2010 Seventh IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems

Improving the Efficiency of Self-Organizing Emergent Systems by an Advisor

Jan-Philipp Steghöfer*, Jörg Denzinger†, Holger Kasinger* and Bernhard Bauer*
*Institute for Software and Systems Engineering
University of Augsburg, Augsburg, Germany
{steghoefer, kasinger, bauer}@informatik.uni-augsburg.de
†Department of Computer Science
University of Calgary, Calgary, Canada
denzinge@cpsc.ucalgary.ca

***Abstract*—Self-organizing emergent systems, also referred to as Decentralized Autonomic Computing systems, are commonly known for their scalability, robustness, flexibility, and adaptivity rather than their efficiency. However, certain application scenarios, in particular in industrial settings, require a high degree of efficiency from these systems as well, in order to keep operational expenditures and energy use small. In this paper, we therefore present the concept of an advisor, designed to improve the efficiency of self-organizing emergent multi-agent systems solving industrial problems with recurring tasks. The advisor autonomously identifies the recurring tasks at runtime and provides the agents with advice for better solutions in the future, if indicated. The advisor does not limit the self-organizing behavior of the underlying system, i. e. all problem-solving decisions are still locally made by the agents. Experiments with instances of dynamic pickup and delivery problems show that the advisor concept can achieve substantial efficiency improvements, even if the recurring tasks change over time.**

***Keywords*-decentralized autonomic computing; control; self-adaptive; software architecture;**

I. Introduction

Self-management, as the essence of Autonomic Computing (AC), is considered to be one of the crucial means of computer systems to adapt to changing conditions at runtime automatically [1]. Because today's computer systems are composed of an increasing number of distributed elements[1], whose actions and interactions cannot be monitored and controlled by a single element anymore, in recent years, self-management solutions started to shift from priorly quite centralized architectures (see e. g. [2]) to more and more decentralized architectures [3]. In Decentralized Autonomic Computing (DAC) systems [4], also referred to as self-organizing emergent systems [5], the desired self-managing behavior along with the required self-* properties emerge on the macroscopic system or application level solely from the local behavior of the system elements on the lower microscopic level. In contrast to the traditional exogenous self-management approaches advocated by AC, where an additional subsystem implements a control loop to adapt the structure or behavior of the system, in such endogenous self-management approaches the system elements thus adapt their structure or behavior to changing conditions themselves and cooperatively realize a system adaptation [6].

By taking inspirations from fields such as biology or physics (see e. g. [7], [8], [9]), for instance, research in the field of self-organizing emergent systems thus in recent years focused to a great extent on the identification of decentralized coordination mechanisms that achieve a desired self-managing and problem-solving behavior. However, experiences with these systems confirmed that the price for such scalable, robust, flexible, and adaptive solutions is a loss of performance, in particular efficiency, which is very detrimental to these systems. For instance, consider a self-organizing emergent multi-agent system (MAS) solution to dynamic pickup and delivery problems[2] (PDPs), as e. g. proposed in [11] or [12]. 'Suboptimal' respectively inefficient local decisions by the autonomous vehicles will not only lead to longer routes and hence higher operational expenditures, but also to higher energy use and pollutive CO_2 emissions regarding the environment.

In this paper we therefore present the general concept of an advisor, able to improve the efficiency of arbitrary self-organizing emergent MASs at runtime. Although this so-called *efficiency improvement advisor* (EIA) is a dedicated agent, which is added to the basic MAS and implements a control loop, in contrast to existing solutions it takes into account the low observability and poor controllability of self-organizing emergent systems, considers their openness and autonomy, and preserves the basic self-organizing emergent behavior. As a result, the approach maintains all beneficial properties of the systems mentioned above. The EIA collects the local history of the other agents and, based on the aggregated global history, autonomously detects recurring task patterns the system had and potentially will have to fulfill, but which are currently solved far from optimal. Based on a global optimization function, which in particular

[1]System elements can be autonomous software entities such as agents as well as autonomous real-world entities with computing and networking capabilities such as servers, mobile devices, robots, or modern cars

[2]In a dynamic PDP [10], a previously unknown set of goods has to be transported between a set of origins and a set of destinations by a set of vehicles using an appropriate transportation network.

incorporates the energy use, the EIA calculates the optimal solution for these task patterns and provides the agents with advice in form of exception rules that the agents can add to their own problem solving behavior. Because as a result all problem solving decisions are still locally made by the agents, the latter will continue to work, even if the EIA fails. By this approach we obtain the benefits of a central control but avoid its associated problems such as a single point of failure or a bottleneck.

The rest of this paper is organized as follows: Section II discusses the general challenges for achieving optimality in self-organizing emergent MASs in more detail. Section III formally describes the general concept of the EIA, whereas Section IV presents an instantiation of this general concept to the domain of PDPs. Section V presents and evaluates the efficiency improvements that we could achieve by means of the EIA in experiments with a self-organizing emergent MAS solution to dynamic PDP instances. Section VI reviews related work, whereas Section VII finally presents some concluding remarks.

II. Improving the Efficiency of Self-organizing Emergent Multi-Agent Systems

More and more real-world application scenarios require to solve highly dynamic, complex, and often unpredictable problem instances, such as e. g. transportation scenarios. A key problem is the fact that the problem instances, in more detail the tasks that have to be fulfilled, change dynamically. Therefore, solving such instances by self-organizing emergent MASs optimally requires on the one hand as optimal local agent decisions as possible and on the other hand appropriate system adaptations preferably maintaining a high degree of autonomy.

In order to make 'optimal' local decisions on its own, an agent would have to be in possession of an abundance of relevant information, including information about the current and future state of the system environment, in particular the problem-relevant tasks, the environment topology (e. g. networks, machines, customers, ...), and the current and future intended behavior of other agents. This would not only force the agents to quickly gather real-time information from a large number of (possibly unknown) entities, but also to be able to "look into the future", such that a dynamically appearing task can be assigned to the best agent (with respect to global optimality of the solution), while other tasks are already executed by the agents.

Because this would be a very complex endeavor, in the literature, hybrid solution approaches very often can be found that consist of one or more additional hierarchy levels, e. g. defined on physical or organizational boundaries (see [13], [14], for instance). Thereby, an agent on a higher level is in charge of optimizing respectively adapting the agents on the lower level, usually by implementing a control loop. However, when considering the engineering of hybrid solutions for self-organizing emergent MASs, the following additional challenges and constraints have to be respected:

1) *Consider the openness and autonomy of the underlying system:* A higher level agent has to autonomously adapt the behavior or structure of a basically open MAS depending on current, past, and future situations, which were potentially unexpected and unforeseeable at design time. This process should not limit the autonomy of the underlying system, i. e. all problem solving decisions must still be made locally by the agents on the lower level themselves.
2) *Take into account the low observability and poor controllability:* A higher level agent will neither be able to observe all (inter)actions of lower level agents, if ever, nor be able to gather all relevant information at the time of occurrence, just because of the nature of self-organization and emergence [15]. Thus, the higher level agent will not be able to optimally adapt or influence the behavior, intention, or upcoming action of any lower level agent yielding immediate effects.
3) *Preserve the basic self-organizing and emergent behavior:* A higher level agent may neither limit the basic system's self-organizing and emergent problem-solving behavior, nor limit its scalability, robustness, flexibility, or adaptivity. This implicates that the higher level agent may not act as a central controller and thus as a bottleneck and single point of failure. Consequently, if the higher level agent crashes, the agents on the lower level still have to function properly.

Although there naturally exist a couple of hybrid solution approaches in various fields (see Section VI), they very often either take over central control in certain situations, assume to be able to observe and control the underlying MAS at every point in time, do not preserve the basic beneficial properties, or are unable to "look into the future".

III. Efficiency Improvement Advisor Approach

In this section, we formally describe our concept of an advisor as an appropriate higher level agent, which respects the just mentioned challenges and constraints and is to some extent able to "look into the future". We first introduce the general formal scheme, which we will instantiate in order to describe the participating autonomous agents on the lower level, as well as provide basic notations about the general problem setting these agents are supposed to solve.

A. Basic Definitions

A very generic definition of an agent $\mathcal{A}g$ is as a 4-tuple $\mathcal{A}g = (Sit, Act, Dat, f_{Ag})$, where Sit is the set of situations the agent can face (i.e. its possible view of the environment), Act is the set of actions $\mathcal{A}g$ can perform, Dat the set of possible values of the agent's internal data areas and $f_{Ag} : Sit \times Dat \rightarrow Act$ the agent's decision function, describing how $\mathcal{A}g$ selects an action based on its current situation and

the current value of its internal data areas (i. e. its perceptions of the world and its current knowledge status). This assumes that there is an action for every combination of activities the agent can do. If f_{Ag} is not much influenced by the value of Dat, $\mathcal{A}g$ is called reactive. A MAS is then a group of agents $A = \{\mathcal{A}g_1, \ldots, \mathcal{A}g_n\}$ that share an environment $\mathcal{E}nv$. The agents in A might be heterogeneous, i. e. they may all have different sets of situations, actions, internal data area values, and also different decision functions.

The general structure of problems that have to be solved by a set of agents A we focus on consists of tasks out of a set T that are announced to A at some times within a given time interval $Time$ to form a *run instance* for the system A. Usually, there will be a sequence of run instances that A has to solve. For instance, a run instance could be all the tasks A has to solve at a particular day, whereas a sequence of run instances are the tasks to solve over several days. Naturally, a task for a concrete application will be described by a set of features, but for the general description generic tasks and the time of their announcement are enough.

Formally, we describe a run instance as a sequence $((ta_1, t_1), (ta_2, t_2), \ldots, (ta_m, t_m))$, with $ta_i \in T$, $t_i \in Time$ and $t_i \leq t_{i+1}$. A sequence of run instances of length k is then described as $((ta_{11}, t_{11}), (ta_{21}, t_{21}), \ldots, (ta_{m_1 1}, t_{m_1 1})), \ldots, ((ta_{1k}, t_{1k}), (ta_{2k}, t_{2k}), \ldots, (ta_{m_k k}, t_{m_k k}))$. A solution sol generated by A for a run instance is again a sequence $sol = ((ta'_1, \mathcal{A}g'_1, t'_1), (ta'_2, \mathcal{A}g'_2, t'_2), \ldots, (ta'_m, \mathcal{A}g'_m, t'_m))$ where $ta'_i \in \{ta_1, \ldots, ta_m\}$, $ta'_i \neq ta'_j$ for all $i \neq j$, $\mathcal{A}g'_i \in A$, $t'_1 \leq t'_{i+1}$, $t'_i \in Time$. A tuple $(ta'_i, \mathcal{A}g'_i, t'_i)$ means that task ta'_i will be started by $\mathcal{A}g'_i$ at time t'_i.

Please note that $\{t_1, \ldots, t_m\}$ and $\{t'_1, \ldots, t'_m\}$ do not have to be related in any way, i. e. tasks do not have to be immediately started by one of the agents when they are announced. This, at least theoretically, allows for the possibility that the agents in A can be more than purely reactive. Also, solving ta'_i might require a sequence of actions by $\mathcal{A}g'_i$. Depending on the application there might be additional restrictions, for example, because sometimes not every agent can perform every task, we require that $\mathcal{A}g'_i$ can indeed perform ta'_i.

As mentioned, users or operators of a self-organizing emergent system A associate with a solution sol a quality $qual(sol)$, which is naturally dependent on the particular application that the agents in A have been created for. Apparently, A is expected to produce a solution that is of optimal quality. Under some circumstances engineering an A that produces optimal solutions is easy, but under many circumstances it is difficult (e. g. for NP-complete problems) or even impossible. In particular if a task ta can arrive at any point in time within $Time$, then the requirement that all tasks need to be started within $Time$ (which often is accompanied by the additional requirement that all tasks also need to be fulfilled within $Time$) will often lead to suboptimal solutions.

Consequently, in this paper we are interested in self-organizing emergent MASs that solve dynamic run instances for which at least some of the tasks are announced to the agents later than other tasks, i. e. there is at least one t_i such that $t_i < t_{i+1}$ (and usually there are more than just one such t_i). Since the agents do not know at the beginning of the interval $Time$ what all tasks will be, it is in most cases impossible for the agents to solve the whole (dynamically developing) run instance optimally.

B. Architectural Overview

Based on these definitions and notations, our approach to improve the efficiency of a self-organizing emergent system $A = \{\mathcal{A}g_1, \ldots, \mathcal{A}g_n\}$ over a sequence of run instances is to add a dedicated higher level agent $\mathcal{A}g_{EIA}$, the efficiency improvement advisor, to A (see Figure 1) that collects the local histories of all agents, creates a global view of the history of A (and the environment around A as far as possible), identifies sequences of recurring tasks, calculates the optimal solution of these task sequences, creates advice for the individual agents, allowing them to better deal with the task they did not solve very well, and makes this advice available in form of so-called exception rules[3]. This approach requires the following conditions to be fulfilled:

- Each agent is able to transmit a history of its local behavior to the advisor at least once during or after a run instance
- Each agent's decision function can be extended to deal with exception rules (stored in its internal data area)
- A sequence of run instances must have a (sub)set of similar tasks in (nearly) each instance of the sequence

While the first two conditions usually are achieved easily, the third condition seems very restrictive at a first glance. But in everyday life, there are many problems that fulfill this condition, e.g., transportation companies usually have daily recurring tasks together with one-of-a-kind tasks. In the following, we explain the functional architecture of an $\mathcal{A}g_{EIA}$ along with its actions more precisely.

C. Functional Overview

Due to the advisory role, Sit_{EIA} contains information about the communication with members of A. Dat_{EIA} represents the data received from the $\mathcal{A}g_i$ and the intermediate results by $\mathcal{A}g_{EIA}$'s actions towards creating advice for the $\mathcal{A}g_i$s. Its decision function f_{EIA} creates the following steps represented by the indicated actions out of Act_{EIA}[4]:

[3] $\mathcal{A}g_{EIA}$ not necessarily has to be a new agent, it can also be a role of one of the $\mathcal{A}g_i$ or all agents in A can share performing the actions of $\mathcal{A}g_{EIA}$. However, this requires extensive communication between the $\mathcal{A}g_i$ and might require more computing power in an $\mathcal{A}g_i$ than is possible in a particular application. A stationary agent with lots of computing power and occasional communication with the $\mathcal{A}g_i$s is a reasonable extension to many existing systems for the kind of problems we are interested in, which is why we present our approach in this manner.

[4] These steps also implicitly define what Sit_{EIA} and Dat_{EIA} have to contain.

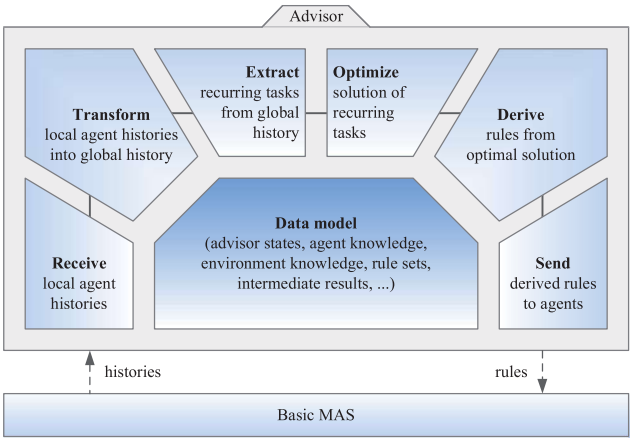

Figure 1. Functional advisor architecture

1) **receive**($\mathcal{A}g_i$, ((s_i^1, d_i^1, a_i^1), …,(s_i^o, d_i^o, a_i^o))) collects the local history H_i for each agent $\mathcal{A}g_i$, if $\mathcal{A}g_i$ is able to communicate, while A performs a run instance. H_i = ((s_i^1, d_i^1, a_i^1), …, (s_i^o, d_i^o, a_i^o)), with $s_i^l \in Sit_i$, $d_i^l \in Dat_i$, $a_i^l \in Act_i$, is the history of $\mathcal{A}g_i$ since the sequence of run instances started.
2) **transform**(H_1, …, H_n) creates the global history $GHist$ out of the received histories of all agents. $GHist$ essentially contains the sequence of run instances (ri_1, …, ri_k) = ((ta_{11}, t_{11}), …, ($ta_{m_1 1}$, $t_{m_1 1}$)), …, ((ta_{1k}, t_{1k}), …, ($ta_{m_k k}$, $t_{m_k k}$)) A has solved so far and the solution sol_j for each run instance ri_j that A created for it.
3) **extract**($GHist$) extracts from this history, more precisely the sequence of run instances (ri_1, …, ri_k), a sequence of recurring tasks (ta_1^{rec}, …, ta_p^{rec}).
4) **optimize**(ta_1^{rec}, …, ta_p^{rec}) computes the optimal solution opt^{rec} = ((ta_1^{rec}, $\mathcal{A}g_1'^{rec}$, $t_1'^{rec}$), …, (ta_p^{rec}, $\mathcal{A}g_p'^{rec}$, $t_p'^{rec}$)), $\mathcal{A}g_j'^{rec} \in A$, $t_j'^{rec} \in Time$, if ta_1^{rec}, …, ta_p^{rec} were the only tasks A had to perform and they would be all known at the beginning of $Time$. It then compares $qual(opt^{rec})$ with the quality $qual(last)$ of the last emergent solution $last$ for the tasks (ta_1^{rec}, …, ta_p^{rec}) A has created. If
$$qual(last)/qual(opt^{rec}) > qualthresh,$$
the work of $\mathcal{A}g_{EIA}$ is done until new information arrives, since A performs well.
5) **derive**(opt^{rec}, (ta_1^{rec}, …, ta_p^{rec}), $GHist$, $last$) otherwise creates for each agent $\mathcal{A}g_i$ a set R_i of exception rules, where R_i can also be empty.
6) **send**($\mathcal{A}g_i$,R_i) communicates the set R_i of exception rules to an agent $\mathcal{A}g_i$ the next time communication with $\mathcal{A}g_i$ is possible.

Apparently, concrete realizations of these actions depend on the application at hand and on the realization of the $\mathcal{A}g_i$s, including their coordination principles. However, the actions **extract** and **derive** allow for different general ways of realization, which we discuss further in the next two subsections. At the end we comment on the other actions.

1) Extracting Recurring Tasks: In principle, there are several ways to find recurring tasks in a sequence of run instances. But for many applications, the problem is more complicated than just finding tasks that occur in each run instance. Thus, we are not only interested in tasks that are identical in all run instances but also in tasks for which *similar* tasks exist in all (or at least most) of the run instances. For example, the task of delivering something to a particular house in a street is usually not very different from delivering to a neighboring house, so that having one delivery each day to one of the two houses should put this task into the sequence of recurring tasks.

More precisely, we assume the existence of an application dependent similarity measure $sim : T \times T \rightarrow \mathbb{R}^+$ that is used by the action **extract**. This measure can then be used to cluster the tasks in ri_1, …, ri_k according to sim, in order to identify the recurring tasks in these run instances[5]. A clustering method that is useful for this problem is Sequential Leader Clustering (SLC), see [16], because it does not require initially stating the number of clusters it should produce. SLC in its original form works on the tasks in ri_1, …, ri_k one after the other. If up to task ta in one of the ri_j it has produced the clusters C_1 , …, C_x with $c_i \in C_i$ being the representative of the cluster C_i, then we compute $sim(ta, c_i)$ for all clusters. If cluster C_q is the one with the biggest similarity to ta, then ta is added to C_q if $sim(ta, c_q) > clustthresh$ for a given parameter $clustthresh$. If it is added, then it needs to be checked if the representative for C_q has to be changed. The determination of the representative depends on the description of the task. If ta is not similar enough to any of the clusters, then a new cluster $C_{x+1} = \{ta\}$ is created.

If C_1, …, C_x is the result of the clustering process, then the next step is to determine all clusters that are big enough to indicate that they represent recurring tasks. With k run instances, these are all clusters C_i with $|C_i| \geq minocc \cdot k$, with $0 < minocc \leq 1$ a user determined parameter. If C_1', …, C_y' are all clusters fulfilling this condition, then we put the $c_i' \in C_i'$ with $sim(c_i', c_i)$ is minimal into the set of recurring tasks. If there are C_i' with $|C_i'| \geq (1 + minocc) \cdot k$, this indicates that the task represented by this cluster is usually occurring several times in the run instance. Thus, $\mathcal{A}g_{EIA}$ not only puts the $c_i' \in C_i'$ with $sim(c_i', c_i)$ is minimal into the set of recurring tasks, but also the $c_i'' \in C_i' \backslash \{c_i'\}$ with $sim(c_i'', c_i)$ minimal (and so on, if $|C_i'| \geq (2 + minocc) \cdot k$, etc.)

It should be noted that for applications where the recurring tasks can change over time, **extract** should not use all k

[5]Other techniques from the area of data mining that are able to identify recurring event sequences in data could be used as well, of course.

run instances from the beginning of A's work, since most probably after some time the set of recurring tasks will become very small or even empty. In such cases, a parameter k_{max} has to be defined, only the run instances $ri_{k-k_{max}}$, $ri_{k-k_{max}+1}$, ..., ri_k are used in the clustering, and k_{max} is used in the conditions using $minocc$. While a change in the recurring tasks obviously will not be noticed immediately, at the latest k_{max} run instances after the change $\mathcal{A}g_{EIA}$ will be aware of the change and will create new advice for the agents in A. Naturally, if new changes happen faster than k_{max} run instances, $\mathcal{A}g_{EIA}$ will not be able to detect recurring tasks very well and A will have to rely on the basic decision making of its agents.

2) Deriving Exception Rules: Deriving advice for the agents in A from opt^{rec} and $last$ depends on what kind of exception rules the agents in A can handle. In general, there are two possible aims for exception rules: (1) they can be used to encourage an agent to take on a certain task or (2) they can be used to detract an agent from taking a certain task. In a MAS, this opens a wide spectrum of possibilities. But for the considered kind of problems, detraction from a task is the only possibility, since it is not easily possible to encourage an agent to execute a task that it cannot know about, because it will only be announced to it later. A detraction rule should be rather specific about the circumstances when an agent should follow it, so that it really is only an exception and, for example, the not-occurrence of an expected task will not block an agent for too long, if an exception rule was created to detract this agent from other tasks in anticipation of the expected task. Specific circumstances in rules can also allow for not having to come up with a conflict resolution mechanism for the agents that deals with determining what to do if several exception rules are applicable to a situation.

An exception rule for an agent $\mathcal{A}g_i$ has the form $cond_{exc}(s,d) \rightarrow \neg a_{ta}$, with $s \in Sit_i$, $d \in Dat_i$ and $a_{ta} \in Act_i$ (with a_{ta} indicating the action to start performing task ta). The effect of such a rule on $\mathcal{A}g_i$'s behavior can be described as creating a variant $f'_{\mathcal{A}g_i}$ of $\mathcal{A}g_i$'s decision function $f_{\mathcal{A}g_i}$. Such an exception rule to detract the agent from its "normal" action a_{ta} (as indicated by $\neg a_{ta}$) creates

$$f'_{\mathcal{A}g_i}(s',d') = \begin{cases} f_{\mathcal{A}g_i}(s',d'), & \text{if } cond_{exc}(s',d') = \text{false} \\ a', & \text{with } a' \neq a_{ta}, \text{else} \end{cases}$$

The action a' should be the action that $\mathcal{A}g_i$ would take without knowing that task ta needs to be done. Detracting an agent from an action is a rather weak form of control, since the agent is only told what *not* to do and still needs to figure out itself what to do. This way preserves the basic decision making of the agents as much as possible. Note that $cond_{exc}$ can be defined so that the agent's current Dat-value does not matter, which allows $\mathcal{A}g_{EIA}$ to give advice to agents it does not know much about. But including the Dat-value will allow for better targeting of the exception.

To decide, which exception rule to create for which agent, $\mathcal{A}g_{EIA}$ has to compare the two solutions opt^{rec} and $last$. If opt^{rec} = $((ta_1^1, \mathcal{A}g_1^1, t_1^1), \ldots, (ta_p^1, \mathcal{A}g_p^1, t_p^1))$ and $last$ = $((ta_1^2, \mathcal{A}g_1^2, t_1^2), \ldots, (ta_p^2, \mathcal{A}g_p^2, t_p^2))$, then $\mathcal{A}g_{EIA}$ looks for the first j with $ta_j^1 \neq ta_j^2$ or $\mathcal{A}g_j^1 \neq \mathcal{A}g_j^2$. Since solutions are sorted according to the t_i-values, this is really the first assignment of a task to an agent for which the agents in A deviated from the optimal solution for the recurring tasks.

The created exception rule is then for agent $\mathcal{A}g_j^2$ and naturally has the form $cond_{exc}(s',d') \rightarrow \neg a_{ta_j^2}$. For determining $cond_{exc}(s',d')$, $\mathcal{A}g_{EIA}$ looks up in $GHist$ the triple $(s,d,a_{ta_j^2})$ that represents in the history of $\mathcal{A}g_j^2$ the point when it chooses to do $a_{ta_j^2}$. $cond_{exc}(s',d')$ is then an abstraction of s and d that is application dependent and tries to cover not only an activation of ta_j^2, but the whole cluster from the **extract** step of which ta_j^2 is a member.

For the application we present in Section IV, this exception rule for $\mathcal{A}g_j^2$ was all we created in one working cycle of $\mathcal{A}g_{EIA}$. However, for certain applications it might be better to test what the agents in A will do after the exception rule is communicated. If $\mathcal{A}g_{EIA}$ knows enough about the agents in A and has enough time, it can simulate which new solution A would produce for the recurring tasks and if this new solution $last'$ is still too bad, the above rule creation steps can be repeated until the emergent solution of A is good enough. Then all rules created using the simulation would be communicated to the real agents in A.

3) Other Actions: The action **transform** can be very easily realized if the environment $\mathcal{E}nv$ is known to $\mathcal{A}g_{EIA}$ and the only events happening are the announcement of tasks and the actions taken by the agents in A. Without a readily available global view, **transform** essentially has to create some kind of environment "map" out of the perceptions of the $\mathcal{A}g_i$ as represented by their local view on the situations they encountered. In this case, it is also possible that not all tasks will be observed, simply because no agent might be in a situation to observe a particular task being announced. This is one of the reasons why we do not require that a recurring task has to appear in every run instance.

The action **optimize** requires $\mathcal{A}g_{EIA}$ to have an optimization system that handles the static optimization problem to the dynamic problem that A tries to solve. Although this optimizer does not have to look into the future, the static optimization problem still can be very difficult to solve. Then, or if the time between run instances is too short, only searching for a very good solution for the recurring task sequence (perhaps combined with a lower $qualthresh$-value) instead of the optimal one is more appropriate.

Determining the emergent solution created by A for the recurring tasks is not trivial. There will often be other tasks mixed in into fulfilling the recurring tasks, or in the last run instance not all of the recurring tasks might have occurred (the size of the clusters representing the recurring tasks

can be smaller than $k!$). The fact that the agents fulfill other tasks while fulfilling the recurring tasks means that $\mathcal{A}g_{EIA}$ cannot determine the quality based on measuring what really happened. For example, between fulfilling two tasks of the recurring task set in a transportation domain, an agent might have to drive to a far off location to fulfill a not recurring task in a particular run instance. Adding the traveled distance of this agent between the two recurring tasks to the travel cost (if this is the quality criterion) would deteriorate the emergent solution, although in other run instances the recurring tasks are solved well. But for such an application a lower bound for the costs that would emerge can be provided, if there were no additional tasks, namely the distance the agent has to travel after the first recurring task to start performing the second one. Such lower bounds are possible to be determined for many applications and many quality criteria. If the quality of such a lower bound is far from the optimum, then advice from $\mathcal{A}g_{EIA}$ will be useful for many run instances. Given that $last$ and its quality are already an approximation, the problem of a recurring task not occurring in the last run instance can now be solved, too. $\mathcal{A}g_{EIA}$ determines, which agent fulfills the task in the emergent solution by going back one more run instance (or several). This also allows to determine the correct position of the task in the sequence of tasks the agent performs.

IV. Instantiating the EIA for PDPs

In this section, we first briefly describe PDPs as an instantiation of the general problem class the EIA approach aims at. We then describe a self-organizing emergent MAS solution based on digital infochemical coordination (DIC) [9] instantiated for PDPs. To improve its efficiency, we finally instantiate the EIA approach from the last section to PDPs, in order to be used for the DIC-based solution.

A. Pickup and Delivery Problems

The general PDP [17] is a well-known problem class that has instantiations such as transportation problems in internal and external logistics. Many of these instantiations fulfill the conditions mentioned in the last section, i. e. not all tasks to perform are known in advance and there are tasks that appear in many run instances. Additionally, vehicles usually return to a depot after a run instance, which can be used to house the EIA. In this paper, we instantiate the PDP with time windows (PDPTW), which generalizes PDP to require delivery within a task dependent time frame. A task ta_{PDP} for this problem consists of a location l_{pickup} where goods needs to be picked up, a location $l_{delivery}$ where the goods have to be dropped off, the needed capacity $ncap$ and times t_{start} and t_{end} in $Time$ defining the time window in which both pickup and delivery have to happen, i. e. ta_{PDP} = (l_{pickup}, $l_{delivery}$, $ncap$, t_{start}, t_{end}). An agent $\mathcal{A}g$ has a transport capacity $cap_{\mathcal{A}g}$ and has first to perform the pickup and then the delivery to accomplish a task.

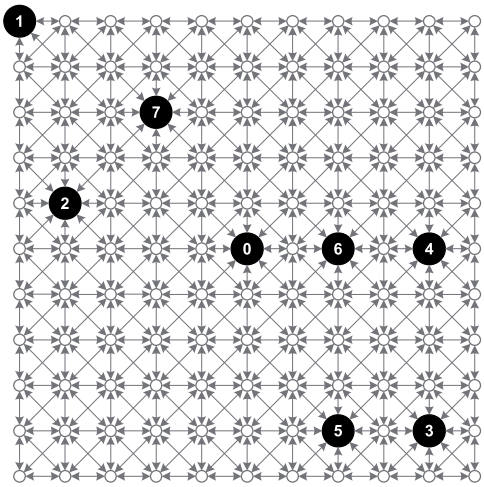

Figure 2. A "bad" PDP run instance

B. DIC-based Solution to PDPs

In principle, there are different ways how transport agents can coordinate their actions in a self-organizing manner to perform pickup and delivery tasks. DIC, which generalizes pheromone-based coordination [7], represents an approach that is applicable for various problems providing a higher variety of (digital) chemicals to be used for the coordination of agents, to achieve effects that are beneficial for chemical emitting agents, receiving agents, or both.

A MAS solution for PDPs based on DIC was already presented in [11], thus we only briefly describe this solution, concentrating on the decision making of the agents, which will be later advised by the EIA. A DIC-based system achieves coordination between transportation agents $\mathcal{A}g_1$, ..., $\mathcal{A}g_n$ solely using digital infochemicals that are propagated through the environment the agents are situated in. For the experiments, this environment is a map organized in a grid (see e. g. Figure 2) and an $\mathcal{A}g_i$ can "access" all digital infochemicals of the location it is currently situated on, which provides it with a very local view of the system. Naturally, arbitrary maps are possible, too.

A task ta_{PDP} is given to the system by "creating" two emitter agents in the environment, one at l_{pickup} emitting a so-called synomone (a type of infochemical) that specifies the location and the transportation requirements, i. e. $ncap$ and t_{start}, and one at $l_{delivery}$ also specifying the task via a synomone. In the experiments, the task announcement was done at t_{start}. All such synomones are propagated through the environment and a location receiving them stores their existence and intensity. The chemical evaporates after a certain time, which is why an emitter agent repeats the synomone emission from time to time until it has been served by a transportation agent.

A transportation agent $\mathcal{A}g_i$ "smells" all chemicals at its current location and computes a utility for each task represented. In addition to the intensity of a chemical, the utility is also influenced by the agent's current status. If it has e. g. already picked up the goods for a task, it gives priority to delivering it. Factors such as how close a task's t_{end} is are used as well. Additionally, other infochemical types influence the utility computation: a pickup agent emits so-called allelochemicals as soon as an $\mathcal{A}g_i$ served it. This indicates to an $\mathcal{A}g_j$ that the task execution at this location has already started, which prevents it from being unnecessarily attracted to this location by unevaporated synomones, i. e. it will not give any utility to this task anymore. Also, transportation agents emit pheromones indicating the task they currently intend to perform. These pheromones are – in contrast to synomones – only propagated in a very small area, but other agents crossing such a pheromone trail then know not to choose the task for themselves. After an $\mathcal{A}g_i$ has computed the utility for all tasks it perceives, it selects the task with highest utility and moves directly towards the synomone emitter representing this task.

This DIC-based approach is a typical example for a self-organizing emergent MAS the EIA is applicable for. The transportation agents use only local information for the decision making. The system as a whole is very robust, since the breakdown of a transportation agent might lower the global efficiency but does not lead to its breakdown. After the pheromones of a broken-down agent have evaporated, the system has adapted to operate with one less agent.

C. EIA for DIC-based Solutions to PDPs

In order to instantiate the EIA approach for such DIC-based solutions to PDPs, first the quality of a solution has to be defined. For PDPs, very often the quality function is based on a single measure, such as minimizing the travel costs of the transportation agents. However, we decided to use a function that combines several measures of interest for PDPs and that in particular aims to combine energy use related measures, like travelled distance, with other kinds of measures. Thus, $qual(sol)$ is defined as

$$\frac{100000}{\kappa \cdot qual_{dist}(sol) + \lambda \cdot qual_{order}(sol) + \mu \cdot qual_{tw}(sol)}$$

where κ, λ and μ are weight parameters and $qual_{dist}(sol)$ is the distance traveled by all $\mathcal{A}g_i$ according to sol, $qual_{order}(sol)$ is the penalty accumulated by sol for fulfilling tasks in a different order than their announcement to the system, and $qual_{tw}(sol)$ is the penalty for sol for all tasks that are not completed within the time window for the task. The factor 100000 is used to end up with $qual$-values that are larger than 1. For $qual_{order}$, the difference between the start times of all pairs of tasks ta_1 and ta_2 is summed up, if the start time of ta_1 is before ta_2 but in sol ta_2 is finished before ta_1. For $qual_{tw}$, the difference between the finishing time for a task and t_{end} is summed up, if the finishing time is later than t_{end}. Obviously, the three measures of interest are not aligning well with each other, so that many solutions have rather similar $qual$-values, which makes improvement by the EIA even harder, since it is less likely to have a big difference in quality between the emergent solution and the optimal solution. Please note that penalizing for tasks not done in order also favors making decisions based on local information, favoring the self-organizing system, again. Note also that for this $qual$-function higher values represent better solutions.

The instantiation of the actions **receive** and **transfrom** requires to create a description of the announced tasks only from the histories of all $\mathcal{A}g_i$ and their local perceptions. Having the run instances available in the form (ri_1, …, ri_k), recurring tasks can be identified by **extract** using

$$\begin{aligned} sim(ta_1, ta_2) \quad = \quad & \alpha \cdot Ed(l_{pickup,1}, l_{pickup,2}) + \\ & \alpha \cdot Ed(l_{delivery,1}, l_{delivery,2}) + \\ & \beta \cdot |ncap_1 - ncap_2| + \\ & \gamma \cdot (|t_{start,1} - t_{start,2}| + |t_{end,1} - t_{end,2}|) \end{aligned}$$

where Ed is the eucledian distance. The instantiation of the action **optimize** requires to create the optimal solution for the identified recurring tasks. Although the PDP is rather well researched, we were not able to find an efficient public-domain optimizer, so we had to implement a simple branch-and-bound-based optimizer for the proof-of-concept. This optimizer is not very sophisticated and as a result the size of problems it could tackle in acceptable time is limited. However, the experiments in Section V show that the optimizer was good enough to demonstrate the improvement abilities of the EIA approach.

To determine the $qual$-value of the emergent solution, we compute $qual_{dist}$ by using the direct distances between the emitter agents for the recurring tasks. The $qual_{order}$- and $qual_{tw}$-values are directly available out of the last run instance containing the particular recurring tasks, but as mentioned $qual_{tw}$ can be heavily influenced by the non-recurring tasks of this run instance.

The instantiation of **derive** has to be able to detract an agent from a particular task, resp. all similar tasks. For the DIC-based system, we therefore add exception rules to the synomone utility computation performed by an agent: any synomone, which is sufficiently similar to an abstracted synomone ab given in the condition of an exception rule, is not considered, resp. its utility is zero. ab consists of the elements $l_{pickup,ab}$, $ncap_{ab}$ and $t_{start,ab}$ of the task that should not be taken by the agent. To determine the similarity of a synomone representing a concrete task ta_1 to the abstracted synomone ab, $\mathcal{A}g_{EIA}$ computes the value

$$\begin{aligned} dist_{syn}(ab, ta_1) \quad = \quad & Ed(l_{pickup,ab}, l_{pickup,1}) + \\ & |ncap_{ab} - ncap_1| + \\ & |t_{start,ab} - t_{start,1}| \end{aligned}$$

If the distance to the abstract synomone $dist_{syn}$ is below a given threshold $synthresh$, then the associated exception

rule will be applied. Each exception rule is only applied for a limited time, which prevents tasks being ignored for too long, if the agent the EIA designated to serve the ignored task fails or is busy with servicing other tasks.

V. Experimental Evaluation

To evaluate the usefulness of the EIA approach, we have performed several experiments with the instantiation described in the last section. Creating experimental sequences of run instances that allow an appropriate evaluation is not a straightforward task, since the evaluation requires sequences that on the one hand include some recurring tasks, but on the other hand also have enough randomness to allow the argument that the approach will work for many problem instances.

To test the capabilities of the EIA approach, we thus have created two kinds of scenarios: those crafted to not being solved well by the underlying DIC-based system (starting with "craft-..."), see Table I) and those with a randomly created sequence of intended recurring tasks (starting with "rand-..."). The creation of all sequences of run instances in these scenarios started with a sequence of tasks that forms the set of recurring tasks the EIA should identify. Each of these tasks (or a slight modification of it within sim) is included in each run instance of a sequence with a given probability (95%, except for the craft-4-I-... scenarios, where the probability was 100%). Then some randomly created tasks were added to each of the instances, whose number was chosen randomly from between 10 to 30 percent of the number of intended recurring tasks. The length of the sequence of run instances that forms a scenario in Table I depends on the number of intended recurring tasks (indicated by the Arab number in the name), namely 10 run instances for scenarios with 4 recurring tasks, 20 run instances for 6 recurring tasks, and 40 run instances for 8 recurring tasks. Scenarios with 4 and 6 recurring tasks are situated on a 11x11 grid, scenarios with 8 recurring tasks on a 21x21 grid. All scenarios engage two transportation agents.

Figure 2 exemplarily illustrates the crafted sequence of recurring tasks for craft-4-I-... scenarios. The grid node numbered with 0 is the depot for the transportation agents. The basic set of tasks are ta_1 = (4,6,20,25,0), ta_2 = (3,5,20,25,0), ta_3 = (2,7,20,50,0) and ta_4 = (1,7,20,50,0) where a t_{end} value of 0 indicates that no time window is set. The figure displays grid field numbers instead of coordinates to make it easier to envision the tasks on the grid. The underlying system solves these four tasks (if no other tasks are there) by having both agents go to the lower right corner to perform tasks ta_1 and ta_2 and then both agents move to the upper left corner to do the other two tasks, getting quite some late delivery penalties (if we enforce the given time windows). A better strategy would have only one agent go down into the lower right corner performing ta_1 and ta_2, while the other agent waits in the depot until ta_3 and ta_4 are announced and then performs those two tasks. The crafted recurring task sequence for the craft-6-... scenarios used a similar weakness of the reactive DIC-based system.

Table I
Results for not-changing recurring tasks

Scenario	w/o EIA	w/ EIA	Impr.
craft-4-I	1319.37	1415.51	7.29%
craft-4-I-TW	210.94	240.31	13.92%
craft-6-I	704.06	797.63	13.29%
craft-6-I-TW	140.46	170.41	21.32%
rand-6-I	627.26	737.41	17.56%
rand-6-II	566.05	726.69	28.38%
rand-8-I	177.96	186.05	7.49%
rand-8-II	181.08	185.41	2.39%

Table II
Results for changing recurring tasks

Scenario	w/o EIA	w/ EIA	Impr.
chang-6-I	836.45	956.74	14.38%
chang-6-II	800.74	936.90	17.00%
chang-6-III	728.74	874.22	19.96%

For all scenarios, we used the following parameter settings: $qualthresh$ was set to 95%, $clustthresh$ = $synthresh$ = 20, $minocc$ = 0.7, k_{max} = 20, κ = 1, λ = 15, μ = 0 (except for the ...-TW scenarios, which enforce the time windows, then we used μ = 3), α = 0.3, β = 0.1, and γ = 0.3. The time to live for a exception rule was 100.

As shown in Table I, the EIA approach leads to quite some efficiency improvements for most scenarios. Remember that a higher $qual$-value (in columns w/o EIA and w/ EIA) indicates a better solution. Even some of the scenarios with a randomly created set of recurring tasks have a two digit percentage improvement. The two scenarios enforcing the time windows for the deliveries show higher improvements than the scenarios not enforcing them, which is a very good result since a PDPTW is more difficult to solve and therefore more likely to not being solved well.

Due to lack of space we can not present the detailed run instance by run instance results, but as expected there are often a few run instances where the randomly created additional tasks had as result that the advised agents were not as good as the not-advised ones, but the gains in the other run instances make more than up for this. It should be noted that we could have used this fact to boost the improvements by simply having more run instances in a scenario, but we think that the chosen numbers of runs are sufficient to show that the EIA approach is successful.

An important aspect of real-life transportation problems of the kind we are interested in is that the set of recurring tasks can change over time, e.g., a company might loose a customer contributing to the recurring task set or new such customers might be added. The EIA approach is able to deal with this, demonstrated by the experiments shown in

Table II. Scenario chang-6-I consists of 20 run instances using one set of 6 recurring tasks (the instances were created with additional random tasks and probabilities for the recurring tasks as described before). Then follow 12 run instances created using a different randomly created set of 6 recurring tasks, after which follow another 20 run instances with the first set of recurring tasks. The number 12 was chosen, because it is just too small to allow for a change in advice by the EIA. For chang-6-II, the same sets of recurring tasks have been used as in chang-6-I, but having 24 instead of 12 run instances before changing "back". As one can see by the improvement between chang-6-I and chang-6-II, if the "change" is for long enough, the EIA will adapt the agents to it. Finally, chang-6-III has three blocks of run instances using three different (random) sets of recurring tasks, each set for 20 run instances. Also for this type of scenario, the EIA demonstrates its usefulness.

Overall, the experimental evaluation shows that the EIA approach is able to provide the agents with good advice, without undermining the important and useful properties of the underlying self-organizing emergent system necessary to deal with the dynamic nature of the problem.

VI. Related Work

As mentioned, there exist a couple of related approaches representing hybrid solutions. The observer/controller (O/C) approach [18], although being very similar to the EIA, however, requires to be able to observe and control the underlying system at every point in time. Similarly, the management-by-exception approach [19] involves a higher level agent taking over total control of the agents, if certain performance conditions are not fulfilled. Both approaches do not preserve the beneficial properties of the basic system. The distributed hybrid approach presented in [6], which rather focuses on self-healing, only uses a set of rigid repair plans that cover a predefined set of faulty states. Whereas none of these approaches has the limited ability to "look into the future", in [20] a model-based control framework is presented that uses limited lookahead control (LLC) to optimize the forecast behavior of the basic system over a limited prediction horizon. However, the ability to "look into the future" is based on a fixed stochastic model and does not regard the actual system history.

Hybrid MAS optimization approaches in the PDP domain are either based on a predefined stochastic model as well (e. g. [21]), or do not even incorporate learning (see [13], [14]) and thus have to permanently control the basic system. Although learning of behavior is a technique widely applied in MASs (see [22] for an overview), the emphasis in this field is on learning complete behaviors, usually for achieving one specific goal but not many constantly changing goals. Either all agents are centrally controlled by the learner or each agent tries to figure out its role on its own.

In [23], three levels of behavior in animals and humans are described and then transferred to artificial systems: reaction, which deals with predefined, undeliberated responses to sensory input; routine, the level on which learned behavior is executed and the consequences of actions are assessed; and reflection, the level on which a system deliberates about itself, its past and future behavior. According to this categorization, the EIA operates on the third level (reflection), as it analyses the actions of the system and adapts its constituent parts to increase the system's efficiency. The original vision of Autonomic Computing [2] did not directly refer to these three levels but already mentioned the need for a system to learn and optimize – albeit referring to these two requirements as being distinct instead of complementary. Both activities, however, require self-reflection and a method to test the consequences of the learned behavior, usually achieved with simulations on a model of the system [24]. Following the function of dreams in humans, [25] defines a "dreaming" state for applications in which operations can be executed and evaluated on the system itself but data modified by the actions is not persisted. This resembles a simulation on the real system with current, real data.

In summary, most hybrid optimization and self-reflection approaches do not have the limited ability to "look into the future" based on the past situations the basic system had to cope with. Thus, the higher level agent has always to monitor and control the basic system centrally. By contrast, due to its limited ability to "look into the future", the EIA approach focuses on an adaptation of the basic system, so that it can cope with potentially inefficient future situations on its own. Moreover, because the EIA control loop is decoupled with regard to execution time, scalability is not a major issue.

VII. Conclusion and Future Work

In this paper, we presented the efficiency improvement advisor approach. The advisor is able to improve the efficiency of self-organizing emergent MAS solutions to dynamic optimization problems containing recurring tasks, by providing a limited capability to "look into the future", while preserving the basic beneficial properties of the underlying system. The experimental evaluation demonstrated its usefulness in scenarios that are known to be solved not very well by the underlying system, even if the recurring tasks were changing or random.

Future work includes the improvement of the optimizer allowing for more complex run instances, followed by applying the approach to other self-organizing MAS solutions also in other problem domains. Additionally, there are more possibilities for advice than just telling an agent to ignore a task, e. g. encouraging an agent to move to a certain area in anticipation of a task that is likely to appear in the vicinity without having actually accepted the task yet.

REFERENCES

[1] R. Sterritt, M. Parashar, H. Tianfield, and R. Unland, "A concise introduction to autonomic computing," *Advanced Engineering Informatics*, vol. 19, no. 3, pp. 181–187, 2005.

[2] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[3] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing—degrees, models, and applications," *ACM Computing Surveys*, vol. 40, no. 3, pp. 1–28, 2008.

[4] T. De Wolf and T. Holvoet, "A taxonomy for self-* properties in decentralised autonomic computing," in *Autonomic Computing: Concepts, Infrastructure, and Applications*. CRC Press, 2007, pp. 101–120.

[5] M. Jelasity, O. Babaoglu, and R. Laddaga, "Self-management through self-organization," *IEEE Intelligent Systems*, vol. 21, no. 2, pp. 8–9, 2006.

[6] D. Weyns, R. Haesevoets, B. V. Eylen, A. Helleboogh, T. Holvoet, and W. Joosen, "Endogenous versus exogenous self-management," in *Proceedings of SEAMS 2008*, 2008, pp. 41–48.

[7] S. Brückner, "Return from the ant - synthetic ecosystems for manufacturing control," PhD thesis, Humboldt-Universität, Berlin, 2000.

[8] M. Mamei and F. Zambonelli, "Co-fields: A physically inspired approach to motion coordination," *IEEE Pervasive Computing*, vol. 3, no. 2, pp. 52–61, 2004.

[9] H. Kasinger, B. Bauer, and J. Denzinger, "Design pattern for self-organizing emergent systems based on digital infochemicals," in *Proceedings of EASe 2009*, 2009, pp. 45–55.

[10] G. Berbeglia, J.-F. Cordeau, and G. Laporte, "Dynamic pickup and delivery problems," *European Journal of Operational Research*, vol. 202, no. 1, pp. 8–15, 2010.

[11] H. Kasinger, J. Denzinger, and B. Bauer, "Digital semiochemical coordination," *Communications of SIWN*, vol. 4, pp. 133–139, 2008.

[12] D. Weyns, N. Boucké, and T. Holvoet, "A field-based versus a protocol-based approach for adaptive task assignment," *Autonomous Agents and Multi-Agent Systems*, vol. 17, no. 2, pp. 288–319, 2008.

[13] K. Fischer, J. P. Müller, M. Pischel, and D. Schier, "A model for cooperative transportation scheduling," in *Proceedings of ICMAS 1995*. MIT Press, 1995, pp. 109–116.

[14] M. Mes, M. van der Heijden, and A. van Harten, "Comparison of agent-based scheduling to look-ahead heuristics for real-time transportation problems," *European Journal of Operational Research*, vol. 181, no. 1, pp. 59–75, 2007.

[15] G. D. M. Serugendo, M.-P. Gleizes, and A. Karageorgos, "Self-organisation and emergence in MAS: An overview," *Informatica*, vol. 30, no. 1, pp. 45–54, 2006.

[16] J. A. Hartigan, *Clustering Algorithms*. John Wiley & Sons, 1975.

[17] M. W. P. Savelsbergh and M. Sol, "The general pickup and delivery problem," *Transporatation Science*, vol. 29, pp. 17–29, 1995.

[18] J. Branke, M. Mnif, C. Mller-Schloer, H. Prothmann, U. Richter, F. Rochner, and H. Schmeck, "Organic computing - addressing complexity by controlled self-organization," in *Proceedings of ISoLA 2006*, 2006, pp. 200–206.

[19] R. Schumann, A. D. Lattner, and I. J. Timm, "Management-by-exception - a modern approach to managing self-organizing systems," *Communications of SIWN*, vol. 4, pp. 168–172, 2008.

[20] S. Abdelwahed and N. Kandasamy, "A control-based approach to autonomic performance management in computing systems," in *Autonomic Computing: Concepts, Infrastructure, and Applications*. CRC Press, 2007, pp. 149–167.

[21] J. Koźlak, J.-C. Créput, V. Hilaire, and A. Koukam, "Multi-agent approach to dynamic pick-up and delivery problem with uncertain knowledge about future transport demands," *Fundamenta Informaticae*, vol. 71, no. 1, pp. 27–36, 2006.

[22] L. Panait and S. Luke, "Cooperative multi-agent learning: The state of the art," *Autonomous Agents and Multi-Agent Systems*, vol. 11, no. 3, pp. 387–434, 2005.

[23] D. Norman, A. Ortony, and D. Russell, "Affect and machine design: Lessons for the development of autonomous machines," *IBM Systems Journal*, vol. 42, no. 1, pp. 38–44, 2003.

[24] R. Sterritt, "Autonomic computing," *Innovations in systems and software engineering*, vol. 1, no. 1, pp. 79–88, 2005.

[25] A. Butler, M. Ibrahim, K. Rennolls, and L. Bacon, "On the persistence of computer dreams - an application framework for robust adaptive deployment," in *Database and Expert Systems Applications, 2004. Proceedings. 15th International Workshop on*, Aug.-3 Sept. 2004, pp. 716–720.