jong also write in [Potter and De Jong, 2000]:

> To successfully apply evolutionary algorithms to the solution of increasingly complex problems, we must develop effective techniques for evolving solutions in the form of interacting coadapted subcomponents. One of the major difficulties is finding computational extensions to our current evolutionary paradigms that will enable such subcomponents to "emerge" rather than being hand designed.

Evolutionary computation gets most of its inspirations from biology. So we should have a look at what made cooperation evolve in nature. In other words: How has evolution built cooperative organizations out of self-interested components? John Stewart gives in [Stewart, 2000] a general answer to this question:

> A mechanism that ensures individuals capture all the effects of their actions would completely overcome the barrier to the evolution of cooperation. If individuals capture the effects of their actions on others, self-interest would no longer stop an individual from helping another. Helping the other would be as profitable to the individual as helping itself.

In nature, this condition is very often ensured by depriving the individuals of their capability to reproduce individually. Such individuals have a clear evolutionary advantage if they cooperate because that makes the reproducing compound fitter. This makes organelles like Mitochondria, which once seem to have been separate organisms [Margulis, 1981], cooperate in a single cell. It makes the cells in a multicellular creature cooperate. And it makes bees, ants and termites cooperate. John Maynard Smith and Eörs Szathmáry write in [Maynard Smith and Szathmáry, 1995]:

> To the extent that individual ants, bees or termites have lost the capacity to reproduce, they can propagate their genes only by ensuring the success of the colony, just as somatic cells can propagate theirs only by ensuring the success of the organism. Hence, the colony can be expected to have features adapted to ensure its success, and it is reasonable to apply concepts of optimization to it, rather than to the individual ...

This concept is also used for allowing the evolution of cooperation with object-oriented ontogenetic programming. The OOOP units (the objects) can only reproduce together (as they are all based on the same genome which is used for reproduction) and they are not assigned

[11] This could also be used for mobile entities if they move not fast enough to make the Doppler-effect render the used wavelengths inseparable or inidentifiable.

separate fitness values. Their fitness is determined by the fitness of the team. If you consider genes as the units this also holds even though they have separate fitness values in OOOPS and can migrate to other genomes. One reason for this is that their fitness is based on the fitness values of the genomes that contain them. This will be described in more detail in section 3.2. Hassan Masum and Franz Oppacher also saw that gene regulation is a great method for evolving cooperative units[12]. They write in [Masum and Oppacher, 2001]:

> With embryogenetic mechanisms, regulatory networks ... provide a simple distributed coordination method – the same overall program is being run in different ways, and since only one program is evolving, it will implicitly evolve to perform well when instantiated in multiple cooperating cells.

Masum and Oppacher also make another important point here: Even though the resulting problem solution consists of many different units (concerning their actions), the evolutionary computation system only has to handle one single genome. This allows the evolution of heterogeneous teams[13].

Perhaps, the flexibility and adaptivity of gene regulation as a coordination method can even compensate to some extent the ruggedness of the GP fitness landscape by choosing which genes to use. This influence can perhaps be supported by making the gene fitness also depend on how often or whether the gene has been used. These issues are interesting questions for further research.

As we will see in section 3.2, the object-oriented ontogenetic programming system even allows meta-evolution like evolving the type of cell update (e.g. fixed order, random order or random update) or the environment model.

Emergent differentiation

Differentiation of cells can be seen as emergent division of labour for the good of the creature. In terms of computational problem solving, this means that the interactions between cells that are used in OOOP can produce emergent problem decomposition and task assignment. John Maynard Smith and Eörs Szathmáry write in [Maynard Smith and Szathmáry, 1995] about cell differentiation in nature:

> The development of a multicellular organism, with differentiated cells, requires that three problems be solved:
>
> *Gene regulation ...*
>
> *Cell heredity, and the dual inheritance system.* Not only are different genes switched on in different cells, but the states of differentiation are heritable through cell division ...
>
> *Spacial patterns.* Differentiated cells are arranged in space in a specific and repeatable pattern. How does this come about? There are two possibilities, which we will call 'self-organization' and 'external specification'.

[12] But I only read their paper in April 2002, when I had already finished implementing the object-oriented programming system. So they simply had the same idea.

[13] The advantage of OOOP to combine the benefits of homogeneous and heterogeneous teams is discussed in more detail later in this section.

All these preconditions are given in OOOP. As we know, gene regulation is the basis of object-oriented ontogenetic programming. We also know that an important feature of OOOP is the virtual space in which the cells are located and in which they build spacial patterns. Both possibilities for their formation exist in OOOP: self-organization[14] is allowed by a random influence on cell growth. When a cell divides, the newly created cell takes a random free position around the original cell. External specification is given by special message sources, that can be arbitrarily located in the environment. This is also already implemented in OOOPS. Also the second precondition for cell differentiation is realized in OOOP. Because of its object-oriented approach to modelling cells, the internal state of a cell is naturally heritable through cell division by just copying the cell object.

Combines advantages of homogeneous and heterogeneous teams

Sean Luke and Lee Spector try in [Luke and Spector, 1996] to evolve coordination in teams of interacting agents and describe the following two approaches:

> In one commonly used scheme, teams consist of clones of single individuals; these individuals breed in the normal way and are cloned to form teams during fitness evaluation. In contrast, teams could also consist of distinct individuals ...
>
> ... heterogeneous tasks can only be solved (or solved easily or effectively) with multiple individuals each of which uses a distinct specialized algorithm.

With object-oriented ontogenetic programming, this contrast does not exist any more. It combines the benefits of both approaches: It can easily be handled with the usual breeding strategies because as there is only one genome, all the agents are homogeneous concerning variation (i.e. in genotype). But in the evaluated program, the agents (i.e. the objects) differentiate to perform specialized actions which makes them heterogeneous concerning execution (i.e. in phenotype). So the ontogenetic processes used in OOOP allow to combine the easy evolution of homogeneous teams with the problem solving power of heterogeneous teams.

For centralized and distributed problem solving

As we have seen in section 2.1.1, object-oriented ontogenetic programming started as a method for evolving a new sort of modular programs. This new type of modularization was meant to enhance the power of GP in finding good building blocks and it provided a new type of communication between the modules that allowed extremely complex interaction networks and promises good adaptive qualities for the evolved programs. The spacial locations of the objects were only a way to introduce even more possibilities for adaptation and interaction, they did not have any meaning. Therefore it seemed to be sufficient to let the objects take only positions in a regular grid and let them stay there until they die. In this approach, the virtual space is only a tool for making the communication between the modules even more powerful and the programs even more adaptive[15] compared to using only the genes of a single cell as modules. It does not model any real space. And the problem environment is not modeled

[14] By this, Maynard Smith and Szathmáry mean that spacial patterns emerge because the homogeneous distribution of cells is not stable. Already a little random symmetry breaking mechanism like the Brownian motion starts the formation of stable but inhomogeneous patterns.

[15] By introducing growth processes for building patterns of objects in this space.

in this space. The space it located in the problem solving program which is located in the environment. This is totally different for the second approach, which evolved much later but seems to be even more interesting: The ideas of OOOP can be extended to evolve control programs for separate cooperating units which interact in an environment on the basis of the OOOP communication paradigm. In this approach, the environment is modeled in the virtual space and the location of the objects in the space corresponds to their location in the real environment. For modelling mobile robots in the real world, the locations of the objects would for example be extended to take arbitrary values and the objects would be enabled to change their position. Because in this approach the environment is in the same space as the objects, the communication between them is directly influenced by the environment. The intensity of messages at an object's position is computed on the basis of the environment model. There is no need for special message sources that give information from the environment into the object space. Moreover, the problem solution is very different to the one of the centralized approach: It is (usually) not the whole evolved program including the virtual space and the functions for ensuring the diffusion of messages in this space, but the problem solution is simply the genome which controls the actions and communication of the objects in the space. In this approach, the space is no more a tool for communicating but a simulation of the real communication medium. This method for evolving control programs for multiagent teams (which I named "Swarm-Programming") will be described in more detail in section 4.2.

Both approaches to using ontogenetic processes for problem solving have their pros and cons, but they both open up new possibilities for evolving intelligent systems. Both approaches should be investigated much further because they seem to have a lot of power and many common advantages compared to traditional approaches to genetic programming. Most advantages discussed in this section apply both to the centralized and the distributed approach to ontogenetic problem solving.

Close to nature

As you have seen, OOOP is quite close to the natural example of growing cell clusters. This is not only an advantage because the great abilities of cell clusters give a hint about the power of this approach also for solving other problems, but this also means that OOOP could be used as a simulation of natural processes. In section 1.5.1, we saw that there have been already several works on simulating the processes used in OOOP, but nevertheless, this object-oriented approach introduces some new possibilities and is very open for extensions, which makes it a good tool for modelling gene interaction networks. More importantly and in contrast to some other approaches, OOOP also provides a good basis for simulating multicellular interactions and growth processes.

Open and easily extendable

One of the major advantages of OOOP both for simulation and for problem solving is that it is not based on a strict mathematical model like most other simulations but on an object-oriented program which can easily be extended to arbitrary levels of detail or to include any interesting new feature. This openness is the last of the here presented advantages of the object-oriented ontogenetic programming paradigm, but that does not mean that it is the least important. Perhaps, it is even one of the most important features of OOOP, because the effect is that you can make out of the OOOP idea what you like and what suits your special needs

and interests. This is not to say that good OOOP programs can easily be written. Because of the complex interactions, the genetic parameters like genetic code, activation conditions and message production can not be hand-coded to produce a desired behaviour (except for extremely simple examples). Therefore, these parameters are optimized with evolutionary methods in the Object-Oriented Ontogenetic Programming System described in chapter 3. But as this system and the meta structure of the OOOP program (describing for example the diffusion model or the gene regulation mechanism) are written in the traditional way, these parts (that implement the OOOP model and the breeding mechanisms) can easily be adjusted or extended by hand. There is hardly any feature that cannot be added by simply changing or adding some code at easily locatable parts of the system. This openness and the facility of performing changes are well known advantages of object-oriented programming.

There are many examples of extensions that could easily be introduced and could be important for specific simulation or problem solving goals. Here are some suggestions:

- There are no restrictions in dimensionality. It might be interesting to take a look at the differences in interaction and evolvability when using different numbers of dimensions. We do not have any idea how cell clusters in a 10-dimensional space behave. Of course, there is no 10-dimensional space in nature, but this space could nevertheless be used as a tool for communication in modular programs if it gives rise to new and useful levels of interaction.

- In contrast to many other approaches to simulating units that interact on the basis of some sort of diffusible messages (e.g. cellular automata or *amorphous computers* which will be introduced in section 4.1.1), the virtual space used in OOOP is not restricted in any way. Any object in the space can have an arbitrary location which can be extremely far away from the other objects without having any negative influence on execution time. The only precondition is that the used numbers can get large enough to specify the location of the object. This allows for simulating very irregularly placed units in an environment with no known borders.

- In the current implementation, diffusion of messages is instantaneous: If a gene is activated and produces a message, this message can be received at once even by distant cells if its intensity is high enough for reaching them. This is good for modelling fast propagating signals[16] like radio, sound or light. But substances produced by cells in a living creature do not spread as quickly. For modelling cell interactions in natural cell clusters it would be better to provide a slow diffusion mechanism where a message needs some time to reach a distant recipient. This can easily be modeled by making the concentration of a message at a location not only depend on the distance to the sender but also on the time when it was sent. The distance and the diffusion speed determine how far into the history of production of this message the recipient looks. Like we can only see what happened with distant galaxies millions of years ago, the concentration of a message at a distant receiving cell depends on the intensity of production which happened several cell cycles ago. For enabling this, a cell would not only save the current production but a history of the production as far into the past as is needed for the most distant receiver. This possibility is even already implemented in OOOPS but has not yet been used.

[16] This means fast compared to the update frequency of the cells. If you model very big distances, even the here mentioned signals can not be simulated as spreading fast.

- Another extension which is very promising is to use complex templates as keys for the message types instead of simple ID numbers. In nature, the influence of a substance on the activation of a gene depends on a complex template matching mechanism based on the molecular 3D structure of the substance. Hence, introducing this template matching into the activation of object functions in OOOP would make it even more similar to nature. But it could also provide a big advantage for the power of OOOP in problem solving. As discussed by Sean Luke et al. in [Luke et al., 1999], a good template matching mechanism could allow to change a gene's influence on another gene without changing its influence on a third gene. This would for example be the case if the changes happened in a region of the produced message template that is important for matching the second gene's activation template but which is not needed for matching the third gene's activation template[17]. This opens up more possibilities for subtle changes in the interaction dynamics by evolutionary variation.

- For evolving teams of autonomous units with OOOP, we need to model their environment. This environment model can be extended to be arbitrarily complex. At present, it does not even exist. The computation of message concentration solely bases on the distance to the sending object. But it is no problem to introduce a dependency on the modeled environment into the concentration function (called `howMuch()` in the current implementation as described in section 2.2).

- As I have already mentioned, for evolving mobile agents acting in a real environment, the grid-oriented positioning can be extended to mobile objects with arbitrary locations given by floating point numbers.

- Growth processes (cell division and cell death) are another point that is already implemented but could still be extended in different ways.

- A big advantage of the approach is also, that the objects as acting units or agents can be extended to be arbitrarily complex and powerful. They cannot only be simple reactive agents (in terms of multiagent systems) but can become sophisticated *deliberative agents*[18] with their own memory and learning. The only barrier to using great numbers of very complex deliberative units in OOOP is execution time. Because this barrier is a serious restriction, OOOPS has been designed to run in parallel on many computers. This will be discussed in more detail in section 3.2.

- The GP function set can easily be extended depending on the used GP model. The current implementation works with a linear GP approach which means that you only have to add the new function to the set. Perhaps you also have to change some grammar rules that provide inputs for the functions, but that is no real problem. Changing the function set is one of the easiest modifications.

- A last example for extensions that have already been made is the sequencing of cell updates in the GP individual. The cells could always be updated in the same order

[17] Of course, there can be more than one template influencing the activation of a gene, but that does not change the principle explained here.

[18] This is also a term used in multiagent systems research for describing an acting unit whose actions in contrast to a reactive agent do not simply depend on its sensory input but also on an internal status which can be seen as knowledge.

Pitfalls of OOOP
Slow evolution
Slow execution for big cell clusters
Difficult to analyse

Table 2.3: Pitfalls of object-oriented ontogenetic programming.

(This was the first implementation.), or they could be updated in a random order, or they could be randomly chosen for update without ensuring that all cells are updated. The first and the last example have been used for experiments which are described in section 5.1.

2.1.4 Pitfalls

There are three main problems that come with the OOOP approach. Table 2.3 summarizes these problems which will be discussed in the following.

Slow evolution

OOOP has the great advantage that it can easily be handled with artificial evolutionary approaches like genetic programming. But it also has some attributes that make the genetic programming runs for its evolution quite slow:

- Object-oriented programming languages are always high-level languages. This means that they must either be interpreted on execution time, which makes the program run very slow, or they must be compiled before being executed. This produces a faster running program but the compilation step itself takes a lot of time. This step can easily take as much time as the whole run of the program (which is needed for the fitness evaluation)[19] or even longer depending on the problem. Usually, object-oriented languages even compile slowly compared to other high level programming languages. There are many ways of speeding up the compilation step. In OOOPS, always several individuals are compiled together, which already saves a little time. Another way of saving time would be to not compile individuals which were not varied since the last compilation. Moreover, one could split up the individuals and only compile the part that was varied. So you see that there are various possibilities for solving the problem of compilation time or at least reducing its severity.

- The compilation of object-oriented programs usually produces large executable files. Moreover, there are many functions in the OOOP programs which are only needed for communication with the system controlling the evolution or for simulating an environment or for managing the message diffusion. All this additional code also pumps up the executable. And this takes time when running the program for fitness evaluation. This is not a major influence on the evolution time, but it is nevertheless important to watch out when happily programming and adding more and more features. Moreover (which is perhaps even more important), all new program code also extends the compilation time.

[19] If you do not understand everything here, take a look at section 3.1.1. There, we will see in more detail how genetic programming systems work.

- Another minor influence on evolution time are the search operations needed for automatically varying the program code. As not all code is evolvable, the right places for variation have to be found in the string containing the code. The time needed for these operations could also be saved if the evolvable code would be saved in separate files. But the code would have to be rearranged so that still not much more than one file would have to be varied on average, because file operations take a lot of time.
- A second major influence that could make the execution of the program slow down considerably and with that take much time for fitness evaluation is the method used in the OOOP individual for computing the concentration of messages at an object's position. This problem is the heart of the next disadvantage of the object-oriented ontogenetic programming approach in its current form discussed in the following.

It has to be said, that there is another important approach to tackling the problem of slow evolution. Genetic programming is an optimization technique which can easily be executed in parallel. Because of the foreseeable time problem with OOOP, the object-oriented ontogenetic programming system has been designed so that the population of OOOP individuals can be distributed on many computers. It has also been optimized for creating as little net traffic as possible. Additional individuals and computers can always be added during an evolutionary run. There are many more interesting features of this system, but this is not the right place to list them all. They will be described in sections 3.2.1 and 3.2.2.

Of course, you could also wait for faster computers. This will make OOOP more and more interesting. But faster computers cannot solve the next discussed problem.

Slow execution for big cell clusters

The diffusion of messages in OOOP is very different to that used in cellular automata and many other approaches. It is not based on a definite regular grid (sometimes called reaction-diffusion matrix) where the concentrations at each gridpoint are regularly updated and only depend on the concentrations of the adjacent points. OOOP does not need any regular grid because only the concentrations at the cell positions are computed based on the message sources, their distance and possible influences of the environment modeled in the space. This allows the units to be at arbitrary locations in an unlimited space, it allows to model the influence of environmental local factors on diffusion, and it makes several more extensions possible which have been mentioned in the last section. This is much faster than the matrix approach if there are few acting units irregularly placed in a big space. But if the naive approach for determining the concentrations at a specific location is used, which consists of scanning all message sources and adding their influences on the message concentrations at the current location, this approach leads to a quadratic dependence of execution time on the number of cells or (more general) units. For every cell, all cells have to be scanned for computing the influence of their message production on the actual cell. This results in quite slow execution of OOOP programs with a big number of interacting units, especially if environmental influences on the diffusion have also to be taken into account.

Of course this naive approach to computing the concentration is not the last word spoken on this issue. There are surely many possibilities for speeding up the interactions between cells. For example, cells which are out of reach in any case could be skipped. Finding efficient algorithms for determining the concentration of messages in arbitrary locations of a space with many message sources is a very important step for improving the power of OOOP.

Fortunately, current computers are already quite fast, so that even with the naive algorithm OOOPS already produces good results. Moreover, if OOOP is used for evolving control programs for teams of agents in a real environment, the here discussed influences on execution time do not remain in the final solution. In final usage, nature takes care of producing the right concentrations at the right locations. So in this case, the influence of the diffusion algorithm on execution time only matters for how long the evolution of good solutions will take. In contrast, if the solution consists of the whole program and the diffusion and virtual space are tools for enabling the complex interaction between program modules which are part of the solution, the influences of the diffusion algorithm remain in the final solution and are therefore of greater importance.

Difficult to analyse

Object-oriented ontogenetic programming is not based on a simple mathematical model but on an algorithmic approach that can easily be extended. It is difficult to analyse its dynamics on the one hand because there are many influences on them and on the other hand because these influences can easily change. The paradigm has been constructed with the goal to be powerful, open, easy to use and easy to extend. As it is a heuristic approach anyway who primarily has to work well—no matter how, the weight has not been put on making it easily analysable. This is an attribute that OOOP shares with most other directions in programming, especially genetic programming. If object-oriented ontogenetic programming was designed to be easily analysable, that would necessarily limit its other advantages.

Of course, the fact that OOOP is not based on a simple mathematical model does not mean that it makes no sense to analyse its behaviour. A closer look at the dynamics of OOOP programs can help a lot in getting good ideas for improving its performance. Perhaps even because it is not easily analysable, there are a vast number of possible and interesting subjects for theoretical research.

2.2 Realization of a Multicellular Program

A simple multicellular program works as shown in figure 2.1. First, the cells (here named entities because even though they communicate like cells they normally represent something else like for example robots or sensory units or simply program modules) have to be instantiated from the genome and stored into a datastructure that can easily be run through for updating the cells. In the current implementation (which is written in C++ and two-dimensional) this datastructure is a map of maps which contain the cells. As the cells are instantiations of the class Genome, this datastructure is declared as follows:

```
map<float, map<float, Genome> >  cells;
```

The key values of the maps contain the position of the cell. This is not necessarily so. If we would have entities that move (e.g. robots) it would probably be a better idea to not care about the keys and only change the position stored in the entities themselves.

After this initialization, the main function runs through the `cells` datastructure and calls the `update()` function of each cell. This function first computes the total concentration of messages[20] at the cell's position. For that purpose it runs through all cells and adds the

[20] This corresponds to the proteins in nature.

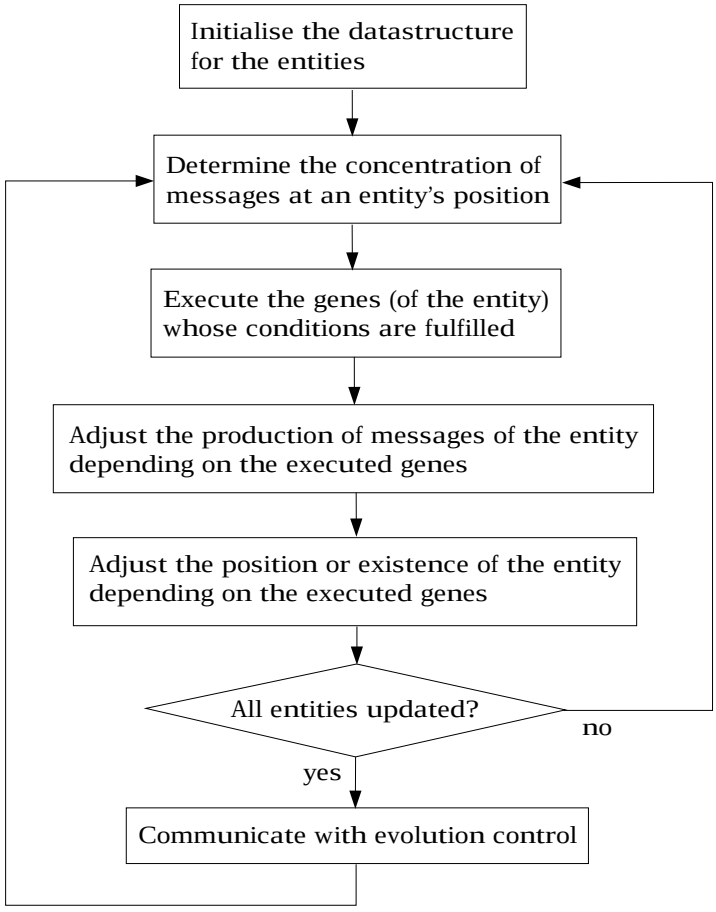

Figure 2.1: Control flow of an individual.

influence of their message productions on the local concentration. This influence is determined on the basis of the distance between the message producer and the updated cell and can include arbitrary variations by a modeled environment in the virtual space between the two cells. The calculation of the influence of a produced message on the local concentration is taken care of by a function called `howMuch()` that has access to the simulated environment.

In the next step, the `update()` function calls all the genes of the cell. The genes themselves test all their conditions for activation before executing their evolved program code and producing a message. This ensures that only genes whose activation conditions are met get executed.

After that, the `update()` function first updates the message production of the current cell depending on which of its genes got executed (and therefore could produce a message). Then, it adjusts the position (in the case of moving entities) or the existence of the entity (in the case of cell death) or even of other entities (in the case of cell division). This could also be done by the genes directly, but as these are important actions for the functioning of the individual, they are presented as separate steps in figure 2.1.

The communication with the part of the system that takes care of the evolution of good individuals[21] can be done after each single update or even directly in the genes, but here it is assumed that most of this communication happens after all the cells have been updated[22].

Major parts of the individual

The OOOP individual (i.e. the multicellular program) consists of the following parts:

- the class `Genome`,
- the `cells` datastructure which holds the cell objects (of type `Genome`),
- the environment model,
- the function `howMuch()` which determines the concentration of a message at a specified position (on the basis of the distance to the sender and the environment model),
- the functions for communication with the evolutionary control modules (for determining the fitness, exchanging commands, etc.),
- the main function that ensures the control flow described in figure 2.1 by calling the corresponding functions.

Major parts of the genome

The class `Genome` consists of the following parts:

- the genes (i.e. member functions with evolvable code),
- the conditions for the execution of genes[23],
- a datastructure containing the actual message production of the cell (e.g. `map<messageID, intensity>`),
- the actual location of the cell (in the virtual space),
- possibly more temporal memory and datastructures that can be used by the evolvable code of the genes,
- the `update()` function which in its simplest form first determines the message concentrations at the current cell by running through all cells and adding their influence on the local concentration (using `howMuch()`), which then calls all the genes of the current cell, and which at last updates the cell's message production depending on which of its genes have really been executed.

[21] This part of the system is described in section 3.2.

[22] It is possible (and as we will see in section 5.1 it could also be desirable at least for some applications) that the cells are chosen randomly for update, so that one cell might have been updated twice while another has not yet been updated at all. In this case, it could make sense to communicate after each single update, but it would also be possible (for not creating too much communication which takes time) to communicate after as many single updates as there are entities in the individual. This does not ensure that all entities have been updated, but it is at least a frequency of communication that is comparable with that of the individual with ordered update as shown in figure 2.1.

[23] These conditions for each gene consist of several message type and concentration pairs that either have to be reached or must not be reached for executing the corresponding gene. So we have enhancing and inhibiting effects of messages.

Major parts of a gene

A gene consists of the following parts:

- test clauses that stop the execution of the gene if its activation conditions are not all met (*execution conditions*),
- the section with the evolvable program code (*evolvable code* or *genetic code*),
- the message returned (i.e. produced) by the gene (type and intensity) (*message production*).

Chapter 3

How to Breed Multicellular Programs

3.1 Genetic Programming

In section 1.5.2, you already read about the major pros and cons of genetic programming as a strategy for breeding problem solutions. In the last sections, we talked about some impacts of genetic programming as a basis of OOOPS. But we have not yet seen how genetic programming really works. Moreover, you will probably ask why we breed the OOOP programs and not try to design them manually. Why not use the techniques that are common in multiagent systems research? Until now, manual computer programming has produced much more powerful and complex programs than automatic evolution. Is it really a good approach to let the OOOP programs emerge on the basis of chance?

Well, for being able to judge about GP, we first have to know how it tries to reach its goal of constructing good programs for a given problem. So we will first have a look at the GP approach and then talk about its use for developing OOOP programs.

3.1.1 How does it work?

In this section, we will run through a little "GP in a nutshell" introduction to genetic programming[1]. As we have seen in section 1.3, natural evolution uses two main principles for developing good solutions: variation and selection. The genotype is varied and if this variation results in advantages for the phenotype, the latter has a bigger probability of being selected for reproduction and not dying early. In natural evolution, the individual is not selected by "somebody" for inducing a specific direction of development. The selection is an implicit result of the competitive and dangerous nature of the environment. The direction of development emerges on this basis and explores the best strategies for coping with the problems given by the environment.

This feature of natural evolution is used when breeding creatures or plants to meet certain human conceptions or preferences. The environment is created or influenced by the breeder so to enforce a specific direction of development. Only individuals with "good" attributes are being given the possibility to reproduce. Evolutionary computation uses the same principles.

[1] For more information refer for example to [Banzhaf et al., 1998].

The environment is given by the human designer and includes a fitness function which determines how good the individual solves a given problem. The resulting fitness value sets the probability for the individual to reproduce or die. In the case of GP: if the individual (i.e. the program) meets well the features defined as "good" in the fitness function (which mainly but not necessarily only[2] means that it produces "good" results when executed), it gets a high fitness value. This in turn causes the evolutionary algorithm to give this program a high probability of survival and reproduction. What does that mean?

"High" is a relative word. If there is nothing to compare with, you cannot denote something as high. The fit individual has a high probability of survival and reproduction because there are other individuals that have a lower probability. Selection needs a set of possibilities to choose from. So in a GP run, not a single individual but a *population* of individuals is evolving. Also in nature, evolution can only work with a population of individuals. For evolution to continue, this population has to ensure that there remain different possibilities for selection to choose from. If all the individuals in a population have the same genome, there cannot be any progress any more.

This said, we can understand what it means for a GP individual to have a high probability of survival and reproduction. It means that it is very probable that the program remains in the population and it has a high probability to multiply by being chosen to replace other less fit individuals. As natural offspring always has a slightly different genome compared to its parents and thus ensures variation in the population, these "copies" of the fit program that replace other died individuals are also being varied. But to speed up the evolutionary search, also most parent individuals are being varied after the selection step. Only some of the best individuals (the elite) might survive without being changed. This ensures that the best solution is not lost again.

In a nutshell, the execution of an evolutionary algorithm like genetic programming can be described as the *evaluation–selection–variation loop* shown in figure 3.1[3]. A pass through this loop is called one *generation*, because new offspring has come into being.

First, an initial (often random) population is created. All the programs in this population are then executed and their fitness evaluated. If a termination criterion[4] is met, the best program is presented as the problem solution. If the breeding has not yet reached its goal, selection decides on the basis of the computed fitness values what to do with the different individuals. There are many different possible selection methods, but all base on chance and on the fitness of the individual[5]. A fitter individual has a greater probability to survive and produce offspring, but that does not mean that an unfit individual cannot get these privileges. It just has a smaller probability to do so.

Most individuals are then varied. There are two main types of variation which are shown in figure 3.2. We met these two mechanisms that can change the genotype already in section 1.3. Mutation makes random changes to the genome (which in GP is the program code) of one individual and *crossover* mixes parts of the genomes of two individuals. Depending on

[2]"Good" could also mean that the resulting program is short, which is not an attribute of the execution (the phenotype) but of the program code (the genotype). The resulting selection pressure towards small programs is called *parsimony pressure* and is a good way to reduce code that does not have any influence on the execution results.

[3]This picture describes the functioning of many evolutionary algorithms (including GP), while some variants (e.g. most of ES) work slightly differently.

[4]This could for example be a specific fitness value that has to be reached by the best individual.

[5]One variant—*tournament selection*—bases only implicitly on the fitness of the individual, because in this case, the winner of a competition between several individuals is chosen for survival and reproduction.

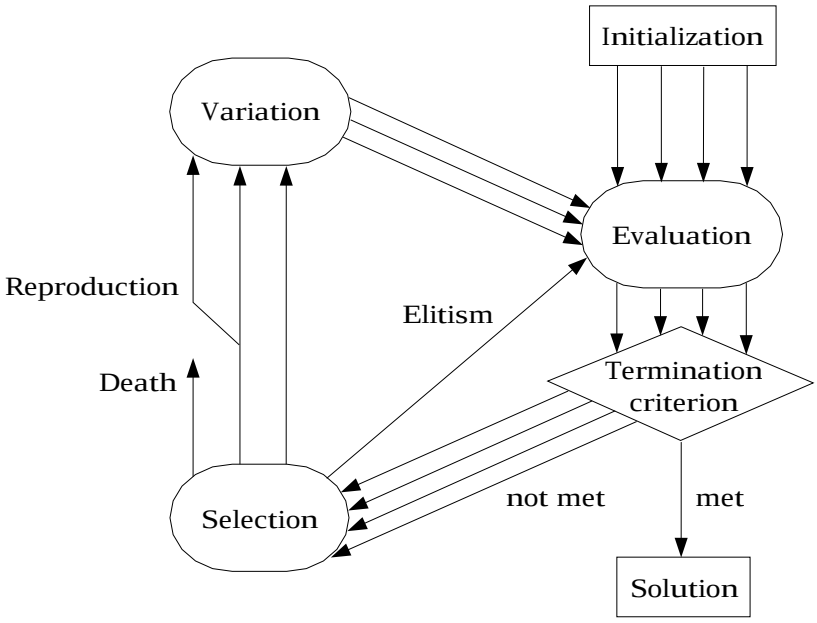

Figure 3.1: Execution of the evolutionary algorithm.

the type of programming language and on the used function set[6], the variation operators have to work very differently for producing valid programs. A simple example of a mutation in GP would be to change one command in the program code.

The varied programs are then run and evaluated again to begin a new generational loop.

[6]Here, it has to be said, that you do not only have to choose a function set for GP, but also a *terminal set*. Terminals are constants or variables that serve as parameters to the functions of the function set. When I speak of the function set, I implicitly also mean the terminal set, because functions without terminals make no sense.

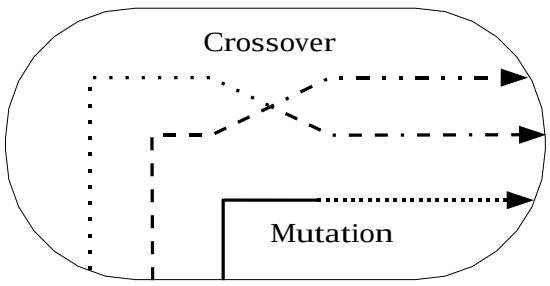

Figure 3.2: Variation.

3.1.2 Why not use another technique of program development?

Now, you might say: "OK, this is a nice simulation of natural evolution. But why should we use this for designing problem solving systems? Are the traditional techniques for program development not good enough?" Yes and no. For many problems the traditional program design processes are the best or even the only feasible way of creating good solutions. It is impossible to evolve a word processor like Starwriter[7] with genetic programming techniques. Firstly, it would be impossible to define a suited fitness function except perhaps user satisfaction. Secondly, the evolution (or more exactly: breeding) would take ages. And thirdly, EC which relies on *interactive evolution*—which uses an evaluation of individuals based on humans rating their quality—produces many problems like, for example, the huge amount of time and human resources needed for evaluation. But interactive programs can only be evolved by interactive evolution or with a simulated user. And simulating a user of a word processor is much harder than writing the word processor itself.

But this is not our goal. We do not want to breed an office suite. We know quite well how to program such systems with the traditional techniques. But there are problems that cannot be solved with these techniques. We already talked about this in section 2.1.3 where I pointed out the advantages of OOOP compared to traditional programming techniques. So let us say we want to use object-oriented ontogenetic programming because of its many advantages but we do not want to breed the OOOP programs but write them by hand. OK, but how do you think the genome should be programmed so that the intended cooperation between the cells emerges? That is the problem: The interactions of genes and cells in OOOP are so complex, that we do not have any idea how to change the genotype (the program code) for getting a desired effect in the phenotype (the running program). Current molecular biology research at present has reached a similar problem: We now know the "program code" of the human species (the genome), but that does not help us a lot because we have no idea what to change for getting a desired effect. Most attributes of human beings (like heritable diseases, which are the most accepted candidates for gene therapy) are based on the interaction of many genes. And even if you change all of them, you never know which side effects this will have on other interactions and resulting attributes. Fortunately, we do not breed humans. But the experiences in breeding other creatures and plants show, that breeding works quite well and is very easy compared to direct gene manipulation.

In nature, molecular approaches to changing attributes of creatures are very interesting because breeding takes ages whereas manual gene modifications produce results already in the next generation. As we know, also genetic programming takes quite a lot of time and manually changing the program code is a much faster method for improvements if one knows what to change. But we do not know what to change until now, and it will probably take very long until we find good methods for manual design and optimization of systems using gene regulation processes. Since artificial evolutionary approaches like genetic programming are quite easy to apply and seem to work much faster than breeding natural creatures, this is the current method of choice for developing systems based on object-oriented ontogenetic programming.

As we will see in section 4.2, OOOP can be extended to be an ideal method for programming swarms of cooperating units. And at the moment, genetic programming is the ideal method for developing such OOOP programs, because there is no other feasible way. Trying different strategies by hand like it has been done in many works on swarm interac-

[7] This is the word processor of the great free office suite StarOffice 5.2.

tions [Stephens and Merx, 1990; Korf, 1992; Mataric, 1995; Hemelrijk, 1997; Bonabeau et al., 1999] is very improbable to lead to good solutions for difficult problems. Moreover, searching a good solution manually takes much human time whereas a genetic programming run only takes much computer time. The human user can do other things while waiting for the computer to find the solution. Unlike human work, computing resources get cheaper and faster all the time. And trying different changes in programs is not a work for which it would be easy to find employees, even if they are well paid. This is also one of the main problems of interactive evolution. If it cannot be organized so that the evaluation by the user is a by-product of other work or leisure, it has not a big chance for success because it will be difficult or extremely expensive to find enough users for doing the evaluations. OOOPS does not have these problems. It can work on a problem without any user interaction because it includes a simulation of the problem environment, and with using genetic programming it finds good solutions on itself. David Evans writes in [Evans, 2000]:

> We are a long way ... from solving the deeper problem of how to program swarms of computing devices.

The combination of object-oriented ontogenetic programming and genetic programming used in OOOPS and even more its extension to "Swarm-Programming" introduced in section 4.2 could be a big step towards reaching that goal.

3.2 The Object-Oriented Ontogenetic Programming System

The Object-Oriented Ontogenetic Programming System is an ensemble of programs which work together for breeding multicellular programs. The underlying techniques are object-oriented ontogenetic programming for the multicellular programs and genetic programming for the breeding. For coping with the problems of OOOP and GP that are described in sections 2.1.4 and 1.5.2, OOOPS introduces some extensions to GP. First, we will look at the architecture of the system and its functioning. In section 3.2.2, we will then discuss the extensions of OOOPS to the basic GP approach shown in section 3.1.1.

3.2.1 The System

It is impossible to describe OOOPS in all detail here, because it is such a big system that its complete description would fill a whole book of its own. This section tries to give at least a good overview of the main features and workings of the Object-Oriented Ontogenetic Programming System. Because of limited space, there are also several unexplained technical terms which are common knowledge for programmers.

As was already mentioned in section 2.1.4, the Object-Oriented Ontogenetic Programming System is designed such that it can run in parallel, distributed on several computers. For allowing this, OOOPS is organized as a client–server architecture. There are three different programs that have been created to build OOOPS:

1. The *evolution manager* (*Ooopse*). This is the server. It runs on a central computer and has access to the *Ooopse database* (*Ooopsed*) which is used for storing data about the current population.

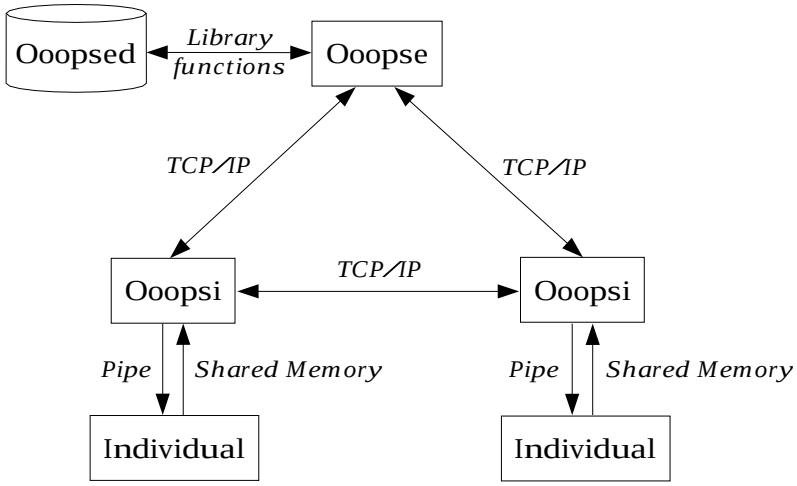

Figure 3.3: Parts of OOOPS and their communication.

2. The *individual manager* (*Ooopsi*). There are several of those programs which act as clients. They can run on different computers and communicate via a network connection and TCP/IP. They communicate as well with the Ooopse as with each other.

3. The *individual*. These are the OOOP programs that are being evolved. One Ooopsi can manage several individuals. The individuals are varied, compiled, run and evaluated by the Ooopsi. For control and information flow between the Ooopsi and the individual, two types of inter-process-communication are used: piping and shared memory.

Figure 3.3 shows these different parts of OOOPS and their interactions. All parts are described in more detail in the following.

Database (Ooopsed)

Ooopsed is a relational database which stores all important information about the evolutionary run. As database management system, PostgreSQL is used because it is free and has a lot of practical features. Plus, there are many developers and a good documentation. The database has the following structure:

< **count** | {int4 indid, int4 geneid, int4 evaluations} >

< **gene** | {int4 geneid, float4 genefitness, int4 numcommands, int4 numrequire, int4 numinhibit} >

< **ind** | {int4 indid, float4 indfitness, text address} >

< **indgene** | {int4 indid, int4 geneid, int4 used} >

< **parent** | {int4 indid, int4 numdescs, text code, text results} >

The `count` table includes counters for the individual ID, the gene ID and the number of evaluations already performed by the system. A new individual ID is needed when an individual dies and is replaced either by the descendant of another individual or by a newly initialized individual. A new gene ID is given to every gene when varied and of course also to newly initialized genes. Individuals only exist once. They have a specified address which consists of the host address, the port number of the Ooopsi and the individual number in the Ooopsi (because there can be more than one individual in an Ooopsi). But the same gene can exist in several individuals because genes are often copied into another individual by conjugation[8]. Therefore, a gene gets a new ID when it is varied to distinguish it from the other copies. Thus, a gene ID clearly denotes one type of gene described in the `gene` table. And the `indgene` table contains all the gene copies with their host individuals and the information how often the gene has been executed during the last evaluation of the individual. In contrast, an individual can keep its ID when it is varied because there are no other individuals with the same ID anyway. The `numcommands`, `numrequire` and `numinhibit` fields of the `gene` table are only used for making statistics about the evolutionary development.

The `parent` table is needed for storing information about offspring. As OOOPS works asynchronously[9] and therefore only can vary or select one individual at a time, offspring cannot be inserted into the population at once when an individual is selected for reproduction. A reproducing individual does not die. But a new individual can only replace a died individual and take its address. So the children have to wait in the database (in the `parent` table) until another individual dies and they can take its place. The `indid` is the ID of the parent individual, `numdescs` contains the number of children it is allowed to produce, `code` is the sourcecode of the parent individual and `results` contains the evaluation results which made Ooopse select this individual for reproduction. When an individual dies, a random child is chosen from the parent table to take its place.

Evolution manager (Ooopse)

Ooopse is the central server program which controls the evolutionary run by initializing new individuals, taking care of the Ooopsed database, deciding on selection and variation and organizing the reproduction. Ooopse is divided into several modules that take on different parts of the work. As the whole system is implemented in object-oriented manner with the C++ programming language, these modules are realized as different objects. Figure 3.4 gives an overview of the architecture of the Ooopse program.

The `server()` function is the central module which is constantly called by the `main()` function. It answers requests coming from Ooopsis by forking the Ooopse program[10] and calling functions of the other modules. The `server()` gets requests from an Ooopsi and

[8] As already mentioned in section 1.3, conjugation replaces crossover in this system. The use of conjugation is necessary because the system works asynchronously which means that variation can only change one individual at a time. In contrast to crossover, conjugation only changes one of the two participating individuals. But viewed from an evolutionary perspective, together with gene deletion it can have the same effects as crossover.

[9] Asynchronous GP approaches are also called *steady-state* and synchronous approaches *generational*. So the asynchrony here means that not all individuals of the population do the steps of the evaluation–selection–variation loop together but every individual follows its own rhythm of running through these steps. When one individual is varied, another is perhaps just being compiled or evaluated. This approach is ideal for distributed execution because in a distributed system, synchrony would be difficult to ensure and would imply individuals waiting for each other and thus wasting time.

[10] This means that the whole Ooopse program is copied and while one copy answers the request, the other can wait already for the next demand. This is a common behaviour of server processes on Unix systems.

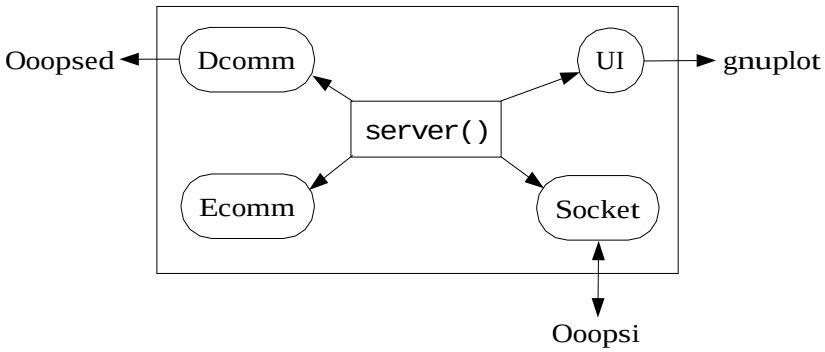

Figure 3.4: The Ooopse program.

answers them through the `Socket`. More precisely, there are two sockets, a `ServerSocket` which listens for incoming requests and a `ClientSocket` with which the Ooopse could request the source code of a specific individual from its hosting Ooopsi. This is not needed for the functioning of the system, but it allows the user at the central computer to get the sourcecode of any individual that is registered in the database.

`Ecomm` contains all the other functions necessary for communicating with the Ooopsis, for example functions for converting information which is to be sent into a serial format and back to the original datastructures.

`UI` is meant to be the object controlling the user interface. As there is nearly no user interaction until now, its only function is to save statistical data to the harddisk and draw graphs by sending the data to gnuplot[11] through a pipe.

`Dcomm` is the most important of the Ooopse modules. It not only contains functions for saving the results of individual evaluations to the database, but it also includes all functions that decide on what to do with each single individual. So this module also decides on selection and variation. It makes sense not to construct separate objects for these functions because they must operate directly on the database. All decisions are based on the information contained in the database and the functions have to return information from the database as for example the address of the donor individual for gene conjugation. But except deleting an individual from the database when it dies, `Dcomm` only decides but does not execute its decisions itself. The decisions are sent to the Ooopsi for execution. Later in this section, we will see more exactly how the different parts of OOOPS work together by looking at the control flow of a typical evolutionary run.

Individual manager (Ooopsi)

The Ooopsi is a client program that contains the sourcecode of some individuals of the population and takes care of everything concerning these individuals. It compiles, runs and evaluates one individual after the other, sends the results to the Ooopse and then executes the commands (e.g. concerning variation) received as an answer. Thereafter, it compiles, runs and evaluates again. The main parts of the Ooopsi program are shown in figure 3.5.

[11] This is a good free graph drawing program which can be found in all Linux distributions and is very widely used.

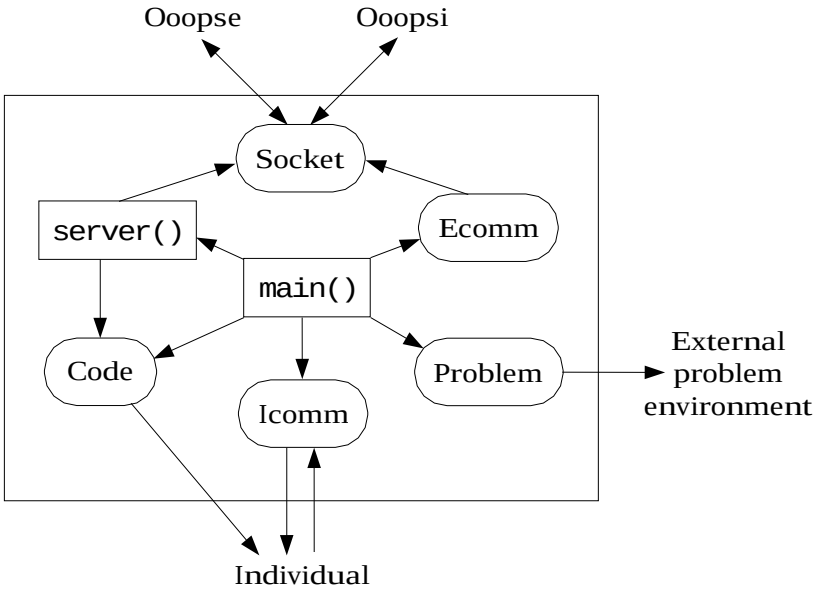

Figure 3.5: The Ooopsi program.

In the Ooopsi, the `main()` function not only calls the `server()` function, but also manages all the other activities. The `server()` function here is only used for answering requests for individuals or genes that are managed by the Ooopsi. These requests can either come from another Ooopsi (which is the standard for gene conjugation) or from the Ooopse. As in the Ooopse, the `server()` function uses the `Socket` object for communication via TCP/IP. It also has access to the `Code` module where the sourcecodes are stored.

The `Code` module not only stores the sourcecode of the individuals but it also compiles and varies them based on the instructions received from the Ooopse. It can probably be seen as the most important part of the Ooopsi. It takes care of everything related to the program code of the individuals. Especially complex and important are the different possibilities for variation. They are described separately later in this section.

The `Icomm` module handles the communication with the individual. So in contrast to the `Code` object which manages the individual code including the compilation, the `Icomm` object is in charge of the individual executable. It starts and stops the runs of the individual, sends other instructions (which are mostly problem specific) through a pipe and reads information from the individual through shared memory. This information can for example be the current state of the individual for visualization of its dynamics in a user interface, it can be reactions of the individual which are fed back into the external problem environment managed by the `Problem` module, or it can be outputs of the individual that are used for evaluating its fitness.

The `Problem` module is only used for some types of problems. If OOOPS is used for the evolution of distributed intelligence where the problem solution is only the genome of the units and the problem environment is modelled in the individual, the `Problem` module is not used as there is no external problem environment. But if OOOP is used as a modularization technique and the problem solution is the whole individual, the problem environment is located outside of the individual. In this case, the problem environment will usually be read by the `main()`

function (using the `Problem` object) which will then feed the environmental inputs into the individual through the `Icomm` object. These environmental inputs can for example be fed into the individual by varying the message production of special *external message sources* that can be arbitrarily located in the cell space[12]. Like this, usual GP training- and testcases can be fed into the individual[13].

The `Ecomm` module does in the case of the Ooopsi not only serve as function package for conversion of message formats used for the communication between the Ooopse and the Ooopsis. Except for the `server()` function which directly uses the `Socket` object[14], the Ooopsi does not directly access the `Socket` but communicates through the `Ecomm` module. So this module handles the whole communication with the other Ooopsis and the Ooopse except answering gene or individual requests.

Individual

The individual has already been described in section 2.2.

Selection

Selection in OOOPS is carried out by the Ooopse[15]. First, the individual is ranked by its fitness[16]. Depending on this rank, the individual can be selected to die, in which case it is neither a candidate for reproduction nor (of course) for variation. There is a parameter `mortalPercentage` that defines which percentage of the population has a death probability greater than zero. The individuals that appertain to the mortal part of the population are those that have the lower fitness ranks. The individual with the lowest fitness rank has a death probability of almost 1[17] and this probability decreases linearly with increasing fitness rank until it reaches 0 at the first individual that belongs to the immortal part of the population.

If the individual does not die, the `Dcomm.select()` function decides whether the individual is to be varied itself or if it is part of the elite which continues unvaried into the next generation. There is a parameter `elitistNum` which defines how many individuals belong to the elite. An individual remains unvaried in the population if it ranks among the `elitistNum` fittest individuals.

At last, the `Dcomm.select()` function decides whether the individual is allowed to produce a child. The individual's reproduction probability is computed analogous to its death probability but with the difference that the probability is 1 for the fittest individual and it decreases linearly with decreasing fitness rank until the first infertile individual is reached. As every child can only replace a dead individual, for not producing too many children the average reproduction probability should be about the same as the average death probability. Therefore, `numFertiles` is set equal to `numMortals`. As described earlier in this section,

[12] These sources correspond to external substances getting into a natural cell cluster and as such being one basis for symmetry breaking in ontogeny and for interaction with the environment of the multicellular creature.

[13] Of course this can also be done with the usual method of providing them as arguments when starting the individual. But the possibility of using the pipe (or possibly even the shared memory) during execution allows the individual to interact with dynamically changing problem environments.

[14] Or more exactly: the `ServerSocket` object.

[15] More exactly by its `Dcomm.select()` function.

[16] The used selection method is called *ranking selection* because as we will see, the selection only depends on the fitness rank instead of the absolute fitness value.

[17] $deathProb = 1 - \frac{fitnessRank}{numMortals}$ with $fitnessRank = 1$ and $numMortals$ as total number of mortal individuals in the population.

children are not produced directly but noted in the database until they can replace a died individual. Because the parent will probably not stay unvaried until the child is used, the code of the parent is transferred and saved to the database when it is selected for reproduction. This not only ensures that the child will not be produced from a parent that has already lost its good qualities (or has even died) but it also means that there are some copies of the fittest individuals in the database most of the time. The user can take a look at them at every stage of the evolutionary run without requesting separate transmissions[18].

Variation

Variation in OOOPS is handled both by the Ooopse and by the Ooopsis. The Ooopse function `Dcomm.vary()` decides on which type of variation is to be executed on the individual and who is the donor for gene or code conjugations. These decisions are then sent to the Ooopsi which follows the instructions by deciding on how exactly to execute the variation types, requesting the needed code parts from the donor and executing the variations. There are three main types of variation (gene conjugation, code conjugation and mutation) and several subtypes (gene deletion, gene insertion, code deletion, code insertion and different types of mutation).

Gene conjugation is an exchange of complete genes between individuals. It is divided into the two subtypes gene deletion and gene insertion[19]. As the name implies, *gene deletion* removes genes from the individual genome. On the basis of chance and probabilities that are adjustable by parameters, `Dcomm.vary()` decides if gene deletion is to be applied and if yes how many and which genes are to be taken away. Analogously, it decides on *gene insertion*[20] with additionally deciding about the donor individual for the inserted genes.

Code conjugation exchanges parts of the evolvable code of two distinct genes. If the donor gene is part of an individual in another Ooopsi, this gene also has to be requested for transfer from this Ooopsi by the host Ooopsi of the varied individual. Like gene conjugation, code conjugation can be divided into two subtypes *code deletion* and *code insertion*. The same reasons for this apply as for the two subtypes of gene conjugation. Code conjugation most closely resembles classical GP crossover, because it can choose arbitrary parts of the code for the mixture. In contrast, gene conjugation differs from classic GP crossover in that the boundaries of the exchanged parts (which correspond to the crossover points) are given by the division of the program into separate genes. These boundaries are not a product of chance in the crossover process, but their content evolves[21]. This is a reason why the division into genes realized in object-oriented ontogenetic programming is a promising technique for finding good

[18] If an individual ID is already in the `parent` table, the new parent code replaces the old and the value in the field `numdescs` is increased. Though this means that children are not necessarily produced of the individual code that were originally selected for producing them, this is no problem because the new code also has a high fitness value. Else, it would not have been selected for reproduction again.

[19] Conjugation in nature usually does not include deletion, but it makes more sense in this context to see gene deletion as a subtype of gene conjugation and not as a form of mutation, because it uses nearly the same mechanisms as gene insertion and is necessary for showing that gene conjugation can have the same evolutionary effects as crossover. If two individuals exchange genes by gene insertion (so one is the donor for the second and the second is the donor for the first) and the copied genes are deleted in the donor individuals by gene deletion, the effect is the same as if the two genes would have crossed over. This has already been mentioned several times, but it is very important because OOOPS is one of the very few (In fact, I don't know any other.) GP systems that use conjugation instead of crossover.

[20] In this case, genes are copied from a donor individual into the genome of the varied individual.

[21] The advantages of this difference will be discussed in more detail in section 3.2.2.

building blocks that can be combined by gene conjugation into a better overall program[22].

In the first experiments with OOOPS, code conjugation was not yet used. Perhaps, it is not even needed at all[23]. One could argue with Kevin J. Lang and Peter J. Angeline [Lang, 1995; Angeline, 1997] that the classical GP crossover is not more than a macromutation operator that does not help in finding and reusing good building blocks but only makes big changes to the code and as such could be an important part of GP in providing big jumps through the search space. But in OOOPS, classical crossover (resp. the corresponding code conjugation) or another form of more explicit macromutation is perhaps not needed because of two reasons:

1. If there are no children waiting in the `parents` table of Ooopsed when an individual dies (which periodically seems to happen), a new random individual is being initialized for replacing the died individual. These new individuals provide totally new starting points for the evolutionary search and as such serve the same purpose as a macromutation operator.

2. As Hassan Masum and Franz Oppacher claim in [Masum and Oppacher, 2001], the use of gene regulation mechanisms for developing the phenotype from the genotype has the effect that small changes in genotype can lead to big but still viable changes in phenotype. This permits macromutations on the phenotypic level without the deleterious effects that often result from macromutations on the genotypic level. As such, this mechanism also provides some sort of (possibly even better) macromutation without the use of crossover.

Mutation in OOOPS is a random variation of the evolvable parts of a single gene. As there are several evolvable sections in a gene, there are also several different types of mutation. The Ooopse only decides which gene is to be mutated and the Ooopsi then chooses the type and if necessary the exact place of mutation and changes the individual code accordingly. First, there is the evolvable code of the gene. A mutation of the evolvable code in the current implementation of OOOPS is simply a deletion of a random single command or the insertion of a single command at a random position. Of course, only commands which are taken from the function set can be inserted. The function set is created depending on the problem that is to be solved. But until now, only a function set suited for linear programs was used because this makes the implementation of code conjugation and mutation much easier compared to the use of tree structures which are common for classical GP[24].

But the OOOP paradigm implies some more variable parameters of the genes that do not exist in classical programming but are central determinants of the behaviour of object-oriented ontogenetic programs. These are the message production of the gene (which consists of the message type and the intensity of production) and its execution conditions. In OOOPS, the execution conditions are divided into *requirements* and *inhibitors* (which both also consist of

[22] As mentioned in section 2.1.1, this advantage for the application of the schema theorem for GP was one of the roots for the creation of the OOOP approach.

[23] It will be interesting to compare the performance with and without code conjugation (for runs with a powerful function set). As we will also see in the following sections, there is a vast number of issues that can and will hopefully be investigated in future experiments with OOOPS.

[24] Moreover, the special execution structure of OOOP can produce all sorts of loops, recursions and dependencies even with linear programming inside the genes, provided a good choice of the function set and temporal variables inside the objects.

a message type and an intensity). Requirements define a concentration of a specific message that must be reached at the cell's position for executing the genetic code. Inhibitors in contrast to this define the concentration of a message that must not be reached if the gene is to be executed. All these parameters can be mutated by exchanging either the type or the intensity by a random new value (within predefined limits)[25].

In OOOPS, all types and possibilities of variation have their own probabilities and can all be applied to the same individual if all happen to be chosen. This also enables a sort of macromutations, but the single probabilities should normally be chosen so that in average not much more than one mutation type at a time is applied.

Control flow of a typical evolutionary run

A typical evolutionary run with OOOPS proceeds as follows:

1. First, the Ooopse server is started. It signals that it is ready to answer requests. Then, you can start as many Ooopsis as you want. The number of Ooopsis and the number of individuals managed by one Ooopsi determine the population size. As you can start a new Ooopsi at every stage of the evolutionary run and every Ooopsi can theoretically be set to manage a different number of individuals, the population size is not necessarily static.

2. When started, an Ooopsi first contacts the Ooopse for getting an initial set of individuals. These are created by the Ooopse on the basis of an individual template and a random but limited number of initial genes that can contain random values[26] for the variable parts.

3. Then, the main evolutionary loop is entered in the Ooopsi. For doing the first evaluation, it has to compile and start the individuals. For that purpose, the sourcecode of all individuals is put together so that it can be compiled as one program but the different individuals can still be separately executed. This code is then written to a temporary folder on harddisk and compiled by running the gnu C++ compiler on it.

4. Next, the first individual is started, and depending on how the individuals are to be evaluated, the communication between the individual and the Ooopsi begins. When the Ooopsi has gathered enough information for rating the fitness of the individual, the latter is stopped and evaluated. The results are then sent to the Ooopse.

5. When the Ooopse receives such a results message containing the individual ID, the individual fitness, the address, the gene IDs and some more statistical information about the genes, it first saves the results to the database. By doing this, the Ooopse puts the first individual into the population. An individual is only part of the population, if it is registered in the database. And it can only be registered in the database by sending evaluation results to the Ooopse. This ensures that there are no individuals without any known fitness in the population. If the individual is already registered in the database, its entries are updated with every results message.

[25] Of course, it would also be possible to mutate the intensity by increasing or decreasing it by a little amount. This would give more control over the size of the changing step. But as this control only holds for the genotype and not for the phenotype (see section 1.5.2 on GP and Masum & Oppacher's claim earlier in this section), this would probably not lead to a great improvement.

[26] As always between the limits of certain user-defined parameters.

6. The `Dcomm.saveResults()` function of the Ooopse not only saves the information contained in the message to the corresponding database fields. It also computes and updates the fitnesses of the genes contained in the individual. Now, we have a *credit assignment problem*: How do we know what a specific gene has contributed to the fitness of the individual? David E. Moriarty and Risto Miikkulainen have solved a similar problem of determining the fitness of single identifiable neurons in artificial neural networks in [Moriarty and Miikkulainen, 1997] by computing their fitness as the average fitness of the five best networks they participate in. Analogously, the fitness of a gene in OOOPS is computed as the average fitness of all individuals that contain and use this gene.

7. As a next step, the Ooopse saves the new information to data files readable by the graph drawing program gnuplot and tells this program to redraw the graph for visualizing the development of best and average fitness of the individuals in the population and possibly other statistical data.

8. After that, the Ooopse starts the `Dcomm.select()` function on the individual. As we have discussed earlier in this section, this function decides on what is to be done with the individual.

9. If the `Dcomm.select()` function has decided to kill the individual, this is then deleted from the database. As children are always varied copies of their parents and a died individual will be replaced by some child waiting in the database, the deletion of an individual is always followed by deciding about what types of variations to apply to the replacing individual.

10. Also in the case of a surviving individual which has been selected to get varied, the `Dcomm.vary()` function is called for determining the type of variation and the donor individual as described earlier in this section.

11. If the individual has been selected to produce offspring, the Ooopse sends a request for the individual's sourcecode to the Ooopsi for saving it to the database. With this information, the `parent` table is updated to add the new offspring to the children waiting to be born.

12. At last, the instructions about what to do with the individual are sent back to the Ooopsi which is still waiting for an answer to its message with the evaluation results.

13. The instructions from the Ooopse are then executed by the Ooopsi. If the individual has been selected to die, the Ooopsi sends a request to the Ooopse for transferring a parent sourcecode from the database and replaces the old individual with this code. Finally, it executes the variations on the individual code as described earlier in this section.

14. Now, the run continues at step 4 with starting and evaluating the next individual in the Ooopsi.

15. This continues until the last individual is varied. Then, the run continues at step 3 by compiling the varied set of individuals.

16. The evolutionary run is stopped either by the user at any stage of the development or by the Ooopse when new results added to the database make that the population meets a prespecified termination criterion.

Innovations of OOOPS for GP
Introduces genes (used as building blocks)
A new modularization technique
Combines steady-state with global selection methods
Uses conjugation instead of crossover
New reproduction management
Dynamic population size
Multilevel evolution
Cooperative coevolution of genes
Metaevolution possible
Homology possible

Table 3.1: Innovations for genetic programming.

3.2.2 Innovations for Genetic Programming

OOOPS is a big system with many features of which some have been described in the last section. Now we will take a closer look at some features of OOOPS (or OOOP) that are new to genetic programming or very rarely realized until now. Table 3.1 gives an overview of the most important innovations that will be discussed in the following.

Introduces genes (used as building blocks)

Genetic programming did get its name from genetics (because of the operators), but until now it did usually not include genes. As we know, genes are the units in the genotype that describe a separate part of the functioning of the cell (a protein). In crossover, genes are normally not split. We also know that genes can be activated or inactivated. Separate parts of the program that have these genetic attributes are introduced by combining genetic programming with OOOP. We already discussed the advantages of using gene regulation in section 2.1.3. But separate genes might also have another advantage for GP that was already mentioned in section 2.1.1: genes are a new approach to modularization of programs.

A new modularization technique

Genes as separate blocks of code that can evolve through mutation and are recombined into different individuals by gene conjugation are a new technique for evolving reusable program modules. One might argue that only by having the ability to migrate into other individuals and influencing their fitness, which in turn influences the spread of the individual (and with that of the gene) in the population, the gene will evolve towards being a good reusable building block. As genes can spread independently in the population through gene conjugation, it is plausible that not only good individuals but also good genes have a higher probability of survival than bad genes. This implicit genetic evaluation and selection is supported in OOOPS by adding an explicit gene fitness and favouring fitter genes for gene insertion. This has already been described in section 3.2.1 and we will also come back to it later in this section.

In genetic programming, *introns*[27] (or substitutions like *explicitly defined introns* [Nordin et al., 1996]) have been used as emergent borders for good building blocks. They might also

[27] That is: blocks of unused or effectless program code.

play such a role in nature. But nature additionally has other (more important) mechanisms for defining the borders of a gene. Using introns or their substitutions as emergent borders takes much computing resources during execution of the program or for extracting the intron code before execution or for the genetic operators (in the case of explicitly defined introns).

By using explicit separate genes like in OOOP in which the code evolves, we have predefined borders but emergent content. This has many advantages. First, the crossover operator is very simple, because it only has to mix the genes. It does not have to take care of anything else, neither of producing valid programs nor of computing the probabilities for different crossover points. Second, this simplicity of the crossover operator is not only easy to program but also saves computational resources. Third, this allows to very easily identify and transfer the building blocks between several computers for distributed evolution. Fourth, it enables the use of gene regulation whose pros have been extensively discussed in section 2.1.3. I believe that this technique is also more effective in finding good building blocks than previous techniques (among other things because it enables to apply multilevel evolution as discussed later in this section). This remains to be proved.

Compared to other modularization techniques like automatically defined functions and automatically defined macros, the use of genes in OOOPS does not require to predefine static parameters like for example the number of genes. We have already talked about these advantages of OOOP in section 2.1.3.

Combines steady-state with global selection methods

As we have seen in section 3.2.1, the object-oriented ontogenetic programming system is what I call an *asynchronous GP* system. This means that the GP individuals reach the different states of the evaluation–selection–variation loop not at the same time. Their life-cycles are chronologically independent of each other. The term "steady-state" refers to the fact that in an asynchronous GP system the biggest part of the population usually is not also just being varied when one individual changes.

Until now, steady-state approaches are intimately tied to *tournament selection*, because this is the only known selection method that does not compare the evaluation results of all individuals in the population. With tournament selection, the individuals taking part in the tournament can be selected independently of the rest of the population. Tournament selection is a good selection method. It has many advantages on the other methods: A central unit for selection is not needed, there is no need for a global fitness function that produces absolute fitness values[28], the evaluation and selection takes less computational resources, this selection method is closer to the natural example than the others, etc. But tournament selection also has some disadvantages:

- Tournament selection is not truly an asynchronous method. You always have to synchronize several individuals to build a tournament. One might think: "This is not really a problem. You can just always compare the next n individuals that are ready for evaluation if n is the chosen tournament size." But this is not a good solution. It not only requires a central unit to register individuals that are ready for evaluation (which counteracts the advantage of no need for a central selection instance), but it also produces another much more severe problem: If all individuals would run through the evaluation–selection–variation loop at approximately the same speed (which is quite

[28] Instead, a few individuals—the participants in a tournament—are just evaluated relative to each other.

probable to be the case if the individuals are distributed on comparable and not otherwise used computers or even on the same computer), the tournaments would always consist of the same individuals or their descendants. But in order to compensate for the local quality of evaluation in tournament selection, the tournament composition must be a good random mixture of individuals from the population which should strive to be as representative of the whole population as possible and which should vary as much as possible from one evaluation to the next. If the competitors do not mix enough between different tournaments in the course of the evolutionary run, tournament selection will enforce distinct societies of individuals that evolve separately. This is not the goal. But how can you ensure a good random mixture of participants in all tournaments? For choosing random individuals from the population for each tournament (which is a simple, often used and theoretically good answer) you not only need a central unit which has an overview of the population, but you also destroy the advantages of asynchronous GP and waste a lot of time: The individuals always have to wait for each other to get ready for the competition[29].

- The organization of tournaments in a system with distributed competitors poses some problems. How should distributed individuals be compared? There are two obvious solutions: 1. Use a global fitness function for separately evaluating the individuals and only compare the results via net communication. 2. Let the individuals come together for making a real tournament. The first solution destroys the most important advantage of tournament selection that it does not require a global fitness function, whereas the second solution produces a lot of net traffic and leaves the question unanswered about where to arrange the tournament.

- The parameters influencing the functioning and results of tournament selection are not very fine tuneable. Tournament selection usually works as follows: First, $n = \textit{tournament size}$ individuals are being selected to compete. Then the competition takes place which determines relative fitness values for the individuals. The best $d \leq \frac{n}{2}$ individuals are then allowed to produce children which replace the worst d competitors. This means that in every tournament, d individuals die. With this parameter d you can adjust the *selection pressure*, which is a very important measure in evolutionary algorithms. If the selection pressure is chosen too high, the genetic variability in the population can quickly get lost and the algorithm will easily get stuck in local fitness optima[30] because bad solutions have a very low survival and reproduction probability. If the selection pressure is chosen too low, the evolution advances very slowly because bad solutions have not a much lower survival and reproduction probability than good solutions which makes it difficult for good solutions to spread in the population. The bigger the tournament size, the finer you can adjust the selection pressure[31]. But the bigger the tournament size, the more tournament selection resembles other selection methods and loses its advantages.

- Tournament selection produces a lot of net traffic in distributed systems. If the com-

[29] This does not only waste individual time but also computational time if it is used with distributed evolution where one individual can not easily benefit from the resources unused by another individual.

[30] You remember the fitness landscape mentioned in section 1.5.2? The goal is to find points in this landscape that are not only locally but also globally (approximately) optimal.

[31] With a tournament size of 2, you can not adjust it at all.

petitors come together for the tournament, nearly all their sourcecodes (which can be very long) have to be transferred through the net. This happens for each tournament and produces a huge individual-traffic. But even if a distributed tournament is arranged which requires only little net communication, tournament selection produces more individual-traffic than other selection methods. This is because it usually includes a relatively high death rate[32] and every died individual requires one individual-transfer for the replacing code[33]. Moreover, tournament selection includes a relatively high death probability for good individuals. If a tournament for example only consists of relatively good individuals which would normally not be selected to die, still the worse d of them are going to die. When evolving a big population on fast computers or on the internet, the net traffic could become a bottleneck and therefore should be kept at the minimum. Tournament selection is not the right method for reaching this goal.

As we have seen, tournament selection seems to be a rather bad selection method for distributed evolutionary systems. Therefore, it is not used for OOOPS. Instead, I developed a new organization of the GP system, that allows to use all non-tournament selection methods (like fitness-proportional selection, truncation selection or ranking selection) with truly asynchronous execution of the GP run[34]. This organization required several other innovations for GP of which the two most important are described in the following.

Uses conjugation instead of crossover

As you already know from other sections (e.g. 1.3 and 3.2.1), OOOPS uses conjugation instead of crossover. This decision was made because conjugation only changes one of the two participating individuals which is ideal for asynchronous genetic programming. In the asynchronous approach we always only have one individual which is guaranteed to be at a specific state of the evaluation–selection–variation loop. When this individual is to be varied, with conjugation it can be chosen to get parts of any other registered individual regardless of the state of the last. But if crossover was to be used, the first individual would have to wait for the second to also reach the state of variation so that both would be ready for changes.

This is not the first time that conjugation is proposed for use with evolutionary algorithms. Peter W. H. Smith suggested a form of conjugation in [Smith, 1996] for GAs and GP which consists of copying a part of the donor genome to the identical position in the recipient genome replacing the original recipient code at that position. This conjugation only partly resembles the natural example, because in nature, the transferred genes do usually not replace the original genes but are simply added to the recipient genome.

The OOOPS conjugation follows the natural example in this respect by providing gene insertion without deleting the original code. But it also provides gene deletion for being a good replacement for crossover (we discussed this earlier) and for in average not blowing up the code length. The division of conjugation into insertion and deletion enables a great variability in code length with still (by using the same probabilities) not predetermining any direction of length development.

[32] Every tournament produces exactly d died individuals. A tournament size of 2 enforces that every second individual in the population dies in each generation.

[33] So a tournament size of two would require to transfer half of the population in each generation. This is unacceptable.

[34] Its workings have been described in section 3.2.1.

New reproduction management

The reproduction management of OOOPS has already been explained in section 3.2.1. An individual which is selected to produce offspring is registered in the `parent` table of the database. When an individual is selected to die, a random individual from this database table is used as parent of the replacing child and deleted from the table. If there is no entry in the table, a newly initialized individual is used as parent. This organization of the reproduction process is not only perfectly adapted to asynchronous distributed evolution (with a central unit), it also as a by-product has the advantage of refreshing the diversity of the population and trying new starting points for the exploration of the search space from time to time by inserting new random initial individual into the population. This only happens if there are no parents in the database, but as the experiments until now suggest, this is periodically the case. It is an interesting question for further research why this seems not to happen with a random frequency but in periodically changing phases of high and low offspring production. Could it perhaps be compared with periodic changes in natural population developments?

Dynamic population size

In section 3.2.1, we also already mentioned that new individuals can always be added to the population by starting new Ooopsis. So the population size can be varied in the course of the evolutionary run if desired. This is a good and important quality for distributing the evolutionary runs on the internet, because new users can always join in and start an Ooopsi on their computer. Yet, distribution of OOOPS on the internet would require some changes in the communication because Ooopsis then regularly change their address and are often unreachable. Consequently, it would make more sense to not let the Ooopsis communicate directly with each other but instead only let them talk to the Ooopse. This means that all individual sourcecodes should also exist in the Ooopsed database[35]. Moreover, some security features would have to be added.

Multilevel evolution

As mentioned earlier, OOOPS not only determines fitness values for the individuals but also for the genes. Not only the individuals are selected, but also the genes underly a separate selection process which depends on their fitness. Genes are selected for being the parent of inserted genes in gene conjugation depending on their fitness rank. Of course, they are also varied (by mutation and possibly code conjugation). So not only individuals are evolving in the evaluation–selection–variation loop. Also their subunits (genes) follow these steps. This means that we have two levels of evolution in OOOPS. Such multilevel evolution is not unnatural. In fact, we can view many systems on very different levels in nature as evolutionary systems. Elliott Sober writes in [Sober, 1992] about this issue:

> ... If an object and its offspring resemble each other, the system will evolve, with fitter characteristics increasing in frequency and less-fit traits declining.
>
> This abstract skeleton leaves open what the objects are that participate in a selection process. Darwin thought of them as organisms within a single population.

[35] This centralization of the communication would also enhance the behaviour of the system in case of the extinction of an Ooopsi.

> Group selectionists have thought of the objects as groups or species or communities. The objects may also be gametes or strands of DNA, as in the phenomena of meiotic drive and junk DNA.
>
> Outside the biological hierarchy, it is quite possible that cultural objects should change in frequency because they display heritable variation in fitness. If some ideas are more contagious than others, they may spread through the population of thinkers. Evolutionary models of science exploit this idea. Another example is the economic theory of the firm; this describes businesses as prospering or going bankrupt according to their efficiency.

Cooperative coevolution of genes

Genes in OOOPS have their own selection and they also reproduce. But they do not reproduce independently of their hosts because their fitness strictly depends on the individual fitnesses[36]. As has been mentioned in section 2.1.3, this is an important precondition for cooperation between the genes to emerge. They must capture all effects that their behaviour has on others. So the genes in OOOPS evolve separately, but they evolve to cooperate. This type of evolution is called *cooperative coevolution*.

Metaevolution possible

All environmental influences on the individual are defined in the individual itself, either with the external message sources that feed the externally modelled environment into the individual or with the internal environment model in the distributed problem solving approach of OOOP[37]. Also the whole functioning of ontogeny with the virtual space and the organization of diffusion and gene regulation is implemented in the individual. These parts of the individual are normally not evolvable because some are not part of the genome and therefore not really constitute the genotype of the individual (which is the only thing usually varied in artificial evolution) and they all do not correspond to attributes of natural genomes that change during the evolution of natural species[38]. But because they are part of the OOOP individual, they can also be evolved with OOOPS. A very simple example would be that you initialize half of the population with individuals that update its cells in a prespecified and static order and the other half of the population with individuals that randomly choose the next cell that is to be updated. This difference does not exist in nature, but it can make a great difference in the behaviour of the object-oriented ontogenetic program. In the course of the evolutionary run, the individuals whose update method produces better results will supersede the others. This example is a very simple form of evolution, because the search space only contains two points. But it shows that OOOPS also allows to evolve parameters of the multicellular model itself and not only the genome of the modeled cell cluster. Such an evolution of the model can be viewed as a kind of *metaevolution*.

[36] As I have described in section 3.2.1, the fitness of a gene is computed as the average fitness of all individuals in which it is contained and has been executed during the last evaluation.

[37] These two different approaches to use OOOP have been discussed in section 2.1.3.

[38] Of course, the gene regulation mechanism has evolved itself, too. But as it can be found in all living creatures and always works quite similarly, we can think of it as not changing.

Homology possible

Nature uses some tricks to reduce the probability that crossover will have disastrous effects on the individual by destroying or removing important genes. Two important such tricks are *speciation* and *homology*. Both limit the possibilities of crossover on different levels. Speciation only allows individuals of similar species to produce offspring together. As crossover only happens with sexual reproduction, this means that crossover only happens between the genomes of similar species (which have similar genomes). Homology describes the fact, that crossover tends to exchange very similar parts of the genetic code between the two genomes which mostly serve the same function. This results in quite small changes and greatly reduces the probability to remove code parts providing important functions from a genome. It is very difficult to model this on the computer for enhancing evolutionary algorithms, because the similarity of function between genome parts cannot easily be measured. Nature apparently also does not measure function but in natural crossover, the similarity of the DNA sequence seems to be a sufficient indication of function for providing the advantages of homology. This usually does not hold in evolutionary computation. There has been some work on trying to model homology for GP either by only defining homologous code as being found at the same position in the genome [D'haeseleer, 1994; Altenberg, 1995] or by introducing a much more sophisticated definition of homology by comparing both structure and function of the code as proposed in [Banzhaf et al., 1998]. Though the results did not yet indicate a great breakthrough, it might be a good idea to try some kind of homology on OOOP.

One approach to homology for OOOP could be to introduce a genealogy for genes. Provided a good datastructure for storing the sequence of all past IDs of a gene, one could easily determine the *genealogical distance* between any two genes. You only would have to find the last ID common to both lines and add the number of IDs following this common ID including the actual gene IDs. In case of no common ID, the genes would not be related. As a gene changes its ID with every variation, this genealogical distance would provide a very easy measure of structural similarity between two genes[39].

In the current implementation of OOOPS, homology makes no sense because insertion and deletion are separate variation operators. Homology can only enhance genetic operators in which a part of a genome is replacing a part of another genome[40], because only then these two parts can be compared. But the only thing one has to change to enable homology for gene conjugation in OOOPS is to exchange gene insertion and gene deletion with a single gene conjugation operator which replaces a recipient gene by a donor gene. Then you can ensure some kind of homology by only allowing gene conjugation to exchange genes with a genealogical distance below a certain limit.

If you would analogously combine code insertion and code deletion into a single code conjugation, you could even view this genealogically limited code conjugation as a form of speciation for genes. A genealogical distance below the specified limit would mean that the genes are of the same "gene species" and can therefore be recombined.

[39] Of course, this measure is not an exact edit-distance (which would count the minimal number of changes needed to make one gene out of the other), but it is an approximation which can be computed in nearly no time at all.

[40] ... or in which a part of a genome replaces another part of the same genome ...

Chapter 4

The Goal: Evolution of Distributed Intelligence

An object-oriented ontogenetic program is made of several units that interact for creating a joint behaviour that the single units are not capable of producing. No matter how we define intelligence, the object-oriented ontogenetic program can be called *distributed intelligence* if we have the opinion that the behaviour of the program shows intelligence. This is obvious if OOOP is used in the explicitly distributed problem solving approach described in section 2.1.3, in which the units represent and are later realized as separate physical objects that interact in the real world with messages on the basis of radio, sound, light or the like. But also in the centralized OOOP approach, in which it is used as modularization technique for a single program, the intelligence it includes can be seen as distributed. In the world of the program, the different modules are separate units that have to work together to solve a given problem. So in this case the intelligence is distributed between them.

In this context, I understand intelligence (in contrast to its common meaning in *Artificial Intelligence* as we will see in section 4.3) as follows:

> *Intelligence* is a property of a system that shows intelligent behaviour which in turn cannot be intersubjectively and statically defined. If the behaviour of a system (for arbitrary reasons) is seen by a person as being intelligent, this makes the system include intelligence.

There are many programs that exhibit a behaviour that makes some people think: "Wow, that's intelligent!" With the above definition, the program will therefore have intelligence. It's creation will be a development of intelligence. But that does not mean that it will keep the honor of this characterization forever. Computer programs that have been seen as being intelligent in the 60s are now being laughed about. Animals can suffer a similar fate[1]. A long time, bees were marvelled at for their great cooperation and building abilities. But the more we find simple behavioural rules for single bees that can create such a cooperation, the less we are tempted to call their behaviour intelligent.

As we are evolving programs that solve problems which we ourselves cannot solve and in a way that we cannot understand, we will be tempted to call their behaviour intelligent and

[1] Though I doubt if they really suffer. Like computer programs, they are probably not interested in how we denote their behaviour and whether we think that they are intelligent.

can therefore call the evolution of such programs an evolution of intelligence (or of distributed intelligence in the case of OOOPS).

4.1 Distributed Intelligence – Related Work

Distributed intelligence is a subject that has until now mainly been studied by other parts of the computer science community which use quite different methods for developing distributed intelligent systems. Probably the most prominent area of research trying to create distributed intelligence is the field known as *Distributed Artificial Intelligence* (*DAI*) and its subfield called *Multiagent Systems* (*MAS*). This field will be shortly characterized in section 4.1.2. But in section 4.1.1, I will first describe another approach to creating distributed intelligence that has primarily been developed by an equally called research group at the Massachusetts Institute of Technology: *Amorphous Computing*. The reason for this order of presentation is, that the idea of amorphous computing is much closer related to the multicellular approach of OOOP than multiagent systems, so that OOOPS can very easily be used for evolving amorphous computers. Evolving multiagent systems with OOOPS requires to leave the notion of cell clusters and extend the *Multicellular Programming* approach of OOOP to *Swarm-Programming* as described in section 4.2.

Section 4.3 finally tries to fit the new field denoted as *Evolution of Distributed Intelligence* (*EDI*) into a taxonomy covering all the major current approaches to creating intelligent problem solutions on the basis or with the help of computers.

4.1.1 Amorphous Computing

The idea of amorphous computing is to randomly distribute a great number of similar computing elements on a surface or throughout a volume. These *computational particles* are sensitive to the environment, may affect actions and are all programmed identically[2]. They should be as self-sufficient as possible and work asynchronously, but they can interact with each other using local communication based on diffusible carriers like short-distance radio or chemical substances. Each particle has only modest computing power and a modest amount of memory, but a large group of such particles can self-organize to build a single amorphous computer which can solve new types of problems.

The computational particles constituting an amorphous computer can be such different entities as tailor-made biological cells or very small chips integrating many microelectronic mechanical components. The development in both microfabrication and fundamental biology will probably enable the production of huge numbers of almost-identical information-processing units at almost no cost in the future, provided that all the units need not work correctly and that their geometric arrangement and interconnections need not be precisely manufactured. There are already some research groups that build such cubic millimeter-scale microelectromechanical computing units with the desired physical abilities [“Smart Dust” Project, ; Ultralow Power Wireless Sensor Project, ; Wireless Integrated Network Sensors (WINS) Project,]. Eventually, it may be possible to construct even smaller and cheaper microfabricated particles using molecular devices [Rotman, 2000].

Harold Abelson et al. of the Amorphous Computing Project at MIT write in [Abelson et al., 1999]:

[2]But they can store a local state and generate random numbers.

> Yet fabrication is only part of the story. Digital computers have always been constructed to behave as precise arrangements of reliable parts, and almost all techniques for organizing computations depend upon this precision and reliability. So while we can envision producing vast quantities of individual computing elements—whether microfabricated particles or engineered cells—we have few ideas for programming them effectively. The opportunity to exploit these new technologies poses a broad conceptual challenge, which we call the challenge of amorphous computing:
>
> How does one engineer prespecified, coherent behavior from the cooperation of immense numbers of unreliable parts that are interconnected in unknown, irregular, and time-varying ways?

This question could be answered with: "By using multicellular programming." Indeed, we have seen that OOOPS as a realization of multicellular programming exactly does what is being requested here. This is not surprising, because the OOOP paradigm is inspired by the same natural examples as is amorphous computing. Still, the researchers working on the field of amorphous computing do until now not use the natural way of producing such systems. To my knowledge, there is no approach until now that uses evolutionary algorithms for breeding good particle programs for amorphous computers (except OOOPS). The main reason for this is probably that the necessary interactions in an amorphous system seemed to be too complex for the evolutionary power of current evolutionary algorithms. There was not yet a programming paradigm like OOOP that fits exactly the amorphous computing notion of distributed intelligence on the basis of the interaction between many small identically programmed units and which can easily be combined with evolutionary algorithms.

But why did the amorphous computing researchers not even try evolutionary computation? Might another reason be a misunderstanding about EC? Harold Abelson et al. note in [Abelson et al., 1999]:

> As engineers, we must learn to construct systems so that they end up organized to behave as we a priori intend, not merely as they happen to evolve.

This comment is not targeted on evolutionary computation but on artificial life research about self-organizing systems. Nevertheless, it should have been followed by a remark like: "One possible approach is to direct the evolutionary development with breeding methods. This is called evolutionary computation."

If you compare descriptions of the amorphous computing challenges with what you have read in previous chapters about OOOP and OOOPS, you will notice that the object-oriented ontogenetic programming system is ideally suited for developing programs for the particles in amorphous computers. All the following quotes are taken from [Abelson et al., 1999].

> Traditionally, one seeks to obtain correct results despite unreliable parts by introducing redundancy to detect errors and substitute for bad parts. But in the amorphous regime, getting the right answer may be the wrong idea: it seems awkward to describe mechanisms such as embryonic development as producing a "right" organism by correcting bad parts and broken communications. The real question is how to abstractly structure systems so we get acceptable answers, with high probability, even in the face of unreliability.

> ...
>
> Wave propagation with hop counts, as Nagpal [Nagpal, 1999] remarks, is evocative of the gradients formed by chemical diffusion that are believed to play a role in biological pattern formation. Consequently, we can attempt to organize amorphous processes by mimicking gradient phenomena observed in biology.
>
> ...
>
> All of the particles have the same program. As a result of the program, the particles "differentiate" into components of the pattern.
>
> ...
>
> In general, it is plausible to expect that the most powerful techniques for amorphous computing will be ones that will tie computation intimately to particle activation and mobility, and to physical constraints of the environment.

All these quotes could as well be taken from a description of the OOOP paradigm[3]. Even though OOOP was developed without having any idea about amorphous computing, it is a nearly exact realization of the amorphous computing ideas, just because the ideas in both cases are taken from the natural example of cooperation between cells in a multicellular creature.

Amorphous computing as realized by the MIT project does not include mobile particles even though this possibility is included in the amorphous approach described in [Abelson et al., 1999]:

> In general, the individual particles might be mobile, but the initial programming explorations described here do not address this possibility.

A project called "Programming the Swarm" initiated by David Evans works on programming methodologies for amorphous computers including mobility of the particles ["Programming the Swarm" Project,]. OOOPS is also perfectly suited for developing particle programs for these types of systems. For doing this, only the multicellular approach of OOOP has to be extended to a swarm approach as described in section 4.2. This extension also enables OOOPS to breed control programs for the interacting units of multiagent systems and with that opens up a huge field of possible applications for the object-oriented ontogenetic programming paradigm and for the OOOP communication paradigm.

4.1.2 Multiagent Systems

Compared to amorphous computers, multiagent systems do not concentrate on large numbers of very small interacting entities. Instead, they can consist of arbitrarily complex and powerful units called *agents*. Such an agent could for example be a robot in an automated production process which would be much more flexible than the current industrial robotic systems. Or it could be one of several robots autonomously exploring a distant planet. The cooperation of several robots would have the advantage over one single explorer that they could help each other in difficult situations and that the exploration could be done much faster. The agents do not have to be physical entities; they can also be separate programs that interact either on a

[3]Though the OOOP communication bases on a much more realistic diffusion model than hop counts.

single computer or in a network. But the example of real mobile robots each solving complex tasks shows best the mentioned difference to amorphous computing and the extensions to OOOP that are necessary for developing such systems. These extensions will be discussed in section 4.2.

As the field of multiagent systems is already very well known and spans a wide area of possible applications, there are several introductory publications that can be recommended for getting more information about this subject. [Schmutter, 2000b] is a short introduction to the main ideas and approaches. [Stone and Veloso, 2000] is more detailed concentrating on machine learning methods. [Weiss, 1999] finally is a book extensively covering most current MAS approaches and problems.

The research field of multiagent systems has come into being as a subfield of distributed artificial intelligence which itself is a subfield of Artificial Intelligence (AI). As we will see in section 4.3, artificial intelligence uses a quite different approach to creating intelligence than evolutionary computation. This explains, why also in multiagent systems research there are very few works trying to use evolutionary methods for developing distributed intelligence, even though this research field is older and much larger than amorphous computing.

One of the few approaches to using genetic programming with multiagent systems is described by Thomas Haynes et al. in [Haynes et al., 1995]. They write:

> The identification, design, and implementation of strategies for coordination is a central research issue in the field of Distributed Artificial Intelligence (DAI) [Bond and Gasser, 1988]. Current research techniques in developing coordination strategies are mostly off-line mechanisms that use extensive domain knowledge to design from scratch the most appropriate cooperation strategy. It is nearly impossible to identify or even prove the existence of the best coordination strategy. In most cases a coordination strategy is chosen if it is reasonably good ... We believe that evolution can provide a workable alternative in generating coordination strategies in domains where the handcrafting of such strategies is either prohibitively time consuming or difficult.

Sean Luke and Lee Spector describe another approach to using genetic programming for the evolution of distributed intelligence in [Luke and Spector, 1996]. These first steps into EDI have already been discussed in section 1.5.2 where I also described some disadvantages of the presented models. In spite of these pitfalls, especially of the simplicity of the models, the real reasons for their little impact on other multiagent systems researchers have to be sought for somewhere else. The main reason might be the large separation of the EC and the AI research communities. The relation between the different approaches to creating intelligence will be discussed in section 4.3.

4.2 Multicellular Programming and Swarm-Programming

Multicellular Programming (*MP*) is the combination of the OOOP paradigm with genetic programming as realized in the current object-oriented ontogenetic programming system (OOOPS). It has this name because the different program modules interact with each other like the cells in a multicellular creature. Most cells have a definite position in the creature and cannot move freely. But they communicate with each other by producing diffusible substances. The production of these substances is based on and is the basis for gene regulation.

In multicellular programming, the OOOP units analogously have static positions in a grid which they can only leave by dying. A new unit then can take that place only by division of an adjacent unit. They also interact on the basis of gene regulation which controls the production of diffusible messages.

Swarm-Programming (*SP*) is an extension and generalization of multicellular programming which allows to use the advantages of OOOP and GP also for mobile entities and very elaborate systems such as many multiagent systems. The name is written with a hyphen to distinguish it from other swarm approaches [Hiebeler, 1994; Evans, 2000]. The main differences between multicellular programming and swarm-programming are the following:

1. The swarm units are allowed to be mobile.

2. They take arbitrary positions in the environment. There is no grid any more.

3. While the multicellular units usually are as small as possible, the swarm units tend to get much bigger and very complex.

4. The swarm units' program often has to consist of much more than the OOOP part which controls behaviour and communication. Like it is not possible and makes no sense to evolve a word processor, it also makes no sense to try breeding OOOP units which can control all aspects of a real robot in a multiagent system. Such a robot has to solve so many complex subproblems for which already good solutions exist (like analysing visual information or controlling motions) that OOOP only takes on the task of organizing the interaction between the swarm units (i.e. in this case between the robots). This means that the programs bred with swarm-programming only control the possibilities provided by the specialized subsolutions that can be developed with other techniques. The function set of the GP algorithm then for example includes functions of the communication or visual or motor control modules of the robot program. The interaction with the environment only happens indirectly through these specialized modules.

5. Swarm-programming can include other forms of communication than the diffusible messages.

6. Swarm-programming usually includes a much more complex environment model[4] than multicellular programming.

Both multicellular programming and swarm-programming are methods for breeding intelligent distributed problem solutions. They are new and promising approaches to the hopefully soon more quickly growing field of *Evolution of Distributed Intelligence* (*EDI*).

4.3 A Taxonomy for Artificial and Computational Intelligence

In the last sections and chapters, we have met ANN, DAI, EC, EDI, EP, ES, GA, GP, MAS, MP, OOOP and SP. These are all abbreviations for problem solving approaches that try to create intelligence as defined in the introduction to chapter 4. All these methods are commonly

[4] As described in section 2.1.3 for the distributed problem solving approach of OOOP.

summarized under two catchwords that sound similar but usually describe different parts of computer science: Artificial Intelligence and Computational Intelligence.

Artificial Intelligence (*AI*) is the oldest and best known research field which has the goal of creating intelligent systems. There are some people that use AI as the generic term for all approaches with that goal and define it for example like John McCarthy in [McCarthy, 2001]:

> Q. What is artificial intelligence?
>
> A. It is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable.
>
> Q. Yes, but what is intelligence?
>
> A. Intelligence is the computational part of the ability to achieve goals in the world. Varying kinds and degrees of intelligence occur in people, many animals and some machines.

This definition of intelligence poses the question on how we can determine whether some unit achieves a goal. Does it have to internally represent this goal in a specific way or is it sufficient that the unit's actions make sense for an observer which interprets them as steps for achieving a goal? In the first case: then how do we know that humans or other animals achieve goals? How do they internally represent goals? We have no clue about this question, so we cannot define intelligence by using explicitly represented goals. This means that the ability to achieve goals in the world is in effect only the ability to do things that make sense for a human observer given either the goals that he preset for a machine or the goals that he can imagine for a person or an animal.

Stuart J. Russell and Peter Norvig also use this definition of intelligence in [Russell and Norvig, 1994] and they call it rationality. A system for them is rational if it does the right thing. The right thing is usually the one which best (or at least well) helps the system in achieving its goals. Stuart J. Russell and Peter Norvig denote this as the "rational agent approach".

There is a slight difference between these definitions of intelligence and the one which has been presented in the introduction of chapter 4 and which is used in this publication. If a system is intelligent or "rational" if it does things that help it in achieving its goals, every correctly working machine and nearly all living beings are intelligent. The machine follows the goal for which it was produced and the creature or plant follows the goal to survive and reproduce. A species that would not follow this goal would die out. But we would usually not call every such system intelligent. This definition therefore seems to be too wide. We call systems intelligent that can perform tasks that seem very complex to us, not ones that can perform any tasks. But whether a performed task is complex enough to call a specific system intelligent is a very subjective and dynamic judgment. Therefore, my definition does not describe intelligence like McCarthy, Russell and Norvig with the added precondition that it must be able to perform complex tasks. Instead, it is formulated by explicitly referring to the judgment of an observer. Using this definition, evolutionary algorithms can create intelligence. But even though they seem to be included in the definition of AI used by McCarthy, Russell and Norvig, they are usually not seen as a part of artificial intelligence. In [Russell and Norvig, 1994], which is one of the best known books about AI, you find nearly nothing about

EC, a few pages about Fuzzy Logic and only 35 pages (of 859) about ANNs. This is probably one of the reasons, why the important fields of EC, Fuzzy Logic and ANN have joined forces under the name "computational intelligence".

There are many AI researchers that use more restricted definitions of artificial intelligence (which in my opinion better describe the common orientation of AI). Richard E. Bellman defines AI in [Bellman, 1978] as

> ... the automation of activities that we associate with human thinking, activities such as decision making, problem solving, learning ...

Richard Stottler [Stottler, 1999] defines AI as follows:

> Artificial intelligence is the mimicking of human thought and cognitive processes to solve complex problems.

Patrick Henry Winston's definition of AI in [Winston, 1992] is:

> Artificial Intelligence is ... the study of the computations that make it possible to perceive, reason, and act. From the perspective of this definition, Artificial Intelligence differs from most of psychology because of the greater emphasis on computation, ...

Many online encyclopedias define it similarly restricted to human intelligence. The Webopedia [Webopedia, 2002] defines AI as:

> The branch of computer science concerned with making computers behave like humans.

Whatis.com [Whatis.com, 2001] defines AI as follows:

> Artificial intelligence is the simulation of human intelligence processes by machines, especially computer systems.

The AI depot [AI Depot, 2001] uses the following definition:

> Artificial Intelligence is a branch of Science which deals with helping machines find solutions to complex problems in a more human-like fashion. This usually involves borrowing characteristics from human intelligence, and applying them as algorithms in a computer friendly way.

As we can see in these definitions, most of AI is very human oriented. Russell and Norvig distinguish in [Russell and Norvig, 1994] between AI approaches centred around humans and approaches centred around rationality. As an example for the rationality approach they quote Winston's definition of AI. But as we have seen, he continues by distinguishing AI from psychology by saying that AI concentrates more on what computers can do: computation. With saying this, he acknowledges that AI bases mainly on psychological theories. This also shows in the weighting of subjects in his book (like in most AI books). But Psychology is the study of human intelligence on the basis of behavioural experiments. According to the UNESCO definition, it belongs to the social sciences and not the natural sciences. Also nearly all the other sciences inspiring artificial intelligence research are social sciences. Aaron Sloman writes about these influences in [Sloman, 1998]:

> If we construe AI in this way (as studying how information is acquired, processed, stored, used, etc. in intelligent animals and machines) then it obviously overlaps with several older disciplines, including, for instance, psychology, neuroscience, philosophy, logic, and linguistics.

This listing of influences includes the natural science *neurophysiology*, but that discipline only overlaps with AI as far as we see the research area of artificial neural networks as a part of AI. ANNs are the most important example of the so-called connectionist approach to AI which contrasts to the approach of the classical computationalism. [Internet Encyclopedia of Philosophy, 2001] describes this last approach (which clearly is the central approach of AI) as follows:

> According to classical computationalism, computer intelligence involves central processing units operating on symbolic representations. That is, information in the form of symbols is processed serially (one datum after another) through a central processing unit. Daniel Dennett, a key proponent of classical computationalism, holds to a top-down progressive decomposition of mental activity.

This approach, whose influences are described in [Sloman, 1998] as follows, is often called *classical AI*.

> It should be clear from all this that insofar as AI includes the study of perception, learning, reasoning, remembering, motivation, emotions, self-awareness, communication, etc. it overlaps with many other disciplines, especially psychology, philosophy and linguistics. But it also overlaps with computer science and software engineering ...

Most AI researchers consider ANN as a part of AI, because most of us think that the human intelligence is located in the brain. And modelling the brain with an artificial neural network seems to be another approach to modelling human intelligence. But neural cells are an important part of the intelligent functioning and behaviour of many different multicellular creatures, not only of humans. Current ANNs more closely resemble for example the nervous system of the sea-snail Hermissenda than that of humans. But these systems still do a good job in solving complex problems. So when using the human centred definitions of AI, ANN is not really a part of AI, because it does not simulate human intelligence. Moreover, in most publications about AI, it plays a minor role even though it is an important approach to creating intelligent systems. This has lead many ANN researchers to see themselves more as a part of CI than of AI, because artificial intelligence is still often identified with classical AI. This is fostered by the fact that classical AI is the only area approaching the creation of intelligence, that does not have an agreed own name but usually simply calls itself AI.

Computational Intelligence (*CI*) is even more difficult to characterize than AI as it is more a collection of all the approaches to creating intelligent systems excluding traditional AI than a coherent research field. As the chairs of the ICSC congress "Computational Intelligence: Methods and Applications" put it on the introductory page to [Kuncheva and Porter, 2001]:

> Defining "Computational Intelligence" is not straightforward. Several expressions compete to name the same interdisciplinary area. It is difficult, if not impossible, to accommodate in a formal definition disparate areas with their own established

> individualities such as fuzzy sets, neural networks, evolutionary computation, machine learning, Bayesian reasoning, etc.

The three main research ares united under the term "computational intelligence" are EC, ANN and *Fuzzy Logic*[5]. The last is a generalization of traditional (Boolean) logic which allows to mathematically represent and handle uncertainty and vagueness. Its underlying principles, introduced in the 1960s by Lotfi A. Zadeh, can also be applied to set theory and other areas and have been successfully used for many complex controlling tasks in industry[6].

Another possibility to distinguish CI from AI is to stress the fact that CI uses *subsymbolic knowledge processing* whereas classical AI uses symbolic approaches. The CI project at the University of Dortmund [Collaborative Research Center Computational Intelligence, 2002] for example remarks:

> In contrast to the traditional field of Artificial Intelligence (AI) CI makes use of subsymbolic, i.e. numerical, knowledge-representation and -processing. The probably most well known techniques of Computational Intelligence are Fuzzy Logic, Artificial Neural Networks, and Evolutionary Algorithms.

For understanding this difference, you have to understand what is meant by *symbolic representation*. The article [Dictionary of Philosophy of Mind, 2001] provides a good introduction to the difference between the symbolic approach of AI and the distributed representations used in CI. In a nutshell, it can be explained as follows: In a symbolic representation, the knowledge can be decomposed into symbols (e.g. a concept in a semantic net or a proposition in a logic representation) which each have a particular meaning. In contrast, in distributed or subsymbolic representations, a meaning or specific part of the knowledge cannot be clearly located. The knowledge is represented in the whole state of the system. The system produces its own meanings that cannot be understood by humans.

In this sense, human society, its sciences (e.g. psychology) and AI are symbolic while nature, its sciences (e.g. neuroscience) and CI are subsymbolic. The knowledge representations of AI are nearer to human understanding, but the representations in CI are nearer to how nature works. I personally think that the natural approach has a more promising future even though it is easier for us to build symbolic systems which are therefore still often the best choice. The reason for this belief is the following line of thought:

I believe that neither humans nor any other animal thinks in symbols. Language is pure symbols. But we do not think in language. When we think of a concept which can be described by a symbol (a word of the language), we do not think of the symbol itself, but of all the associations that we have when we hear this specific word. These associations can often also be described by symbols, but they are no symbols themselves. They are made of many recollections of past internal states of the system human being produced by sensory inputs combined with the previous states of the system. Put easier, we think in the combined recollections of many images, sounds, smells and other past sensory experiences. Symbols (which represent such a combination) are only a tool for communication between individual creatures. The approximate meaning of the symbols is common knowledge of the

[5] See for example [World Congress on Computational Intelligence, 2002] and the aims & scope section of [International Journal of Computational Intelligence and Applications, 2001].

[6] A good web resource which also distinguishes between CI (EC, ANN and Fuzzy Logic) and classical AI is [Keller and Kangas, 2001].

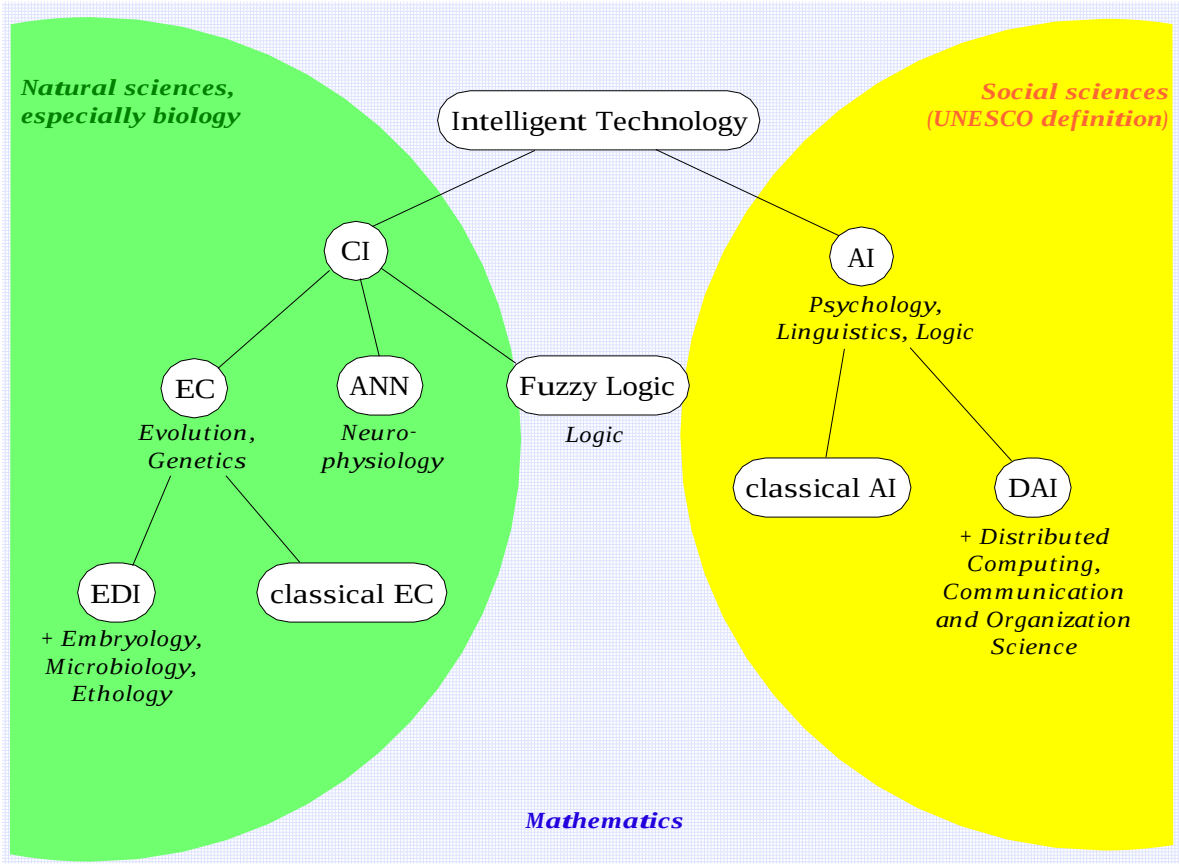

Figure 4.1: Research fields trying to create intelligent systems.

creatures which helps them understand each other. A system of symbols meets best the requirements of the communicating creatures if they can develop it themselves. Also the knowledge representation meets best the requirements of a creatures if it can produce its own meanings. Every creature has to adapt its view of the world to its own sensory system and its own needs. It makes not much sense to force human-made meanings and symbols upon a totally dissimilar creature which has to live in a totally other environment than humans. Every artificial system is such a totally dissimilar "creature" in a totally other environment, which means that the most promising approach to making it behave intelligently in its environment should be to let it develop its own view of the world in its own representation. Of course, we cannot give it the choice between all imaginable representations. But we can give it as much freedom as possible by providing it with a powerful, flexible and low-level representation without imposed human meanings. Given a good choice of the function set, GP seems to best solve this problem. The only problem is that the more freedom we give to the creature, the longer it takes to breed it. But with faster computers, distributed computation and better GP methods this problem constantly loses of importance.

Figure 4.1 shows the described relations between Artificial Intelligence, Computational Intelligence, Evolutionary Computation, Artificial Neural Networks, Fuzzy Logic and the distributed versions of EC and AI: Evolution of Distributed Intelligence and Distributed Artificial Intelligence. It shows the different underlying philosophies of AI and (most of) CI

and it shows the main influences inspiring the various approaches.

EC and ANN are inspired by the natural sciences while AI is inspired by social sciences. Fuzzy Logic is a little difficult to position, because it is seen as a part of CI even though its inspirations come partly from human thinking and it is very closely related to symbolic approaches[7]. But clearly its main influences are mathematics and logic. Mathematics is important for about every area in computer science, hence it is presented as the background of the whole taxonomy. Philosophy is in my opinion the basis of and an inspiration for every science. Therefore, it is not noted with AI, even though AI researchers like to mention it as an influence.

Taxonomies are always problematic, because they serve in dividing different approaches. But the best science is the one that does not know any frontiers. It should be clear, that a taxonomy cannot completely define a research field. There are always relations, combinations and variations that cannot be displayed. On the other hand, a taxonomy also serves in getting an overview of a big field and as such can also make you look beyond your frontiers. Likewise, it can make you understand some relations between different approaches which can help in combining them if this seems promising.

[7]It could even be seen as a symbolic approach itself. Therefore, CI is also sometimes defined as being adaptive, fault tolerant and approximative. Lotfi A. Zadeh, the inventor of fuzzy logic, coined the term *soft computing* which is defined similarly such that fuzzy logic is clearly part of it and which is mostly used synonymously to computational intelligence.

Chapter 5

First Experiments

In the last chapters, you have learnt a lot about the object-oriented ontogenetic programming system OOOPS and about the underlying paradigms and innovations, but you might say: "Nice ideas! But do they really work?" For proving that they actually work, I have performed some simple first experiments that do not take a long time to prepare and execute. Of course, this is not sufficient for supporting all the claims of the previous chapters, but it shows that the system and the paradigm can produce solutions for interesting distributed problems. There is a vast number of other experiments with very different problems from a variety of domains that can and will be carried out with OOOPS in future work. The goal of the here presented work was only to create the ideas and a system realizing these ideas.

5.1 Paintable Computing: developing paint that signals and shows defects in a wall as soon as they appear

The characteristics of amorphous computing have been explained in section 4.1.1. I claimed that OOOP is a suitable system for developing programs for amorphous computers because it realizes exactly the features that are necessary for this task. As a first validation, I have performed some simple experiments which on the one hand support the claim that OOOP is capable of evolving programs for amorphous systems and which on the other hand serve as a proof of principle for OOOP and OOOPS.

One easily understandable concept for the application of amorphous computers has been developed by Bill Butera at the MIT amorphous computing project. It is called *Paintable Computing*. The idea is to mix amorphous computing particles into a sort of paint that could be applied as a coating to bridges, buildings or planes for sensing and reporting on problems like defects or tensions in the material. The hardware (i.e. the microfabricated particles) has not yet reached a state which would allow to realize this idea now, but as technology is evolving quite quickly, one should already think about how to program these systems.

We will discuss three experiments approaching a paintable computing scenario on different levels of abstraction. The simulated amorphous computing particles communicate with each other via short-distance radio signals of different frequencies (which define the OOOP message types) and different intensity. Apart from that, the particles can only behave in two distinct ways: They can set a signal (which in reality would mean to change the colour of the surrounding paint) or they can unset it (making the paint return to the original colour). As given by the paintable computing approach, the computing particles are located in a paint

OOOPS parameter	Value
Population size	20
Individuals per Ooopsi	5
Dimensions of cell space	2
Number of different produceable message types	4
Mortal percentage	40
Elitist size	2
Maximal number of genes in new individual	5
Maximal number of commands in new gene	1
Maximal number of requirements for new gene	4
Maximal number of inhibitors for new gene	4
Maximum value for a requirement	20.0
Maximum value for an inhibitor	20.0
Maximum intensity for message productions	9.9
Mutation probability if selected for variation	0.7
Gene conjugation probability if selected for variation	0.5
Code conjugation probability if selected for variation	0.0
Command deletion probability if mutation is called	0.4
Command insertion probability if mutation is called	0.4
Probability for changing produced message type if mutation is called	0.2
Probability for changing produced message intensity if mutation is called	0.4
Probability for deleting a requirement if mutation is called	0.2
Probability for inserting a requirement if mutation is called	0.2
Probability for deleting an inhibitor if mutation is called	0.2
Probability for inserting an inhibitor if mutation is called	0.2

The used function set:	signal = 0; signal = 1;

Table 5.1: OOOPS parameters used in the three paintable computing experiments.

coating a wall and shall report on defects in this wall. If a piece of material breaks out of the wall leaving a hole in the coating, the remaining computing particles shall take notice of the loss and quickly change the paint colour in a big area around the hole so that the defect can instantly be seen and easily be located by just determining the centre of the discoloured area.

The OOOPS parameters common to all three experiments are listed in table 5.1. In all three experiments, there is a (2-dimensional) cluster of cells in the grid on initialization. The growth genes (death and division) are inactivated. After some time during which the communication can settle, the Ooopsi determines the number of cells which have set the signal. Then, a block of several cells in the centre is removed and the remaining cells again get a short time for noticing the loss and spreading this news. At last, the number of set signals is determined again and the run is stopped. The greater the difference between the first and the second count, the more cells change their surrounding paint colour and thus the better one can see the defect. Consequently, the fitness of the individual (which is the whole amorphous computer) is set to this difference between the number of signals before and after the damage in the wall (modelled by the removal of the block of cells). The differences

between the three experiments are the following:

In the first experiment, the cell cluster was initialized by inserting a cell at every second grid point in an area of $21 * 21$ points and then letting it grow for a short time. The growth genes were preset so that there would occur cell death if every grid point around was taken, and cell division otherwise. After the growth time, the genes for death and division were inactivated. This was thought to provide an irregular cell cluster because cell division chooses a random place next to the original cell for the new cell and the death gene would prevent a regular cluster with a cell at every grid point. But this initialization process was not a very clever idea, because for saving time (which is important as we have seen in section 2.1.4), the growth time was reduced to one update per cell which resulted in a nearly fully occupied regular cell cluster. The reason for this was the following: In this first update, every cell (which still had an adjacent free gridpoint) was dividing. As the message intensity was not high enough in the initial cell cluster for activating any death gene and there was only one update, the death gene could not do what it was supposed to do. But even if the growth time would have been prolonged, the results would probably not have been satisfying: During the next update, the message intensity would perhaps have activated all death genes which would have had the effect of deleting every single cell. This shows how difficult it is to hand-code the behaviour of an amorphous computer. Even trying to produce such a simple behaviour like this random initialization on the basis of growth processes can easily have totally unwanted effects. But the results of the first experiment still have value, because this initialization process resulted in an approximately regular cell cluster with a cell at nearly all the $21 * 21$ grid points. So the optimum fitness for this experiment lies at about $21 * 21 = 441$. A really irregular random positioning was then realized in the next experiment. As already mentioned, the update sequence was ordered as described as the standard case in section 2.2. In both the first and the second experiment, the cell cluster was given the time of one update per cell for settling before the block of cells was removed and the time of two updates per cell for spreading the news after the removal of cells.

For the second experiment, the same update order was used, but the initialization process was altered so that the cell locations better simulated a random placement of the amorphous computing particles on the wall. In this case, a single program loop simply inserted a cell at every grid point in an area of $21 * 21$ points with a probability of $\frac{3}{4}$. This results in an average optimum fitness of $21 * 21 * \frac{3}{4} \approx 331$. Of course, in this experiment, the best reachable fitness is more variable because the total number of cells depends on chance (more than apparently in the first experiment).

The third experiment was performed using the same initialization procedure as in the second experiment, but the update order was changed. The ordered update used in the first two experiments does not correspond well to the attribute of amorphous computers that the particles work asynchronously. So in this run, the cells were updated randomly. This means, that for every update, a random cell was chosen. The result is, that one cannot guarantee every cell being updated in a specific period, but this method simulates true asynchrony, even with the additional problem of possibly very different working speeds for the particles. Because the initialization is equivalent to that of the second experiment and produces the same number of cells, the optimum fitness for this experiment also varies around 331. The time for settling was in this case set to 1000 cell updates[1] and the time for spreading the news about the defect was set to 3000 updates.

[1] As the cells were updated randomly in this experiment, it is not possible to count the number of updates per cell. Therefore, the total number of updates has to be the measure of choice.

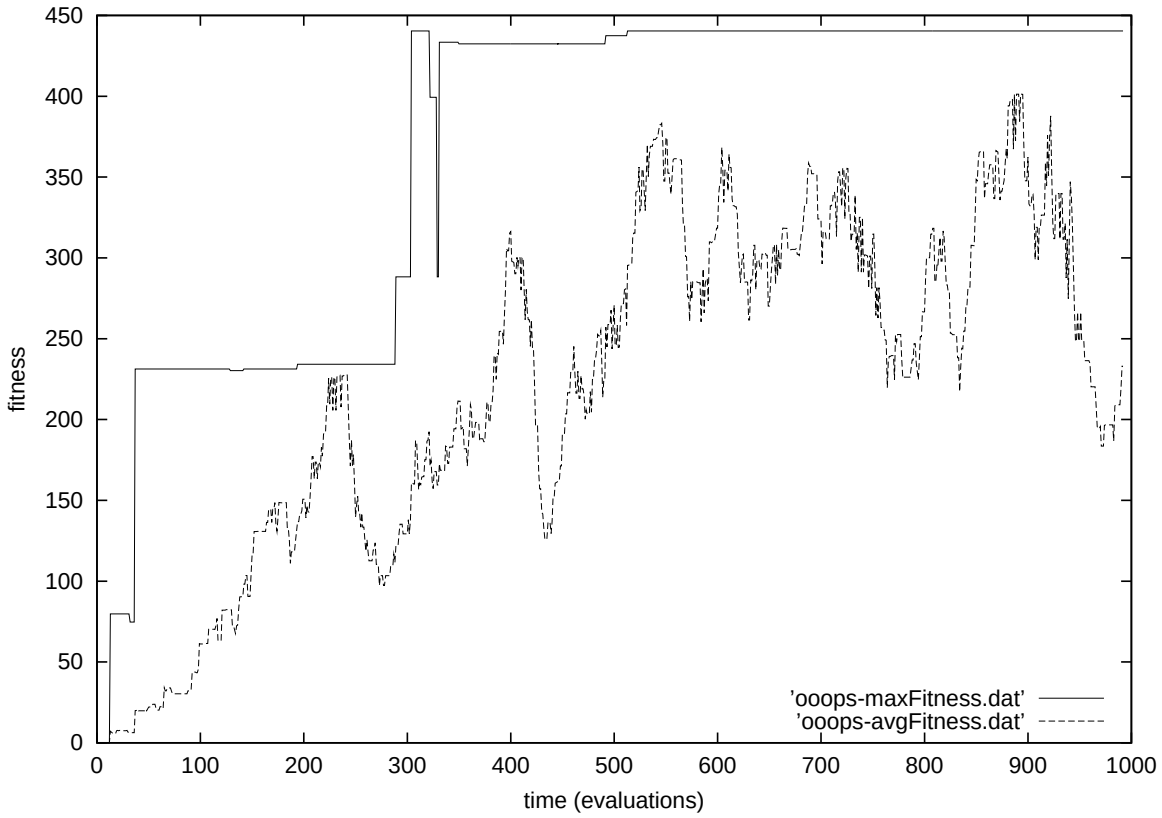

Figure 5.1: The paintable computing experiment with regular locations and ordered update. (1 generation $\hat{=}$ 20 evaluations)

5.2 Results

Figure 5.1 shows the development of the best fitness and the average fitness in the population during the first experiment. The optimum fitness of 441 is reached very quickly. The x-axis shows the number of evaluations and the y-axis the fitness. So if you are used to thinking in generation numbers, which is common in most evolutionary algorithms, you have to divide the value on the x-axis by the population size which in this case is 20. The optimum fitness is therefore already reached after about $\frac{300}{20} = 15$ generations for the first time. And it is finally reached after about 510 evaluations which corresponds to less than 26 generations.

You might wonder why the best fitness sometimes decreases. The elitist approach which does not change the two best individuals should ensure not to lose the best result reached. In fact, the best individual is not lost, but it is possible that the same individual does not produce the same good results in its next evaluation. This is because there is randomness in the evaluations, for example in the initial setup. If you compare the curve of the first experiment with that of the other two runs, you see that the best fitness is much more stable in the first experiment than in the others. This is because there is much less influence of chance in the first run compared to the others: As has been described, the second and third experiment include a lot more randomness in the initial setup of the cells and the third even includes randomness in the update order.

You can see that the average fitness of the population is increasing quite steadily most

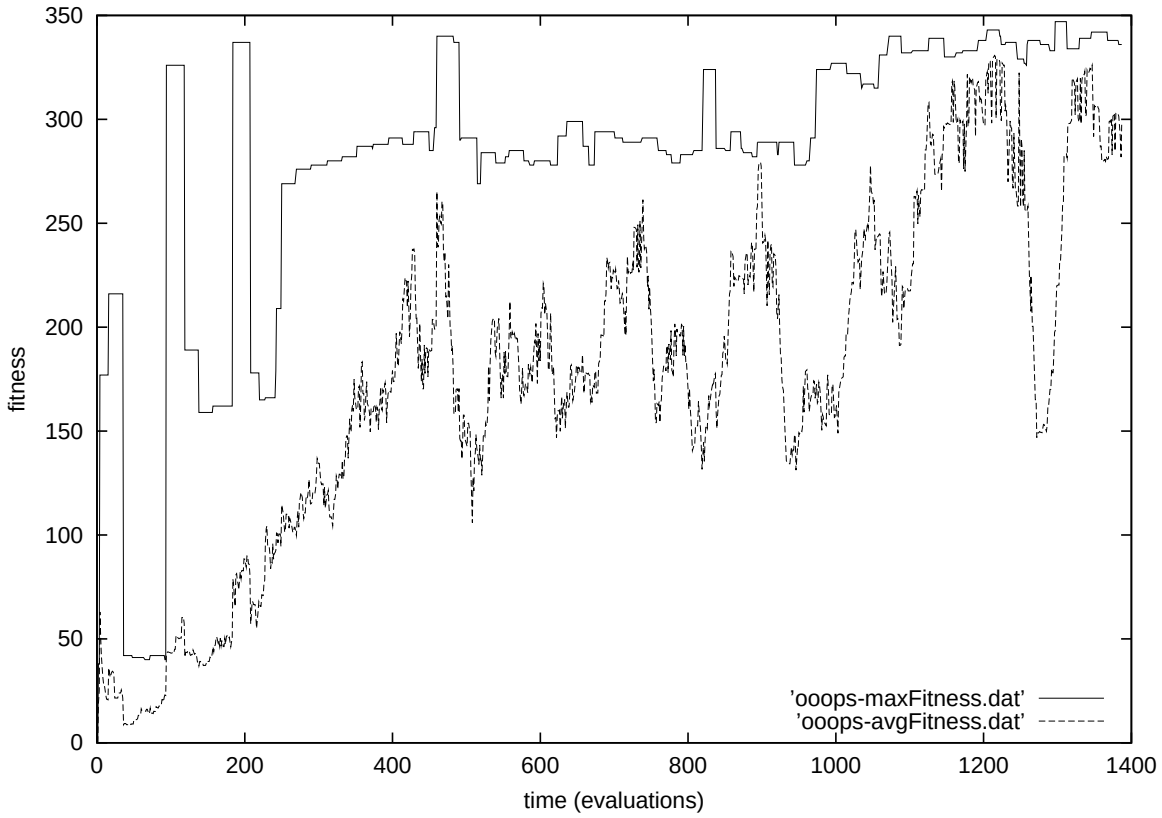

Figure 5.2: The paintable computing experiment with random locations and ordered update. (1 generation $\hat{=}$ 20 evaluations)

of the time, but you will also notice that there are astonishingly regular periods where the average fitness heavily decreases. But this does not have any bad influence on the best fitness. In the third experiment, one even gets the impression that it sometimes helps the population to make new jumps to a better fitness level. This can perhaps be explained with the new reproduction handling introduced in OOOPS. The periods of decreasing average fitness possibly follow periods where there are no parents left in the database. The lack of parents results in many newly initialized individuals which have a high probability of getting a low fitness value. This results in the decrease of average fitness when these new individuals are evaluated. But another result of the new individuals is that totally new ideas for solving the problem can come into the population.

In the second experiment (shown in figure 5.2), the best fitness is very unstable (as was already explained). The average optimum fitness of 331 is already reached after approximately 190 evaluations for the first time. But it is quickly lost again. Roughly, the best fitness increases in four steps:

A level of about 40 is already reached in the randomly initialized starting population. This is easily explainable with the fact that the number of removed cells for simulating the defect reaches approximately this value. All cells in an area of $7 * 7$ grid points are removed which results for the second experiment in an average of $7 * 7 * \frac{3}{4} \approx 37$ cells. So if all cells always keep the signal set, this already results in a difference of set signals of nearly 40 (and

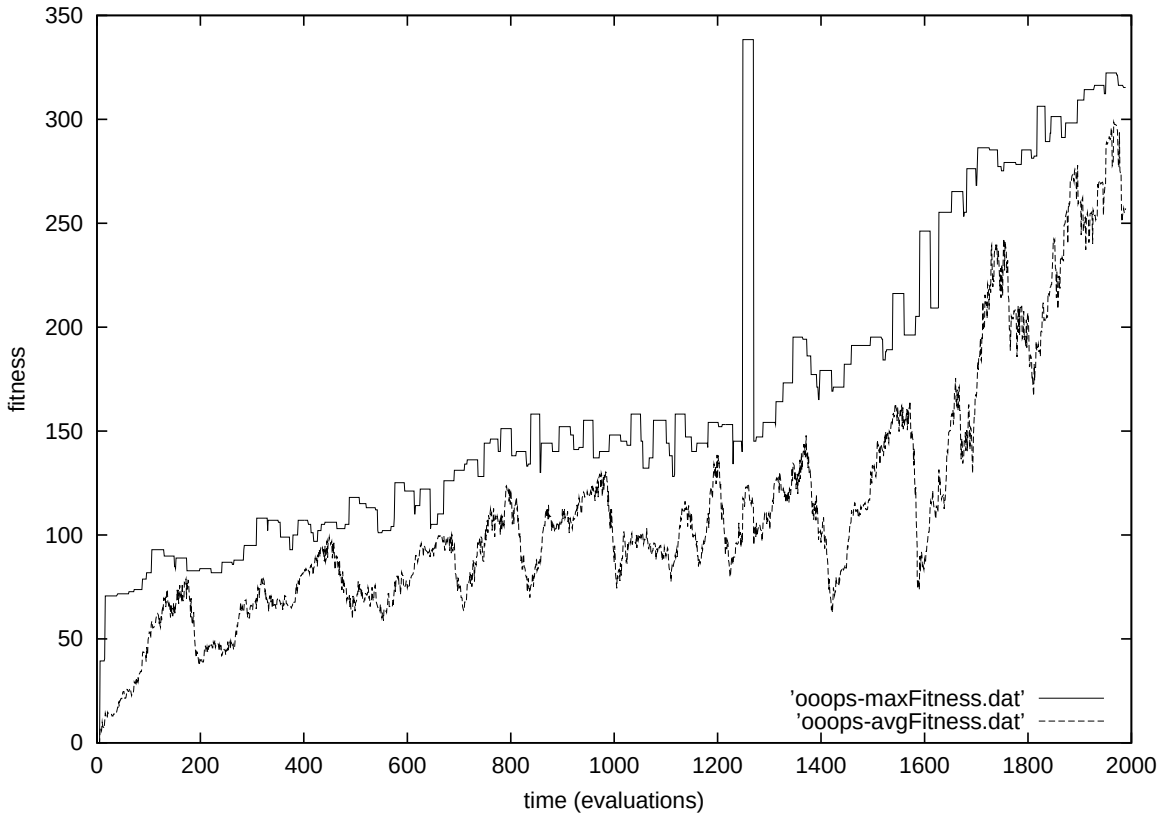

Figure 5.3: The paintable computing experiment with random locations and random update. (1 generation $\hat{=}$ 20 evaluations)

consequently a fitness with that value).

After about 100 evaluations, the best fitness reaches a level of approximately 165. This is half of the optimum fitness. It is the expected fitness if all cells set and unset their signals randomly.

After about 300 evaluations, a fitness level around 290 is reached by the best fitness curve. This level can be explained by the number of removed cells again. If no signal is set before the removal and all cells react to the defect by setting the signal, the fitness value will be equal to the number of remaining cells which is about $331 - 37 = 294$.

The last fitness level is that of the optimum fitness which can only be reached if all cells have initially set their signals and react to the defect by unsetting it. This level is reached in the second experiment after about 1050 evaluations which corresponds to about 53 generations.

In the third experiment (shown in figure 5.3), the first best fitness reached also has a value of 40. This is (analogously to the second experiment) the case where every cell constantly keeps its signal set. But the best fitness quickly leaves this level for increasing quite slowly but constantly starting from a level of about 70. At a level of about 145, it stagnates for some time until it continues to grow after a single eye-catching peak which reaches the optimum fitness at once but loses it as quickly as it got there. Considering the relative smoothness of the rest of the curve, this peak is really astonishing. It cannot be the result of an ingenious

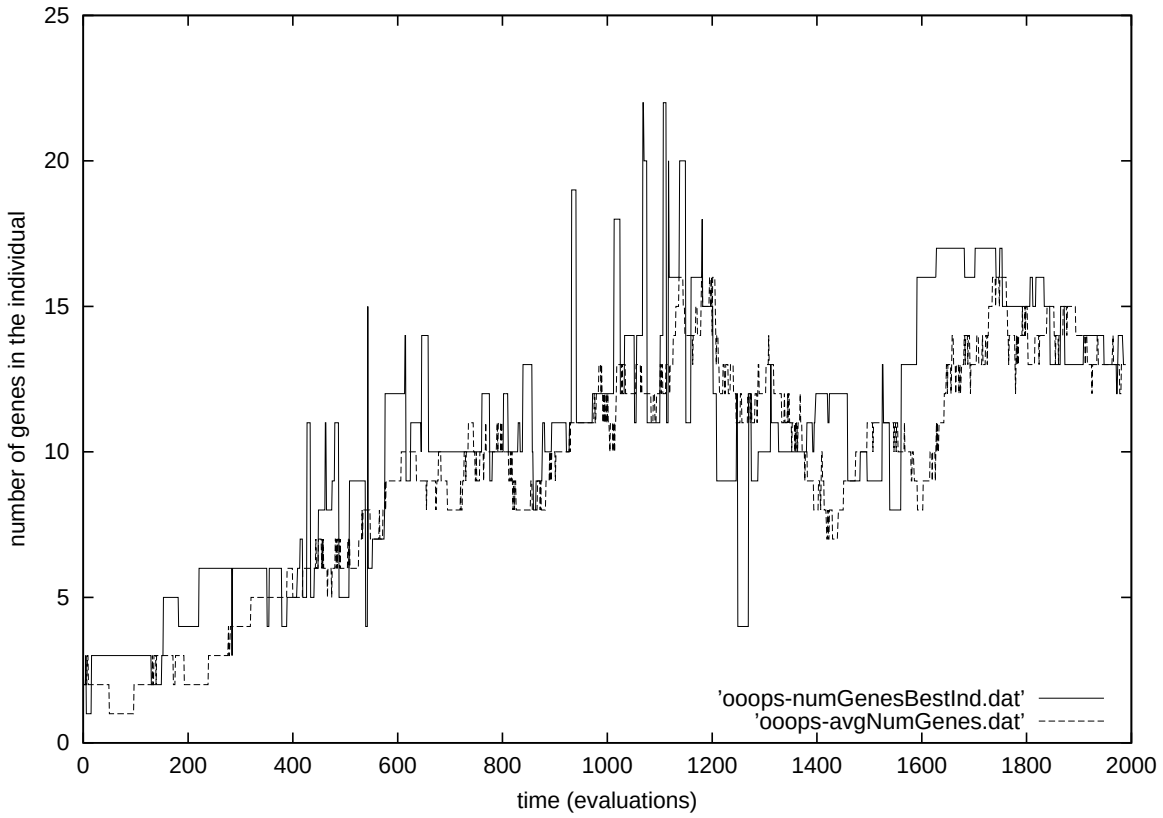

Figure 5.4: The dynamics of the number of genes in the third experiment. (1 generation $\hat{=}$ 20 evaluations)

improvement of the genome for spreading the news about the defect between the cells, because this improvement would have stayed in the population as the individual was part of the elite. Perhaps, there was a coincidence in the update order that helped in unsetting all signals. Or there was an error in the program that resulted in the removal of all cells instead of only the small block. Both explanations are not very convincing, but in any case this particularity does not seem to be important because it does not have much if any influence on the overall development of best and average fitness in the population.

The problem modelled in the third experiment seems to be much more difficult to solve than the other two. Firstly, it takes more than 2000 evaluations (or 100 generations) to reach the optimum fitness. Secondly, the best fitness curve does not stay at different levels that can be explained by simple problem solving strategies. Instead, the best fitness quite constantly (with the deviations caused by random influence) grows by improving the strategy to spread the news of the defect between the remaining cells.

There are some other evolutionary dynamics at which we will take a short look. We will only do this for the third experiment as it is the most interesting of the three.

Figure 5.4 shows the development of the number of genes per individual during the evolutionary run. This graph is quite interesting, because you can see that the number of genes in the individuals slowly grows until it reaches a maximum after about 1100 evaluations. Then, it starts to vary around a number of about 13 genes until the end of the run. Though the

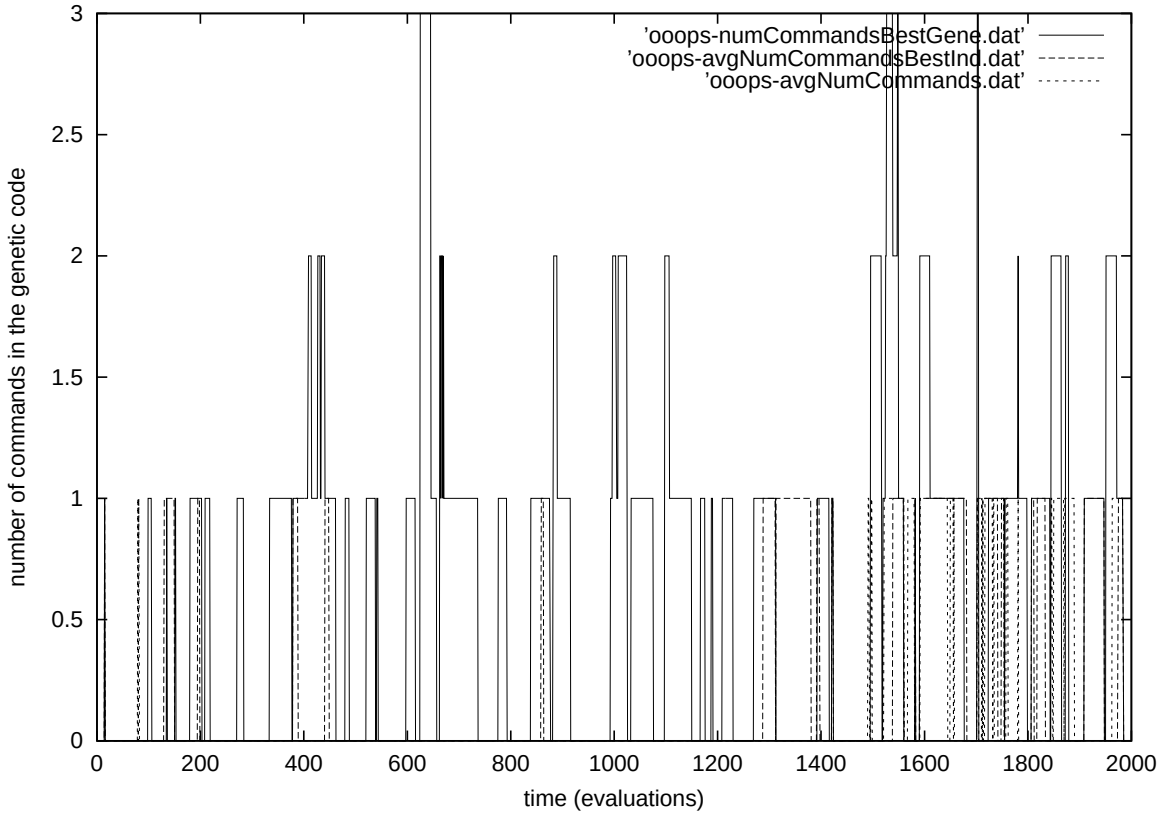

Figure 5.5: The dynamics of the number of commands in the evolvable code of the genes in the third experiment. (1 generation $\hat{=}$ 20 evaluations)

deviations are large, it seems as if this number is a good value for ensuring the functioning of the cells. The number of genes in the best individual of the population (`numGenesBestInd`) is most of the time approximately equal to the average number of genes in all individuals of the population (`avgNumGenes`). The value of the best individual only changes more often and shows many peaks that leave the average into both directions. In most peaks however, `numGenesBestInd` has a higher value than `avgNumGenes`. This also holds for some longer periods where the two values differ from each other.

If you look a little closer, you will notice another interesting detail: The best individual nearly always determines the direction of the development of the number of genes. When `numGenesBestInd` shows many peaks to higher values, the development goes towards more genes per individual. When the peaks point into the other direction, the average number of genes decreases. This can be explained by the fact, that the best individual frequently reproduces and thereby puts many new individuals into the population which contain a similar number of genes.

Another astonishing fact is, that exactly at the same time as when the fitness reaches the single eye-catching peak to the optimum, the number of genes in the best individual reaches an eye-catching peak to a low value of only four genes. This is interesting, because it seems to show that a number of about 13 genes is not really necessary for reaching the optimum fitness. Still, it is puzzling that not only the fitness of this best individual was unusual but

also its number of genes shows an astonishing peak.

Figure 5.5 depicts the dynamics of another part of the genetic parameters during the third evolutionary run: the number of commands used in the evolvable code of the genes. This graph shows the development of three parameters: the number of commands in the best gene, the average number of commands in all genes of the best individual and the average number of commands in all genes. The only parameter that from time to time reaches a value of more than 1 is the number of commands in the best gene. This is also the only parameter that is not an average which explains its larger deviations. Even though the frequency of higher values seems to increase after about 1500 evaluations, the plot shows no real tendency of the number of commands per gene to increase. The average stays below 1 most of the time. Perhaps, the growing frequency of higher values is already a harbinger of a coming strong code elongation (called *bloat* in GP) after the optimum has been reached. As only the last command of a gene has any influence on the result in this setup, putting more commands into the genetic code would be a possible strategy for the individuals and genes to protect themselves against mutation[2].

At last, we will have a look at the "result" of the evolutionary run: the best individual. The following list shows all the variable parameters of the 18 genes of this individual. You might wonder why there are 18 genes even though figure 5.4 only showed 13 genes for the best individual at the end of the run. This is because the run was actually stopped only shortly after the 2000th evaluation. The graphs were simply cut before that for better fitting into the scales. In the following list of genes of the best individual, the first line always shows the genetic code, the second line describes the message production in the format (type, intensity) and then follow the requirements and inhibitors in the same format:

- no code
  (2, 4.912)
  require: (1, 3.425)
  inhibit: (4, 15.634)

- signal = 0; signal = 0; signal = 0;
  (3, 9.300)
  require: (4, 3.094) (3, 12.641)
  inhibit: (1, 12.899)

- signal = 0;
  (3, 4.890)
  require: (4, 3.094) (3, 12.641)
  inhibit: (3, 18.828)

- signal = 0;
  (3, 4.358)
  require: (4, 3.094) (3, 12.641)

- signal = 0; signal = 1; signal = 0;
  (3, 5.820)
  inhibit: (4, 15.634) (2, 4.299)

[2]This common behaviour of genetic programming systems to produce bloat at the end of the evolutionary run is discussed in more detail for example in [Banzhaf et al., 1998].

- signal = 1; signal = 0;
  (4, 2.580)
  inhibit: (4, 3.785) (3, 10.868) (1, 8.786)
- no code
  (2, 4.912)
  require: (4, 14.746) (1, 3.425)
- signal = 1;
  (3, 4.912)
  require: (1, 3.425)
  inhibit: (4, 14.950)
- signal = 1;
  (4, 2.580)
  require: (4, 9.376)
  inhibit: (4, 3.785) (3, 10.868) (1, 8.786)
- signal = 0;
  (3, 2.008)
  require: (3, 16.848) (1, 9.692)
  inhibit: (3, 2.688)
- signal = 0; signal = 0;
  (2, 6.496)
  inhibit: (2, 14.102)
- signal = 1;
  (4, 6.638)
  require: (2, 11.240)
  inhibit: (4, 15.247)
- no code
  (1, 3.275)
  require: (4, 13.831) (3, 16.537) (2, 8.389) (1, 4.012)
  inhibit: (4, 2.470) (2, 15.005) (1, 0.886)
- signal = 0; signal = 0;
  (3, 5.820)
  inhibit: (2, 4.299) (4, 15.634)
- signal = 0;
  (3, 4.890)
  require: (4, 3.094) (3, 12.641)
- signal = 1; signal = 0;
  (4, 2.580)
  inhibit: (4, 3.785) (3, 10.868) (1, 8.786)
- signal = 1;
  (1, 1.473)
  inhibit: (1, 7.800) (2, 18.160) (4, 15.247)

- signal = 0; signal = 0;
  (2, 9.824)
  inhibit: (2, 14.102)

In this example you can easily see the attribute of object-oriented ontogenetic programs that their functioning is very difficult to analyse and understand for humans, because firstly they include very complex interactions and secondly there are no parts that represent any separable meaning. This is a subsymbolic representation as discussed in section 4.3. Without much more detailed analysis, it is not possible to understand how this individual solves the given problem. This was not the goal of the here presented work. Without further analysis, we can conclude from the fact that the individual reaches a high fitness value, that it solves the modelled problem, no matter how. This was all that I wanted to demonstrate.

5.3 Conclusion

The three presented experiments show that OOOPS is able to quite quickly (in about 100 generations for a population size of only 20 individuals, which took one day on a single 1000 MHz pentium computer) breed multicellular programs that can solve computational problems which seem to be reasonable models for a complex real-life task. In combination with the clues from the natural example and from other publications as discussed in the previous chapters, this can be seen as a good indication of the utility of the Object-Oriented Ontogenetic Programming Paradigm and the correct functioning of the Object-Oriented Ontogenetic Programming System. The experiments and arguments make expect interesting results in future analyses and applications. But that is all they can serve for. These simple experiments with little analysis can of course not in any way validate all the claims of the previous chapters. Nor can they give a reasonable impression about the real capabilities of the system and the underlying new methods. There is a vast possibility and necessity for further experiments and deeper analysis which is the project for the future.

Bibliography

[Abelson et al., 1999] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F. Knight Jr., Radhika Nagpal, Erik Rauch, Gerald Jay Sussman, and Ron Weiss. Amorphous computing. Technical report, MIT Artificial Intelligence Laboratory, 1999.

[AI Depot, 2001] AI Depot. Artificial intelligence. `http://ai-depot.com/`, 2001.

[Altenberg, 1995] Lee Altenberg. Genome growth and the evolution of the genotype-phenotype map. In Wolfgang Banzhaf and Frank H. Eeckman, editors, *Evolution and Biocomputation: Computational Models of Evolution*, pages 205–259. Springer Verlag, 1995.

[Angeline, 1997] Peter J. Angeline. Subtree crossover: Building block engine or macromutation. In John Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 9–17. Morgan Kaufmann Publishers, 1997.

[Astor and Adami, 1998] Jens C. Astor and Christoph Adami. Development and evolution of neural networks in an artificial chemistry. In Claus Wilke, Stephan Altmeyer, and Thomas Martinetz, editors, *Proceedings of the Third German Workshop on Artificial Life*, pages 15–30. Verlag Harri Deutsch, 1998.

[Banzhaf, 1994] Wolfgang Banzhaf. Genotype-phenotype-mapping and neutral variation – a case study in genetic programming. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Männer, editors, *Parallel Problem Solving from Nature III*, volume 866 of *Lecture Notes in Computer Science*, pages 322–332. Springer Verlag, 1994.

[Banzhaf et al., 1998] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction: On the Automatic Evolution of Computer Programs and Its Applications.* Morgan Kaufmann Publishers, 1998.

[Bellman, 1978] Richard E. Bellman. *An Introduction to Artificial Intelligence: Can Computers Think?* Boyd & Fraser Publishing Company, 1978.

[Benzer, 1957] Seymour Benzer. The elementary units of heredity. In William D. McElroy and Bently Glass, editors, *A Symposium on The Chemical Basis of Heredity*, pages 70–93. Johns Hopkins University Press, 1957.

[Bonabeau et al., 1999] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems.* Oxford University Press, 1999.

[Bond and Gasser, 1988] Alan H. Bond and Les Gasser. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann Publishers, 1988.

[Brooks, 1992] Rodney A. Brooks. Artificial life and real robots. In Francisco J. Varela and Paul Bourgine, editors, *Towards a practice of autonomous systems: Proceedings of the first European Conference on Artificial Life (ECAL 91)*, pages 3–10. MIT Press, 1992.

[Bull, 1997] Larry Bull. On the evolution of multicellularity. In Phil Husbands and Inman Harvey, editors, *Proceedings of the Fourth European Conference on Artificial Life, ECAL 1997*. MIT Press, 1997.

[Collaborative Research Center Computational Intelligence, 2002] Collaborative Research Center Computational Intelligence. About CI. University of Dortmund. `http://sfbci.cs.uni-dortmund.de/home/English/CI/top.html`, 2002.

[Colorni et al., 1992] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. An investigation of some properties of an "ant algorithm". In Reinhard Männer and Bernard Manderick, editors, *Proceedings of the Second Conference on Parallel Problem Solving from Nature (PPSN 92)*, pages 509–520. Elsevier Publishing, 1992.

[Darwin, 1859] Charles Darwin. *On the Origin of Species*. John Murray, 1859.

[Deneubourg and Goss, 1989] Jean-Louis Deneubourg and Simon Goss. Collective patterns and decision-making. *Ecology, Ethology and Evolution*, 1:295–311, 1989.

[Descartes, 1996] René Descartes. *Meditations on First Philosophy*. Cambridge University Press, 1996.

[D'haeseleer, 1994] Patrik D'haeseleer. Context preserving crossover in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 256–261. IEEE Press, 1994.

[Dictionary of Philosophy of Mind, 2001] Dictionary of Philosophy of Mind. Distributed representation. `http://www.artsci.wustl.edu/~philos/MindDict/distributedrepresentation.html`, 2001.

[Dittrich et al., 2001] Peter Dittrich, Jens Ziegler, and Wolfgang Banzhaf. Artificial chemistries - a review. *Artificial Life*, 7(3):225–275, 2001.

[Eggenberger, 1996] Peter Eggenberger. Cell interactions as a control tool of developmental processes for evolutionary robotics. In Pattie Maes, Maja J. Mataric, Jean-Arcady Meyer, Jordan Pollack, and Stewart W. Wilson, editors, *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, pages 440–448. MIT Press, 1996.

[Eggenberger, 1997] Peter Eggenberger. Evolving morphologies of simulated 3d organisms based on differential gene expression. In Phil Husbands and Inman Harvey, editors, *Proceedings of the Fourth European Conference on Artificial Life, ECAL 1997*, pages 205–213. Springer Verlag, 1997.

[Eggenberger and Dravid, 1999] Peter Eggenberger and Raja Dravid. An evolutionary approach to pattern formation mechanisms on lepidopteran wings. In *Proceedings of the 1999 Congress on Evolutionary Computation*, volume 1, pages 470–473. IEEE Press, 1999.

[Elman et al., 1996] Jeffrey L. Elman, Elizabeth A. Bates, Mark H. Johnson, Annette Karmiloff-Smith, Domenico Parisi, and Kim Plunkett. *Rethinking Innateness: A Connectionist Perspective on Development*. MIT Press, 1996.

[Evans, 2000] David Evans. Programming the swarm. NSF Proposal. `http://swarm.cs.virginia.edu/nsf-proposal.pdf`, 2000.

[Ferreira, 2001] Cândida Ferreira. Gene expression programming: A new adaptive algorithm for solving problems. *Complex Systems*, 13(2):87–129, 2001.

[Fogel et al., 1965] Lawrence J. Fogel, Alvin J. Owens, and Michael J. Walsh. Artificial intelligence through a simulation of evolution. *Biophysics and Cybernetic Systems*, pages 131–155, 1965.

[Friedberg, 1958] R M. Friedberg. A learning machine - part I. *IBM Journal of Research and Development*, 2(1):2–11, 1958.

[Furusawa and Kaneko, 1997] Chikara Furusawa and Kunihiko Kaneko. Emergence of differentiation rules leading to hierarchy and diversity. In Phil Husbands and Inman Harvey, editors, *Proceedings of the Fourth European Conference on Artificial Life*, pages 172–181. MIT Press, 1997.

[Furusawa and Kaneko, 1998] Chikara Furusawa and Kunihiko Kaneko. Emergence of multicellular organisms with dynamic differentiation and spatial pattern. In Christoph Adami, Richard K. Belew, Hiroaki Kitano, and Charles E. Taylor, editors, *Proceedings of the Sixth International Conference on Artificial Life*, pages 43–52. MIT Press, 1998.

[Gilbert, 1994] Scott F. Gilbert. *Developmental Biology*. Sinauer Associates, 4th edition, 1994.

[Gruau, 1994] Frederic Gruau. Genetic micro programming of neural networks. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, pages 495–518. MIT Press, 1994.

[Hart et al., 1995] William E. Hart, Thomas E. Kammeyer, and Richard K. Belew. The role of development in genetic algorithms. In Darrell L. Whitley and Michael D. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 315–332. Morgan Kaufmann Publishers, 1995.

[Haynes et al., 1995] Thomas Haynes, Sandip Sen, Dale Schoenefeld, and Roger Wainwright. Evolving multiagent coordination strategies with genetic programming. Technical Report UTULSA-MCS-95-04, The University of Tulsa, 1995.

[Hemelrijk, 1997] Charlotte K. Hemelrijk. Cooperation without genes, games or cognition. In Phil Husbands and Inman Harvey, editors, *Proceedings of the Fourth European Conference on Artificial Life*, pages 511–520. MIT Press, 1997.

[Hiebeler, 1994] David Hiebeler. The swarm simulation system and individual-based modeling. In J.M. Power, M. Strome, and T.C. Daniel, editors, *Decision Support 2001. 17th Annual Geographic Information Seminar and the Resource Technology '94 Symposium*, pages 474–494. American Society for Photogrammetry and Remote Sensing, 1994.

[Hirsch-Kauffmann and Schweiger, 1996] Monica Hirsch-Kauffmann and Manfred Schweiger. *Biologie für Mediziner und Naturwissenschaftler*. Georg Thieme Verlag, 3. edition, 1996.

[Hoffmeyer, 1997] Jesper Hoffmeyer. The swarming body. In Irmengard Rauch and Gerald F. Carr, editors, *Semiotics Around the World. Proceedings of the Fifth Congress of the International Association for Semiotic Studies*, pages 937–940. Mouton de Gruyter, 1997.

[Hoile and Tateson, 2000] Cefn Hoile and Richard Tateson. Design by morphogenesis. In Mark A. Bedau, John S. McCaskill, Norman H. Packard, and Steen Rasmussen, editors, *Proceedings of the 7th International Conference on Artificial Life*, pages 141–145. MIT Press, 2000.

[Holland, 1975] John Holland. *Adaptation in natural and artificial systems*. MIT Press, 1975.

[International Human Genome Sequencing Consortium, 2001] International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, (409):860–921, 2001.

[International Journal of Computational Intelligence and Applications, 2001] International Journal of Computational Intelligence and Applications. `http://www.worldscinet.com/ijcia/ijcia.html`, 2001.

[Internet Encyclopedia of Philosophy, 2001] Internet Encyclopedia of Philosophy. Artificial intelligence. `http://www.utm.edu/research/iep/a/artintel.htm`, 2001.

[Kaneko and Yomo, 2000] Kunihiko Kaneko and Tetsuya Yomo. Sympatric speciation from interaction-induced phenotype differentiation. In Mark A. Bedau, John S. McCaskill, Norman H. Packard, and Steen Rasmussen, editors, *Proceedings of the 7th International Conference on Artificial Life*, pages 113–121. MIT Press, 2000.

[Kant, 1781] Immanuel Kant. *Kritik der reinen Vernunft*, 1781.

[Keller and Kangas, 2001] Paul E. Keller and Lars J. Kangas. Cognitive Systems at Pacific Northwest National Laboratory. `http://www.emsl.pnl.gov:2080/proj/neuron/`, 2001.

[Keller and Banzhaf, 1996] Robert Keller and Wolfgang Banzhaf. Genetic programming using genotype–phenotype mapping from linear genomes into linear phenotypes. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 116–122. MIT Press, 1996.

[Kim, 2001] Jan T. Kim. Transsys: A generic formalism for modelling regulatory networks in morphogenesis. In Jozef Kelemen and Petr Sosík, editors, *Proceedings of the 5th European Conference on Artificial Life, ECAL 2001*, volume 2159 of *Lecture Notes in Computer Science*, pages 242–251. Springer Verlag, 2001.

[Kitano et al., 1997] Hiroaki Kitano, Shugo Hamahashi, Jun Kitazawa, Koji Takao, and Shin ichirou Imai. The virtual biology laboratories: A new approach of computational biology. In Phil Husbands and Inman Harvey, editors, *Proceedings of the Fourth European Conference on Artificial Life*, pages 274–283. MIT Press, 1997.

[Korf, 1992] Richard E. Korf. A simple solution to pursuit games. In *Working Papers of the 11th International Workshop on Distributed Artificial Intelligence*, pages 183–194, 1992.

[Koza, 1992] John Koza. *Genetic Programming: On the Programming of Computers by Natural Selection.* MIT Press, 1992.

[Koza, 1994] John Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, 1994.

[Koza et al., 1996] John R. Koza, Forrest H. Bennett III, David Andre, and Martin A. Keane. Automated design of both the topology and sizing of analog electrical circuits using genetic programming. In John S. Gero and Fay Sudweeks, editors, *Artificial Intelligence in Design '96*, pages 151–170. Kluwer Academic Publishers, 1996.

[Koza et al., 1999] John R. Koza, Forrest H. Bennett III, David Andre, and Martin A. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving.* Morgan Kaufmann Publishers, 1999.

[Kuncheva and Porter, 2001] Ludmila Kuncheva and Tim Porter. ICSC Congress "Computational Intelligence: Methods and Applications". `http://www.icsc-naiso.org/conferences/cima2001/`, 2001.

[Lang, 1995] Kevin J. Lang. Hill climbing beats genetic search on a boolean circuit synthesis of Koza's. In Armand Prieditis and Stuart J. Russell, editors, *Proceedings of the 12th International Conference on Machine Learning*, pages 340–343. Morgan Kaufmann Publishers, 1995.

[Liang et al., 1998] Shoudan Liang, Stefanie Fuhrmann, and Roland Somogyi. Reveal, a general reverse engineering algorithm for inference of genetic network architectures. *Pacific Symposium on Biocomputing*, 3:18–29, 1998.

[Luke et al., 1999] Sean Luke, Shugo Hamahashi, and Hiroaki Kitano. "genetic" programming. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the 1999 Genetic and Evolutionary Computation Conference (GECCO 99)*, pages 1098–1105. Morgan Kaufmann Publishers, 1999.

[Luke and Spector, 1996] Sean Luke and Lee Spector. Evolving teamwork and coordination with genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 150–156. MIT Press, 1996.

[Margulis, 1981] Lynn Margulis. *Symbiosis in cell evolution.* W. H. Freeman and Company, 1981.

[Masum and Oppacher, 2001] Hassan Masum and Franz Oppacher. Regulatory networks and genomic algorithms. In Nagib Callaos, Ivan Nunes da Silva, and Jorge Molero, editors, *Proceedings of the World Multiconference on Systemics, Cybernetics and Informatics 2001 (SCI 2001/ISAS 2001)*, volume III. IIIS, 2001.

[Mataric, 1995] Maja Mataric. Designing and understanding adaptive group behavior. *Adaptive Behavior*, 4:51–80, 1995.

[Mataric, 1997] Maja Mataric. Learning social behaviors. *Robotics and Autonomous Systems*, pages 191–204, 1997. Special Issue on "Practice and Future of Autonomous Agents".

[Maynard Smith and Szathmáry, 1995] John Maynard Smith and Eörs Szathmáry. *The Major Transitions in Evolution*. Oxford University Press, 1995.

[McCarthy, 2001] John McCarthy. What is artificial intelligence? `http://www-formal.stanford.edu/jmc/whatisai.html`, 2001.

[Meinhardt, 1995] Hans Meinhardt. *The Algorithmic Beauty of Sea Shells*. Springer Verlag, 1995.

[Mitchell, 1996] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[Moriarty and Miikkulainen, 1997] David E. Moriarty and Risto Miikkulainen. Forming neural networks through efficient and adaptive coevolution. *Evolutionary Computation*, 5(4):373–399, 1997.

[Nagpal, 1999] Radhika Nagpal. Self-organizing a global coordinate system from local information. MIT AI Memo 1666, 1999.

[Nordin et al., 1996] Peter Nordin, Frank D. Francone, and Wolfgang Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In Peter J. Angeline, editor, *Advances in Genetic Programming 2*, chapter 6, pages 111–134. MIT Press, 1996.

[Pfeifer et al., 2000] Rolf Pfeifer, Hanspeter Kunz, and Marion Weber. Artificial life. Lecture notes. `http://www.ifi.unizh.ch/groups/ailab/teaching/AL00.html`, 2000.

[Plato, 1901] Plato. *The Republic*. P.F. Collier & Son, 1901.

[Popper, 1935] Karl Popper. *Logik der Forschung*. Julius Springer Verlag, 1935.

[Potter and De Jong, 2000] Mitchell A. Potter and Kenneth A. De Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8(1):1–29, 2000.

["Programming the Swarm" Project,] "Programming the Swarm" Project. University of Virginia. `http://swarm.cs.virginia.edu`.

[Pschyrembel, 1998] Willibald Pschyrembel, editor. *Klinisches Wörterbuch*. Walter de Gruyter Verlag, 258. edition, 1998.

[Rechenberg, 1970] Ingo Rechenberg. *Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. PhD thesis, Technical University of Berlin, 1970.

[Reil, 1999] Torsten Reil. Dynamics of gene expression in an artificial genome - implications for biological and artificial ontogeny. In Dario Floreano, Francesco Mondada, and Jean-Daniel Nicoud, editors, *Proceedings of the 5th European Conference on Artificial Life, ECAL 1999*, pages 457–466. Springer Verlag, 1999.

[Reynolds, 1987] Craig W. Reynolds. Flocks, herds, and schools: a distributed behavioral model. *Computer Graphics*, 21:25–34, 1987.

[Rosca and Ballard, 1994] Justinian P. Rosca and Dana H. Ballard. Hierarchical self-organization in genetic programming. In William W. Cohen and Haym Hirsh, editors, *Proceedings of the Eleventh International Conference on Machine Learning*, pages 251–258. Morgan Kaufmann Publishers, 1994.

[Rotman, 2000] David Rotman. Molecular computing. *Technology Review*, 2000. `http://www.techreview.com/articles/rotman0500.asp`.

[Russell and Norvig, 1994] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1994.

[Schmutter, 2000a] Peter Schmutter. Artificial Embryology (Arbeiten von Peter Eggenberger). `http://www.ps-it.com/publications/index.html`, 2000.

[Schmutter, 2000b] Peter Schmutter. Eine Einführung in Multiagentensysteme und deren Verwirklichung im Roboterfußball. `http://www.ps-it.com/publications/index.html`, 2000.

[Schopenhauer, 1819] Arthur Schopenhauer. *Die Welt als Wille und Vorstellung*, 1819.

[Schwefel, 1965] Hans-Paul Schwefel. *Kybernetische Evolution als Strategie der experimentellen Forschung in der Strömungstechnik*. Diploma thesis, Technical University of Berlin, 1965.

[Sipper, 1997] Moshe Sipper. *Evolution of Parallel Cellular Machines: The Cellular Programming Approach*. Springer Verlag, 1997.

[Sloman, 1998] Aaron Sloman. What is artificial intelligence? `http://www.cs.bham.ac.uk/~axs/misc/aiforschools.html`, 1998.

["Smart Dust" Project,] "Smart Dust" Project. University of California, Berkeley. `http://robotics.eecs.berkeley.edu/~pister/SmartDust`.

[Smith, 1996] Peter W. H. Smith. Conjugation – a bacterially inspired form of genetic recombination. In John R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1996 Conference*, pages 167–176. Stanford Bookstore, 1996.

[Sober, 1992] Elliott Sober. Learning from functionalism—prospects for strong artificial life. In Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, volume 10 of *SFI Studies in the Science of Complexity*, pages 749–765. Addison Wesley, 1992.

[Spector, 1996] Lee Spector. Simultaneous evolution of programs and their control structures. In Peter J. Angeline and Kenneth E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, pages 137–154. MIT Press, 1996.

[Spector, 2001] Lee Spector. Autoconstructive evolution: Push, pushgp, and pushpop. In Lee Spector, Erik Goodman, Annie Wu, Bill Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *GECCO-2001: Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann Publishers, 2001.

[Spector and Stoffel, 1996] Lee Spector and Kilian Stoffel. Ontogenetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 394–399. MIT Press, 1996.

[Stephens and Merx, 1990] Larry M. Stephens and Matthias B. Merx. The effect of agent control strategy on the performance of a dai pursuit problem. In *Proceedings of the 1990 Distributed AI Workshop*, 1990.

[Stewart, 2000] John Stewart. *Evolution's Arrow*, chapter 3–5. The Chapman Press, 2000. Also published on `http://www4.tpg.com.au/users/jes999/index.htm`.

[Stone and Veloso, 2000] Peter Stone and Manuela Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383, 2000.

[Stottler, 1999] Richard Stottler. `http://www.shai.com/ai_general/quotations.htm`, 1999.

[Turing, 1950] Alan Turing. Computing machinery and intelligence. *Mind*, 59:433–460, 1950.

[Ultralow Power Wireless Sensor Project,] Ultralow Power Wireless Sensor Project. Massachussetts Institute of Technology. `http://www-mtl.mit.edu/~jimg/project_top.html`.

[von Neumann, 1966] John von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966. Edited and completed by A. W. Burks.

[Webopedia, 2002] Webopedia. Artificial Intelligence. `http://www.webopedia.com/TERM/a/artificial_intelligence.html`, 2002.

[Weiss, 1999] Gerhard Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 1999.

[Whatis.com, 2001] Whatis.com. Artificial Intelligence. `http://searchEBusiness.techtarget.com/sDefinition/0,,sid19_gci211597,00.html`, 2001.

[Winston, 1992] Patrick Henry Winston. *Artificial Intelligence*. Addison Wesley, 3rd edition, 1992.

[Wireless Integrated Network Sensors (WINS) Project,] Wireless Integrated Network Sensors (WINS) Project. University of California, Los Angeles. `http://www.janet.ucla.edu/WINS`.

[World Congress on Computational Intelligence, 2002] World Congress on Computational Intelligence. `http://www.wcci2002.org/`, 2002.

Index

This index is supposed to be used as a glossary and therefore only contains the occurences at which the technical terms are explained. If there is more than one page noted for a term, this term is used in different contexts or both passages are important for understanding it.