Offline Emergence Engineering For Agent Societies

Michael Zapf and Thomas Weise

Distributed Systems Group, Kassel University,
Wilhelmshöher Allee 73, 34121 Kassel, Germany
{zapf|weise}@vs.uni-kassel.de

Abstract. Many examples for emergent behaviors may be observed in self-organizing physical and biological systems which prove to be robust, stable, and adaptable. Such behaviors are often based on very simple mechanisms and rules, but artificially creating them is a challenging task which does not comply with traditional software engineering. In this article, we propose a hybrid approach by combining strategies from Genetic Programming and agent software engineering, and demonstrate that this approach effectively yields an emergent design for given problems.

Presented at EUMAS'07.
Fifth European Workshop on Multi-Agent Systems.
Hammamet, Tunesia
December 13-14, 2007
http://www.atia.rnu.tn/eumas/

1 Introduction

Emergence is defined as the appearance of properties in a system, created by a possibly large number of individuals. These individuals all have some observable behavior, but it cannot easily be explained how these individual behaviors actually contribute to the behavior of the whole group. Emergent behaviors that have been extensively studied often turn out to be based on very few, very simple rules, and they often become manifest in self-organizing systems. One example from biology are schools of fish, which form as a means of protection against predators. These swarms seem as if they had some central control, but in fact, the movement of the swarm is determined by the movement of the individuals. Such flocking behaviors, for instance, only need the individuals to comply with three rules:

- If you are too close to your neighbors, increase the distance.
- Steer towards the average heading of your neighbors.
- Steer towards the average position of your neighbors.

With these simple rules such schools of fish convincingly demonstrate the robustness and self-healing capabilities of self-organizing systems. A school of fish will maintain its formation, even if multiple fish are removed, join in, or if its general heading suddenly changes. Still, one may wonder how the above rules lead to such a stable system, and this is commonly considered a good example of an emergent phenomenon.[1,2]

Understanding the mutual influences within swarms is still a challenge, even though such phenomena are examined for a long time. In computer science, self-organizing systems have been a hot research topic in the recent years, and autonomous agents have often successfully been employed as key building blocks for their implementation. Moreover, especially in the context of Ambient Intelligence and adaptive systems, we see a rise of adaptive agent systems, which are designed to host agents capable of adapting to the environment. We have been exploring such adaptive systems in the IST projects MADAM[1] and its successor MUSIC[2].

The term *Emergence Engineering*, as a new idea of software engineering using emergent phenomena, sounds like an oxymoron: One one hand, emergence is characterized by our inability to deduce the group behavior from the individual behavior. On the other hand, engineering – as understood until now – is an application of the classic *divide-and-conquer* strategy: We decompose a large problem in several sub-problems, trying to solve each of them separately, and assemble the overall solution from the partial solutions. That is, we know very well how the partial solutions contribute to the overall solution.

Large-scale self-organizing systems, usually modeled by a set of autonomous agents, impose new challenges for program design. The mere size of such systems requires methods which extend beyond the classic design. Emergence engineering could be an interesting approach to self-organizing systems, but a common understanding how to proceed on these new paths is still to be found.

One specific challenge is the complexity of the environment for each agent. The environment not only depends on the configuration of accessible resources but also on the rest of the agent community. A promising idea is to design systems which constantly attempt to adapt to the environment in order to maintain their functionality. This can be achieved by explicitly designing *adaptive agents* which compose the overall system. We basically find two approaches here:

- Include specific code for all adaptation situations. This leads to very heavy agents in terms of code lines. Still, their behavior is often fragile when it comes to unforeseen situations.
- Allow agents to set up their behavior in response to local adaptations. Agents know nothing about the "big picture", but try to mutually cooperate as well as possible. However, it is unclear when – if at all – some cooperative behavior appears.

[1] http://www.ist-madam.org/
[2] http://www.ist-music.eu/

While the first case does not make use of emergence, the second case utilizes it as part of the productive execution phase. Agents are started and expected to settle on some group behavior, hopefully the one which was desired. We call this *online emergence engineering*, because emergence is planned to occur during the execution.

In this paper, we present another approach which we call *offline emergence engineering*. This approach allows agents to let some group behavior emerge *before* they are actually put into the real environment. Thus, we may give the emergent process enough time to reach an acceptable solution, preventing a potentially harmful effect on the environment in early phases. There exists a vast amount of other instances of emergence in evolutionary biology [3], leading us to argue that emergent behavior can be synthesized by using the same mechanisms that drive the natural evolution. The core of our approach is Genetic Programming, a member of the family of evolutionary algorithms.

This article is structured as follows: In Section 2, we discuss emergence in general before elaborating on how emergent behavior can be evolved with Genetic Programming. We illustrate our approach by means of the well-known load-balancing problem in Section 3. Some links to related work are presented in Section 4, and we finally conclude the paper in Section 5.

2 Emergence and Genetic Programming

At first we give some background on our view of emergence and how Genetic Programming may contribute to Emergence Engineering.

2.1 Emergence

In the sense that we (and many other authors, see for instance [4]) use the term *Emergence* within, a system with emergent properties has identifiable individuals which show some behavior that can be characterized by specific (possibly multiple) degrees of freedom. For instance, mobile agents are able to relocate themselves on the network; communicative agents negotiate some trade, choosing from a set of communication forms, performatives, or protocols. Emergent properties need to be persistent and reproducible, so that one can reason about them. Even a dynamic equilibrium is, as such, a persistent property, although the individuals do not behave statically. Therefore, any emergent property of a collection of individuals should be assumed to be persistent, at least for a given period of time.

Any persistent property is a manifestation of some order, limiting the entropy of the system. We do not expect that by simply exploiting all degrees of freedom, the society of individuals will create a stable macroscopic configuration. We distinguish between the macroscopic level, related to the group of individuals, and the microscopic level, related to the individuals. Only if there is some counterforce coming from the macroscopic level, affecting the microscopic level, there is a chance that a system state may *emerge* which presents a perceivable property.

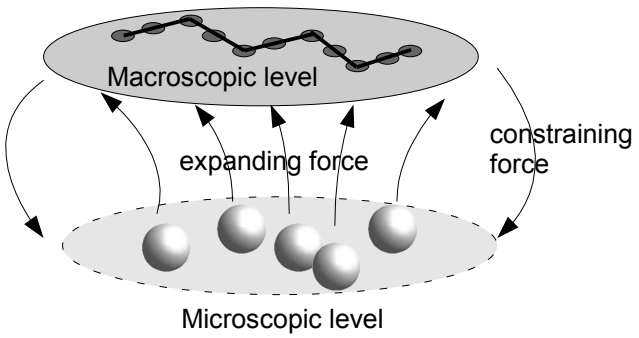


Fig. 1. Macro-micro level interaction, leading to emergent properties

Figure 1 shows the common understanding of the microscopic and macroscopic level interaction in a scenario exhibiting emergent behavior.

Approaching emergence, according to Figure 1, we have to consider three basic aspects:

1. An agent, by exploiting its freedom, must have some noticeable effect on the overall configuration. For instance, by migrating, it may cause another agent to leave this place if just one agent is expected to be present; or previous partner agents may now fail to keep up their communication channels.
2. The macro level must be capable of displaying a perceivable, persistent property.
3. The macro level must have some noticeable influence on each individual agent. In order to let some persistent property appear, this influence must constrain the agent's freedom. Creating a group, for example, may have some advantage for agents (characterized by mutual reachability or access) and leaving this group may have an adversarial effect on reaching the goal. Here, the group formation (or swarm, if created by a large number of individuals) is the actual emerging property.

While agent technology may seem to be a natural choice for engineering emergent phenomena, emergence does not necessarily imply agent technology. It is not even required that the individuals will be identifiable, distinguishable entities in the application. We will comment on this in the next sections.

As the current state-of-the-art in software engineering involves an analysis based on decomposing a complex problem, it remains unclear how emergence can be included in the software creation process. Whenever we think about solving a problem with an agent system, we intuitively consider sequences of actions which we expect the agent to perform, or we presume the agent will apply some techniques that we already know. Such implicit ideas exclude sets of behaviors that do not fit to our usual trains of thought, probably including many of the emergent algorithms.

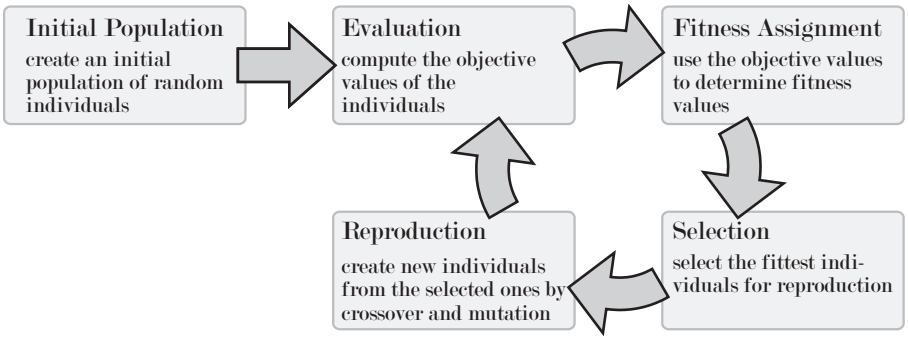


Fig. 2. The basic cycle of evolutionary algorithms

Autonomous agents, especially those designed with the BDI model in mind, are expected to process sensory input according to some given rules, trying to resolve open desires using various plans. Here, the problem arises where to plug in emergent behavior: How can the agent rationally decide on some action while not knowing the global goal, or how its actions contribute to the global goal?

Usually, agents with self-organizing capabilities are expected to develop emergent properties as a result of continuous adaptation. Our approach allows to build emergent behavior outside of the real environment. In order to make use of emergent behavior, it could be wise to consider creating the behavior in a simulation, and when that behavior proved to be appropriate, use it in the real environment.

2.2 Genetic Programming

Evolutionary algorithms (EA) [5,6,7] are generic, population-based meta-heuristic optimization algorithms that use biology-inspired mechanisms like mutation, crossover, natural selection, and survival of the fittest.

The advantage of evolutionary algorithms compared to other optimization methods is that they make only few assumptions about the underlying fitness landscape and therefore perform consistently well in many different problem categories. All evolutionary algorithms proceed in principle according to the scheme illustrated in Figure 2:

1. Initially, a population of individuals with a totally random genome is created.
2. All individuals of the population are tested. This evaluation may incorporate complicated simulation and calculations.
3. With the tests, we have determined the utility of the different features of the solution candidates and can now assign a fitness value to each of them.
4. A subsequent selection process filters out the individuals with low fitness and allows those with good fitness to enter the mating pool with a higher probability.

5. In the reproduction phase, offspring is created by varying or combining these solution candidates, and is integrated into the population.
6. If the termination criterion is met, the evolution stops here. Otherwise, it continues at step 2.

Genetic Programming is a class of evolutionary algorithms for breeding programs, algorithms, and similar constructs [8,9]. The roots of Genetic Programming go back to Friedberg who used a learning algorithm to stepwise improve a fixed-size program in 1958 [10,11]. In the mid-1980s, Cramer utilized genetic algorithms and tree-like structures to evolve programs [12]. The standard tree-based Genetic Programming, which is most often used in practical applications and as reference model, was formalized by Koza a few years later [8]. Since then, many different approaches like grammar-guided Genetic Programming [13] and linear Genetic Programming [14] have branched off.[3]

During the course of our research we have applied different forms of Genetic Programming for deriving distributed algorithms [15,16,17]. In this paper, we take this idea a step further by using rule-based Genetic Programming [18] in order to obtain rules for creating a cooperative, adaptive, and robust behavior for multi-agent systems.

2.3 Genetic Programming and Emergence

The approach that we are about to describe may be called *offline emergence engineering*, due to the fact that the agent behavior actually emerges from Genetic Programming, but within a simulated environment *before* the agents are deployed in the real environment where they are supposed to run.

If we claim to use Genetic Programming for emergence engineering, we should be able to analyze this approach with respect to the comments in Section 2.1. A good starting point here is the natural evolution of species, which, as it turns out, is emergent by itself. Calvin describes it as *The River That Flows Uphill* in his book of the same title [3]. Over billions of years it created more and more complex life forms which contradicts the general tendency of the universe to maximize entropy. The same contradiction can be observed in Genetic Programming: from a set of randomly shaped programs in the initial population, step by step well-formed programs emerge.

Here, the individuals as shown in Section 1 are not different *agents* but rather the *programs* in the population. The expanding force (exploiting the degrees of freedom) on the micro level corresponds to the various kinds how they may be modified by the genetic operations *crossover* and *mutation*. The constraining forces on the macroscopic level are the *objective functions*, the *fitness assignment process* which computes a fitness value by combining the objective functions, and the *selection mechanism* which decides which individuals may reproduce on basis of their fitness. Eventually the Genetic Programming process converges to a state where the population is dominated by programs solving the specified

[3] A more thorough discussion of these different variants can be found in [7].

problem. Although mutation and crossover still carry on to produce new inferior individuals, their fraction in the population roughly remains constant since they cannot withstand the selection pressure. If a new superior species arises, the equilibrium will tip over, followed by short phase of mass extinction, until a new stable state is reached.

As a result, we obtain a program which serves as an evolved behavior for the autonomous agent. As this program has been grown within a simulated environment, a potentially long learning phase within the real environment is prevented. In addition, we can employ emergence engineering without constraints on the set of involved agents: The programs can be used for a set of collaborative agents, for separate agents, or possibly for just one agent in its environment. As our experiments show, systems of multiple evolved agents cooperate and reach a common goal where each agent exercises its degrees of freedom by the action parts of its rules. The agent selects a suitable rule according to its environment – including the other agents, their influence on the place where it is, and the communications received from them.

This approach currently targets at creating one behavior for one agent or agent type; if we plan to have different types of agents, we either need multiple, separate evolution processes (in which case we must make sure that any mutual influence in modeled adequately), or we need an evolution process which explicitly grows different kinds of individuals (*co-evolution).* In our experiments described below, we only require one agent type, appearing in multiple instances.

3 Offline Emergence Engineering

In order to explore this approach, we have applied Genetic Programming to an interesting aspect of distributed systems, the *load balancing* problem. In load balancing scenarios, tasks continuously enter a system consisting of multiple workstations. Each task needs a different amount of time to finish. The goal is to reduce the overall waiting time by distributing the workload equally between the stations. Traditional forms of load balancing are performed by a central instance assigning the tasks to the different processors; modern methods rely on decentralized cooperation of the workstations.

In our version of the load balancing scenario, we pack each task into a single agent who has to decide itself on which station it wants to run. The goal of this agent society is to evenly distribute its workload on all computers in a grid.

3.1 Requirements analysis

Before Genetic Programming may be applied to a given problem, we need a thorough requirements analysis. This analysis has two goals:

- Consider the operations needed in any possible solution.
- Reduce this set to a minimum.

This reduction helps to increase the probability of obtaining useful solutions. In our case described later, we wish to have mobile agents which can migrate to different workstations. They also need messaging capabilities for cooperation.

Here we should be cautious, for this is already a typical, and misleading assumption: We expect the evolutionary process to breed some specific behavior. But if communication fails to show a salient advantage, or less complicated characteristics can be as same as effective, it will simply not be used – a mechanism well known from Nature.

Nevertheless, we allow for agents exchanging messages in the form of single integer numbers. They are furthermore equipped with the mathematical operations $+$, $-$, $*$, $/$, and *mod*, read-only variables denoting their total and remaining required processing time, the number of other agents using the same workstation, and three multi-purpose variables to be used to hold values.

In the simulations needed to evaluate the grown behaviors, we assume the workstations to form a loose network where each node knows a set of neighbors and their approximate workload. An agent may decide to migrate to any of these neighboring stations by invoking a special command.

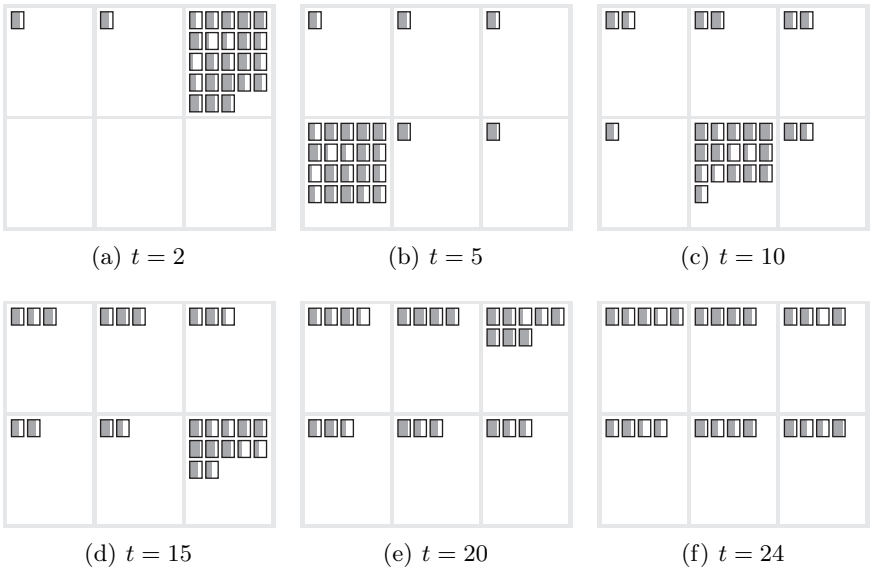
Finally, we have to decide about the properties of a suitable solution. These properties need to be formalized as *objective functions*. Most importantly, those functions should allow for having an appropriately large set of comparable values, for example real numbers in the interval [0..1], which can express the degree of adequateness. Hence, the evolution can approach a solution step-by-step. Obviously, Boolean objective functions are not well suited.

3.2 Experiment I

In our first experiment, we guide the evolution with two objective functions subject to minimization: the variance of the number of agents on the workstations (which grows for asymmetric load distribution) and the number of rules in an agent's program (since we want to find simple behaviors). After some time, we got the following algorithm as a result:

- Perform a time slice of your work on the current node.
- Get the list of neighboring nodes.
- Migrate to some node on this list, sometimes using the least-loaded neighbor, sometimes the worst-loaded neighbor. This decision is based on mathematical computation generating some sort of pseudo-random numbers.

This result is somewhat discouraging: The task was adequately solved, but the solution fails to exploit some capabilities which we would like to see; for example, there is no communication between the agents. On the other hand, it distributes the load effectively: Each agent keeps on moving, so after some time all workstations will be more or less evenly inhabited. Stations with few agents working on them will appear at the top of the list and are likely chosen as migration targets. The agents do not always choose the least-loaded host – which makes sense, because always choosing the queue with the shortest length will quickly overload the free resource.



(a) $t = 2$ (b) $t = 5$ (c) $t = 10$

(d) $t = 15$ (e) $t = 20$ (f) $t = 24$

Fig. 3. Some execution phases of Experiment II

On the bottom line, the created behavior was emergent: By moving to different workstations in a more or less confusing way, the agents modify the workloads on these stations. This serves as some form of implicit communication. Like fish in a school using the distance to their neighbors to find their trajectories, the agents use the loads on neighboring stations in an obscure decision process about whether to migrate or to stay.

3.3 Experiment II

Learning from the experience of the first attempt, we redesigned the scenario to trigger explicit communication. We need to add a constraint to Genetic Programming which enforces communication, making our goal attractive to the evolution. In order to put pressure against wild random agent movements, we also introduce an additional objective function which minimizes the total number of migrations. Starting the experiment again, we get a new class of load balancing algorithms after about 200 generations which all work in the same way.

To verify the adequateness of the generated algorithm, we use a simulation environment. Figure 3 shows an excerpt of the phases of the simulation. The six white rectangles represent the workstations, and the black framed rectangles inside them are the inhabiting agents. The earliest phase is shown in the upper left subfigure, continuing to the right, and the phase closed to the end is on the bottom right. The phases are executed within one time step each.

The agents have different resource demands, denoted by the gray bar inside them, and all enter the system via the same workstation. In each time step, all

Algorithm 1 One of the Resulting Algorithms of Experiment II

$(0 < a_t) \vee false \;\Rightarrow\; send_{t+1} = send_t * start_t$
$true \wedge (remainingWork = 1) \;\Rightarrow\; migrate(1)$
$(remainingWork \geq 0) \wedge (0 \neq 1) \;\Rightarrow\; send_{t+1} = start_t$
$(remainingWork \geq 0) \vee (requiredWork \neq send_t) \;\Rightarrow\; send_{t+1} = start_t$
$(1 \leq 1) \vee (receive_t = send_t) \;\Rightarrow\; send_{t+1} = send_t - send_t$
$(knownHostCount \neq knownHostCount) \vee (send_t > requiredWork) \;\Rightarrow\; a_{t+1} = a_t - requiredWork$
$(a_t \geq 0) \wedge (requiredWork \neq send_t) \;\Rightarrow\; send_{t+1} = start_t$
$(remainingWork \geq 0) \wedge (requiredWork \neq start_t) \;\Rightarrow\; send_{t+1} = start_t$
$[useless] false \vee false \;\Rightarrow\; agentCount_{t+1} = agentCount_t - receive_t$
$(requiredWork \geq 0) \wedge (requiredWork \neq send_t) \;\Rightarrow\; send_{t+1} = start_t$
$[useless](agentCount < 1) \wedge (send_t \geq knownHostCount) \Rightarrow receive_{t+1} = receive_t + 0$
$(remainingWork \geq requiredWork) \wedge true \;\Rightarrow\; migrate(1)$

agents that just have arrived on a host select one amongst them who is allowed to stay by exchanging different messages. The rest of the agents migrates to one of the neighboring stations with minimal workload. There, the same election is repeated. In the configuration outlined in Figure 3, this leads to a clockwise circular movement of the agent swarm through the network, dropping one agent off on each node. This is repeated until all agents found a place for their work and the system settles in the last shown step.

In this scenario, we also have an information exchange between the neighboring workstations above, below, left, and right of each station. When the workload of a workstation decreases because it has finished some jobs, a few neighboring agents may decide to migrate to it in order maintain the balance. Whenever new agents enter the system, they will also be distributed fairly.

Algorithm 1 shows the complete evolved agent code of Experiment II, formulated in the rule-based language which we used during evolution [18,7]. In each time step t, the rule set is checked for conditions which are satisfied. For all these conditions, the actions on the right side are performed. Operations typically consist of an assignment of a value to some variable; the new value will be available during the next time step. Note that the lines are unordered; this means that the behavior of the code may vary if the same variable is affected by different rules.

- a_t is a multi-purpose variable holding a value at time step t.
- $migrate(n)$: Migrate to the workstation at the n^{th} position in the list. The list is locally delivered by each node, with the first item being the neighbor in clockwise direction.
- $send_t$: Transmit the value written to this register (received by $receive_s$ with $s > t$).
- $start_t$: a volatile variable, set to 0 at every time step; set to 1 right after starting the agent.

Conditions which are detected to be unsatisfiable are labeled *useless.* These lines will most likely be removed in later generations because they do not contribute

to the functional fitness, but as they increase the program length, the overall fitness will be higher without these lines, due to the second objective function.

3.4 Discussion

The complete algorithm is oddly complicated and very hard to understand. This should not be surprising: Any kind of solution of the problem, written by a human author, would be easier to analyze; we could assume some *intention* behind the sequence of code lines, allowing us to interpret what the author expects the code to do. This interpretation also allows us to detect errors and to repair the program.

The result of Genetic Programming has nothing to do with intention, as far as the *way to reach the goal* is concerned. As we already mentioned in the experiment descriptions, we should refrain from expecting specific features to evolve. Not only the form of the code but also the way how the problem is solved may be completely different than expected. It is even unclear whether this code fragment actually defines an absolutely correct solution, because we neither have any formal criterion to verify the correctness, nor do we have a certainty that we exposed the code to all relevant scenarios. We can only state that this code works efficiently in the many randomly create scenarios it has been tested during its evolution. It might thus be more appropriate to speak about agents which *behave acceptably*, rather than correct programs.

Genetic Programming bears another, normally unwanted side-effect: If the code generation happens within a static environment, the program code may become *overfit*: The fitness may have reached an acceptable state, but only because the code reflects too many details of the current environment. In that case, the winning individual may actually fail when put in a similar, but not identical environment. In order to prevent this overfitting, we can apply changes to the scenario, used as a perturbance in order to break out of local minima.

A-priori code creation with an evolutionary process may seem to contradict a design of adaptive applications. Indeed, at first sight, the evolved behavior is fixed as soon as we stop the evolutionary process. But adaptive applications are not necessarily composed of components which are adaptive by themselves. The agent behavior strongly depends on the simulated environment used during the evolution. Modifying this environment during the evolution process will either lead to no useful individuals, or we will get solutions which prove to be adaptive in environments which are subject to those changes previously simulated during the evolutionary process. Of course, unanticipated changes will most likely not be handled adequately and may lead to unforeseeable effects.

The objective functions indirectly represent the global goal of the system, so one might object that the whole process is not emergent in the proper sense. However, the objective functions as such do not define the way how a task is solved; they only help to evaluate the adequateness of the solution. The agents, finally equipped with the result of the evolutionary algorithm, do not have any information about the objective functions and about the fitness of their current actions.

3.5 Obfuscation

This observation could prove to be useful in a completely different application area: *code obfuscation*. Mobile agents always face the threat of being analyzed by a malicious host, so some researchers conceived approaches like the obfuscation of the program code in order to prevent analysis, at least for some time until the agent leaves the platform again [19]. The usual process of obfuscation is to apply some transformations on code which is initially well understandable. These transformations keep the functionality, while making it very difficult for a human reader or intelligent system to understand the semantics of the code.

With the genetically generated programs, as shown above, we have a similar effect: We formulate the goal in a clear way, but the Genetic algorithm produces adequate code with obscure structure. The only chance for a malicious host would be to simulate the agent behavior in order to predict its behavior. But it should be practically impossible to derive the semantics of the code by simple inspection.

4 Related Work

Adaptive multi-agent systems (AMAS) are a new idea of building an environment where we may observe that some behavior of the whole group emerges from the actions of the individual agents, without clearly defining their behavior as such. The basic notion is that each agent has to evaluate whether it finds itself in a cooperative or an uncooperative situation with other agents. The AMAS theory [20] states that an agent society will implement a functionally adequate behavior if it can ensure cooperativeness among each other, including the environment. In that way, such an agent society is inherently adaptive, and it inherently prevents antinomic effects to the environment.

The challenge of creating such a system is first to create agents with an evaluation capability (taking into account that detecting non-cooperativeness may be difficult even for really intelligent beings), and second to allow such a system to approximate and finally reach an appropriate behavior after an unknown period of time, during which the actions of the agents emerge through a sequence of more or less antagonistic behavior. On the other hand, of course, we eventually get an emergent, appropriate, and adaptive behavior.

Using Genetic Programming for developing multi-agent systems [21] is not a new approach as such, with applications in food foraging algorithms [22], rendezvous scenarios [23], and pursuit problems [24]. With our work, we attempt to bridge the areas of Genetic Programming and emergence, introducing a complementary approach to enable Emergence Engineering for adaptive agents.

5 Conclusions

Emergence engineering may sound like a contradiction, but as we showed in this article, a viable approach exists that exploits emergent phenomena for the creation of agent behavior. Yet it is profoundly different from the classical idea of

software engineering. For an emergent process to occur and to reach a reasonable result, the goals must be carefully formulated, and the capabilities of the agents must be defined, but specific expectations should be avoided. Instead of running this process *in situ*, we propose to develop the agent behavior in a simulation environment, and afterwards implant it into the actual agents. For the experiments we used the Distributed Genetic Programming Framework, which has been developed in our group [15,16,17]. We described a relevant sample scenario and illustrated how the results of such a generation look like. We evolved agent behavior as fixed sets of rules that accomplish the desired objectives.

But do we still talk about autonomous agents? – Similar to Nature, we envisage to grow a program which has proved to be useful through all generations. We may call it *instinct*, if we want to use a biological term. We need not restrict the behavior to instinct solely; rather, it could prove worthwhile to implement some part of the behavior depending on reasoning, while some other part relies on an evolutionary process. How instinct and reasoning may be orchestrated is certainly an open issue.

We are currently checking further scenarios like an electronic market place, where agents use certain performatives to trade goods, and try to derive principles for a more generic engineering procedure. One specifically interesting point is the relation between requirements analysis and genetic approach.

References

1. Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21:25–34, July 1987. ACM SIGGRAPH '87 Conference Proceedings, Anaheim, California, July 1987.
2. Lee Spector, Jon Klein, Chris Perry, and Mark Feinstein. Emergence of collective behavior in evolving populations of flying agents. In *Proceedings of Genetic and Evolutionary Computation - GECCO 2003, Genetic and Evolutionary Computation Conference, Part I*, 2003, volume 2723/2003 of *Lecture Notes in Computer Science (LNCS)*, pages 61–73, ISBN: 978-3-540-40602-0, ISSN: 0302-9743 (Print) 1611-3349 (Online). http://hampshire.edu/lspector/gecco2003-collective.html.
3. William H. Calvin. *The River That Flows Uphill: A Journey from the Big Bang to the Big Brain.* Macmillan Pub Co, ISBN: 0792445228, July 1986.
4. Giovanna Di Marzo Serugendo, Marie-Pierre Gleizes, and Anthony Karageorgos. Self-organization and emergence in multi-agent systems. *The Knowledge Engineering Review*, 20(2), 2005.
5. Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Oxford University Press, ISBN: 0195099710, January 1996.
6. Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz, editors. *Handbook of Evolutionary Computation.* Computational Intelligence Library. Oxford University Press, Bristol, New York, ISBN: 0750303921, April 1997.
7. Thomas Weise. *Global Optimization Algorithms – Theory and Application.* Thomas Weise, November 16, 2007 edition. http://www.it-weise.de/.
8. John R. Koza. *Genetic Programming, On the Programming of Computers by Means of Natural Selection.* A Bradford Book, The MIT Press, Cambridge, Massachusetts, 1992 first edition, 1993 second edition, ISBN: 0262111705, 1992.

9. Jörg Heitkötter and David Beasley, editors. *Hitch-Hiker's Guide to Evolutionary Computation: A List of Frequently Asked Questions (FAQ)*. ENCORE (The EvolutioNary Computation REpository Network), 1998. http://www.cse.dmu.ac.uk/ rij/gafaq/top.htm.
10. Richard M. Friedberg. A learning machine: Part I. *IBM Journal of Research and Development*, 2:2–13, 1958.
11. Richard M. Friedberg, B. Dunham, and J. H. North. A learning machine: Part II. *IBM Journal of Research and Development*, 3(3):282–287, 1959.
12. Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the 1st International Conference on Genetic Algorithms and their Applications*, Carnegie-Mellon University, Pittsburgh, PA, USA, July 24-26 1985, pages 183–187, Mahwah, NJ, USA. ISBN: 0-8058-0426-9.
13. Robert Ian McKay, Xuan Hoai Nguyen, Peter Alexander Whigham, and Yin Shan. Grammars in genetic programming: A brief review. In *Proceedings of the International Symposium on Intelligence, Computation and Applications*, 2005, pages 3–18. http://sc.snu.ac.kr/PAPERS/isica05.pdf.
14. Peter Nordin. A compiling genetic programming system that directly manipulates the machine code. In *Advances in genetic programming*, volume 1, ISBN: 0-262-11188-8, chapter 14, pages 311–331. MIT Press, Cambridge, MA, USA, April 1994.
15. Thomas Weise and Kurt Geihs. DGPF – an adaptable framework for distributed multi-objective search algorithms applied to the genetic programming of sensor networks. In *Proceedings of the Second International Conference on Bioinspired Optimization Methods and their Application, BIOMA 2006*, 2006, pages 157–166, ISBN: 978-961-6303-81-1. http://www.it-weise.de/documents/files/W2006DGPFc.pdf.
16. Thomas Weise, Kurt Geihs, and Philipp Andreas Baer. Genetic programming for proactive aggregation protocols. In *Proceedings of Adaptive and Natural Computing Algorithms, 8th International Conference, ICANNGA 2007, Part I*, Warsaw University of Technology, Warsaw, Poland, April 11-14 2007, volume 4431/2007 of *Lecture Notes in Computer Science (LNCS)*, pages 167–173, ISBN: 978-3-540-71589-4, ISSN: 0302-9743. http://www.it-weise.de/documents/files/W2007DGPFb.pdf.
17. Thomas Weise, Michael Zapf, Mohammad Ullah Khan, and Kurt Geihs. Genetic programming meets model-driven development. In *7th International Conference on Hybrid Intelligent Systems (HIS 2007)*, 2007. IEEE Computer Society, ISBN: 0-7695-2946-1. http://www.it-weise.de/documents/files/WZKG2007DGPFg.pdf.
18. Thomas Weise, Michael Zapf, and Kurt Geihs. Rule-based genetic programming. In *Proceedings of the 2nd International Conference on Bio-Inspired Models of Network, Information, and Computing Systems*, 2007. (to appear).
19. Fritz Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. *Lecture Notes in Computer Science*, 1419:92–113, 1998.
20. C. Bernon, M.-P. Gleizes, S. Peyruqueou, and G. Picard. ADELFE: A methodology for adaptive multiagent systems engineering. *Lecture Notes in Computer Science*, 2577, 2002.
21. Forrest H. Bennett. Automatic creation of an efficient multi-agent architecture using genetic programming with architecture-altering operations. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, 1996, pages 30–38.
22. Forrest H. Bennett. Emergence of a multi-agent architecture and new tactics for the ant colony foraging problem using genetic programming. In *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior: From animals to animats 4*, Cape Code, USA, 1996, pages 430–439, ISBN: 0-262-63178-4.

23. Mohammad Adil Qureshi. Evolving agents. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, 1996, pages 369–374.
24. Mohammad Adil Qureshi. *The Evolution of Agents.* PhD thesis, University College, London, London, UK, July 2001. online available: http://www.cs.bham.ac.uk/ wbl/biblio/gp-html/qureshi_thesis.html.