2011 Eighth IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems

Pro-active Advice to Improve the Efficiency of Self-Organizing Emergent Systems

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

***Abstract*—An advisor autonomously improves the efficiency of self-organizing emergent multi-agent systems at runtime. It identifies tasks that the system has to perform recurringly. If the system performs these tasks inefficiently, the advisor will create so-called exception rules for the agents that enable them to perform these tasks more efficiently in the future. In contrast to the previously presented concept of ignore rules, a type of exception rule which only keeps the agents from doing certain tasks, in this paper we present the concept of pro-active rules. This type of exception rule allows the advised agents to already prepare for tasks before they are even announced to the system. Our experimental evaluation shows that a combination of these two rule types for the domain of dynamic pickup and delivery problems utilizes the advantages of both, improves previously badly handled problem instances, and additionally offers slight improvements for randomly created instances.**

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

I. Introduction

Self-organizing emergent systems [10], which often also are called Decentralized Autonomic Computing (DAC) systems [5], exhibit several desired properties, like scalability, robustness, flexibility and adaptability to the environment as well as to the given problem. However, the efficiency with which a self-organizing emergent system solves a problem respectively performs its tasks is not a property that has been focused on very much. This is not totally surprising, since many of the application problems that are of interest for the use of such systems are of a dynamic nature. Thus, for these problems seems to be no hope to produce optimal solutions, since this would require to be able to "look into the future" in some sense. Nevertheless, especially for applications that involve physically embodied agents as the components of the self-organizing emergent system, efficiency (e. g. in terms of required agents, solution costs, ...) is of high importance, since the cost of adding additional agents is definitely not neglectable.

Recently, in [16] we presented a concept to improve the efficiency of a self-organizing emergent system while maintaining the beneficial properties mentioned in the first paragraph. This concept is a so-called *efficiency improvement advisor* (EIA). The EIA uses learning techniques to identify recurring tasks the self-organizing emergent system has to perform, standard optimization techniques to create nearly optimal solutions for these recurring tasks, and the concept of exception rules that are given to the agents of the system as a form of advice. This advice helps the system in future situations to solve the recurring tasks more optimal. The advisor performs its work mostly off-line from the base system and only communicates with the other agents when they are ready and/or in range. This is why the whole system retains the basic beneficial properties, while having a substantially improved efficiency compared to the performance of the base system.

The exception rules (i. e. the type of advice) we have presented in [16] were of the type of so-called *ignore rules*, i.e. a rule suggests to an agent to ignore a particular task. Ignore rules represent a rather small influence on an agent and, as [8] has shown, have a rather low potential for being abused. But, as pointed out in [12], there are several inefficiencies in a self-organizing emergent system for which ignore rules are ineffective.

In this paper, we present the concept of so-called *pro-active exception rules* that allow for a stronger influence of the EIA on the other agents. As the name suggests, a pro-active rule tries to get an agent to start working on a task before the task is announced to the system. Obviously, this is only possible, if a task consists of several steps to be performed by an agent, while performing the first step does not commit the agent to having to perform the other steps. But many problems of interest for self-organizing emergent systems, especially involving physically embodied agents, do have some kind of preparation steps in their tasks, thus allowing for pro-active advice. Examples include set-up times for machines in manufacturing plants or having to drive to a location to pickup loads.

Similar as in [16] and [8], in this paper we use pickup and delivery problems (PDP) [14] as application domain for the experimental evaluation, which allows for a comparison of the results. These results show that pro-active exception rules indeed can deal with problem instances for which ignore rules are not capable to improve the behavior of the basic self-organizing emergent system. But in certain

cases they can be exploited to worsen the behavior of the base system. A combination of both ignore and pro-active rule types not only provides promising results for selected problem instances, for which only one of the individual types is very good, it also improves slightly the performance on randomly created scenarios. As such pro-active exception rules add to the tools available to the developers of self-organizing emergent systems that can be used to fulfill the specific requirements of an application domain and the task profiles particular users have.

The rest of this paper is organized as follows: Section II provides basic definitions that are used throughout this paper. This includes the definition for the kind of problems that an EIA can be used for, which are *dynamic task fulfillment problems* with recurring tasks. Section III briefly recalls the concept of an EIA from [16] as well as the assumptions that have to be fulfilled in order to make good use of an EIA. Section IV presents the concept of pro-active exception rules (also in combination with ignore rules) and the concept of revoking already published rules. Section V instantiates the concepts from Section III and Section IV for the application domain of PDP. Section VI presents our experimental comparison of pro-active rules with ignore rules and the combination of both. We conclude with a look at related work in Section VII and at possible future work in Section VIII.

II. Basic Definitions

A rather generic definition of an agent $\mathcal{A}g$ is a 4-tuple $\mathcal{A}g$ = $\{Sit,Act,Dat,f_{\mathcal{A}g}\}$, where Sit is the set of all situations that $\mathcal{A}g$ can find itself in, Act is the set of $\mathcal{A}g$'s actions, Dat is the set of all possible values of the agent's internal data areas (i.e. $\mathcal{A}g$'s knowledge and memory), and $f_{\mathcal{A}g}$: $Sit \times Act \rightarrow Act$ is $\mathcal{A}g$'s decision function. $f_{\mathcal{A}g}$ defines how $\mathcal{A}g$ decides on an action in its current situation and given its current knowledge from Dat. A multi-agent system (MAS) is a group of agents A = $\{\mathcal{A}g_1,...,\mathcal{A}g_n\}$ that are acting in a shared environment $\mathcal{E}nv$. If all agents in A have the same sets Sit, Dat, and Act and the same $f_{\mathcal{A}g}$, they are called homogeneous, else they are heterogeneous.

The general structure of the kind of problems we are interested in solving using a set of agents A consists of tasks out of a given set T that are announced at some time within a given time interval $Time$, usually not all at the same time, to some or all agents in A. We call a sequence of such tasks a *run instance*. There is usually a whole sequence of run instances that A has to solve allowing for tasks to reoccur in different run instances in this sequence. In this paper, we require a task to require at least two actions to be performed by an agent in order to fulfill the task.

More precisely, a run instance is a sequence of task-time pairs

$$((ta_1, t_1), (ta_2, t_2), ..., (ta_m, t_m))$$

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

$$(((ta_{11}, t_{11}), (ta_{21}, t_{21}), ..., (ta_{m_1}, t_{m_1})), ...,$$
$$((ta_{1k}, t_{1k}), (ta_{2k}, t_{2k}), ..., (ta_{m_k}, t_{m_k})))$$

A solution sol compiled from the emergent responses of the agents in A for a run instance is then

$$sol = ((ta'_1, \mathcal{A}g'_1, t'_1), (ta'_2, \mathcal{A}g'_2, t'_2), ..., (ta'_m, \mathcal{A}g'_m, t'_m))$$

where $ta'_m \in \{ta_1, ..., ta_m\}$, $ta'_i \neq ta'_j$ for all $i \neq j$, $\mathcal{A}g'_i \in A$, $t'_i \leq t'_{i+1}$, $t'_i \in Time$. A tuple $(ta'_1, \mathcal{A}g'_1, t'_1)$ means task ta'_i was started by $\mathcal{A}g'_i$ at time t'_i. Since we assume that a task requires at least two actions by an agent $\mathcal{A}g_i$, we identify the first (or preparation) action of a task ta by $prep(ta)$ and all other actions to perform ta by $rest(ta)$. Please note, if we have heterogeneous agents, it is possible that not every agent can perform every task.

In order to be able to talk about efficiency of A in solving a run instance, we need to be able to associate with each solution sol a quality measure $qual(sol)$. The nearer the solution produced by A comes to an optimal quality the more efficient A is. Quality measures are naturally dependent on the particular application problem A was created for, but even for a particular application there usually are still several possible measures so that in the end the users have to decide on how $qual$ is defined. For real dynamic problems (i.e. we have at least one i such that $t_i < t_{i+1}$ in a run instance) it is usually not possible for A to produce an optimal solution, except if solving the static variant of the problem is very easy.

III. The Efficiency Improvement Advisor

The EIA is an additional agent ($\mathcal{A}g_{EIA}$) that is added to A. By recalling our definition of an agent, the components of $\mathcal{A}g_{EIA}$ consist of a Sit_{EIA} that contains the information currently received by $\mathcal{A}g_{EIA}$ from any of the agents in A. Dat_{EIA} contains all the information previously sent by agents in A and all the intermediate results $\mathcal{A}g_{EIA}$ created while doing its work. $f_{\mathcal{A}g_{EIA}}$ goes through several actions (forming Act_{EIA}), which are depicted in the functional architecture of an EIA (see Figure 1) and described in more detail in the following:

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 $\mathcal{A}g_i$, when $\mathcal{A}g_i$ 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 from 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

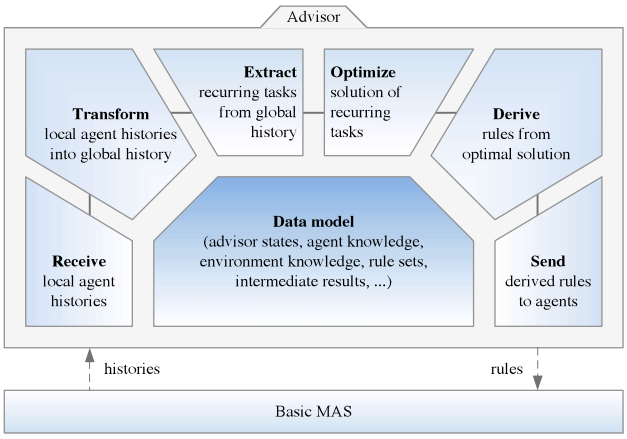

Figure 1. Functional architecture of EIA

so far and the solution sol_j for each run instance ri_j that A created for it (*emergent solution*).

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.

For many of these actions, there are several possibilities how they can be realized for a concrete application problem. Since the EIA has been discussed in detail in [16], we will focus our discussion on details for the different actions that are of importance for the exception rules.

The actions **receive** and **transform** are not of any relevance for how exception rules are created, except for the fact that a task that never has been observed by any of the agents obviously will not find its way into the sequence of run instances that is the result of **transform**. How to find the recurring tasks can be achieved in many different ways. One requirement that influences the exception rules is that in practice we should not require that a task is really occuring in exactly the same way in every run instance. Slight variations of the task itself should be allowed and also very few run instances in which the task (or variations) does not occur. This is achieved by defining an application dependent similarity measure $sim : T \times T \rightarrow \mathbb{R}^+$ and having a threshold value ($minocc$) for how often a task (or a sufficiently similar task) has to occur in the last k run instances to be considered as being recurring. Using only the last k run instances allows for the recurring tasks to change over time.

The method to realize **extract** in [16], which we also use in this paper, uses a clustering algorithm, more precisely Sequential Leader Clustering (see [7]). By using sim and another threshold factor, the tasks of all last k run instances are put into clusters (each cluster representing groups of tasks that are within this similarity threshold) and all clusters with more than $minocc \times k$ elements represent a recurring task (or several). The clustering algorithm also provides for each cluster a representant that can be used by extension rules to refer to this recurring task.

The action **optimize** normally should not have any influence on exception rules, but this is only true if **optimize** is really able to provide an optimal solution for the recurring tasks, as was the case in [16]. Unfortunately, for many interesting problems, finding the optimum for the static variant of the problem is infeasible, so that only very small instances can be solved in a timely manner. Therefore the advisor can also make use of approximation algorithms for the problem, such as for example evolutionary methods or simmulated annealing. Unfortunately, this may cause some problems when comparing experiments with different exception rules, since in different experiments the advisor will often use different solutions to adapt the self-organizing emergent system to.

Since this paper is about different types of exception rules we will discuss the action **derive** in detail in the next section. Moreover, we will also discuss the need for eliminating exception rules that were communicated to the agent before, which is included into the action **send**. This means that R_i now is the complete set of rules that $\mathcal{A}g_i$ should use.

For using an EIA as described above, several requirements have to be met. First, in order to be able to receive the necessary data from the agents in A, the agents need to be able to transmit their local histories to the advisor, preferably at least once during or after a run instance. In theory, an EIA also works if such a transmission is only done after several run instances. However, this means as a consequence that any advice will reach the agents only after several run instances as well (after the advice has been determined), which will endanger gains, if the recurring tasks change often. The second requirement is that the agents in A, respectively their decision functions, can be modified to accept and work with exception rules. Usually, these two requirements are not difficult to fulfill.

A third requirement, at first glance, seems to be harder to fulfill: a sequence of run instances must have a large

enough set of recurring tasks, as defined previously in this section. Fortunately, this requirement is fulfilled by a lot of applications for a lot of users. For example, many transport companies have a rather large number of daily recurring tasks that provide them with a good bottom line, which is enriched by additional tasks that have to be fulfilled only once or very few times.

IV. Pro-Active Rules for the Advisor

In this section, we first use the ignore rules from [16] to present an example for how exceptions rules can be created and then used by the agents in A. Then we present the necessary modifications to allow for the use of pro-active exception rules. Finally, we address the issue of having the EIA try out rules and delete them if they are not successful. This is an issue with pro-active rules as well as with the combination of ignore and pro-active rules.

A. Ignore Exception Rules

As stated in the last section, in order for the advisor to work, the agents in A have to be able to work with exception rules. The idea behind an ignore rule is that it should detract the agent from performing a particular task (and its variations defined using sim) at least for some time. The hope is that due to such a detraction another agent will perform this particular task and that this results in a better overall emergent solution (at least for the set of recurring tasks).

In general, an ignore rule for an agent $\mathcal{A}g_i \in A$ has the form $cond_{ig}(s,d) \rightarrow \neg a_{ta}$, 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). Essentially, every agent architecture can be modified to use such an exception rule by simply creating a variant $f^{ig}_{\mathcal{A}g_i}$ of $\mathcal{A}g_i$'s decision function $f_{\mathcal{A}g_i}$ in the following manner:

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

This can be extended to a set of ignore rules by simply having the action a' not being equal to any of the actions' starting tasks for which the condition of an ignore rule is true in the current situation, given the current values of the internal data areas of $\mathcal{A}g_i$. a' should be the action that $\mathcal{A}g_i$ would select if its original choice (in $f_{\mathcal{A}g_i}$) is not available, to keep as much of the basic decision making of the agent preserved as possible. This also means that $f_{\mathcal{A}g_i}$ is naturally able to do any non-recurring task that comes up while ignoring a task ta.

An obvious way to realize $cond_{ig}$ is to use the task ta that is supposed to be ignored. Usually, only some parts of the task description are used and $cond_{ig}$ will already be true, if a currently active task is similar enough to the task for which the condition was created (at least with regard to the parts of the description that are used in the condition). $cond_{ig}$ will also contain any conditions on how long an agent should ignore the task.

Exception rules in general are created by $\mathcal{A}g_{EIA}$ by comparing the optimal solution opt^{rec} = ((ta^1_1, $\mathcal{A}g^1_1$, t^1_1),...,(ta^1_p, $\mathcal{A}g^1_p$, t^1_p)) for the recurring tasks with the computed emergent solution $last$ = ((ta^2_1, $\mathcal{A}g^2_1$, t^2_1),...,(ta^2_p, $\mathcal{A}g^2_p$,t^2_p)) for these recurring tasks. $\mathcal{A}g_{EIA}$ always targets the first j with either $ta^1_j \neq ta^2_j$ or $\mathcal{A}g^1_j \neq \mathcal{A}g^2_j$ which represents the first assignment of a task to an agent that deviates from the optimal solution. The ignore rule approach creates an ignore rule for $\mathcal{A}g^2_j$ of the form $cond_{ig}(s',d') \rightarrow \neg a_{ta^2_j}$. $\mathcal{A}g_{EIA}$ looks up in $GHist$ the triple $(s,d,a_{ta^2_j})$ and, as already stated, abstracts s and d such that tasks in the whole cluster of ta^2_k will activate the rule.

B. Pro-Active Rules

Pro-active rules advice agents to start tasks before any agent in A even knows (for sure) that a task will have to be performed. If the rules are adequate, there will be obvious efficiency gains: for instance, if efficiency is measured in terms of the distance traveled in a transportation scenario, the agents will travel less far, since they will know where to go next and will not be detracted by other tasks or go back to a depot. In contrast, if efficiency is measured in terms of needed time in a dynamic job-shop scheduling scenario, the agents will use less time, since they can already prepare for the next upcoming job. Naturally, if the rules are inadequate, the effort put into performing the *prep*-part of the task is wasted and preparing for another task might even become more expensive, which reduces the efficiency. Also, non-recurring tasks represent a bigger danger for reducing the efficiency, since using pro-active rules can result in the agents ignoring these tasks for some time, which was not the case for a system using ignore rules. As a consequence, the particular application and the profile of the tasks have much influence, whether pro-active rules should be applied or not.

A pro-active rule for an agent $\mathcal{A}g_i \in A$ has the form $cond_{proa}(s,d) \rightarrow prep(ta)$, with $s \in Sit_i$, $d \in Dat_i$ and $ta \in T$. Having only one such rule at a time, the modification of $f_{\mathcal{A}g_i}$ would be analogous to the case of an ignore rule. But since pro-active rules require an agent to perform a particular action, we need to deal with the cases where we have several rules with their conditions being fulfilled at a time (and suggesting different actions). This requires the use of a conflict resolution function $cr_i : 2^{R_i} \rightarrow Act_i$, where R_i is the (current) set of exception rules of $\mathcal{A}g_i$. If in a situation s an agent has as current value of its Dat_i d, then $R^{s,d}_i$ is the set of all exception rules in R_i with fulfilled conditions. Thus, the modified decision function $f^{exc}_{\mathcal{A}g_i}$ is defined as

$$f^{exc}_{\mathcal{A}g_i}(s,d) = \begin{cases} f_{\mathcal{A}g_i}(s,d), & \text{if } R^{s,d}_i = \emptyset \\ cr_i(R^{s,d}_i), & \text{else} \end{cases}$$

Note that this general function allows the use of both pro-active and ignore rules (hence the superscript "exc"). In general, it should be the goal of the EIA to create exception rules so that a conflict resolution function is not necessary, that is having $|R_i^{s,d}| \leq 1$. But due to solving dynamic problems including non-recurring tasks we can never rule out that the conditions of more than one rule are fulfilled. An obvious choice for cr_i is using the FIFO principle, i.e. the agent continues "working" on the task suggested by a rule until the task is either fulfilled or an explicit time span associated with a rule has gone by. Only then the next triggered exception rule (timewise) will be worked on.

To create a pro-active rule, $\mathcal{A}g_{EIA}$ again looks for the first index j in which the optimal solution opt^{rec} is different from the emergent one $last$. But this time a rule for $\mathcal{A}g_j^1$ is created, which has as action $prep(ta_j^1)$. With regard to the issue how to trigger a pro-active rule there are two general categories: task-triggered and time-triggered. A task triggered pro-active rule uses a task as the trigger, which is usually the recurring task an agent performed before the task the rule tries to prepare the agent for. As in the case of ignore rules, the condition will use only parts of the task description (depending on the application) and will also use sim and a threshold to allow for sufficiently similar tasks (or an approximation on the used parts) to be the trigger. A time triggered pro-active rule, as the name suggests, uses a set time to trigger the rule. This time reflects the time necessary to perform the preparation of the task and the variation of the occurence of the task from $GHist$[1]. For both kinds of triggers, the condition includes a maximum time limit for the rule being active, to avoid having the agent essentially taken out of the run instance in case that the particular recurring task is not appearing in the particular run instance.

Naturally, $\mathcal{A}g_{EIA}$ can be allowed to use both ignore and pro-active rules. Given the greater potential dangers in using pro-active rules, we suggest to prefer ignore rules whenever possible and only to use pro-active rules if the ignore rules are not able to achieve the necessary effects in the agents in A. As our experiments in Section VI show, this way of combining the two types of exception rules is very successful.

C. Deleting Rules

With the possibility to use pro-active rules comes the need for the $\mathcal{A}g_{EIA}$ to instruct agents to delete rules that it communicated to them before. The main reason is that the set of recurring tasks can naturally change and such a change might mean that the task that a pro-active rule is supposed to prepare for is not a recurring task anymore, while the trigger task is still there, respectively a time trigger is used. Without deleting this rule, the rule would still be triggered frequently, resulting in quite some inefficiency. Also, a rule created by the EIA might not have the intended effect, which often can only be observed after the rule has been given to an agent and $\mathcal{A}g_{EIA}$ needs then to be able to delete the rule. Deleting rules becomes also very important, if $\mathcal{A}g_{EIA}$ is allowed to use both types of rules, because trying out one type and not deleting it when trying out the other type creates additional problems for the conflict resolution function (FIFO would obviously not work anymore).

Fortunately, it is not too difficult for $\mathcal{A}g_{EIA}$ to deal with the problems above. For each rule that is active in an agent in A, $\mathcal{A}g_{EIA}$ needs to remember the recurring task the rule was created for. For each task, there should be at most one pro-active rule. If $\mathcal{A}g_{EIA}$ creates a pro-active rule for a task for which already another pro-active rule exists, this other rule gets deleted. Also, if a task is not recurring anymore, any exception rule for it gets deleted.

V. Instantiation for PDP

In this section, we instantiate the concepts from the last two sections for an application domain, namely dynamic pickup and delivery problems (PDP), similar as in [16]. We will first describe PDP as instance of dynamic task fulfillment, then present the instantiation of the advisor with ignore rules from [16] using digital infochemical coordination (DIC, see [11]) as basic decentralized coordination model We will focus, again, mainly on what is important for creating and using exception rules. Then we instantiate pro-active exception rules to this application domain.

A. Pickup and Delivery Problems

The general pickup and delivery problem (see [14]) is a well-known problem class with many instantiations (see e. g. [1], [2]). Formally, a task ta of a dynamic PDP consists of a pickup location l_{pickup}, a delivery location $l_{delivery}$ and the needed transportation capacity $ncap$. The capacity is mirrored on the agent side by a transportation capacity $cap_{\mathcal{A}g}$. Most real-world problems require solving dynamic instances of this basic problem. Many of these instantiations fulfill the conditions necessary for using an advisor. Since the basic problem requires agents to move to a location, pickup goods that have to be transported to another location, move to this other location, and drop of the goods, PDP also fulfills the necessary requirement for pro-active exception rules. This is having the action of moving to the pickup location as the $prep$-action for such a task.

With regard to efficiency, i.e. $qual$, there are several possibilities. In this paper, we define as $qual$ the sum of the traveled distances by all agents (for performing all tasks in the run instance), including moving out of and getting back to the depot. This definition of $qual$ is the same as in [8], which allows us to use the testing approach presented there for the evaluations.

[1]This can be rather difficult if the announcement times for tasks vary a lot.

B. Using EIA for DIC for PDP

The self-organizing emergent system that is the basis for our evaluations uses digital infochemical coordination (see [11]), a generalization of pheromone-based coordination (see [3]), as underlying decentralized coordination approach. The system achieves coordination between the transportation agents in A using digital infochemicals that are propagated through $\mathcal{E}nv$. An $\mathcal{A}g_i$ accesses all the infochemicals at its current location and bases its decisions purely on this local view of the environment.

Tasks are introduced into the environment by creating two so-called emitter agents, a pickup agent at l_{pickup} and a delivery agent at $l_{delivery}$. Both agents emit so-called synomones, a specific type of infochemical. These synomones are propagated through the environment and a location receiving them stores their existence and intensity. A synomone on a location evaporates after a certain time, so that an emitter agent repeats the synomone emission from time to time until it has been served by a transportation agent. The intensity and direction of a synomone emitted from l_{pickup} gives a transportation agent hints on the emitters location as well as $ncap$, while the synomone emitted from $l_{delivery}$ gives hints on the delivery location.

For an $\mathcal{A}g_i$ to decide what to do, it computes a utility for each task it "smells" and then chooses the task with the highest utility. The utility is influenced by the intensity of an infochemical as well as by the agent's current status. For example, if it has already picked up the goods for a certain task, it gives priority to delivering it. Additionally, other infochemical types influence the utility computation: a pickup agent emits so-called allelochemicals, another type of infochemical, as soon as an $\mathcal{A}g_i$ served it. This indicates to an $\mathcal{A}g_j$ that the task execution has already started, which prevents it from being unnecessarily attracted to this task by unevaporated synomones. In this case, 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 selected a task it moves directly towards the synomone emitter representing it.

Adding an advisor to this base system is not very difficult. Since the synomones from the emitter agents contain all the necessary information about a task, the histories of the transportation agents are sufficient for creating the actions **receive** and **transform** that identify the run instances. The similarity measure used for the clustering in **extract** is defined by

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

where Ed is the Euclidean distance of the locations and α and β are weighting parameters.

The static variant of our dynamic PDP is unfortunately already a hard problem to solve. Since our experiments required the optimization of many instances of this static variant, we decided not to use the immature optimizer used in [16], but an approximation method using a genetic algorithm, which uses or-tree-based search to make sure that the genetic operators only create solutions that fulfill the hard constraints of the PDP (this is the same approximation method as used in [8]). While this allows us to limit the time needed for the optimization, it unfortunately introduces an additional random element into our experimental evaluation (see Section VI). The quality of the emergent solution is computed by using the direct distances between the emitter agents for the recurring tasks.

The instantiation of **derive** for PDP for ignore rules makes use of the fact that the utility evaluation performed by an agent is very similar to a rule-based system. each infochemical comes with a "rule" describing its contribution to the utility of the task it was created for and the conflict resolution adds up the contributions of all triggered rules. Therefore, ignoring a particular task can be achieved by setting its utility to 0, overriding all other infochemical-based rules for the task. This also automatically achieves that the agent will go after the next-best task. In [16], **derive** used for $cond_{ig}$ an abstracted synomone ab containing the pickup location and the required capacity of the task representing the cluster associated with a recurring task (by **extract**). Consequently, $cond_{ig}$'s task part was fulfilled, if a task ta was within a given threshold $synthresh$ for the function $dist_{syn}$ with

$$\begin{aligned} dist_{syn}(ab, ta) \quad = \quad & \alpha \cdot Ed(l_{pickup,ab}, l_{pickup,ta}) + \\ & \beta \cdot |ncap_{ab} - ncap_{ta}| + \end{aligned}$$

The second part of $cond_{ig}$ requires that the first activation of the rule (in the current run instance) was less than a given number of time units ago.

C. Pro-Active Rules for PDP

As already stated in Section IV-B, there are two different general types for how to trigger pro-active exception rules, time-triggered and task-triggered. In our preliminary experiments, task-triggers clearly outperformed time-triggers by being much more flexible (respectively time-triggers essentially destroyed the flexibility of the basic system), so that here we will concentrate on task-triggered pro-active rules. The obvious problem of how to create a trigger for the first recurring task of a run instance was solved by having an artificial begin task synomone emitted by the depot at the start of each run instance.

The general idea of a task-triggered pro-active rule for our system solving PDP is to create a rule that sends an agent to the pickup location for a task ta_{targ} that the EIA wants it to

perform (instead of what its decision function is suggesting). And this rule will be triggered by the recurring task ta_{trig} this agent performed in the optimal solution before the task the agent should now perform (or by the begin task), respectively by having fulfilled this trigger task. While this seems to be rather straightforward, there are a few problems in the details. For example, the time period between a trigger task and the targeted task can vary a lot for the different recurring tasks the EIA identifies. They can even vary for the same task pair between run instances. So, just sending the agent to the pickup location and then having it follow its "standard" decision function can result in the agent going away again, if the task ta_{targ} is not announced within a short time after arriving there. Providing an additional time span to stay at that location runs the opposite risk: the agent might not be "freed" soon enough to prevent other agents from fulfilling ta_{targ}. Our solution is to provide the agent with an additional time limit and a target synomone ab_{targ} describing the task ta_{targ} it is supposed to do.

So, formally, for PDP a pro-active rule for ta_{targ} consists of two subrules: $cond_{proa1}(s,d) \rightarrow prep(ta_{targ})$ and $cond_{proa2}(s,d) \rightarrow a_{rest(ta_{targ})}$, where $a_{rest(ta_{targ})}$ is the first action to take to fulfill $rest(ta_{targ})$, i.e. picking up the goods. As described above, $cond_{proa1}(s,d)$ is fulfilled, if the abstracted synomone ag_{trig} to ta_{trig} or a synomone sufficiently similar to it has been perceived by the agent in this run instance (which can be determined looking at s and d) and this has happened within a given time limit (the so-called timeout). The condition $cond_{proa2}(s,d)$ consists of $cond_{proa1}(s,d)$ being fulfilled and the abstracted synomone ab_{targ} to the target task ta_{targ} being perceived (or a sufficiently similar synomone). These rules are integrated into the utility computation of an agent by boosting the infochemicals associated with them above the maximum utility a task can achieve otherwise. The EIA creates the abstracted synomones for these rules the same way it creates the abstracted synomone for an ignore rule and then uses $dist_{syn}$ for similarity (also as described for ignore rules).

As already suggested in the last section, the EIA tries first to improve the emergent solution by creating an ignore rule and only if there is no change in the emergent solution after adding this ignore rule it deletes the ignore rule and tries a pro-active rule. On the agent side, if there are two pro-active rules with their conditions fulfilled (which can happen, if tasks are announced in very short intervals), the conflict resolution function chooses the one that was triggered first (which is also most probably the one that the agent is already working on).

VI. Experimental Evaluation

In order to evaluate the usefulness as well as the risks of pro-active exception rules, in this section we report on several experiments, in which we compare variants of the system described in the last section by using only ignore rules, only pro-active rules, and using both. The first experimental series is aimed at showing the potential of the three variants. It reports on problem instances for which the variants achieve very good improvements. For finding such instances, we have used the learning approach of [8], which we will briefly explain in the first subsection. Since this approach was available, we also used it to find problem instances for which the different variants have certain problems with, which is our second experiment series. Finally, in the third experiment series we report on various problem instances that were created randomly to also provide a picture of the average performance of the three exception rule variants.

A. Automatic Testing of Self-Adaptive Systems

Getting an idea of the potential, but also dangers, of a self-organizing emergent system for a particular application is anything but trivial: while it is naturally easy to evaluate such systems on given test instances, one of the reasons we are interested in this type of system is that they offer the ability to solve problem instances the developers and potential users have not thought about (hence the name self-organizing). It is even more interesting to get an idea what are instances for which the system is not working well. [8] presented a way to find such instances using Machine Learning.

The general idea is to use an evolutionary learning approach in order to create PDP run instances that fulfill some intended conditions for the behavior of the self-organizing emergent system. To do this, we start with a set of randomly created run instances (25 in our experiments), called individuals. Each individual is evaluated by having the self-organizing emergent system with the advisor perform the run instance repeatedly. This way, the advisor can adapt the whole system to this run instance. In the case of our system (see Section V), we use the qualities of the emergent solution to the run instance rs before the adaptation by the advisor $qual(sol_{before}(rs))$, after the complete adaptation $qual(sol_{after}(rs))$ and the (nearly) optimal solution for the run instance $qual(opt(rs))$ to create a fitness value for the individual. The way how the three values are combined depends on what the intended condition for this testing is (see later).

Then evolutionary operators are applied to selected individuals from the current set of individuals creating new individuals. Finally, the worst individuals of the current set are replaced by the new individuals. This is repeated for a given number of iterations (100 in our experiments). The used operators are rather standard: on the one hand side changing a task or an announcement time in an individual as a single-point mutation and on the other side choosing the tasks randomly from the tasks of two "parent" individuals

as a crossover[2]. The parents for these operators are selected using tournament selection. This means that randomly a certain number of individuals are selected that then are pitted against each other in a tournament, where the individual fitness determines the winners.

In the next subsection, we are interested in two types of run instances: first, run instances, for which the advised system with the particular variant of exception rules creates large improvements (compared to the base system), which provides an idea of the potential the variant has. Second, run instances, for which the advised system creates bad results (again, compared to the base system), which provides an idea of the potential risks the variant can cause. This requires two different fitness measures. The fitness measure fit_{pot} for the potential of an individual rs is defined by

$$fit_{pot}(rs) = \frac{5 \cdot theo(rs) + 500 \cdot adapt^{+}(rs)}{qual(opt(rs))}$$

where $theo(rs) = qual(sol_{before}(rs)) - qual(opt(rs))$ indicates how much potential for improvement the individual has and $adapt^{+}(rs) = max[0, qual(sol_{before}(rs)) - qual(sol_{after}(rs))]$ indicates how much improvement the advisor achieved. By maximizing this fitness measure we get individuals that are very good for the particular variant of exception rules. Note that having $theo(rs)$, even with such a small weighting factor, helps in the beginning of the search to guide the learner towards individuals that allow for improvement.

The fitness measure fit_{risk} for the danger potential of an individual is defined by

$$fit_{dang}(rs) = \frac{5 \cdot theo(rs) + 10 \cdot pract(rs) + 500 \cdot adapt^{-}(rs)}{qual(opt(rs))}$$

where $theo(rs)$ is as before, $pract(rs) = max[0, qual(sol_{after}(rs)) - qual(opt(rs))]$ indicates how far away the adapted emergent solution is from the optimal one and $adapt + -(rs) = max[0, qual(sol_{after}(rs)) - qual(sol_{before}(rs))]$ indicates how much worse the emergent solution got after adaptation. Again, the major factor is $adapt^{-}$, with the other factors helping to steer the evolutionary algorithm (EA) initially in the right direction. The EA tries to maximize this fitness measure.

B. Evaluating Potential and Risks

The experiments in this subsection aim at evaluating both the potential and the dangers that the different exception rules for the advisor offer, using the testing approach from the previous subsection. The environment for the transportation agents is a 10×10 grid with the depot in the middle. Initial experiments showed that already with 2 agents and 4 recurring tasks we are able to find run instances that allow us to evaluate the differences between the different variants of the system. Due to the use of the GA as optimizer, we performed five simulation runs for each of the run instances and report in Tables I and II on the average distance (in grid fields) travelled by the agents. Since our goal with these experiments is to find the best and the worst behavior, we did not use any non-recurring tasks and also did not make use of any similarity. The timeout for the pro-active rules was 5 time units.

[2][8] uses additional operators that are targeted at sequences of run instances. But since we are only interested in a single run instance, they are not necessary in our experiments.

Exp.	base	ig	proa	both
$gd_{ig,1}$	53.84	25.80	31.54	26.38
$gd_{ig,2}$	69.74	31.46	45.60	24.10
$gd_{ig,3}$	62.87	34.38	58.21	34.38
$gd_{ig,4}$	81.36	35.56	55.36	28.73
$gd_{ig,5}$	61.21	29.56	57.45	31.96
$gd_{proa,1}$	42.38	28.73	17.66	14.24
$gd_{proa,2}$	72.08	57.07	34.73	26.47
$gd_{proa,3}$	67.84	64.38	24.14	22.45
$gd_{proa,4}$	64.91	64.91	33.46	26.63
$gd_{proa,5}$	55.11	35.80	23.31	21.31
$gd_{both,1}$	67.01	45.70	30.53	27.80
$gd_{both,2}$	80.08	51.11	45.60	34.63
$gd_{both,3}$	57.36	45.70	58.21	28.38
$gd_{both,4}$	71.50	52.24	55.36	30.97
$gd_{both,5}$	71.25	71.25	57.45	29.31

Table I
RESULTS FOR THE EVALUATION OF THE POTENTIAL

Table I shows that for each of the 3 variants (only **ig**nore rules, only **proa**ctive rules and **both**) we have run instances, where the variant substantially improves the basic self-organizing emergent system (indicated by base). But the variant using both rule types is better than the ignore variant in two of the examples created for the ignore variant (gd_{ig}) and is as good for one of those examples. Even more, the both-variant is better than the pro-active variant in all of the examples created for the pro-active variant (gd_{proa}). Finally, the both-variant produces the best results when used for the examples created for it (gd_{both}). The gd_{proa} and gd_{both} examples also show that just using pro-active rules can produce better results than ignore rules.

Looking more closely at the examples, the strength of pro-active rules is in changing the sequence in which an agent performs its tasks. While ignore rules achieve that the "correct" agent performs a task (i.e. the agent that performs it in the optimal solution), these rules cannot influence the sequence that the agent chooses to do "its" tasks in. But by sending an agent to the right task before it tackles a wrong one, pro-active rules can deal with this problem. If we only use pro-active rules, there is one additional potential problem, namely that another agent picks up the task that an agent should do before the agent gets there. But by giving this other agent an ignore rule for the task, this problem can be avoided. This is the reason for the very good performance of the combined variant.

Exp.	base	ig	proa	both
$bd_{ig,1}$	48.63	67.60	48.63	29.31
$bd_{ig,2}$	44.87	65.84	45.17	26.42
$bd_{ig,3}$	29.31	43.80	53.98	23.31
$bd_{ig,4}$	46.87	68.67	49.94	37.50
$bd_{ig,5}$	45.70	60.18	57.60	27.56
$bd_{proa,1}$	38.87	47.92	77.88	29.73
$bd_{proa,2}$	44.73	46.33	204.15	167.28
$bd_{proa,3}$	29.90	29.90	175.56	114.76
$bd_{proa,4}$	34.38	32.38	240.97	181.58
$bd_{proa,5}$	29.31	23.31	245.56	133.12
$bd_{both,1}$	42.04	42.04	222.04	238.04
$bd_{both,2}$	38.87	38.87	222.38	238.28
$bd_{both,3}$	38.04	38.04	241.70	241.70
$bd_{both,4}$	45.11	45.11	80.39	175.46
$bd_{both,5}$	30.97	28.87	109.51	236.97

Table II
RESULTS FOR THE EVALUATION OF THE RISKS

Table II presents the results of the three variants for instances specifically created to be solved badly by one of the variants. The bad examples for ignore rules (bd_{ig}) show that it is possible to produce worse results than the basic system. However, compared to the bad instances for pro-active rules (bd_{proa}) and for the combination of both (bd_{both}) the risk is much more limited (definitely smaller than a factor of 2.0). Using both rules together produces better results on the bad instances for the single rule types they were created for. But using pro-active rules, either alone or in the combination, clearly can be a risk.

Looking more closely at the examples they reveal a weakness of the pro-active rules that is due to the instantiation of the DIC concept used in the system: the pickup emitter synomone does not contain any information about where to deliver the goods. As a result, if two tasks have the same pickup location, whichever is announced first to the agents will be performed by an agent send there by a pro-active rule, even if this rule was created for the other task. The learning tester from the last subsection was able to exploit this weakness quite a bit. Note that adding an appropriate ignore rule is not possible, because also ignore rules cannot distinguish between the two tasks.

C. Evaluating Random Instances

The experiments in this subsection aim at evaluating the three variants under "average" conditions. As in [16], this means that we are looking at sequences of run instances with recurring tasks together with non-recurring tasks. However, the recurring tasks have not been selected to be performed badly by the basic system. Instead, they are created randomly and are modified between run instances within the similarity threshold. This means that for a particular sequence of run instances, there might be no need for the advisor to try to change the behavior of the transportation agents. Therefore Table III reports the average of 20 such randomly created sequences of run instances in the rows labeled "Av.". Additionally, we provide data on the best sequence and the worst one of these 20, each in the form of the improvement over the basic system. Each sequence of run instances consists of 30 run instances with $k = 5$. The different columns represent different numbers of tasks per run instance. A column title xRyN means that we have x recurring tasks and y non-recurring tasks in each of the 20 sequences of run instances that were used to create the column entries. Again, the timeout for the pro-active rules was 5.

As Table III shows, using pro-active rules alone results in improvements, but those improvements are not comparable to the other two variants. As the experiments in the last subsection have shown, pro-active rules come with dangers and this is also shown in Table III by the worst results. They all are negative, meaning that the advised system was worse than the basic system. If we compare ignore rules with the variant using both rule types, we see a slight improvement of the average by the both-variant in six of the columns, with one column having an identical value. Of those columns where the ignore variant is better on average, we have three columns where the best runs of both variants have the same improvement. It also has to be pointed out that the combination variant usually will take longer to adapt to a set of recurring tasks, given that in every run instance only one new exception rule is created for the whole system. So, overall the variant using both exception rules is a slight improvement.

VII. RELATED WORK

As pointed out in [16], the main difference between the advisor approach and other related approaches to control the cooperation of agents in a self-organizing emergent system (see e. g. [9] for Autonomic Computing, [13] for Organic Computing, [6] for Autonomous Communication, or [15] for multi-agent systems) is that the advisor approach does not require the ability to observe and directly control the agents at every point in time. In fact, this is why exception rules are an adequate concept to improve the performance of a self-organizing emergent system while keeping its beneficial properties. Thus, all problem solving decisions are still made by the agents themselves. As a consequence, there exist no other approaches similar to pro-active exception rules.

Admittedly, there are quite a few other approaches derived from control theory that try to use learning from the past behavior to predict future states of a system. For example, the Model Predictive Control (MPC) approach (see [4]) tries to use a stochastic, linear model (generated out of the history of the system) to derive a control action by minimizing a quality function over a set of possible control action sequences. Unfortunately, the approach does not include updating the stochastic model, and always takes over total control of the basic system, which takes away essentially all flexibility. Moreover, linear models for self-organizing emergent systems are generally not available. The same

Rul.	Eval	4R0N	4R1N	4R2N	6R0N	6R1N	6R2N	6R3N	8R0N	8R1N	8R2N	8R3N	8R4N
ig	Av.	14.15%	12.03%	8.91%	17.26%	10.04%	7.97%	6.68%	12.48%	8.61%	6.74%	5.48%	4.84%
	Best	29.33%	18.93%	15.93%	28.06%	19.28%	13.78%	13.30%	19.97%	19.03%	12.82%	12.77%	8.53%
	Worst	0%	-1.18%	1.72%	9.49%	-0.38%	2.40%	0.44%	0.12%	-1.69%	-2.25%	0.77%	0.86%
proa	Av.	2.05%	2.26%	2.76%	4.81%	3.39%	3.72%	1.45%	1.97%	3.29%	2.37%	1.73%	1.81%
	Best	11.61%	13.65%	8.83%	21.98%	13.50%	11.42%	7.06%	18.46%	11.83%	8.42%	8.26%	6.58%
	Worst	-9.09%	-20.31%	-2.72%	-10.07%	-6.69%	-6.84%	-704%	-32.02%	-8.37%	-3.01%	-3.39%	-2.58%
both	Av.	14.17%	11.29%	8.96%	17.26%	9.72%	8.05%	6.64%	11.65%	9.79%	6.43%	5.65%	5.03%
	Best	29.33%	18.43%	16.62%	28.06%	19.28%	13.38%	13.30%	19.50%	19.03%	12.82%	12.36%	8.64%
	Worst	0%	-1.18%	1.72%	9.29%	-0.35%	2.25%	-1.45%	0.12%	-1.69%	-2.19%	1.95%	1.62%

Table III
RESULTS FOR RANDOMLY CREATED SEQUENCES OF RUN INSTANCES; IMPROVEMENTS IN PERCENT OVER BASE SYSTEM

holds for non-linear models, which are used by Nonlinear MPC approaches.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we presented the concept of pro-active exception rules for advised self-organizing emergent systems. By allowing agents to already prepare for anticipated tasks, pro-active rules offer the potential for substantial improvements, but also run quite the risk of worsening the performance of the system. Our experimental evaluation showed that pro-active rules can deal with efficiency problems of the basic system due to sequencing issues of agents, which is beyond the ability of the previously proposed ignore rules. The combination of ignore and pro-active rules also has this ability and was slightly better than just ignore rules even for randomly created problem instances.

The experiments revealed that pro-active rules are an useful tool for an advisor for certain run instance profiles, but that more work needs to be done in order to have the advisor only using them for the right sets of recurring tasks. Among the possibilities we want to look into in the future is enabling the advisor to simulate the behavior of the agents of the base system and run instance scenarios. This allows for a much faster adaptation and identifying inappropriate exception rules without consequences in the real world. Another direction of future work will be applying the different exception rule types to other environment-mediated coordination concepts (as suggested in [12]).

REFERENCES

[1] G. Berbeglia, J.-F. Cordeau and G. Laporte: Dynamic Pickup and Delivery Problems, *European Journal of Operational Research* 202, 2010, pp. 8–15.

[2] G. Berbeglia, J.-F. Cordeau, I. Gribkovskaia and G. Laporte: Static Pickup and Delivery Problems: A Classification Scheme and Survey, TOP 15, 2007, pp. 1–31.

[3] S. Brückner: Return from the Ant - Synthetic Ecosystems for Manufacturing Control, PhD thesis, Humboldt-Universität Berlin, 2000.

[4] E. Camacho and C. Bordons: Model Predictive Control, Springer, 2004.

[5] 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.

[6] S. Dobson, S. Denazis, A. Fernandez, D. Gaiti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt and F. Zambonelli: A Survey of Autonomic Communications, *ACM Transactions on Autonomous and Adaptive Systems* 1(2), 2006, pp. 223–259.

[7] J.A. Hartigan: Clustering Algorithms, John Wiley and Sons, 1975.

[8] J. Hudson, J. Denzinger, H. Kasinger, and B. Bauer: Efficiency Testing of Self-adapting Systems by Learning of Event Sequences, Proc. ADAPTIVE 2010, Lisbon, 2010, to appear.

[9] IBM: Autonomic Computing Whitepaper: An Architectural Blueprint for Autonomic Computing, June 2006.

[10] M. Jelasity, O. Babaoglu, and R. Laddaga: Self-Management through Self-Organization, *IEEE Intelligent Systems* 21(2), 2006, pp. 8–9.

[11] H. Kasinger, B. Bauer, and J. Denzinger: Design Pattern for Self-Organizing Emergent Systems Based on Digital Infochemicals, Proc. EASe 2009, San Francisco, 2009, pp. 45–55.

[12] H. Kasinger, B. Bauer, J. Denzinger, and T. Holvoet: Adapting Environment-Mediated Self-Organizing Emergent Systems by Exception Rules, Proc. SOAR 2010, Washington, 2010, pp. 35–42.

[13] U. Richter, M. Mnif, J. Branke, C. Müller-Schloer and H. Schmeck: Towards a generic observer/controller architecture for Organic Computing, In C. Hochberger and R. Liskowsky (eds.): Informatik 2006 - Informatik für Menschen, Springer, 2006, pp. 112–119.

[14] M.W.P. Savelsbergh and M. Sol: The General Pickup and Delivery Problem, Transp. Science 30, 1995, pp. 17-29.

[15] R. Schumann, A.D. Lattner and I.J. Timm: Management-by-Exception – A Modern Approach to Managing Self-Organizing Systems, In Communications of SIWN(4), 2008, pp. 168-172.

[16] J.P. Steghöfer, J. Denzinger, H. Kasinger, and B. Bauer: Improving the Efficiency of Self-Organizing Emergent Systems by an Advisor, Proc. EASe 2010, Oxford, 2010, pp. 63–72.