Applicability of Emergence Engineering to Distributed Systems Scenarios

Michael Zapf, Thomas Weise

Universität Kassel, Wilhelmshöher Allee 73
D-34121 Kassel, Germany
{zapf|weise}@vs.uni-kassel.de

Presented at EUMAS'08,
the Sixth European Workshop on Multi-Agent Systems
Bath, UK, December 18–29, 2008

Abstract Genetic Programming can be effectively used to create emergent behavior for a group of autonomous agents. In the process we call Offline Emergence Engineering, the behavior is at first bred in a Genetic Programming environment and then deployed to the agents in the real environment. In this article we shortly describe our approach, introduce an extended behavioral rule syntax, and discuss the impact of the expressiveness of the behavioral description to the generation success, using two scenarios in comparison: the election problem and the distributed critical section problem. We evaluate the results, formulating criteria for the applicability of our approach.

1 Introduction

Within the area of software engineering, we have to face new challenges for developing programs, especially for systems which are geographically dispersed, or which consist of far too many components to allow a reasonable divide-and-conquer approach. Biologically inspired approaches attempt to learn from nature, for instance, by taking swarm behavior and other self-organizing phenomena as a model for computing technology. However, while we can easily conceive that a collection of autonomous agents cooperate in a way to produce some common, possibly also unexpected behavior, we still do not know how to trigger this development, and specifically, in a given, desired direction.

Utilizing these unexpected – or emergent – phenomena within an engineering process is called Emergence Engineering [1], a rather new area in software engineering. In [2] we presented a first approach to this concept with our offline emergence engineering approach and demonstrated that it is suitable to create an appropriate group behavior within a load balancing scenario. However, we cannot guarantee whether the results of this process actually reliably fulfill the requirements of the scenario, that is, whether the results are really "solutions" as we understand the term. Can such an emergent process create reliable solutions at all?

We attempt to approach this question by briefly introducing our previous approach from [2], adding a new, extended version with richer semantics. Comparing the basic and enhanced version, we will elaborate on two related aspects: On one hand we discuss the contribution of the base language to the evolution success, and on the other hand we comment on the overall practicability of creating solutions by emergence. Two example scenarios, the Election Problem and the Critical Section Problem, will serve as representants of two different classes of problems, and the evolution success of each class will be evaluated.

2 Offline Emergence Engineering

Before we describe our recent experiments, we will summarize the core aspects of our approach [2].

2.1 Rule-Based Genetic Programming

Our approach to emergence engineering is based on Genetic Programming which allows us to breed adequate solutions for a given task. At this point we will only give a brief description of the most important concepts.

Genetic Programming is a class of evolutionary algorithms for breeding programs, algorithms, and similar constructs. All evolutionary algorithms proceed in principle according to the following scheme: Initially, a population of individuals with a totally random genome is created. The objective functions rate the utility of the different features of the solution candidates. Then a fitness value is assigned to each individual. A subsequent selection process filters out the solution candidates with low fitness and allows those with good fitness to enter the mating pool with a higher probability. In the reproduction phase, offspring is created by varying or combining these solution candidates and integrated it into the population. If the termination criterion is met, the evolution stops here. Otherwise, the process is repeated.

Genetic programming has to cope with some unwanted effects inferred by the uncontrolled modification of program code, like *epistasis*. Instead of sequentially executed program lines, the approach as described here features an unordered set of rule lines as the behavior definition. Each rule consists of a condition and an action part. The actions are carried out only if the condition part evaluates to true. The outputs and storage operations performed by the actions are buffered and committed after all rules have been applied. This *Rule-based Genetic Programming*, as we call it, keeps epistatic effects considerably lower than other standard or linear GP methods so that programs become more robust in terms of reproduction operations. We expect that this will lead to a more durable evolution with a higher probability of finding good solutions.

2.2 Engineering Process

Figure 1 provides an overview on the engineering process, especially where Genetic Programming comes into place. The OEE approach consists of five phases:

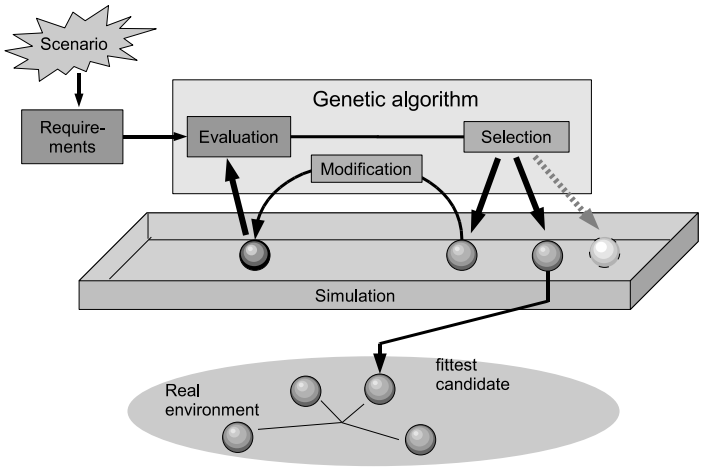

Figure 1: Engineering Process

1. The scenario must be analyzed, resulting in a collection of requirements.
2. Suitable objective functions must be found which determine the fitness of a solution
3. The evolutionary algorithm is run, creating new versions of the individuals and selecting the fittest ones repeatedly.
4. At some point in time, the evolutionary process is halted and the best individual found is picked out.
5. The individual is put into a component as its behavior, being deployed in the real environment.

Creating programs with Genetic Programming by itself is not a new concept. The essential idea of Offline Emergence Engineering (OEE) is to set up this technique as a part of an engineering process for multi-agent systems. In classical approaches and also in recent Agent-Oriented Software Engineering methodologies like Gaia [3], the overall scenario is usually analyzed in order to identify sub-problems which can be handled by a small group of agents. Together, the various agent groups contribute to the overall functionality, and so we can expect to have a suitable overall solution.

To reduce complexity for the evolution process, we let only one type of agent behavior evolve, which must be put into all participating agents. This is no limitation of flexibility; if we had multiple behavior types, we can put them into one type and select the appropriate part of the behavior by means of a state variable.

2.3 Emergence

For the scenarios that we envisage, an analysis may not be applicable, due to the complexity of the overall system, the number of involved agents, their spatial distribution, or simply because we are just unable to find a consistent set of sub-problems. In those

cases, we may still have an idea about the overall functionality of the multi-agent system. That is, a desirable engineering approach should be to provide the overall conditions as input, and getting implementations for the individuals as output. If the process works correctly, and if we have adequately described the target environment with its requirements, the resulting individuals should act in a way so that gradually or immediately, the desired properties appear.

In all cases where we cannot precisely say how the actions of the subparts contribute to the success of the overall system, we refer to this phenomenon as *emergence*. Emergence is typically found in collections of individuals which are interacting with each other. If this collection can exhibit structural properties, emergence is often linked to self-organization although this is not generally the case.

As we have only one type of individual (determined by its behavior), the pressure of selection on the individuals is propagated to the rule set of the individuals. Concerning the effect, a set of rules excerts direct influence on the behavior of the agent, and the individual behavior contributes to the overall behavior which can be measured by the fitness function. Hence, within OEE, we can witness emergence in different levels. The behavior of the group somehow emerges from the behavior of the individuals, and the behavior of the individuals emerges from the rule set.

2.4 Flexibility and Reliability

Unlike classical approaches, we usually cannot verify the fitness of the resulting behavior by looking at the code. In most cases, the outcome of Genetic Programming is hardly understandable – again emphazing the fact that we have an emergent process. Moreover, the result of the creation is fixed; the agents are not able to modify their own behavior. This raises two questions: How flexible are these solutions in the light of changing environments, and how can we rely on the results?

While analyzing the requirements of the scenario, we certainly need to pay attention whether there may be varying conditions. These influences have to be modeled as well; the simulation environment, where the Genetic Algorithm executes its creative and selective process, must mirror those conditions precisely. So if the behavior depends on the presence and the number of specific entities, these must be appropriately varied during the simulation. Agents which can cope with these changes will also be able to do so in the real environment.

Still, we cannot say whether a “solution” will work reliably during the whole time of application. It is certainly possible that a solution will at some time change its behavior in an unexpected way – since we are not able to foretell this effect by looking at the code. This may be a principle problem of emergent behavior. However, we should also think about the nature of a “solution” – is it allowed to deviate from the optimal path, or is it required to strictly fulfill the requirements at all times? In that case, is the scenario suitable for an emergent approach at all?

2.5 Expressiveness of the Behavioral Language

Another issue which we addressed in our recent experiments is the influence of the behavior language. As described above (see also [4, 2]), the Rule-Based Genetic Programming language (RBGP) defines rules with two parts:

$$\textit{Condition}_1 \otimes \textit{Condition}_2 \Rightarrow \textit{Effect}, \text{ with } \otimes \in \{\textit{and}, \textit{or}\}$$

with a conditional part, consisting of two conditions connected by **and** / **or**, and an effect part. Figure 2 shows the correspondence between the genotype and phenotype of RBGP. The evolutionary operations (crossover and mutation) modify the genotype which is then transformed to the corresponding phenotype using the genotype-phenotype mapping (GPM).

Symbol	Encoding
0	0000, 1100
1	0001, 1101
start	0010, 1110
id	0011, 1111
netSize	0100
receive	0101
send	0110
enter	0111
leave	1000
a	1001
b	1010

Comp.	Enc.
>	000
≥	001
=	010
≤	011
<	100
≠	101
true	110
false	111

Concat.	Enc.
∧	0
∨	1

Action	Enc.
=x+y	00
=x-y	01
= x	10
=1-x	11

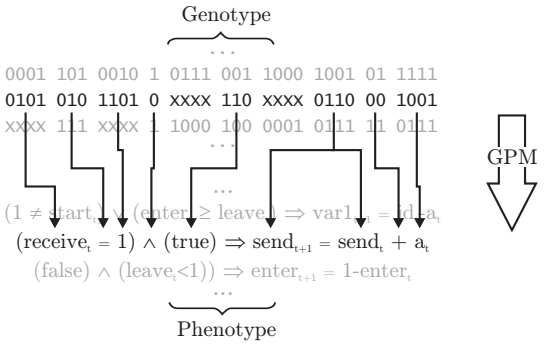

Figure 2: RBGP genotype and phenotype

Although the primitives of Rule-based Genetic Programming are powerful enough to express many of the constructs known from high-level programming languages, the original RBGP as described above has some inherent limitations. The most obvious drawback is the lack of Turing completeness.

In order to illustrate this problem, imagine we would restrict Java data types to primitive, non-constructed types like integers. This would make it hard to create data structures like lists, since we would have to define one variable for every single element. Writing a method for sorting a list of arbitrary length would be unfeasible. The plain

Algorithm 1 Sorting algorithm written in the eRBGP language

$(start_t > 0) \wedge true \Rightarrow a_{t+1} = 0$
$(start_t > 0) \wedge true \Rightarrow b_{t+1} = 0$
$(a_t < l_t) \wedge ([a_t]_t < [b_t]_t) \Rightarrow [a_t]_{t+1} = [b_t]_t$
$(a_t < l_t) \wedge ([a_t]_t < [b_t]_t) \Rightarrow [b_t]_{t+1} = [a_t]_t$
$(b_t \geq a_t) \wedge (a_t < l_t) \Rightarrow a_{t+1} = a_t + 1$
$(b_t < a_t) \wedge true \Rightarrow b_{t+1} = b_t + 1$
$(b_t \geq a_t) \wedge (a_t < l_t) \Rightarrow b_{t+1} = 0$

Rule-based Genetic Programming approach has a similar limitation as the symbols resemble integer variables. In Java, this whole problem is circumvented with arrays, a form of memory which can be accessed indirectly.

To extend the expressiveness, we add indirect memory access to the programming language, using the notation $[a_t]_t$, which stands for the value of the a_t^{th} symbol (at time step t) in the ordered list of all symbols. This new form of Rule-based Genetic Programming which we call eRBGP allows the evolution of list-sorting algorithms and makes it Turing-complete[1]. Algorithm 1 shows an example of list processing.

We included another enhancement in eRBGP: The base RBGP language only allows for two-part expressions. Creating a conjunction or disjunction of three or more expressions requires the evolutionary process to introduce intermediate boolean stores, and to formulate the complete condition in multiple lines. In eRBGP, rules can be built with complex conditions. However, as the expressions do not have a common structure anymore, we lose the ability of using Genetic Algorithms with fixed-size genes for its evolution. Instead we apply Genetic Programming with a tree-shaped genome, as Figure 3 shows. In this case, the evolutionary operations modify the tree structure directly. The text representation of the illustrated eRBGP program can be seen as Algorithm 2.

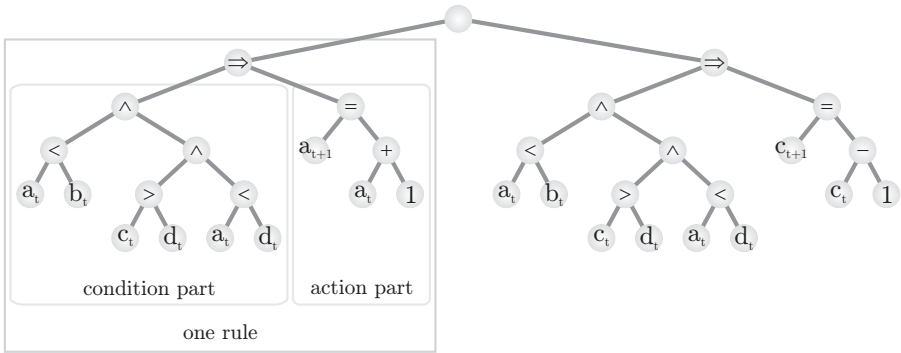

Figure 3: eRBGP genotype and phenotype

[1] The proof for Turing Completeness of Genetic Programming languages with indexed memory [5] can be easily adapted to eRBGP.

Algorithm 2 Textual representation of the tree genome

$((a_t < b_t) \wedge ((c_t > d_t) \wedge (a_t < d_t))) \Rightarrow a_{t+1} = (a_t + c_t)$
$((a_t < b_t) \wedge ((c_t > d_t) \wedge (a_t < d_t))) \Rightarrow c_{t+1} = (c_t - 1)$

3 Experiments

We conducted some experiments in order to determine preconditions for a successful Emergence Engineering:

- *Load balancing*: Distribute load among the available hosts using mobile agents.
- *Election*: Get an agreement on choosing one individual from a given set.
- *Critical section*: Restrict access to a resource so that only one agent may access it at a time.

In a load balancing scenario, we assume that we have a network of n nodes which offer execution resources, and a set of m tasks to be distributed among these hosts. Using mobile agents, we can map the tasks to the agents, where each agent is responsible to migrate to a suitable host by itself. We have already shown that this scenario may be adequately handled by the Offline Emergence Engineering approach [2]. The agents start to circulate from host to host and start to migrate away if they detect that a neighboring host has less load. In the following, we will have a closer look at the other two problems which seem related but show a surpringly different behavior in this approach.

3.1 Election

Election algorithms have many applications in distributed systems. Election is also a very common functionality needed in numerous agent scenarios. A distributed election algorithm can be initiated by any number of agents in the systems and will reach a terminal configuration in which exactly one agent is elected as *leader* and all agents agree to this choice. The challenge of such an election procedure is the distributed nature: It must be ensured that all agents get a consistent view of the election.

Before we start the evolution, we define the operations available to the agents. An agent may send a message containing a single number stored in the variable **out** with the command **send**. Whenever a message is received, its contents appear in the symbol **in**, and the variable **incomingMsg** is set to 1. The communication among the agents is delayed arbitrarily and messages may overtake each other. All agents run in parallel, at different, randomly changing processing speeds. Furthermore the agents have unique identifiers (variable **id**) and two multi-purpose variables **a** and **b**. We expect the identifier of the elected agent to be stored in variable **a** after about 5000 simulated time units.

For our OEE process, we need to define objective functions which guide the selection mechanism. The challenge is to find some properties which, by maximizing or minimizing, indicate the fitness of the individual for the given problem. Thus, binary properties are virtually worthless. In this case, we define the following functions:

f_1 counts the number of different IDs found when comparing all the values stored in the **a** variables of the agents after the simulation and penalizes values that do not denote valid IDs. f_2 determines the behavior size in terms of the number of rules, f_3 counts the time units used for active computations (penalizing useless computation when the node could sleep instead), and f_4 counts the number of messages exchanged. For the best overall fitness, all four functions shall yield minimal values.

The results of this Emergence Engineering process are, in most cases, incomprehensible [2] – due to the fact that there is no "intelligent program designer" with more or less clear and interpretable intentions. Even worse, some effects of these programs are caused by the iterative execution which may require a prohibitive effort to analyze. So we were quite surprised to see that one of the results we got from the evolution process looked very similar to the well-known *Message Extinction* algorithm. Further experiments showed, however, that this easily understandable result was rather an exception, as following results included much more complicated internal computations. They implement valid election algorithms and, as in the previous example, either the node with the highest or lowest ID wins.

RBGP

```
false or (start(t)=incomingMsg(t)) => start(t+1)=1-b(t)
false or (b(t)>=a(t)) => a(t+1)=a(t)+id(t)
(out(t)<=start(t)) or (id(t)!=b(t)) => send
false or (a(t)!=out(t)) => out(t+1)=a(t)
(id(t)=0) and (out(t)>=0) => id(t+1)=id(t)/b(t)
(0=id(t)) or (id(t)<in(t)) => a(t+1)=in(t)
```

eRBGP

```
id(t) => send
[incomingMsg(t)](t) => out(t+1) = id(t)
(out(t) - (incomingMsg(t) or [a(t)](t))) => a(t+1) = id(t)
(in(t) / id(t)) => id(t+1) = in(t)
```

Figure 4: Comparing RBGP and eRBGP solutions of the election algorithm

We conducted our experiments with the standard RBGP language and the new eRBGP language, hoping for simpler solutions as the eRBGP is more expressive. Figure 4 shows one of the delivered individuals for each case. The evolved solution for eRBGP only requires four lines. We evaluated the reliability of multiple solutions delivered by these two approaches by the fraction of scenarios where the election process proceeds correctly. Programs generated with RBGP are reliable in 60% of the scenarios and those from eRBGP achieve correct behavior in 91% of the network simulations. This shows the encouraging result that eRBGP solutions are much more reliable, and it seems to be a strong indication that the expressiveness of the language can help to find better solutions.

3.2 Critical Section (CS)

An interesting problem which has been examined first in the 1960s is the *critical section*. The term critical section denotes the access to a shared resource which must be utilized by only one process at a time. This access therefore requires some form of cooperative arbitration of the requests from multiple agents. The processes running concurrently on different nodes have to communicate by the means of message exchange. Based on the messages sent and received, an agent has to decide whether it is allowed to access the critical section or whether it has to wait. The first algorithms for the mutual exclusion at a distributed critical section were introduced by Lamport and Ricard and Agrawala, followed by Maekawa's solution, which is optimal in the number of exchanged messages.

In order to breed solutions for the CS problem, we again need to analyze the requirements. The first objective function f_1 imposed on the program evolution relates to the number of violations of the mutual exclusion criterion. We simply sum up the time steps where more than one process accessed the critical section. To increase the pressure, we sum up the square of the number of nodes inside the CS. As this function alone would reward solutions where no process ever enters the critical section, we need a second objective function f_2 which represents the number of times each process could enter the critical section *at least* in the fixed time span of the simulation. We also add a value proportional to the total number of accesses of the CS. Finally, $f_3$counts the number of rules and, thus, exerts pressure to drop unnecessary rules. f_1 and f_3 are subject to minimization, f_2 is to be maximized.

$$f_1(P) = \sum_{t=1}^{T} \begin{cases} 0 : \textit{if } n_{cs}(t) \leq 1 \\ (n_{cs}(t))^2 : \textit{otherwise} \end{cases} \tag{1}$$

$$f_2(P) = \min\{\textit{node in cs}\} + 1 - \frac{1}{\sum(\textit{node in cs})} \tag{2}$$

$$f_3(P) = |P| \tag{3}$$

For our experiments we have chosen a population size of more than 7000 individuals. Nine nodes were running in the simulation for 1000 time steps in order to test every evolved program.

The first disappointment we had to face was that during the RBGP evolution, no persistently good solution emerged. We terminated the experiment after more than 2100 generations, although we still observed changes in the population up to that point. We drew the best individuals from the evolved set and put them into a simulation environment.

Table 1 shows the results, evaluating the behavior of the best individuals in 200 random network configurations. The columns contain the ratio of the networks where the respective individual abided by the preconditions of the Critical Section scenario. For evaluating the correct function of the individual, we have to check whether it prevents collisions, and whether it shows fairness to all agents, i.e. ideally, each agent gets the chance to enter the CS.

Table 1: RBGP success ratio for the critical section problem

No collisions	Fairness
98.5%	52.5%
98%	53%
97%	54.5%
90.5%	84%

For RBGP, we found a solution avoiding collisions in 98.5% of all given environments. In 52.5% of the environments, the solution is also fair, allowing more than one process to enter the critical section. As the table shows, lower rates of collisions correlate with lower fairness. From an intuitive point of view this is understandable – involving more agents increases the chances of triggering violations. The last individual, failing to ensure exclusiveness in 1 of 10 test networks, has the best fairness value. Whether such a "solution" is tolerable for employing in the real environment is another question.

Figure 5 shows the RBGP code of one of the individuals. Note that the first rule seems to make this program immediately violate the first condition, as it unconditionally forces the process to enter the critical section at the very first place. However, the evolutionary process "discovered" that in our framework, the rules are only committed if no error occurs when evaluating the remaining lines. In this case, a division by zero in one of the following lines breaks the evaluation, and it only succeeds when the process gets an incoming message. A human software engineer would surely choose a "cleaner" design, but this version obviously had a better fitness than other variants.

Figure 5 also shows an individual from our framework, using the eRBGP language. As we already suspected, the solution is much more compact (6 rules) as the solution of RBGP (15 rules), also defining much more complex conditions. Table 2 presents the evaluation of some individuals in test networks. Unfortunately, the results have become worse.

Table 2: eRBGP success ratio for the critical section problem

No collisions	Fairness
99%	1%
98.5%	1.5%
84%	16%
80%	20%

While the best solution, according to the first criterion, succeeds in 99% of all networks, it fails to be fair in 99% of all cases, by only allowing a single process to enter. The same holds for the other individuals, showing complementary ratios of collision avoidance and fairness. It seems as if fairness was only achieved for that moment when it failed to ensure the exclusiveness.

RBGP

```
true and true => enterCs
(leaveCS(t)>leaveCS(t)) or true => in1(t+1)=in1(t)+1
true or true => send
... [ 9 lines deleted ] ...
(out2(t)!=a(t)) or true => netSize(t+1)=netSize(t)/in1(t)
(netSize(t)>=b(t)) or (d(t)>=out1(t)) =>
    out1(t+1)=out1(t)+leaveCS(t)
true or false => send
```

eRBGP

```
(0 or ((!incomingMsg(t)) or start(t))) => send
((((!start(t)) * in2(t))
    and (((out1(t) - (!incomingMsg(t)))
    and ((0 and b(t)) * -2)) * ((leaveCS(t)=(start(t)
       + (start(t)=b(t)))) + 1)))
       + (((((!incomingMsg(t)) and incomingMsg(t))
          or -1) - in2(t)) + start(t))) => [b(t)](t+1) = 2
(!incomingMsg(t)) => out2(t+1) = start(t)
out2(t) => send(t)
((start(t) and b(t))=(out2(t) - start(t))) =>
     in1(t+1) = ((incomingMsg(t) and a(t))=b(t))
in1(t) => enterCS
```

Figure 5: RBGP and eRBGP solution for the critical section problem

4 Evaluation

The observations of the experiments lead us to some first conclusions. First, increasing the language expressiveness does not necessarily improve the quality of the solution. This is, to some extend, in line with our observation in the Load Balancing scenario [2] where we noticed that the evolutionary process only utilizes those features and properties which yield an evolutionary advantage. The idea behind a more expressive language like eRBGP is that by decreasing the number of rules – using a more expressive syntax – we remove possible non-functional evolutions of the code. Second, we allow for a more complete repository of evolvable functionality due to the Turing completeness. The problem may be that we are still trapped in classical notions of engineering, and we possibly witness that evolution unleashes other problem-solving capabilities.

As we saw in our experiments, some scenarios seem to allow a higher evolutionary success than others. Although the scenarios seem to share some common properties, they are obviously not equally suited for the OEE process. Finding out what makes a scenario suitable for Emergence Engineering is an open question. We can try to approach this question by distinguishing the following special simple cases. The result of the evolution may be

- *any-agent* and
- *all-agent properties or behaviors.*

The former one refers to the observation that eventually, one of the agent instances has got some property or expresses some behavior. The latter case refers to the situation where eventually, the whole collective adops the same properties and behaviors. That way, we can analyze our scenarios as follows:

Load balancing and election seem to be suitable problems for an emergence approach. We believe that this is due to the fact that both problems require *all-agent behaviors*. For load balancing, each agent migrates as soon as there is a host with lower load, while for election, all agents eventually share the same property at the end, knowing the ID of the winner. Election also shows an any-agent property, namely for the elected agent.

At first sight, Critical Section is some kind of an election problem: Choose one agent to enter the CS. But due to the fact that we also want to reach fairness, there are more requirements which need to be fulfilled. More specifically, the CS problem requires an iterative election, and worse, it requires the behavior of the group to change in order to achieve fairness. This is neither an any-agent nor an all-agent behavior.

Obviously, the agents somehow need to *learn* that some agent was already allowed to enter the CS, and so it does not qualify to enter again for some time. While it is not impossible that agents develop learning behavior within the evolutionary process, it is obviously fairly unlikely. Hence, for the applicability of this approach, the analysis must take care whether the group behavior implies learning capabilities or not.

Beyond our presented approach of OEE, the observations are important for other Emergence Engineering approaches as well, as we believe – that is, there may be problems with an inherent difficulty for Emergence Engineering. In the examples, there is no obvious correlation between the design mechanism and the behaviour except for the fact that our agents have to use the behavior that was previously created. But as we described in [2], even though our agents use the fixed ruleset which has evolved, they will show adaptivity if adaptivity has led to an evolutionary advantage. In turn, if we allow an agent community to build up emergent behavior dynamically (“online”), there is neither a guarantee that we reach our goal without subtle side-effects, as demonstrated with the Critical Section example.

5 Related Work

Automatic program generation is already a classic problem and covered by numerous approaches for some decades by now. Genetic Programming is a class of evolutionary algorithms for breeding programs, algorithms, and similar constructs [6, 7]. Cramer utilized genetic algorithms and tree-like structures to evolve programs [8]. 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 [6]. Since then, many different approaches like grammar-guided Genetic Programming [9] and linear Genetic Programming [10] have branched off. Also in the context of multi-agent systems, Genetic Programming is a well-known approach, especially in the context of foraging simulations [11] or rendezvous scenarios [12]. These concepts all address either optimization problems or solutions for specific problem areas. In contrast, in this article we discuss a new engineering concept for multi-agent societies for generic problems. By

utilizing the inherent emergence of Genetic Programming, we envisage to create a part of a larger engineering process.

There is currently only few related work concerning emergence engineering for agent societies. The most notable work here is ADELFE [13]. ADELFE is an engineering approach based on the Rational Unified Process [14] which explicitly exploits emergence among a set of cooperative agents. Basically, in ADELFE the overall system is modelled as an environment in which a set of agents is situated. By its interaction with the multi-agent system, the environment forces the agents to cooperate. Comparing to our approach, we call ADELFE an *online emergence engineering* approach, due to the fact that the agents are situated in the real environment and need to self-organize to respond to changes in the environment.

6 Conclusions

The Offline Emergence Engineering approach, as described in this paper, is a comparably new approach to create implementations for multi-agent systems. Our research is still in an early state where we need to conduct more experiments to find out characteristics of problems which may be suitably handled by this approach. However, we must face the fact that generating programs in this way is extremely time-consuming, literally taking days to weeks until a solution is found. This may certainly be handled by faster hardware, but relying on Moore's Law is a weak excuse.

There are some challenges in Emergence Engineering as we discovered in our approach: First, it is important to find criteria whether a problem is suitable for Emergence Engineering. This is currently a widely open question, but also the most important one. We have presented some simple criteria which may indicate whether EE is likely to produce suitable solutions for a problem. Second, if we have considered a problem to be suitable for EE, we must find appropriate objective functions for the evolutionary algorithms to measure the fitness of individuals. In other approaches [13] this corresponds to evaluating the adequateness of the agent behavior. Third, we have to decide on the expressiveness of the behavioral language and the capabilities of the agents. We found that increasing the expressiveness of the implementation language of the agent behavior need not necessarily yield better solutions. Although we increased the scope of the possible agent behavior, some problems seem to be trapped in sub-optimal areas of the evolution.

We believe that defining some more scenarios will provide us with more experience on the behavior of the evolution process and whether it may be required to introduce additional modifications.

Bibliography

[1] Özalp Babaoğlu, David Hales, Mark Jelasity, et al., editors. *Proceedings of the Workshop on Engineering with Complexity and Emergence (ECE), Satellite Workshop of the European Conference on Complex Systems (ECCS)*, 2005.

[2] Michael Zapf and Thomas Weise from the University of Kassel, FB-16, Distributed Systems Group, Wilhelmshöher Allee 73, 34121 Kassel, Germany. Offline Emergence Engineering For Agent Societies. In *Proceedings of the Fifth European Workshop on Multi-Agent Systems (EUMAS'07)*, December 14, 2007, Elmouradi Hotel, Hammamet, Tunesia. Also presented at the co-located Fifth Technical Forum Group (TFG5). Online available at http://www.it-weise.de/documents/files/ZW2007EUMASTR.pdf [accessed 2009-01-07]. See also [15].

[3] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12(3):317–370, 2003.

[4] Thomas Weise, Michael Zapf, and Kurt Geihs from the University of Kassel, FB-16, Distributed Systems Group, Wilhelmshöher Allee 73, 34121 Kassel, Germany. Rule-based Genetic Programming. In *Proceedings of BIONETICS 2007, 2nd International Conference on Bio-Inspired Models of Network, Information, and Computing Systems*, 2007. In proceedings [16]. Online available at http://www.it-weise.de/documents/files/WZG2007RBGP.pdf [accessed 2009-01-07].

[5] Astro Teller. Turing Completeness in the Language of Genetic Programming with Indexed Memory. In Zbigniew Michalewicz, J. David Schaffer, Hans-Paul Schwefel, David B. Fogel, and H. Kitano, editors, *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pages 136–141. IEEE Press, Piscataway, New Jersey, June 27–29, 1994, Orlando, Florida, USA. ISBN: 0-7803-1899-4. Online available at http://www.cs.cmu.edu/afs/cs/usr/astro/public/papers/Turing.ps and http://www.astroteller.net/work/pdfs/Turing.pdf [accessed 2007-09-17].

[6] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. A Bradford Book. The MIT Press, Cambridge, Massachusetts, USA, 1992 first edition, 1993 second edition, 1992. ISBN: 0-2621-1170-5, 978-0-26211-170-6. Partly online available at http://books.google.de/books?id=Bhtxo60BV0EC [accessed 2008-08-16].

[7] 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. USENET: comp.ai.genetic. Online available at http://www.cse.dmu.ac.uk/~rij/gafaq/top.htm and http://alife.santafe.edu/~joke/encore/www/ [accessed 2007-07-03].

[8] 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*, pages 183–187, 1985. In proceedings [17]. Online available at http://www.sover.net/~nichael/nlc-publications/icga85/index.html [accessed 2007-09-06].

[9] Robert Ian McKay, Xuan Hoai Nguyen, Peter Alexander Whigham, and Yin Shan. Grammars in Genetic Programming: A Brief Review. In L. Kang, Z. Cai, and Y. Yan, editors, *Progress in Intelligence Computation and Intelligence: Proceedings of the International Symposium on Intelligence, Computation and Applications*, pages 3–18. China University of Geosciences Press, April 2005. Online available at http://sc.snu.ac.kr/PAPERS/isica05.pdf [accessed 2007-08-15].

[10] Peter Nordin. A compiling genetic programming system that directly manipulates the machine code. In Kenneth E. Kinnear, Jr., editor, *Advances in genetic programming 1*, volume 1 of *Complex Adaptive Systems*, chapter 14, pages 311–331. MIT Press, Cambridge, MA, USA, April 7, 1994. ISBN: 0-2621-1188-8, 978-0-26211-188-1.

[11] Forrest H. Bennett III. Emergence of a Multi-Agent Architecture and New Tactics For the Ant Colony Foraging Problem Using Genetic Programming. In Pattie Maes, Maja J. Mataric, Jean-Arcady Meyer, Jordan B. Pollack, and Stewart W. Wilson, editors, *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior: From animals to animats 4*, pages 430–439, September 9-13, 1996, Cape Code, USA. MIT Press, Cambridge, MA, USA. ISBN: 0-2626-3178-4.

[12] Mohammad Adil Qureshi from the Genetic Programming Group, Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK. Evolving Agents. 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 369–374. MIT Press, July 28–31, 1996, Stanford University, CA, USA. See also [18]. Online available at http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/AQ.gp96.ps.gz [accessed 2007-09-17].

[13] Carole Bernon, Marie-Pierre Gleizes, Sylvain Peyruqueou, and Gauthier Picard. ADELFE: A Methodology for Adaptive Multi-agent Systems Engineering. 2577, September 16–17, 2002, Madrid, Spain. ISSN: 0302-9743 (Print) 1611-3349 (Online). ISBN: 3-5401-4009-3, 978-3-54014-009-2. doi:10.1007/3-540-39173-8_12. Online available at http://www.agent.ai/download.php?ctag=download&docID=464 and ftp://ftp.irit.fr/IRIT/SMAC/DOCUMENTS/PUBLIS/ESAW02_Bernon.pdf [accessed 2008-12-01].

[14] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley, 2003. ISBN: 0-3211-9770-4.

[15] Michael Zapf and Thomas Weise from the University of Kassel, FB-16, Distributed Systems Group, Wilhelmshöher Allee 73, 34121 Kassel, Germany. Offline Emergence Engineering For Agent Societies. Kasseler Informatikschriften (KIS) 2007, 8, University of Kassel, FB16, Distributed Systems Group, Wilhelmshöher Allee 73, 34121 Kassel, Germany, December 7, 2007. Persistent Identifier: urn:nbn:de:hebis:34-2007120719844. Online available at https://kobra.bibliothek.uni-kassel.de/handle/urn:nbn:de:hebis:34-2007120719844 and http://www.it-weise.de/documents/files/ZW2007EUMASTR.pdf [accessed 2007-11-20], see also [19].

[16] *Proceedings of BIONETICS 2007, 2nd International Conference on Bio-Inspired Models of Network, Information, and Computing Systems*, December 10–13, 2007, Radisson SAS Beke Hotel, 43. Terez krt., Budapest H-1067, Hungary. In-

stitute for Computer Sciences, Social-Informatics and Telecommunications Engineering (ICST), IEEE, ACM. ISBN: 978-9-63979-905-9.

[17] John J. Grefenstette, editor. *Proceedings of the 1st International Conference on Genetic Algorithms and their Applications*, July 24–26, 1985, Carnegie-Mellon University, Pittsburgh, PA, USA. Lawrence Erlbaum Associates, Inc., Mahwah, NJ, USA. ISBN: 0-8058-0426-9.

[18] Mohammad Adil Qureshi. Evolving Agents. Research Note RN/96/4, UCL, Gower Street, London, WC1E 6BT, UK, January 1996. Online available at http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/AQ.gp96.ps.gz and http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/qureshi_1996_eaRN.html [accessed 2008-09-02]. See also [18].

[19] Michael Zapf and Thomas Weise from the University of Kassel, FB-16, Distributed Systems Group, Wilhelmshöher Allee 73, 34121 Kassel, Germany. Offline Emergence Engineering For Agent Societies. In *Proceedings of the Fifth European Workshop on Multi-Agent Systems (EUMAS'07)*, December 14, 2007, Elmouradi Hotel, Hammamet, Tunesia. Also presented at the co-located Fifth Technical Forum Group (TFG5). Online available at http://www.it-weise.de/documents/files/ZW2007EUMASTR.pdf [accessed 2009-01-07]. See also [15].