Universität Augsburg

Learning Task Patterns to Improve Efficiency and Coordination in Decentralized Autonomic Computing Systems

Jan-Philipp Steghöfer[1], Jörg Denzinger[2], Holger Kasinger[1], Bernhard Bauer[1]

[1] Institute of Computer Science, University of Augsburg
[2] Department of Computer Science, University of Calgary

Report 2009-13 June 2009

Institut für Informatik
D-86135 Augsburg

Learning task patterns to improve efficiency and coordination in decentralized autonomic computing systems

Jan-Philipp Steghöfer[1], Jörg Denzinger[2], Holger Kasinger[1], and Bernhard Bauer[1]

[1] Institute of Computer Science, University of Augsburg, Germany
{steghoefer,kasinger,bauer}@informatik.uni-augsburg.de
[2] Department of Computer Science, University of Calgary, Canada
denzinge@cpsc.ucalgary.ca

Abstract. We present the concept of an efficiency and coordination advisor for autonomic computing approaches for dynamic optimization problems. The problem scenarios targeted contain recurring tasks that our advisor identifies over several runs of the autonomous system thus giving it some limited way to "look into the future". If the solutions created by the autonomous agents are much worse than the optimally possible solution, the advisor creates exception rules for the agents that make the wrong decisions for the recurring tasks. This allows them to do better decisions in the future in very specific situations while still retaining all advantages of the autonomic computing approach.
Our experiments with dynamic instances of the pickup and delivery problem that have recurring tasks in it show that with our advisor approach we can improve substantially instances that result in suboptimal behavior of the autonomous agents without advisor. Our advisor approach is also successful if the recurring tasks change over time.

1 Introduction

The goal of autonomic computing is to develop concepts for systems that are able to adapt themselves to the usage that is made of them and to the environment in which they are placed (see [8]). An autonomic computing system does not have to be monolithic, it can consist of a number of autonomic components that are usually called agents. And the idea of a group of more or less autonomous agents solving problems more or less continuously given to them from the outside in a more or less cooperative manner is part of another area from its early beginnings, namely the area of *multi-agent systems* (see [14]).

But, as our phrasing of the last sentence suggests, not every multi-agent system concept fitting the description can be considered an autonomic computing system. The aspect of autonomy of the agents can range from not even perceiving other agents (and naturally not communicating with them and therefore not taking any commands from them, see [3] as an example that cooperation can be achieved without even seeing or hearing your colleagues) to accepting clearly defined tasks from another agent (and being autonomous only in the selection of the method to fulfill it, as, for example, in [14]).

One problematical aspect that is well known but for which we do not really find convincing solutions are dynamically changing problems that essentially require the multi-agent system to look into the future in order to achieve optimal (or even just very good) problem solutions. An instance of this aspect that was created by the multi-agent system itself was already observed in [14], but a very good example for the relevance of this problem in the real world is given in [5]. A group of autonomous agents where their interactions allow for self-configuration of the whole system (i.e. an autonomic computing system)) is currently seen as the best (and perhaps only) way to at least be able to deal with dynamic changes in problem instances, so that solutions are found and performed. However, with regard to self-optimization of such a system the results are often not very convincing, i.e. looking back it often becomes obvious that there are much better solutions than the one that emerged during the run of the system. This is not surprising, since looking into the future is not exactly an ability that is easy to achieve.

But if we look at how experienced humans solve such problems, like, for example, transportation problems like pick-up and delivery (see [12]), then we have to observe that very often their solutions are rather good, as if they could really look into the future. This is due to the fact that in real-life dynamic problems we often find quite a few recurring problem parts (tasks) that humans are able to detect and then create a general problem solution around solutions that are (near) optimal for the recurring tasks. While this comes with the risk that from time to time the overall solution gets worse for a little time – if, for example, an expected recurring task does not occur when expected – the overall performance over longer periods of time is usually better than what a purely reactive solution approach achieves.

In this paper, we present an approach for autonomic computing systems consisting of groups of autonomous agents that mimics the idea of learning recurring tasks and pro-actively expecting these tasks to improve the overall problem solving behavior of the group, while still preserving the features of such groups that allow to do local decisions and to react to dynamic changes in the problem instance to solve. Our approach centers around the idea of an *advisor* (called efficiency and coordination advisor: ECA) that collects the local history of the agents, detects global task patterns that are recurring and that are currently solved far from optimal, and provides the agents with advice in form of exception rules that the agents can add to their own problem solving behavior. In contrast to a central control (and therefore the nearly total loss of autonomy of the agents), the advisor only communicates with the agents when they are ready, its advice can be ignored by the agents (and will be ignored when it is not useful anymore), the agents can still work without it or if it crashes, and every problem solving decision is still locally done by each agent.

Our experimental evaluation of the ECA approach with a group of autonomous agents based on the pollination approach (see [7]) with instances of the pickup and delivery problem showed substantial improvements for examples where the autonomous agents without ECA are known to produce suboptimal solutions. Even for randomly created examples (with recurring tasks) there were improvements and the approach is also able to deal with changing recurring tasks.

This paper is organized as follows: After this introduction, in Section 2 we present basic definitions about the general problem setting for the autonomic computing systems we are interested in. In Section 3, we present the general description of our efficiency and coordination advisor approach, that we instantiate in Section 4 to solving dynamic pickup and delivery problems. In Section 5, we present the experimental evaluation of our proof-of-concept system in this application domain, followed by a discussion of the related work in Section 6. Section 7 concludes the paper and presents some ideas for future work.

2 Basic Definitions

In this section, we introduce the general scheme we will instantiate to describe our agents and we will provide basic notations about the general problem setting our agents are supposed to solve.

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 we have an action for every combination of activities the agent can do (which can always be realized in a model). If f_{Ag} is not much influenced by the value of Dat, then $\mathcal{A}g$ is called reactive.

A multi-agent system (MAS) is then a group of agents A, denoted by $A = \{\mathcal{A}g_1,...,\mathcal{A}g_n\}$, that share an environment $\mathcal{E}nv$. As already stated, the agents in A might all have different sets of situations, actions and internal data area values and they naturally can also have different decision functions.

The general structure of problems for a set A we are interested in consists of tasks out of a set T that are announced to A (or members of it) at some times within a given time interval $Time$ to form a *run instance* for the system A. And there will be a sequence of run instances that A has to solve. We can think about a run instance as being all the tasks A has to solve at a particular day, for example, and a sequence of run instances as the tasks to solve over several days. Naturally, a task for a concrete application will be described by a group of features, but for our purposes generic tasks and the time of their announcement is 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

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

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. Solving ta'_i might require a sequence of actions by $\mathcal{A}g'_i$.

Again, depending on the application there might be additional restrictions, for example if not every agent can perform every task, we need that $\mathcal{A}g'_i$ can indeed perform ta'_i. But note that $\{t_1, ..., t_m\}$ and $\{t'_1, ..., t'_m\}$ 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.

Usually, users of A associate with a solution *sol* a quality $qual(sol)$, which again is dependent on the particular application a user is interested in and the agents in A have been created for. And with the concept of solution quality naturally comes the wish that A produces a solution that is of optimal quality. Under some circumstances creating an A that produces optimal solutions is easy, but under many circumstances it is difficult (for example, finding an optimal solution might be a NP-complete problem for the particular application) or even impossible. 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 enhanced by the additional requirement that all tasks need to be fulfilled within $Time$) will often lead to non-optimal solutions, since in order to create an optimal solution not only has A to be able to find optimal solutions quickly, it also has to be able to "look into the future", so that a new incoming 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. These are the kind of run instances that are of interest to us in the following.

3 The Efficiency and Coordination Advisor Approach

As stated in the previous sections, in this paper we are interested in multi-agent systems that solve 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 to solve the whole (developing) run instance optimally and therefore system developers have concentrated on creating multi-agent systems for such tasks that have other important properties, like robustness against failure or low communication costs, to name just a few. Most of these properties are associated with a high autonomy of the agents, where the individual agents have only local views of the environment and therefore favor reactive behavior (see [10] or [4]). Also, agents in the system might be developed by different people and autonomy is a plus in such situations, again, because with autonomy comes less problems getting the developers to cooperate so that the agents cooperate. But, naturally, the quality of the solution created by the agents is still very important!

In this section, we will present an approach that enhances multi-agent systems for the stated type of problems to allow for better solutions over a sequence of run instances while preserving the other properties of the system, if the following conditions are fulfilled:

- each agent's decision function can be extended to deal with so-called exception rules (that will be stored in the agent's internal data areas),
- each agent is able to "dump" a history of its behavior to a central collection unit at least once during a run instance,
- 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 easily achieved, the third condition seems very restrictive. But in everyday life, there are many problems that fulfill this condition. Delivery companies usually have daily recurring tasks together with one-of-a-kind tasks, to give just one example.

The proposed approach to improve the performance of a team of agents $A = \{\mathcal{A}g_1,...,\mathcal{A}g_n\}$ is to add to A a special agent $\mathcal{A}g_{ECA}$ (**E**fficiency and **C**oordination **A**dvisor) that collects the history of all agents, creates a global view of the history of A (and the environment around A), identifies sequences of tasks that are recurring and that A did not solve very well, creates advice for the individual agents how they should tackle the tasks from the identified sequence, and makes this advice available in the form of the already mentioned exception rules[3]. In the following, we will present the interaction scheme between the $\mathcal{A}g_i$s and $\mathcal{A}g_{ECA}$ more precisely and look at each of the actions of $\mathcal{A}g_{ECA}$ in more detail. Figure 1 gives a graphical representation of the general interaction scheme.

[3] $\mathcal{A}g_{ECA}$ does not have 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 $\mathcal{A}g_{ECA}$. But 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 problem we are interested in, which is why we present our approach in this manner.

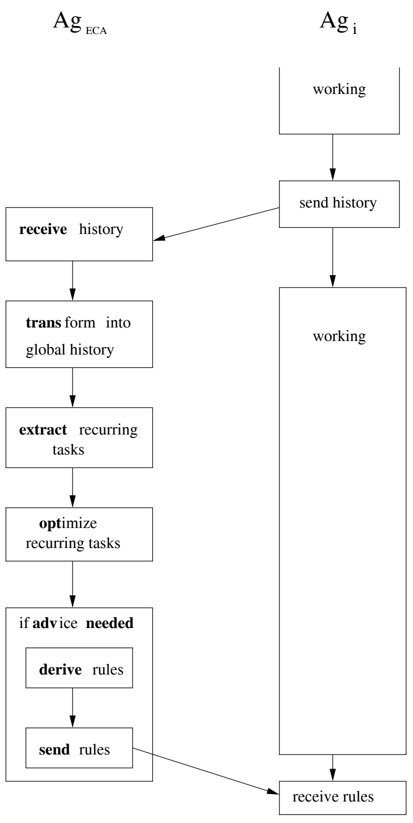

Fig. 1. Typical interaction of an agent with $\mathcal{A}g_{ECA}$

3.1 Using $\mathcal{A}g_{ECA}$

$\mathcal{A}g_{ECA}$ is an agent that can be seen as an external analysis tool for the agents in A, looking out for opportunities to improve the efficiency of A by providing coordination advice. As a consequence, its set Sit_{ECA} centers around the communication with members of A and its set Dat_{ECA} represents the data received from the $\mathcal{A}g_i$ and the intermediate results by $\mathcal{A}g_{ECA}$'s actions towards creating advice for the $\mathcal{A}g_i$s. Its decision function f_{ECA} creates the following steps represented by the indicated actions out of Act_{ECA}[4]:

1. While A performs a run instance, $\mathcal{A}g_{ECA}$ performs `receive`($\mathcal{A}g_i$, $((s_i^1,d_i^1,a_i^1),...,(s_i^o,d_i^o,a_i^o)))$ for each agent $\mathcal{A}g_i$ when $\mathcal{A}g_i$ is able to communicate.
   $H_i = ((s_i^1,d_i^1,a_i^1),...,(s_i^o,d_i^o,a_i^o))$, with $s_i^l \in Sit_i$, $d_i^l \in Dat_i$, $a_i^l \in Act_i$, is the history of agent $\mathcal{A}g_i$ since the sequence of run instances started.

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

2. `trans`(H_1,...,H_n) creates the global history $GHist$ out of the received histories of all agents, which 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 out of the system history – more precisely the sequence $(ri_1,...,ri_k)$ – a sequence of recurring tasks $(ta_1^{rec},...,ta_p^{rec})$.
4. `opt`($(ta_1^{rec},...,ta_p^{rec})$) computes from this sequence the optimal solution

$$opt^{rec} = ((ta_1^{rec}, \mathcal{A}g_1^{'rec}, t_1^{'rec}), ..., (ta_p^{rec}, \mathcal{A}g_p^{'rec}, t_p^{'rec}))$$

   where $\mathcal{A}g_j^{'rec} \in A$ and $t_j^{'rec} \in Time$, 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 $Time$.
5. `advneeded`(opt^{rec},$GHist$) 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 created. If

$$qual(last)/qual(opt^{rec}) > qualthresh$$

   the work of $\mathcal{A}g_{ECA}$ is done until new information arrives, since A performs well. Else, the following actions are performed:
   (a) `derive`(opt^{rec},$(ta_1^{rec},...,ta_p^{rec})$,$GHist$, $last$) creates for each agent $\mathcal{A}g_i$ a set R_i of exception rules, where R_i can be empty.
   (b) for each agent $\mathcal{A}g_i$, `send`($\mathcal{A}g_i$,R_i) is performed the next time communication with $\mathcal{A}g_i$ is possible. Note that Figure 1 might be a little bit deceiving with regard to how much time can be between `derive` and `send`.

Naturally, concrete realizations of these actions of $\mathcal{A}g_{ECA}$ depend on the concrete application problem and on the realization of the $\mathcal{A}g_i$s, including their normal cooperation scheme. But the actions `extract` and `derive` allow for different general ways how they can be realized and in the following subsections we will discuss them further and then close this section with some comments on the other actions.

3.2 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 a little bit more complicated than just finding tasks that occur in each run instance. For these applications, we are not only interested in tasks that are identical in all run instances but also in tasks for which there are *similar* tasks in all (or at least most) of the run instances. To use delivery problems, 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 delivery task into the sequence of recurring tasks.

More precisely, we assume the existence of a similarity measure $sim : T \times T \rightarrow \mathbb{R}^+$ that is used by the action `extract`. Then *clustering* of the tasks in

ri_1,...,ri_k according to *sim* can be used to identify the recurring tasks in these run instances[5]. Again, there are several known clustering methods. One of them, that we found useful for our problem because it does not require to be given initially the number of clusters it should produce, is Sequential Leader Clustering (see [6]). Sequential Leader Clustering in its basic 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. How the representative is determined depends on how a task is described. 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 and this means that we do not only put the $c'_i \in C'_i$ with $sim(c'_i,c_i)$ is minimal into the set of recurring tasks, but also the $c''_i \in C'_i - \{c'_i\}$ 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, this approach for realizing `extract` should not use all k 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} should be defined and only the run instances $ri_{k-k_{max}}$,$ri_{k-k_{max}+1}$,...,ri_k should be used in the clustering and k_{max} is used in the conditions using *minocc*. While a change in the recurring tasks obviously will not immediately be noticed (again, doing that would require to really being able to look into the future), at the latest k_{max} run instances after the change $\mathcal{A}g_{ECA}$ 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, $\mathcal{A}g_{ECA}$ will not be able to detect recurring tasks very well and A will have to rely on the basic decision making of its agents without advice.

3.3 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: they can be used to encourage an agent to take on a certain task or they can be used to detract an agent from taking a certain task.

[5] Naturally, all other techniques from the area of data mining that are able to find recurring event sequences in data could also be used.

And in a MAS, this opens a wide spectrum of possibilities. But for the kind of problems we are interested in, detraction from a task is the only possibility, since it is not easily possible to encourage an agent to do 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 $f_{\mathcal{A}g_i}$.

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 influence, since the agent is only told what *not* to do and still needs to figure out what to do. But since we want to preserve the basic decision making of the agents as much as possible, this is exactly the right thing to do. Note that $cond_{exc}$ can be defined so that the agent's current Dat-value does not matter, which allows $\mathcal{A}g_{ECA}$ to give advice to agents it does not know much about. But including the current Dat-value will allow for better targetting of the exception.

For $\mathcal{A}g_{ECA}$ to decide which exception rule to create for which agent, we need to compare the optimal solution opt^{rec} and the last solution $last$. If opt^{rec} = $((ta^1_1,\mathcal{A}g^1_1,t^1_1),...,(ta^1_p,\mathcal{A}g^1_p,t^1_p))$ and $last = ((ta^2_1,\mathcal{A}g^2_1,t^2_1),...,(ta^2_p,\mathcal{A}g^2_p,t^2_p))$, then $\mathcal{A}g_{ECA}$ looks for the first j with $ta^1_j \neq ta^2_j$ or $\mathcal{A}g^1_j \neq \mathcal{A}g^2_j$. 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 $\mathcal{A}g^2_j$ and naturally has the form $cond_{exc}(s',d') \rightarrow \neg a_{ta^2_j}$. For determining $cond_{exc}(s',d')$, $\mathcal{A}g_{ECA}$ looks up in $GHist$ the triple $(s,d,a_{ta^2_j})$ that represents in the history of $\mathcal{A}g^2_j$ the point when it chose to do $a_{ta^2_j}$. $cond_{exc}(s',d')$ is then an abstraction of s and d that is application dependent and tries to cover not only an activation of ta^2_j, but the whole cluster from the `extract` step of which ta^2_j is a member.

For the application we present in Section 4, this exception rule for $\mathcal{A}g^2_j$ was all we created in one working cycle of $\mathcal{A}g_{ECA}$. But for other applications it might be better to test what the agents in A will do after the exception rule is communicated. If $\mathcal{A}g_{ECA}$ knows enough about the agents in A, it can simulate what 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 and then all rules created using the simulation are communicated to the real agents in A.

3.4 Other actions

The action `trans` can be very easily realized if the environment $\mathcal{E}nv$ is known to $\mathcal{A}g_{ECA}$ 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, `trans` 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 want to require that a recurring task has to appear in every run instance.

The action `opt` requires $\mathcal{A}g_{ECA}$ to have an optimization system for the application that handles the static optimization problem to the dynamic problem that A tries to solve. While this optimizer does not have to look into the future, the static optimization problem still can be very difficult. If it is very difficult or if the time between run instances is short, then it can be necessary to only search for a very good solution for the recurring task sequence instead of the optimal one (and perhaps to have a lower *qualthresh*-value).

Determining the emergent solution created by A for the recurring tasks is in no way trivial. First of all, often there will be other tasks mixed in into fulfilling the recurring tasks and in the last run instance not all of the recurring tasks might have really 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 we 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. If we would add the total traveled distance of this agent between the two recurring tasks to the travel cost (if this is our quality criterion) then the emergent solution is guaranteed to look bad although in other run instances the recurring tasks are solved well.

But for such an application we can provide a lower bound for what the cost is that would emerge, if there were no additional tasks, namely just the distance the agent has to travel after the first recurring task to start performing the second one. And such lower bounds are possible to be determined for many applications and many quality criteria. And if the quality of such a lower bound is far from the optimum, then advice from $\mathcal{A}g_{ECA}$ will be useful for many run instances.

Given that *last* and its quality are already an approximation it is now also easy to solve the problem of a recurring task (or a task sufficiently similar) not occurring in the last run instance. $\mathcal{A}g_{ECA}$ determines which agent fulfills the task in the emergent solution by going back one more run instance (or several). And from there it is also able to determine at what place of the order of tasks for this agent it will perform the task. So, bad results of A for the recurring tasks are due to the wrong agents performing certain tasks or due to a particular agent performing "its" tasks in a bad order.

4 Instantiating the ECA Approach to the PDP

In this section, we present the pickup and delivery problem (PDP) as an instantiation of the general problem class our ECA approach aims at. We then present the digital semiochemical coordination approach (DSC), instantiated for the PDP, as the autonomic computing approach that we want to improve by using the ECA and we instantiate our concepts from the last section for the DSC for solving the PDP.

4.1 The Pickup and Delivery Problem

The general pickup and delivery problem (see [12]) is a well-known problem class that has instantiations like transportation problems in logistics, delivering meals or medication to patients in hospitals or delivering parts in manufacturing plants. Many of these instantiations fulfill the requirements for our ECA approach, i.e. not all tasks to perform are known at the beginning of a run instance and there are tasks that appear in many run instances. Additionally, the transportation agents usually come back to a depot after a run instance, which then can be used to house the ECA.

In the following, we will instantiate the pickup and delivery problem with time windows, 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 a package needs to be picked up, a location $l_{delivery}$ where the package has 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. $ta_{PDP} = (l_{pickup}, l_{delivery}, ncap, t_{start}, t_{end})$. Consequently, an agent $\mathcal{A}g$ needs a transport capacity $cap_{\mathcal{A}g}$ and to perform a task it has to first do the pickup and then the delivery. In general, if $ncap$ for a task is smaller than $cap_{\mathcal{A}g}$, then the agent can do other pickups and deliveries between a pickup and the necessary delivery. In our examples in the next section we will set $ncap$ and $cap_{\mathcal{A}g}$ for all tasks and agents so that this is not possible. We also do not allow agents to switch tasks between them, if they have already done the pickup.

4.2 DSC for the Pickup and Delivery Problem

There are many possible ways how groups of transport agents can perform run instances of pickup and delivery tasks. One concept that focuses a lot at achieving a high autonomy of the individual agents and the resulting resilience of the

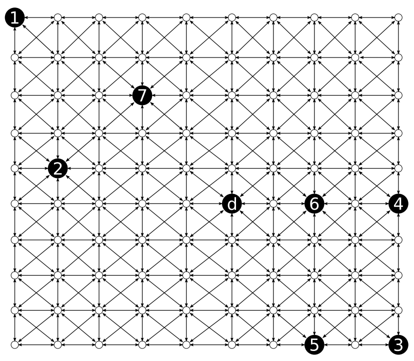

Fig. 2. A “bad” pickup and delivery run instance

whole system is the *digital semiochemical coordination* approach of [7], which generalizes pheromone-based stygmergic approaches, allowing for a wide variety of types of (digital) chemicals to be used between agents of the same type and different types to achieve effects that are beneficial for chemical emitters or receivers or both.

A MAS for pickup and delivery problems was already the example application for DSC in [7] and in the following we will only briefly introduce this application, concentrating on the decision making of the agents, which will be modified by the ECA. A DSC-based system achieves coordination between transportation agents $\mathcal{A}g_1$,...,$\mathcal{A}g_n$ solely using digital semiochemicals that are propagated through the environment the agents are acting in. This environment is essentially a map organized in grids (see Figure 2) and an $\mathcal{A}g_i$ can “access” all digital semiochemicals of the grid field it is currently located on, which naturally has to be considered a very local view for agents.

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 semiochemical) that specifies the location and the transportation requirements, i.e. *ncap* and t_{end}, and one at $l_{delivery}$ also specifying the task via a synomone. In our experiments, the task announcement was done at t_{start}. All such synomones are propagated through the environment and each grid receiving them stores their existence and intensity. The intensity of a chemical is reduced after certain time periods, which is why an emitter agent repeats the synomone emission from time to time until it has been served by a transportation agent. The two synomones for a task are accumulative, i.e. two doses of one of it are combined. So, the overall intensity of a particular chemical on a grid can vary quite a bit over time.

A transportation agent $\mathcal{A}g_i$ “smells” all chemicals at its current location and computes a so-called utility for each task represented. At a location, there can be the following types of semiochemicals around a particular task:

- the synomones indicating the pickup and delivery locations of the task with a certain intensity (as already stated),

- an allomone (another type of semiochemical) emitted by the pickup emitter after a transportation agent has picked up the load, which indicates to other agents that the task has been started,
- a pheromone emitted by another transportation agent that has passed by this location (or very near to it) with the intention of performing the task[6].

The utility for a particular task for agent $\mathcal{A}g_i$ at this moment is then as follows:

- The utility is 0, if either there is an allomone from the pickup emitter indicating that an agent (not $\mathcal{A}g_i$) has started the task or there is a pheromone from another transportation agent for the task.
- The utility has a pre-defined maximum value, if $\mathcal{A}g_i$ has already started this task, i.e. it has done the pickup part.
- Otherwise, the utility is between 0 and the pre-defined maximum value proportional to the intensity of the synomone and the nearness of the current time to t_{end} of the task (if we have time windows active; for more details, please refer to [7]).

After an $\mathcal{A}g_i$ has computed the utility for all tasks it perceives, it selects the task with the highest utility and moves directly towards the emitter representing this task.

The DSC approach is a typical example for a system of autonomous agents that our approach targets. The transportation agents use mainly local information in their decision making, the system as a whole is very robust, since the breakdown of an agent might lower the efficiency of the whole system but does not lead to a general system breakdown and after the pheromones of a brokendown agent have vanished the system has adapted to have one agent less available.

4.3 ECA for DSC for PDP

Before we instantiate our ECA concept, we first need to define the quality *qual* of a solution that our ECA concept wants to improve. There are many possible candidate functions for PDP, like minimizing the distance traveled by the transportation agents or the time needed to complete all tasks. For our experiments we used a function that combines several measures of interest for a PDP. For a solution *sol*, we have

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

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

[6] Note that pheromones are semiochemicals that are used between agents of the same type, in contrast to the other types of semiochemicals used in a system based on DSC.

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 motivation for the 100000 is to end up with *qual*-values that are larger than 1. For $qual_{order}$, we sum up the difference between the start times of all pairs of tasks ta_1 and ta_2 where the start time of ta_1 is before ta_2, but in *sol* we have that ta_2 is finished before ta_1. And for $qual_{tw}$, we sum up the difference between finishing time for a task and t_{end}, if the finishing time is later than the intended end time. Obviously, the three criteria are not aligning well with each other, so that many solutions have rather similar *qual*-values, which makes the task for our ECA harder (since it is less likely to have a big difference in quality between the emergent solution and the optimal solution; note also that penalizing for tasks not done in order also favors making decisions based on local information, favoring the standard autonomous system, again). Note that for this *qual*-function higher values represent better solutions.

The actions `receive` and `trans` for the instantiation of $\mathcal{A}g_{ECA}$ required us to create out of the history of the travels of the $\mathcal{A}g_i$ and their "smell" perceptions a description of the announced tasks[7]. With having the run instances available in the form $(ri_1,...,ri_k)$, we can identify the recurring tasks using the following function *sim*:

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

where α, β, and γ are weight parameters and $Ed((x_1, y_1), (x_2, y_2))$ is the Euclidean distance of two locations. `extract` then employs Sequential Leader Clustering as described before.

For realizing the action `opt`, we needed a way to create the optimal solution for the recurring tasks. While PDP is rather well researched, nevertheless we were not able to find a public-domain optimizer for it, so that we implemented our own branch-and-bound-based optimizer for our proof-of-concept system. This optimizer is not very sophisticated and as a result we were quite limited in the size of problems it could tackle (in acceptable time). But as we show in Section 5, it was good enough to demonstrate the abilities of our ECA approach.

For determining the *qual*-value of the emergent solution, we computed $qual_{dist}$ by using the direct distances between the emitter agents for the recurring tasks, as described in Section 3.4. The $qual_{order}$- and $qual_{tw}$-values are directly available out of the last run instance containing the particular recurring tasks, but obviously $qual_{tw}$ can be heavily influenced by the non-recurring tasks of this run instance.

[7] We also could have simply provided the task announcements to $\mathcal{A}g_{ECA}$, but we wanted to prove that it is possible to create out of the local perceptions an appropriate global picture for $\mathcal{A}g_{ECA}$. Due to lack of space and because the detailed process is not of much consequence for the experiments, we do not go into any more detail regarding these two actions.

Our instantiation of `derive` had to be able to detract an agent from a particular task, resp. all tasks that are similar to it. For DSC we added an additional step to the utility computation. A synomone is compared to the abstracted synomone of an exception rule and – if the comparison shows that they are sufficiently similar – does not consider the synomone, i.e. a bad utility value is assigned to it[8]. This is done for all synomones that were similar enough to any of the exception rules an agent has, but for each synomone-rule-pair we had a time limit after the first application of the rule after which the rule was not applied to this synomone anymore.

The abstracted synomone to a task that should not be taken by an agent consists of the elements l_{pickup}, $ncap$ and t_{start} of the task. To determine the similarity of a synomone representing a concrete task ta_1 to the abstracted synomone ab with $l_{pickup,ab}$, $ncap_{ab}$ and $t_{start,ab}$, we compute the value

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

If $dist_{sem}$ is below a given threshold $semthresh$, then the associated rule is applied.

5 Experimental Evaluation

To evaluate the system described in the last section and with it the usefulness of our ECA approach, we performed several experiments evaluating different aspects of the approach. Creating sequences of run instances that allow such evaluations is not a straightforward task, since the ECA approach only makes sense for sequences of run instances that have some recurring tasks. This means that on the one hand side we have to "construct" such run instance sequences to allow our approach to have a chance, but this construction also needs to ensure that there is enough randomness involved to allow the argument that the approach will work for many instances of the problem.

For all our sequences of run instances with names beginning with "craft" or "rand" (see Table 1) we started with a sequence of tasks that forms the set of recurring tasks the ECA should identify. For each run instance in the sequence each of these tasks (or a slight modification of it within *sim*) is included in the instance with a given probability (95%, except for the craft-4-1 variants, where the probability was 100%). Then some randomly created tasks were added to each of the instances, the number of these tasks 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 1 depends on the number of intended recurring tasks (indicated by the Arab number in the name), namely 10 run instances for the scenarios with 4 recurring tasks, 20 for 6 recurring tasks and 40 runs for 8.

[8] In our implementation, we multiplied the "normal" utility by 0, which eliminated the synomone and therefore the represented task from consideration.

Table 1. Results for not-changing recurring tasks

Scenario	-ECA	+ECA	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%

The ECA approach is intended to improve the performance of the base DSC system. If this base system already performs well, there is not much to improve. Therefore we tested two kinds of scenarios, namely scenarios crafted to not being solved well ("craft-X") and scenarios with a randomly created sequence of intended recurring tasks ("rand-X"). The scenarios with 4 and 6 recurring tasks are done on a 10x10 grid, while all scenarios with 8 recurring tasks are situated on a 20x20 grid. Figure 2 illustrates the crafted sequence of recurring tasks for craft-4-I. The d grid node is the depot for the transportation agents. As in all of our scenarios, we use two transportation agents. The basic set of tasks are ta_1 = (4,6,20,25,30), ta_2 = (3,5,20,25,25), ta_3 = (2,7,20,50,55) and ta_4 = (1,7,20,50,55). We used here the grid node number from the figure instead of coordinates to make it easier to envision the tasks on the grid. The basic DSC system solves these base 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 doing ta_1 and ta_2, while the other agent waits in the depot until ta_3 and ta_4 are announced and then it performs those two tasks. The crafted recurring task sequence for the craft-6-X scenarios used a similar weakness of the basic DSC system.

For all scenarios, we used the following parameter settings: *qualthresh* was set to 95%, $clustthresh = semthresh = 20$, $minocc = 0.7$, $k_{max} = 20$, $\kappa = 1$, $\lambda = 15$, $\mu = 0$ (except for the X-TW scenarios, which enforce the time windows, then we used $\mu = 3$), $\alpha = 0.3$, $\beta = 0.1$, and $\gamma = 0.3$. The time limit for rule application by a transportation agent was 100.

As our experiments reported in Table 1 show, the ECA approach leads to quite some improvements for most scenarios. Remember that a higher *qual*-value (in columns -ECA, i.e. without use of the ECA, and +ECA, i.e. with using advice) means a better solution. Even some of the scenarios with a randomly created set of recurring tasks have a two digit percentage improvement. And the two scenarios enforcing the time windows for the deliveries show higher improvements than the examples that do not enforce the time windows, which is

Table 2. Detailed results of an evaluation run for the scenario craft-6-I

Run Instance	-ECA	+ECA	Rule Created
1	568.18	568.18	
2	1000.00	1000.00	
3	581.39	581.39	
4	694.44	694.44	
5	1388.89	1388.89	
6	343.64	343.64	
7	543.48	543.48	TRUE
8	427.35	467.29	TRUE
9	645.16	636.94	TRUE
10	751.88	617.28	TRUE
11	588.24	806.45	TRUE
12	602.41	699.30	
13	952.38	1136.36	
14	800.00	980.39	TRUE
15	1000.00	1449.28	
16	613.49	884.96	
17	769.23	1123.59	
18	1020.41	1136.36	
19	609.76	826.45	
20	316.47	358.42	

a very good result since PDP with time windows is more difficult to solve (and therefore more likely to not being solved well by just DSC).

Naturally, due to having (randomly created) non-recurring tasks in each run instance, the advise of the ECA does not lead to improvements for every run instance that A tackles. Table 2 presents the detailed results of the sequence of run instances for craft-6-I. The ECA starts providing advice after having seen 7 run instances and, as column "Rule created" indicates, it immediately provides advice to one of the agents. Consequently, for run instance 8 we see the first time a difference between the system with and without the ECA and we see an improvement by 40. The next 4 run instances all see additional exception rules created by the ECA, with instances 9 and 10 having the system with ECA performing worse than the system without it, run instance 10 substantially so. But after that, the system with ECA performs better than the system without.

Table 2 also shows that the randomly created non-recurring tasks result indeed in rather different run instances, as indicated by the quality of the created solutions. Without using the ECA, the quality is between 316 and 1388, which means that the transportation agents used substantially different routes in their solutions. And since the ECA does not influence much the non-recurring tasks, we also see a rather broad solution spectrum between run instances, if it is used.

It should be noted that we could have used the fact that most of the run instances are solved better after the ECA has created the appropriate exception

Table 3. Results for changing recurring tasks

Scenario	-ECA	+ECA	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%

rules to boost the improvements reported in Table 1 by simply having more run instances in a scenario. But we think that the chosen numbers of runs are sufficient to show that the ECA 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 (for example, a company might loose a customer contributing to the recurring task set or new such customers might be added). Our ECA approach is able to deal with this and we demonstrate this by the experiments in Table 3. 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 runs with the first set of recurring tasks. The number 12 for the second set of run instances was chosen, because it is just too small to allow for a change in advice by the ECA. For chang-6-II, we use the same sets of recurring tasks as in chang-6-I, but have 24 instead of 12 run instances before changing "back". As can be seen by the improvement in the improvement between chang-6-I and chang-6-II, if the "change" is for long enough then the ECA can adapt the agents to it. Finally, chang-6-III has 3 blocks of run instances using 3 different (random) sets of recurring tasks, each set for 20 run instances. Again, the ECA shows its usefulness.

Overall, our experimental evaluation shows that our ECA approach is able to provide the autonomous agents with good advice, without destroying the important and useful properties of the underlying agent coordination approach that are necessary to deal with the dynamic nature of the problem.

6 Related Work

The integration of a closed-loop control system, as realized by the ECA, on top of a basic system, which not necessarily has to be a MAS, is a well-known practice originating mainly from control theory. The intention is to overcome the limits of the basic system in guaranteeing an output of a desired quality, in other words to fulfill a desired quality of service (QoS), for dynamic problems itself. Central to this general concept is a device that regulates the output of the basic system (in this case the desired quality of an emergent solution created by A) by adjusting or adapting the system, if the monitored output violates predefined boundaries or thresholds. Instantiations of this general concept can be found in several areas affiliated to multi-agent systems, differing in their strength of central control, their internal realization of the control loop, and in their ability to look into future.

In the area of autonomic computing, an instantiation is presented by the general framework of an autonomic manager (AM) [8], even though on a very coarse-grained level. An AM is a device that reactively controls multiple managed elements, such as processors, databases, or servers and thus focuses more on the optimal performance of IT infrastructures. In contrast to the ECA, the AM acts comparable to a central controller performing a very strong regulation and thus limits the autonomy of the managed elements to a minimal level. In literature, various instantiations of this management framework can be found in different application areas, e.g. power management or data centers.

Another instantiation of the general concept is the observer/controller (O/C) approach [2] in the area of organic computing, even though this instantiation also remains very general itself. This approach equally serves as a framework recommending which functions should be implemented to realize a control loop for the optimization of a self-organizing emergent system. An implementation of this framework is applied to a traffic light control scenario [11] and an evaluation is performed that presents the benefits of the approach compared to traditional systems. Due to the generic description, the ECA fits into this framework, even if it exceeds its capabilities.

In the area of distributed artificial intelligence an instantiation can be found by the management-by-exception approach [13]. Similar to the ECA, this approach tackles the performance of MASs for dynamic optimization problems, at the present time limited to job shop scheduling problems only. If the manager agent (instantiating the control system) detects a violation of the mean flow time of a job (an exception) by monitoring the finished jobs in fixed intervals, it takes over control and orders the shop agents (machines) to work with a fixed strategy that is known to be performing best, as long as the flow time reaches an acceptable range again. Similar to the ECA, the goal is to allow as much flexibility as possible by keeping the time of central control as short as possible, while ensuring an acceptable overall system's performance. However, the management agent does not learn from the past and does not adjust the behavior of the shops on a fine granular level.

Whereas all the approaches mentioned above mainly use classical feedback or reactive control, in [1] 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. The framework is applied to processor power management and distributed signal classification. The online controller within the framework predicts from the current state all possible (or at least a set of) future system states based on a stochastic model up to a prediction horizon and based on that choses a control action that optimizes given constraints. This process is repeated whenever a new system state is available. In contrast to the ECA, this online controller is however not able to adapt the basic system to cope with future situations on its own but controls the system in every situation. The ability to look into the future is based on a stochastic model and not on the actual history. Although the authors state that this model can be obtained via supervised learning where the system is first simulated for various environmental inputs, in contrast to the ECA, the online controller is not able to autonomously adapt its control actions to significantly changed situations that are not covered by the model.

Many approaches for dynamic optimization problems in multi-agent systems are by contrast very specific to the problem. For the PDP, such an approach is presented in [9], where the authors propose a model that incorporates events that occur with a high frequency into the generation of vehicle routes. Routes are planned in a fashion that allows the vehicles to fulfill requests at locations with a high demand without altering the routes. However, the system does not observe past runs to learn from experience which locations frequently generate more requests than others but relies on a predefined stochastic distribution of events. The ECA not only predicts based on past experience but also adapts to changes in the distribution of requests.

When focusing on solutions for a specific dynamic problem, online-algorithms as e.g. presented in [15] are employed. These approaches usually calculate an optimal solution for all problems known at a specific time with a traditional optimization algorithm and apply this solution before calculating the next one for the newly arrived requests. In that case, no learning takes place whatsoever and solutions applied once are not used the next time a similar situation is encountered.

None of the aforementioned approaches has the (limited) ability to look into the future and based on that to adapt the basic agents to cope with similar future situations on their own without violating performance boundaries again. Learning of behavior is a technique that is widely applied in multi-agent systems (we already mentioned [3] as one example), but the emphasis in these approaches is on achieving a goal, which in our scenario would be fulfilling one task, not many constantly changing goals. And either all agents are centrally controlled by the learner or each agent tries to figure out its role on its own. Thus, in the long run, the time the ECA influences the control of the whole system is significantly less compared to the other approaches. Because the agents additionally have the chance of acting autonomously, the difference between a controller and an adviser concept becomes obvious.

7 Conclusion and Future Work

We presented the efficiency and coordination advisor approach that provides an autonomic computing system solving dynamic optimization problems with a limited capability to look into the future, if the problem instances to solve contain recurring tasks. By having the advisor provide individual agents with exception rules, these agents will not react to certain tasks for a given time, allowing other agents to perform these tasks while being available for later tasks for which they are better suited.

Our experimental evaluation showed that examples that are known to be solved not well by the basic system are much better solved when using our advisor approach. Even randomly created examples (with recurring tasks) profited from our approach and it is also able to deal with changes in the set of recurring tasks.

In the future, we want to improve our existing system with a better optimizer allowing for more complex run instances. Naturally, using the ECA approach for other autonomic computing concepts is also on our to-do list. Additionally, there are more possibilities for advice than just telling an agent to ignore a task, like encouraging an agent to go to a place without it accepting a task.

References

[1] S. Abdelwahed and N. Kandasamy: A Control-Based Approach to Autonomic Performance Management in Computing Systems, in M. Parashar and S. Hariri (eds.): Autonomic Computing: Concepts, Infrastructure, and Applications, CRC Press, 2007, pp. 149–167.

[2] J. Branke, M. Mnif, C. Müller-Schloer, H. Prothmann, U. Richter, F. Rochner and H. Schmeck: Organic Computing – Addressing Complexity by Controlled Self-organization Proc. ISoLA 2006, Paphos, 2006, pp. 200–206.

[3] J. Denzinger and M. Fuchs: Experiments in Learning Prototypical Situations for Variants of the Pursuit Game, Proc. ICMAS-96, Kyoto, 1996, pp. 48–55.

[4] M.B. Dias, R. Zlot, N. Kalra and A. Stentz: Market-Based Multirobot Coordination: A Survey and Analysis, Proc. of the IEEE 94(7), 2006, pp. 1257–1270.

[5] K. Fischer, N. Kuhn, and J.P. Müller: Distributed, Knowledge-Based, Reactive Scheduling in the Transportation Domain, Proc. IEEE Conf. on AI and Applications, San Antonio, 1994, pp. 47–53.

[6] J.A. Hartigan: Clustering Algorithms, John Wiley & Sons, 1975.

[7] H. Kasinger, J. Denzinger, and B. Bauer: Digital Semiochemical Coordination, Communications of SIWN 4, 2008, pp. 133–139.

[8] J. Kephart and D. Chess: The Vision of Autonomic Computing, IEEE Computer 36(1), 2003, pp. 41–50.

[9] J. Kozlak, J.-C. Crput, V. Hilaire, A. Koukam: Multi-agent Approach to Dynamic Pick-up and Delivery Problem with Uncertain Knowledge about Future Transport Demands, Fundamenta Informaticae, vol. 71(1), 2006, pp. 27–36.

[10] M. Mamei and F. Zambonelli: Co-Fields: A Physically Inspired Approach to Motion Coordination, IEEE Pervasive Computing 3(2), 2004, pp. 52–61.

[11] H. Prothmann, F. Rochner, S. Tomforde, J. Branke, C. Mller-Schloer, and H. Schmeck: Organic Control of Traffic Lights, Proc. ATC 2008, Stavanger, 2008, pp. 219–233.

[12] M.W.P. Savelsbergh and M. Sol: The General Pickup and Delivery Problem, Transporatation Science 29, 1995, pp. 17–29.

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

[14] R.G. Smith: A Framework for Distributed Problem Solving, UMI Research Press, 1979.

[15] J. Yang, P. Jaillet, and H. Mahmassani: Real-Time Multivehicle Truckload Pickup and Delivery Problems, Transportation Science 38(2), 2004, pp. 135–148.