Testing Self-Organizing Emergent Systems by Learning of Event Sequences

Jonathan Hudson, Jörg Denzinger
Department of Computer Science, University of Calgary
Calgary, Canada

Holger Kasinger, and Bernhard Bauer
Department of Computer Science, University of Augsburg
Augsburg, Germany

Internal Report 2009-949-28
Department of Computer Science, University of Calgary

November 26, 2009

Abstract

We present an approach to test self-organizing emergent systems for unwanted behavior with respect to inefficiencies in task fulfillment based on evolutionary learning of event sequences. By using the differences in produced solution quality versus optimal quality to guide the evolutionary search and by using in addition to standard evolutionary operators targeted ones reflecting knowledge about the tested system, the usual evolutionary learning effects can take place, leading to event sequences that are solved badly by the tested systems. In our experimental evaluation of 2 variants of a self-organizing emergent system for dynamic pickup-and-delivery problems, a system using our learning testing approach created clear evidence that the basic variant of the tested system has problems regarding the efficiency of the solutions it produces and that the efficiency improved version leads even in an extremely negative setting for it to only about double the quality costs.

1 Introduction

Over the last few years, we have seen a tremendous growth of interest in systems capable of self-adaptation and self-organization. While quite a number of conferences, like SASO (Conference on Self-Adaptive and Self-Organizing Systems), ICAC (International Conference on Autonomic Computing), or EASe (Conference on Engineering of Autonomic and Autonomous Systems) to name just a few, show this interest in the scientific research community, initiatives like IBM's Autonomic Computing (see [13]), Intel's Proactive Computing (see [24]), or Hewlett Packard's Adaptive Infrastructure (see [10]) also show the massive interest by industry. The reason for this interest are properties like scalability, robustness, flexibility, and adaptivity that these kinds of systems offer since they provide a solution to the problem of the ever increasing complexity of today's computer systems (see, for example, [15]).

While there is still some confusion with regards to terminology in this area and although current primary research goals focus on achieving the various properties of such systems, it is already very clear that such systems will provide substantial challenges with regards to testing them. Properties like self-adaptation or self-optimization do not only result in rather complex use cases describing what such a system is supposed to accomplish, they also come with dangers like over-fitting (or over-adaptation) in the case of self-adapting systems and emergent misbehavior in case of self-organizing emergent systems (see [18]). From the point of view of testing, these dangers require establishing that there are no (probable) uses of a system of this type that result in overfitting to the environment or emergent misbehavior. This is among the most difficult testing problems and up until very recently purely the domain of human testers (performing so-called *exploratory testing*, see [1]).

Search-based methods, more precisely learning of behavior, offer the only known way to at least partially automate the crucial part of exploratory testing, namely detecting "small" unusual results and expanding them towards detecting an unwanted behavior of the tested system. This has been shown in [4] and [6], but not for any applications with self-organizing emergent systems. In this article, we present an application of testing by learning of behaviors for a general concept for self-organizing emergent systems, namely digital infochemicals (see [11] and [12]), and an improvement of this concept for problems about dynamic task fulfillment that adds the idea of a learning advisor (a so-called efficiency and coordination advisor, ECA) to the agents of the self-organizing emergent system that improves the efficiency of this

system (see [23]). Efficiency is a property that is normally not a strong point of self-organizing emergent systems, but that is for many applications, such as dynamic fulfillment of all kinds of tasks, a desirable goal. Consequently, getting an idea how inefficient general concept and improvement can get for an application and a basic setting is an important question and a typical task for exploratory testing and thus for the search-based learning of behavior.

Our approach to this test problem modifies the approach from [4] and [6] to evolutionary learn sequences of newly announced tasks for the self-organizing emergent system. Since we are interested in dynamic task fulfillment, we also have to associate with each task the time of its announcement to the system, which creates an *event* for the system. The fitness used by the evolutionary learner is the efficiency achieved by the tested system, more precisely a sequence is the better the worse the efficiency measure produced by the system when performing the tasks in the sequence is. For the system enhanced with a learning advisor, we learn two event sequences, one where the advisor creates advice for the agents in the system and a second sequence that the system faces after the adaptation produced by the advice supplied by the advisor. The key component of the fitness function for the evolutionary tester is the difference in the efficiency measure produced by the system for the second sequence with and without the advice supplied by the advisor. And, again, the bigger this difference, the fitter the pair of sequences is.

Our experimental evaluation of this general approach for the pickup-and-delivery system from [23] showed that for the basic approach of the self-organizing emergent system we were able to evolve event sequences that resulted in solutions for the system that were around 3.5 times worse than very good solutions for the particular sequence, which emphasises the problems that these kinds of system have with regard to the efficiency in which they fulfill tasks. When testing the advised system, our testing system was able to create sequences following the adaptation that were solved around two times less efficient than without the adaptation by the advisor showing that the dangers of adaptation are not as serious as the inefficiency potential of the base system, especially given that such radical changes in the tasks to fulfill should not occur very often in the applications for which the advisor approach was developed.

2 Basic concepts: Dynamic task fulfillment by a group of agents

In this section, we first present a general agent definition sufficient for presenting the general class of problems we are interested in and the ECA approach for improving self-organizing emergent systems solving this general class. And then we provide basic notations and definitions from this class of problems, namely having a group of agents fulfill dynamically announced tasks.

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 it acts in), Act is the set of actions $\mathcal{A}g$ can perform, Dat the set of possible values of the agent's internal data areas (i.e. the internal knowledge it can store and work with) 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).

A self-organizing emergent system (as a special kind of multi-agent system) consists then of a group of agents $A = \{\mathcal{A}g_1,...,\mathcal{A}g_m\}$ that share an environment $\mathcal{E}nv$. The agents in A can be heterogeneous, i.e. they can differ in all four components that describe an agent. Starting with Section 4, we will call the agents of a self-organizing emergent system that we test $\mathcal{A}g_{tested,i}$.

The general structure of problems for an A that 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 so-called *run instance* for the system A. And there will be a sequence of run instances that A has to solve. For example, tasks could be deliveries that the agents in A have to make. Then a run instance are all deliveries of a particular day and we are interested in having A perform the deliveries over several days.

More precisely, we describe a run instance as a sequence

$((ta_1,t_1),(ta_2,t_2),...,(ta_l,t_l))$,

with $ta_i \in T$, $t_i \in Time$ and $t_i \leq t_{i+1}$. We also call a pair (ta_i,t_i) an *event*. A sequence of run instances of length k is then described as

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

A solution sol generated by A for a particular run instance is again a sequence

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

where $ta'_i \in \{ta_1,...,ta_m\}$, $ta'_i \neq ta'_j$ for all $i \neq j$, $\mathcal{A}g'_i \in A$, $t'_1 \leq t'_{i+1}$,

$t'_i \in Time$. Here, the 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. Note that solving ta'_i might require a sequence of actions by $\mathcal{A}g'_i$.

Performing the actions that solve a given task usually results in some kind of cost for the agent and/or the whole A. With such a cost comes the idea to prefer solutions that have a low cost, i.e. are of a certain quality according to a quality measure $qual(sol)$ for a solution sol. While there are some instances of our general problem structure for which it is easy to produce optimal solutions, for most instances it is difficult 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 $Time$ 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 the advised self-organizing emergent systems that we are looking at.

3 Optimizing dynamic task fulfillment by a self-organizing emergent system with a learning advisor

Since the problem of optimizing dynamic task fulfillment by a group of agents faces the problem that coming up with an optimal solution might be impossible (which is often the case if there is at least one t_i such that $t_i < t_{i+1}$), system developers have concentrated on creating self-organizing emergent systems of agents that have other beneficial properties, like robustness against failures or adaptivity to environmental changes. 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 [3] or [16]), i.e. in terms of our agent definition there is little influence of an agent's current value from Dat on the decisions of f_{Ag}. But, naturally, the quality of the solution created by the agents still remains important!

[23] presented a generic approach for enhancing all kinds of self-organizing emergent systems for the stated class of problems to allow for better solutions over a sequence of run instances, while preserving the beneficial properties of the underlying system. Under the assumption that it is possible that

- each agent's decision function can be extended to deal with so-called exception rules (which will be stored in the agent's internal data areas),
- each agent is able to "dump" a history of its local 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,

the approach proposes to add to the self-organizing emergent system an additional agent, a so-called efficiency and coordination advisor, ECA or $\mathcal{A}g_{ECA}$.
$\mathcal{A}g_{ECA}$

- collects the history of all agents (action `receive`, performed when individual agent and ECA can communicate with each other),
- creates a global view of the history of A (and the environment around A as far as possible) (action `trans`),
- identifies sequences (ta_1^{rec},...,ta_p^{rec}), $ta_j^{rec} \in T$, of tasks that are recurring and that A did not solve very well (realized by actions `extract`, `opt`, and `advneeded`),
- creates advice for the individual agents how they should tackle the tasks from the identified sequence (action `derive`),
- and makes this advice available in the form of the already mentioned exception rules (action `send`, performed when agent and ECA can communicate with each other).

The general interaction scheme between $\mathcal{A}g_{ECA}$ and the other agents using these actions is depicted in Figure 1.

There are several possibilities for realizing the individual actions of the high-level scheme from above. And how these actions are realized additionally depends on the application problem at hand and on how the agents in the self-organizing emergent system are realized, resp. on what scheme for creating self-organizing emergent systems the system is based. For example, the action `opt` that has to determine for a given sequence of tasks an optimal solution can use any of the known approaches for solving the static optimization problem for the given application. For some applications, the problem of finding an optimal solution for the static variant of the problem is already very difficult, so that the advisor can also use a method that just

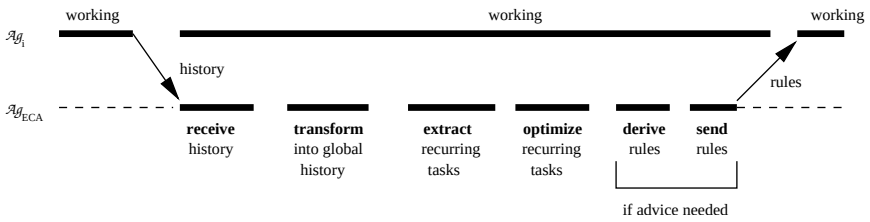

Figure 1: General interaction scheme between an agent $\mathcal{A}g_i$ and the advisor $\mathcal{A}g_{ECA}$

creates a very good solution, i.e. an approximation method. For explaining our testing method, the presented general scheme is sufficient. Readers interested in more information should consult [23].

As stated in the introduction, getting an idea of the efficiency with which a self-organizing emergent system solves instances of the problem it is created to solve is important for practical applications. For the kind of problems the ECA is aimed at, efficiency means how near the quality of the solution produced by the system is to the quality of an optimal solution. There are several reasons, why the produced solution can be non-optimal:

- As already stated, due to the dynamic nature of the problem, the system may already be committed to tasks and therefore newly arriving tasks cannot be handled optimally.
- The basic scheme used by the self-organizing emergent system is not capable to guarantee optimality, due to use of only local knowledge, the used cooperation scheme, or communication restrictions, for example.
- While the ECA should be able to get the system to produce a very good solution, if it can compute an optimal solution to the recurring tasks and the system is given enough time to adapt to the advice of the ECA, there might be also non-recurring tasks in each run instance and for fulfilling these tasks, the basic scheme of the self-organizing emergent system is used.
- The set of recurring tasks the system has been adapted to has just changed dramatically.

With so many sources for non-optimality, it is important to get an idea of how far away from an optimal solution the system might get. Naturally, this depends on individual run instances and on the sequences of run instances

the system faces. And since one of the reasons for using self-organizing emergent systems for this kinds of problems is that instances the system will face can vary quite a lot, there is no convenient set of test instances that can be used to evaluate a particular self-organizing emergent system with regards of its efficiency. In the following section, we will present a way to nevertheless get an idea how "bad" a system is with regard to efficiency, by using learning of event sequences.

4 Testing advised self-organizing emergent systems

In this section, we will first present the general scheme of testing the behavior of systems by learning event sequences. Then we will instantiate this scheme to testing the efficiency of a basic self-organizing emergent system and to testing the efficiency of an advised self-organizing emergent system as described in the last section.

If we look at the general problem of dynamic task fulfillment by a group of agents as described in Section 2 and the problem of how to test systems solving it for efficiency, then the obvious question is what can a tester or testing system manipulate in order to produce a bad or unwanted behavior of the tested system. And the answer is the events in the run instances. Since actions of an agent can also be seen as events, we can generalize the approach from [4] and [6] to testing systems for unwanted behavior by learning event sequences.

In addition to A, the self-organizing emergent system to be tested, and the environment $\mathcal{E}nv$, we add another group of agents $A_{evgen} = \{\mathcal{A}g_{evgen,1},...,\mathcal{A}g_{evgen,n}\}$ that are event generators and that interact with the agents in A by creating events in $\mathcal{E}nv$, i.e. they produce tasks for the agents in A at certain times. And these event generator agents are under the control of a machine learner that learns appropriate event sequences for each agent that achieve the test goal we have for the system.

Formally, each agent $\mathcal{A}g_{evgen,i}$ creates a sequence of events $((ta_1^i,t_1^i),(ta_2^i,t_2^i),...,(ta_l^i,t_l^i))$ and these events and the reactions of the agents in A produce a sequence $e_0,e_1,...,e_x$ of environmental states. The sequence of environmental states is then used by the learner to evaluate the event sequences of the agents in A_{evgen} and to improve them towards the testing goal. Figure 2 presents this testing setup.

Following the lead of [4] and [6], we use an evolutionary learner as the learner in Figure 2. Evolutionary learning is very similar to what an exploratory tester does in his/her tests. An exploratory tester is given a par-

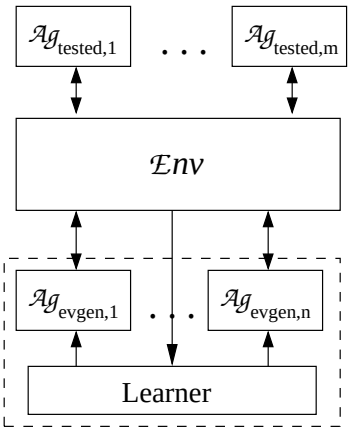

Figure 2: General setting of our approach

ticular test goal and a certain budget (time) for this particular goal. Then he or she will play the role of the agents in A_{evgen} creating events in $\mathcal{E}nv$ that the agents in A will react to producing traces e_0,e_1,...,e_x. The tester then uses his/her knowledge to analyze the traces and to see if there are things that hint at some unwanted behavior and if there is something then (s)he uses knowledge or expertise about application area and testing to come up with other event sequences producing new traces that might provide more hints. And this is repeated until either the tester found an event sequence with A producing the unwanted behavior or the test budget is spent.

Similarly, our evolutionary learner is given a fitness function –that should represent the nearness of a trace e_0,e_1,...,e_x to the test goal– and a certain amount of computing time to fulfill the goal or come as near as possible to fulfilling it. It starts by creating several random individuals, where each individual consists of one event sequence for each $\mathcal{A}g_{evgen,i}$. It then evaluates each individual using the fitness function and, using evolutionary operators and a search control that uses fitness value and random factors to select the individuals the operators are applied to, creates new individuals that are evaluated, again. And this cycle is repeated until either the test goal is fulfilled or the time is over.

For testing the efficiency of self-organizing emergent systems and advised self-organizing emergent systems for dynamic task fulfillment problems, we instantiate this general scheme in the following manner. Testing the efficiency of a self-organizing emergent system essentially means to find a sequence of tasks and their announcement times to the agents in A such that

the quality of the produced solution is much worse than the quality of the best possible solution for the task sequence (or the quality of a very good solution, if the problem of finding an optimal solution even for a static set of tasks is too difficult). Some applications requiring dynamic task fulfillment might require several event generating agents, but in our application in Section 5 we need only one such agent, $\mathcal{A}g_{evgen}$, and this makes describing the evolutionary learning approach easier. Since it is rather obvious how to generalize the one event generating agent case to several, we will in the following describe the one agent case.

So, formally, an individual has the form $((ta_1,t_1),(ta_2,t_2),...,(ta_l,t_l))$ where we do not require $t_i \leq t_{i+1}$ as in a run instance (but we sort the events before running the system that is tested to have real run instances for the interaction) and this event sequence results in a trace $e_0,e_1,...,e_x$, out of which we can determine the generated solution $sol_{gen,-ad}((ta_1,t_1),...,(ta_l,t_l))$ $= ((ta'_1,\mathcal{A}g'_1,t'_1),(ta'_2,\mathcal{A}g'_2,t'_2),...,(ta'_l,\mathcal{A}g'_l,t'_l))$[1], where $ta'_i \in \{ta_1,...,ta_l\}$, $ta'_i \neq ta'_j$ for all $i \neq j$, $\mathcal{A}g'_i \in A$, $t'_1 \leq t'_{i+1}$, $t'_i \in Time$. The individual also has an optimal solution $sol_{opt}((ta_1,t_1),...,(ta_l,t_l))$ and we define the fitness fit_{-ad} of an individual by

$$fit_{-ad}(((ta_1,t_1),...,(ta_l,t_l))) \quad = \quad \frac{qual(sol_{gen,-ad}((ta_1,t_1),...,(ta_l,t_l)))}{qual(sol_{opt}((ta_1,t_1),...,(ta_l,t_l)))} - \frac{qual(sol_{opt}((ta_1,t_1),...,(ta_l,t_l)))}{qual(sol_{opt}((ta_1,t_1),...,(ta_l,t_l)))}$$

if the efficiency goal for the dynamic task fulfillment is to minimize *qual*. Then an individual is the fitter the higher fit_{-ad} is[2] (remember, we want to get an idea how bad the efficiency gets).

Since an individual is a list, we can use standard evolutionary operators for lists, namely a basic single point mutation operator and a basic crossover operator. The basic single point mutation operator takes an individual $((ta_1,t_1),...,(ta_l,t_l))$ and randomly selects an index o. Then the new individual has the form $((ta_1,t_1),...,(ta_{o-1},t_{o-1}),(ta_o^{new},t_o^{new}),(ta_{o+1},t_{o+1}),...,(ta_l,t_l))$, where either $ta_o^{new} \in T$ fulfills $ta_o^{new} \neq ta_o$ or $t_o^{new} \neq t_o$. How ta_o^{new} is created depends on the application, i.e. what a task really is, and there may be several variants for this. The basic crossover operator[3] takes two individuals $((ta_1^1,t_1^1),...,(ta_l^1,t_l^1))$ and $((ta_1^2,t_1^2),...,(ta_l^2,t_l^2))$ and creates the individual

[1] The "-ad" stands for without advice, we will have a solution with advice later when talking about testing an advised system.

[2] If *qual* is to be maximized, then an individual is the fitter, the smaller fit_{-ad} is.

[3] There are several possibilities for a crossover of lists. Some readers might consider the variant we have used as not their first choice and object to using the term "basic". We use "basic" to distinguish this operator from the targeted operators we present later in this section.

$((ta_1^{new},t_1^{new}),...,(ta_l^{new},t_l^{new}))$ where for each i (ta_i^{new},t_i^{new}) is either (ta_i^1,t_i^1) or (ta_i^2,t_i^2) (for example, based on a separate coin toss for each i). There are many schemes for controlling the presented operators and we think that most of them could be used. We will describe the control that we have used in our experiments in the next section.

For testing an advised self-organizing emergent system and thus the concept we presented in Section 3, the goal is different from what is represented by fit_{-ad}. Instead of being interested in the potential size of the difference between the optimal solution and the created solution, of more interest is the contribution of the advisor to any bad efficiency. And naturally we are only interested in cases where the advisor influenced the result of a run instance. This means that we have to modify the just presented approach in the following way.

An individual consists of a pair of event sequences (es_{setup},es_{break}) with es_{setup} being the setup instance that is aimed at creating an adaptation of the tested advised self-organizing emergent system that then results in very bad efficiency of the system when solving the run instance represented by es_{break}. Both event sequences have the form of an individual for the case of just testing the basic self-organizing emergent system. The pair (es_{setup},es_{break}) is evaluated by first having the advised self-organizing emergent system repeatedly solve es_{setup} (repeating this for at least the length of the sequence times is sufficient to achieve the maximum adaptation) and then having it solve es_{break}. The solution $sol_{gen,+ad}(es_{break})$ created should then be compared to the solution $sol_{gen,-ad}(es_{break})$ produced by the system without advisor. But preliminary experiments (see Section 6) showed that the fitness function fit_{+ad} for finding bad efficiency created by the advisor needed more components, resulting in the following definition:

$$fit_{+ad}((es_{setup},es_{break})) = 1.0+\frac{pract(es_{break})+theo(es_{break})+adapt(es_{setup})}{qual(sol_{opt}(es_{break}))}$$

with

$$\begin{aligned}
pract(es_{break}) &= max[(qual(sol_{gen,+ad}(es_{break}))- \\
&\qquad qual(sol_{gen,-ad}(es_{break}))) * w_{pract},0] \\
theo(es_{break}) &= max[(qual(sol_{gen,+ad}(es_{break}))- \\
&\qquad qual(sol_{opt}(es_{break}))) * w_{theo},0] \\
adapt(es_{setup}) &= max[(qual(sol_{gen,-ad}(es_{setup}))- \\
&\qquad qual(sol_{gen,+ad}(es_{setup})) * w_{adapt},0]
\end{aligned}$$

In addition to the difference of the solution quality produced for es_{break} before and after adaptation, weighted by a parameter w_{pract} and measuring the

*pract*ical component of the fitness, we also take into account the difference between the produced solution for es_{break} after adaptation and the optimal solution for es_{break}, weighted by w_{theo} and measuring how far the produced solution is from the *theo*retically possible one, and the difference between the solution produced for es_{setup} without and with the advisor. The later provides an indication how much "change" happened during the *adapt*ation and the more the higher the chance to find a good break sequence.

The evolutionary operators we use for this variant are the obvious extensions of the basic single point mutation operator and the basic crossover operator presented above (with the mutation operator first selecting whether to do the mutation in es_{setup} or es_{break} and then continuing as described and the crossover being performed on both es_{setup} and es_{break} as described) with the addition of a specialized targeted mutation operator that represents some general knowledge about the structure of an individual (which was also the idea behind the targeted operators in [4] and [6]). The targeted twin point mutation operator aims at creating some similar tasks in both the setup and the break instance, so that exception rules created for the setup run instance are triggered when the system fulfills the break instance. This is achieved by aligning a (task,time)-pair in both run instances. Formally, if the individual $(es_{setup},es_{break}) = (((ta_{1,1},t_{1,1}),...,(ta_{1,l},t_{1,l})),((ta_{2,1},t_{2,1}),...,(ta_{2,l},t_{2,l})))$ then the targeted twin point mutation randomly selects a position i and creates the new individual $(((ta_{1,1},t_{1,1}),...,(ta_{1,i-1},t_{1,i-1}),(ta_{1,i}^{new},t_{1,i}^{new}),(ta_{1,i+1}, t_{1,i+1}),...,(ta_{1,l},t_{1,l})),\ ((ta_{2,1},t_{2,1}),...,\ (ta_{2,i-1},t_{2,i-1}),\ (ta_{2,i}^{new},\ t_{2,i}^{new}),\ (ta_{2,i+1}, t_{2,i+1}),...,(ta_{2,l},t_{2,l})))$, where either $ta_{1,i}^{new}$ and $ta_{2,i}^{new}$ are similar tasks (where similar naturally has to depend on the application) or $t_{1,i}^{new} = t + \epsilon_1$ and $t_{2,i}^{new} = t + \epsilon_2$ for a randomly chosen t and small numbers ϵ_1 and ϵ_2 (that are also randomly chosen out of a small interval and that have to make sure that the resulting times are in $Time$).

If we compare the structure of an individual with the structure of the run instance sequences $\mathcal{A}g_{ECA}$ usually deals with, obviously the non-recurring tasks are not represented in an individual. This has several reasons. Learning non-recurring tasks that create bad efficiency would require having many setup run instances in an individual, which greatly enlarges the search space for the learner. Also, we are interested in how much inefficiency can be produced due to the ECA concept. Making it difficult for the ECA to do its job creates only partial advice to the agents, if it creates advice at all. We see the danger of the ECA in an overadaptation of the self-organizing emergent system to a set of recurring tasks and a then following radical change in tasks. And by making finding the recurring tasks easy, we can be sure that our testing reveals the probable extend of this danger.

5 Instantiating testing of advised self-organizing emergent systems to pickup-and-delivery problems

In this section, we present a concrete dynamic task fulfillment problem, namely the dynamic pickup-and-delivery problem (PDP), and a concrete self-organizing emergent system for solving the PDP, which is based on the concept of digital infochemicals. We also present how this system can be enhanced by an advisor and how we instantiate the testing approach from the last section to test the self-organizing emergent system and the advised self-organizing emergent system for the PDP.

5.1 Dynamic pickup-and-delivery problems

The general pickup-and-delivery problem (PDP, see [21]) is a well-known problem class that has instantiations such as transportation problems in logistics or delivering parts in manufacturing plants. Many of these instantiations require solving dynamic instances of the problem and many systems solving these instantiations fulfill the three assumptions mentioned in Section 3 that allow the use of an ECA. Additionally, the transportation agents (i.e. the agents in A) usually return to a depot after a run instance, which can be used to house the ECA. There are also several variants of the PDP, for example the PDP with time windows, that add requirements to the problem. In our experiments, we used the following variant.

A task $ta_{PDP} \in T$ 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, and the needed capacity $ncap$ for transporting the package, i.e $ta_{PDP} = (l_{pickup}, l_{delivery}, ncap)$. An agent $\mathcal{A}g_{tested}$ has a transport capacity $cap_{\mathcal{A}g_{testet}}$ and has first to perform the pickup and then the delivery to accomplish a task. In general, if $ncap$ for a task is smaller than $cap_{\mathcal{A}g_{tested}}$, 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. In our experiments, the environment $\mathcal{E}nv$ is a grid where each grid node represents a possible location.

There are several possible instantiations of the quality measure $qual$ for a solution of the PDP. While [23] uses a rather complex combination of criteria, we have chosen to simply measure the distance travelled by the

agents in A while fulfilling a run instance, including the distances travelled to leave the already mentioned depot and to go back to it after the run instance is over. This measure is not only simpler, it also makes it more likely that the system we test shows bad behavior, since it was not built for this particular quality measure (although it should be able to deal with all possible such measures, according to [23]).

5.2 Solving dynamic PDPs using digital infochemicals

In principle, there are quite a few different ways how transport agents can coordinate their actions in a self-organizing emergent manner to perform pickup and delivery tasks. The coordination model used for the system we want to test –that is applicable for a variety of problems– is the *digital infochemical coordination* (DIC) approach (see [11] and [12]), which generalizes pheromone-based coordination (see [3]) and provides a higher variety of (digital) chemicals to be used for the coordination of agents, to achieve effects that are beneficial for chemical emitting agents, receiving agents, or both.

A self-organizing emergent system solving PDPs based on DIC was already presented in [11], so that in this article we concentrate on the decision making of the agents, which will be later modified by the ECA and which, naturally, is the part of the system that is responsible for its efficiency (and therefore the part that is tested by our approach). A DIC-based system achieves coordination between the tested transportation agents $\mathcal{A}g_{tested,1}$,...,$\mathcal{A}g_{tested,n}$ solely using digital infochemicals that are propagated through the environment the agents are situated in. As already stated, this environment is essentially a map organized in a grid and an $\mathcal{A}g_{tested,i}$ can "access" all digital infochemicals of the field it is currently located on, which provides it with a very local view of the environment and problem to solve.

In [11], a task ta_{PDP} is given to the self-organizing emergent system by "creating" two emitter agents in the environment, one at l_{pickup} emitting a so-called synomone (a type of infochemical) that specifies the location and the transportation requirements, i.e. *ncap*, and one at $l_{delivery}$ also specifying the task via a synomone. All such synomones are propagated through the environment and a grid field receiving them stores their existence and intensity. The infochemical evaporates after a certain time, which is why an emitter agent repeats the synomone emission from time to time until it has been served by a transportation agent. Two synomone emissions for a task are accumulative, i.e. the current intensities of two doses of it are combined. So, the overall intensity of a particular infochemical on a grid field can vary

quite a bit over time.

A transportation agent $\mathcal{A}g_{tested,i}$ "smells" all infochemicals at its current location and computes a so-called utility for each task represented. In addition to the intensity of an infochemical, the utility is also influenced by the agent's current status. If it has, for example, already picked-up the load for a task, it gives priority to delivering it. Additionally, other infochemical types influence the utility computation: a pickup agent emits so-called allomones as soon as an $\mathcal{A}g_{tested,i}$ served it. This indicates to an $\mathcal{A}g_{tested,j}$ ($j \neq i$) that the task execution at this grid field has already started, which then prevents it from being unnecessarily attracted to this field by unevaporated synomones. Thus, an $\mathcal{A}g_{tested,j}$ 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, since an agent that now must be nearer to the field is already taking this task up (although this makes the assumption that this other agent will really fulfill the task, which might not always happen due to the general way of utility computation and the fact that new tasks can come in; but naturally pheromones also evaporate, so that such a task is not permanently blocked). After an $\mathcal{A}g_{tested,i}$ has computed the utility for all tasks it perceives, it selects the task with highest utility and moves directly towards the emitter representing this task.

5.3 Adding an advisor

From the point of view of efficiency, resp. possible unwanted behavior that results in a very bad efficiency, the crucial actions of $\mathcal{A}g_{ECA}$ are `extract`, `derive` and `opt`, the later only if there are time limitations on the advisor and therefore the instantiation of `opt` only produces a good solution without any guarantees for optimality. Therefore all the other actions $\mathcal{A}g_{ECA}$ performs are of not much interest for our testing approach and we refer all readers interested in these actions to [23].

The advisor for the self-organizing emergent system described in the last subsection realizes `extract` by clustering all the tasks performed by the transportation agents with regard to a similarity function sim. The clustering method used is the so-called Sequential Leader Clustering (SLC), see [9], which is a rather old method, but one of the first that did not require to be given initially the number of clusters it should produce. The basic idea of the method is to compute for each cluster a representative and

for every new task the similarity between the representative of each cluster and the new task (with respect to sim) is computed. If the similarity to the most similar representative is greater than a threshold value, then the new task is added to that cluster, else it is the initial element of a new cluster. $\mathcal{A}g_{ECA}$ clusters all tasks from the last k run instances and all clusters that have slightly less than k members represent a recurring task and the representative of such a cluster will also represent it in the computation that instantiates `opt`. The definition of sim for our transportation tasks is not of relevance for our testing, since –as already stated– finding the recurring tasks is not a problem with our approach for testing.

The main idea behind $\mathcal{A}g_{ECA}$ is to identify recurring task sequences within run instances that are not solved very well by the basic system and to create exception rules for transportation agents to nudge them towards creating a better solution the next time the tasks of this task sequence occur. The type of exception rules that are used in the system we test are aimed at detracting the agents that performed tasks in the sequence that they are not performing in the solution suggested by the result of `opt` from the particular task. This can be easily realized by modifying the utility computation of such an agent: any synomone, which is sufficiently similar to an abstracted synomone representing the task the agent should be detracted from, is not considered, resp. has no utility. The abstracted synomone consists of the elements l_{pickup} and $ncap$ of the recurring task that triggered the creation of the exception rule and the time t of its usual announcement to the self-organizing emergent system. The similarity used to decide if the rule is applicable to a task computes the Euclidean distance of the pickup locations and adds to it the differences in capacity and announcement time (which is not really of a big relevance for our testing approach, again). "Sufficiently similar" is realized by providing a threshold value, again. It should be noted that the abstracted synomone does not contain the delivery location (which was not a problem at all in the experiments in [23]).

For realizing `opt` we had to use the variant of not using an algorithm that guarantees finding the optimal solution. While in [23] `opt` was realized using a rather simple branch-and-bound algorithm, this algorithm had rather long runtimes for the examples presented, where after each run instance at most one run of this algorithm had to be performed (if the clusters did not change there was obviously no need to create new advice, therefore `opt` was not executed after each run instance). For our testing by learning of behavior approach, `opt` has to be run for every individual that needs to be evaluated, which means that using the implementation of `opt` from [23] would have meant years to run the experiments from Section 6. Therefore

we used a genetic algorithm using or-trees to realize the creation of random solutions and the genetic operators to create only solutions fulfilling the hard constraints of the static PDP as implementation for `opt` and we provided a maximal resource usage for each call of this algorithm.

5.4 Testing basic system and the advisor improvement

Given the information from the last three subsections, the only components of our testing approach that have not been instantiated for the PDP are the search controls used and the realization of the task mutation for the basic single point mutation operator and the targeted twin point mutation operator.

As already stated, of the three components of a task (l_{pickup},$l_{delivery}$,$ncap$) $ncap$ is chosen so that a transportation agent can only work on one task at a time, so that mutating its value is not useful. Therefore we have, in addition to the mutation of the time when to announce a task, two more variants of the basic single point mutation operator, one that mutates the pickup location l_{pickup} and one that mutates the delivery location $l_{delivery}$. All locations have to be coordinates of a field on the given grid. For the targeted twin point mutation operator, we need to make two events similar, which for the task part of an event means to give the two tasks the same pickup location. This uses the fact that the advisor creates the extension rules using an abstract synomone that only provides information about the pickup location.

The search control for creating a new generation of individuals for the evolutionary learner uses tournament selection to select individuals, whenever it needs to do so. To create the next generation, as usual, a certain percentage of the previous generation is copied over into the new generation. The remaining "positions" in the new generation are created using the operators described before, with given percentages for each of the operators. The first generation is created by creating random individuals.

6 Experimental results

In this section, we will describe several experiments performed with our testing system testing the self-organizing emergent system and the advised self-organizing emergent system described in the previous section. We will first describe the general setting for all the experiments and then present the individual experimental series.

6.1 Experimental settings

Each experimental series uses a 10×10 grid with the depot being located in the middle (coordinates (5,5)). We used as settings for the various parameters of the self-organizing emergent and the advised self-organizing emergent system the values reported in [23]. Our testing system used the following percentages of the new population to be created for the evolutionary operators: we let the 10 percent best indivudals of a population survive. Of the remaining 90 percent of the new generation, 30 percent were generated using crossover, 60 percent using mutation. For testing the advised system, half of the mutations were basic ones, half targeted.

Every experimental series reports on 5 runs of the testing system, due to the random effects within the learning process. In all experiments, we used two transportation agents and tested for 2, 4, 6, 8, and 10 tasks. *Time* was the (integer) interval {0,...,50} for 2 tasks, {0,...,100} for 4 tasks, {0,...,150} for 6, {0,...,200} for 8 and {0,...,250} for 10 tasks. For each number of tasks, we performed one run to get an idea after what generation no improvements seemed to occur anymore and we used this number (which is indicated in the tables) for the other test runs for this number of tasks. In the tables, under efficiency loss factor, we report first the average loss of efficiency and then the maximal loss among the 5 runs. As described earlier, efficiency loss for the basic self-organizing emergent system is defined as the ratio of the quality of the best solution generated by the system to the optimum solution for the set of tasks. For the advised self-organizing emergent system we look at the ratio of the quality of the solution generated by the system for the break run instance after the advisor adapted the system to the setup instance to the quality of the solution generated by the system for the break run instance without having the advisor do anything.

6.2 Testing the self-organizing emergent system for PDP

Table 1 summarizes our experimental results for testing the basic self-organizing emergent system for PDP. As can be seen, with the exception of the 2 task case, the average loss of efficiency is around 3.5, which means that the agents travel around 3.5 times the distance when employing the self-organizing emergent cooperation scheme than is really necessary to solve the problem instances (if a coordination scheme would be used that is able to look into the future). So, at least for the digital infochemicals scheme and its instantiation for the PDP there is evidence that efficiency is a problem.

To give an idea regarding the efficiency of the testing system itself, the

Table 1: Efficiency tests of the basic self-organizing emergent system

# Tasks	# generations	Efficiency loss (avg.)	Efficiency loss (max)
2	40	2.6	2.8
4	80	3.4	4.9
6	120	3.4	3.7
8	160	3.6	4.5
10	200	3.6	4.5

5 runs for 10 tasks took 8.6 hours to complete.

An important reason for the kind of testing we are doing here is to get a better understanding what is going on in the tested system by looking at the event sequences that were produced and how the system reacts to them. Figure 3 presents a visualization of the system reaction to the event sequence that is the result of the max run for 4 tasks[4]. The squares indicate the different grid fields. Our testing system created the run sequence

(((B,F),1),((A,B),17),((C,A),57),((E,D),93))[5]

The travel path of the first transportation agent, $\mathcal{A}g_{tested,1}$, is indicated by the dashed lines, while the normal lines indicate the path of the second agent, $\mathcal{A}g_{tested,2}$. If there are several lines of a type between two nodes this indicates that this part of the path was travelled several times by the agent. If we look at the path produced by the self-organizing emergent system (left picture), we can see that $\mathcal{A}g_{tested,1}$ first does task ((B,F),1), then responds to task ((A,B),17) but does not do it (since $\mathcal{A}g_{tested,2}$ claims it first), returns almost to the depot, then does task ((C,A),57), returns almost to the depot, then does task ((E,D),93), and finally returns to the depot (which is the node indicated by the larger square in the middle of the grid). $\mathcal{A}g_{tested,2}$ does task ((A,B),17), returns to the depot, responds to task ((C,A),57) but does not do it (now $\mathcal{A}g_{tested,1}$ claims this task first) and also not ((E,D),93) and returns to the depot. The travelled distance is 76.2. The optimal solution for this event sequence is to have one agent ($\mathcal{A}g_{tested,1}$) do all tasks in reverse order as announced, i.e. first task ((E,D),93), then task ((C,A),57), then task ((A,B),17) and then task ((B,F),1), after which it returns to the depot. The travelled distance is then only 15.3. This example essentially shows the

[4]Note that while our testing system naturally works on coordinates, for better understandability we have marked the important grid fields by letters and refer to these letters in our description.

[5]We omit here the capacities, since they are, as already stated, not of relevance for our tests.

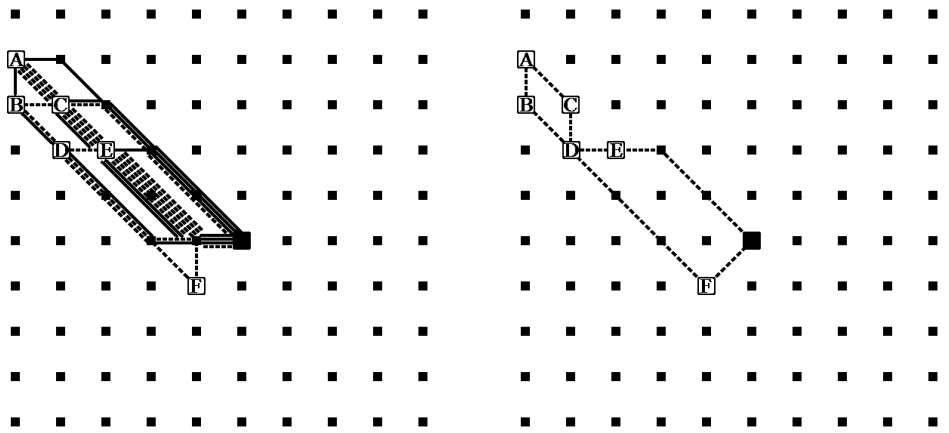

Figure 3: Produced solution (left) vs optimal solution (right)

basic problem behind solving dynamic task fulfillment problems, namely the inability to look into the future, and our testing system is able to find such examples on its own.

6.3 Testing the advised self-organizing emergent system for PDP

Our testing system used as weights in the fitness function for testing the advised self-organizing emergent system w_{pract}=25, w_{adapt}=5, and w_{theo}=1. This means that the primary component is how much worse the break run instance is solved after adaptation by the advisor, compared to the efficiency the system shows for the break instance without the advice. This is also the efficiency loss factor presented in Table 2. As already stated, the other components are there to make sure that the setup instance really leads to an adaptation (which is important in the early stages of the search) and to hint to the search that break instances that are away from being well solved have potential (which, again, is more important in the initial stages of the learning).

As Table 2 shows, the average efficiency loss due to the adaptation is around a factor of two and the worst found examples are also very near to that factor. This means that in the rather extreme situation where a total change of the tasks to fulfill can happen (which is usually not the case in the scenarios for which the advisor was developed) the system user is only two times worse off than without the advisor. We consider that a rather good

Table 2: Efficiency tests for system with advisor

# Tasks	# generations	Efficiency loss (avg.)	Efficiency loss (max)
2	35	1.7	1.9
4	70	1.8	1.9
6	105	2.0	2.1
8	140	1.9	2.1
10	175	2.0	2.0

result for the advisor. With regard to the efficiency of our testing system, the 5 runs for compiling the 10 tasks entry of the table took 19.4 hours to complete.

Figure 4 visualizes one of the run instance sequences found for 2 tasks. The setup run instance is

((C,A),0),((D,F),18)

and the break instance is

((B,C),9),((D,E),18).

Without the advisor, the setup instance is performed by having $\mathcal{A}g_{tested,1}$ (represented by the dotted lines) do task ((C,A),0) and then starting to respond to task ((D,F),18). But since $\mathcal{A}g_{tested,2}$ is also responding to this task, $\mathcal{A}g_{tested,1}$ is abandoning it and returns to the depot (see the left side top part of the figure). The optimal solution for this instance is having one of the agents do both tasks and therefore the ECA creates exception rules for $\mathcal{A}g_{tested,1}$ to detract it from performing any of the two tasks (the abstract synomones are describing tasks starting at C after time 0 and starting at D after time 18). The consequence of this is shown in the bottom part of the left side of Figure 4, where $\mathcal{A}g_{tested,2}$ is first fulfilling ((C,A),0) and then ((D,F),18).

Without the advisor, the system solves the break run instance by having $\mathcal{A}g_{tested,1}$ fulfill first task ((B,C),9) and then ((D,E),18). $\mathcal{A}g_{tested,2}$ responds to the second task ((D,E),18) but then realizes that $\mathcal{A}g_{tested,1}$ is taking the task and goes back to the depot (top of the right side of the figure). This is naturally not the optimal way to solve this instance (which would be for $\mathcal{A}g_{tested,2}$ to ignore both tasks), but it is what the system does. After having adapted to the setup instance, the two agents perform as follows (see bottom grid on the right side): $\mathcal{A}g_{tested,1}$ does task ((B,C),9), but due to the created exception rule about task D, it ignores the task ((D,E),18), which has to be performed by $\mathcal{A}g_{tested,2}$ at an additional cost.

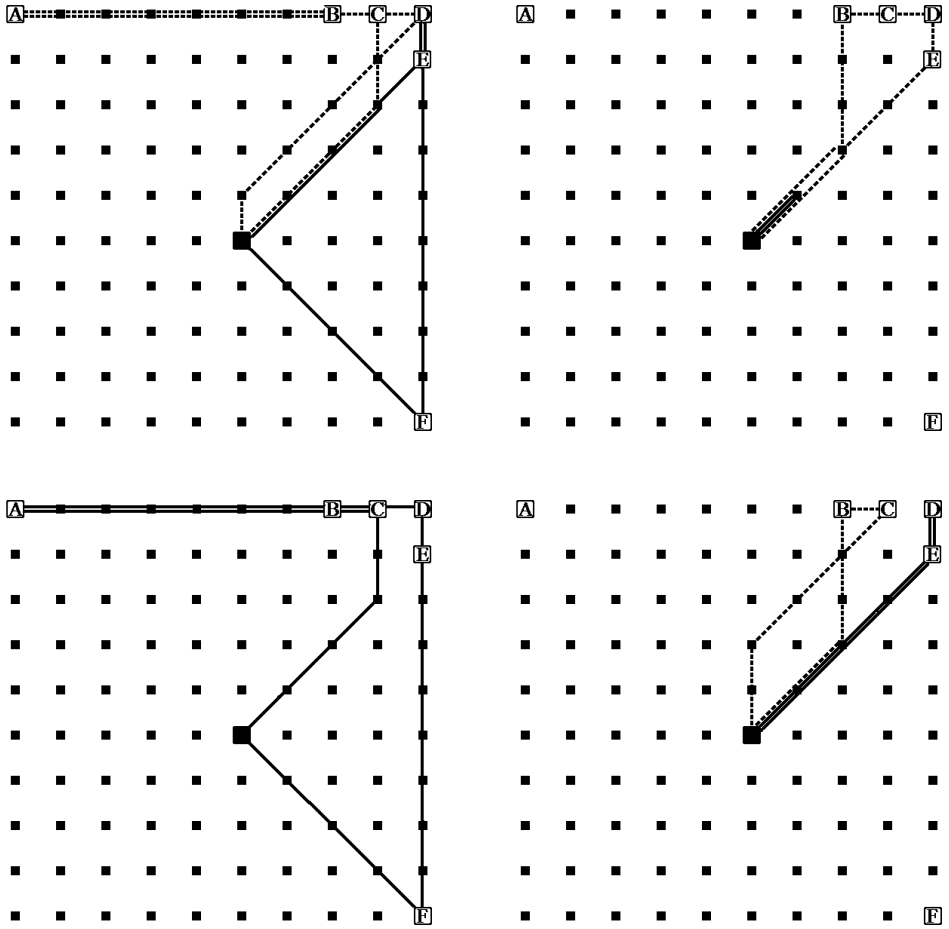

Figure 4: Advisor inefficiency: left side: before advisor (top) vs. after advisor (bottom) for setup instance; right side: before advisor (top) vs. after advisor (bottom) for break instance

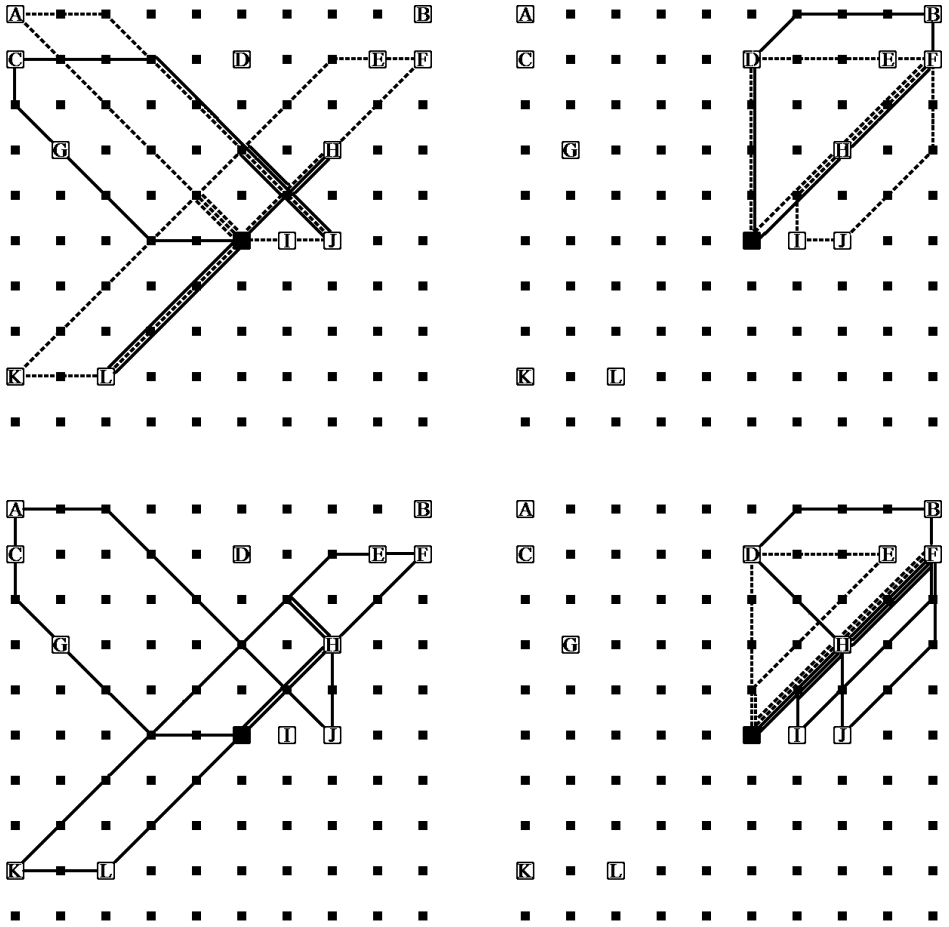

Figure 5: Advisor inefficiency: left side: before advisor (top) vs. after advisor (bottom) for setup instance; right side: before advisor (top) vs. after advisor (bottom) for break instance

Figure 5 visualizes one of the examples with 4 tasks. The setup run instance is

((F,K),19),((L,H),35),((C,G),74),((J,A),75)

and the break instance is

((D,E),18),((F,I),38),((B,D),55),((J,F),58).

Without the advisor, the system solves the setup instance by having $\mathcal{A}g_{tested,1}$ (indicated by the dashed lines, again) fulfill task ((F,K),19), starting to respond to task ((C,G),74) but then switching to task ((J,A),75). $\mathcal{A}g_{tested,2}$ does ((L,H),35), starts to respond to ((C,G),74), switches to start to respond to ((J,A),75) and then switches back to fulfilling ((C,G),74). The advisor realizes that one agent should do all tasks and creates exception rules for $\mathcal{A}g_{tested,1}$ to ignore tasks starting at F after time 19, L after time 35, C after time 74 and J after time 75. As the bottom left part of Figure 5 shows, this does not result in a perfect solution after the advice, since $\mathcal{A}g_{tested,2}$ fulfills ((F,K),19) then ((L,H),35), then starts to fulfill ((C,G),74), but switches to ((J,A),75) and then comes back to fulfilling ((C,G),74) (several of the produced examples showed limitations of the advisor due to only being able to create rules to detract agents from tasks, clearly the advisor can be improved; but note that telling $\mathcal{A}g_{tested,2}$ to ignore ((C,G),74) is not an option, because it would not get back to it after having fulfilled ((J,A),75), respectively it would get back to it only after the time is sufficiently far away from time 74, also in the implementation we tested the advisor is not allowed to give the same exception rule to all agents, exactly to avoid long waits for fulfilling such an excluded task).

Without having the advisor, the break run instance is solved by the system by having $\mathcal{A}g_{tested,1}$ fulfill (((D,E),18), then ((F,I),38) and then ((J,F),58). $\mathcal{A}g_{tested,2}$ fulfills ((B,D),55). As with so many of the instances for a system with two agents, the optimal solution would be to have one agent do all tasks (which was already a suggestion in the all-or-nothing deals of [20], but for higher numbers of task this is not optimal anymore). We also think that examples that essentially require one big loop to fulfill all tasks are the ones allowing for the most inefficiency if not done that way, so that our testing system is biased towards them. After having adapted to the setup instance, the system solves the break instance in the following manner. $\mathcal{A}g_{tested,1}$ fulfills task ((D,E),18), ignores ((F,I),38) because of the exception rule, partially responds to ((B,D),55), discards this task and then ignores ((J,F),58) due to the other exception rule. This means that $\mathcal{A}g_{tested,2}$ fulfills first ((F,I),38), then does ((B,D),55), and then performs ((J,F),58).

Both examples show the importance of having targeted operators that allow to bring knowledge about the application and the system that is tested

into the learning process. The twin point mutation operator allows to connect tasks in the setup and the break instance in such a manner that the other operators are enabled to create events that trigger the exception rules but then result in bad follow-up actions, which is the obvious danger when using an advisor. And our testing system gives us an idea of the potential inefficiency that can come out of this danger.

Naturally, the results produced by our testing system need to be interpreted using the needs of the users of the systems that are tested. For some applications, the factor 3.5 (with potential to get to a factor 5) that we found for the basic self-organizing emergent system might be totally acceptable, because the other properties that such systems exhibit are very important. And for other transportation applications, being by such a factor more expensive might kill the company. But, as with this kind of testing in general, the important point is that examples are found that can be looked at by humans to help with their decision making.

7 Related work

Testing is the major application area for search-based methods in Software Engineering (see [22]). However, there are only very few papers that use search-based methods, respectively learning of behavior, that are even a little bit related to testing for unwanted behavior or testing self-organizing emergent systems. In [2], an approach for evolving the scheduling of presenting given events to a scheduler is presented, with the goal to find times for the announcement of the given events that lead to infeasible schedules generated by the scheduler (resp. a specification of a scheduler). This has some similarities to our testing of the basic self-organizing emergent system, but we do not only evolve the timing of the announcements of tasks, we also evolve the tasks themselves, which is more general. Additionally, we are testing a real system, not a model. And, naturally, we also test a system that adapts (which requires evolving whole events, not just the timing of given tasks).

With regard to using search-based methods around self-organizing emergent systems (which are also called decentralized autonomic systems by some authors), [8] presents an approach to search for behavioral models for autonomic computing systems. While the particular search method is based on the evolution of artificial ecologies, nevertheless the models are for a single agent. Naturally, this is related to the general area of learning of (cooperative) behavior in multi-agent systems ([19] presents, together with

other applications of learning in MAS, an overview of this topic) which is rather well researched and has evolutionary learning as one of the two main learning methods (together with reinforcement learning). In fact, [4] and [6] were inspired by work in this area, namely [5], and this was not the first evolutionary approach for learning of cooperative behavior. The concepts for learning of cooperative behavior usually focus on the particular agent architecture, on how to adapt the general learning approaches to the particular architecture and especially on how to evaluate created behaviors. Also, they have to produce "complete" behaviors, i.e. they want the agents of the system to be able to react to everything that is happening. In contrast, in our testing approach we only need to find *one* specific behavior and an event sequence is not exactly a complex agent architecture, both of which make it possible to be successful in finding something that is rather vaguely defined.

8 Conclusion and future work

We presented a search-based approach to testing the efficiency of self-organizing emergent systems, based on learning event sequences that are not well reacted to by the tested system, resulting in examples for inefficient behavior of this tested system. The approach is also able to test self-organizing emergent systems that learn themselves, thus showing that it is really possible to fight fire with fire. Our experimental evaluation of the approach with a basic self-organizing emergent system and a system enhanced by a learning advisor in the area of dynamic pickup-and-delivery problems showed that our testing system is able to find weak performances of the tested system for various problem sizes.

This work is, in our opinion, only the starting point for a lot of further research. In addition to using the approach to test other concepts for self-organizing emergent systems (or improvements of them) and other application areas for such systems, there are many possible test goals that are of interest and that require the research of ways how to realize them in our approach. For example, for event sequences fulfilling certain conditions, a tester might be willing to accept bad efficiency, so that the test goal is to find bad efficiency event sequences that are not fulfilling those conditions. In general, we have only just started to explore the possible ways how additional knowledge about the application area, the tested system and about testing in general can be integrated into the method and more case studies with so far not used kinds of knowledge are definitely needed to provide users of our testing approach with suggestions on how to use the approach

for their particular problems.

References

[1] Bach J: Exploratory Testing Explained. http://www.satisfice.com/articles/et-article.pdf [13 April 2009].

[2] Briand L, Labiche Y, Shousha M. Using Genetic Algorithms for Early Schedulability Analysis and Stress Testing in Real-Time Systems. *Genetic Programming and Evolvable Machines*, 2006, **7**(2):145–170.

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

[4] Chan B, Denzinger J, Gates D, Loose K, Buchanan J. Evolutionary behavior testing of commercial computer games. Proc. CEC 2004, Portland, 2004; 125–132.

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

[6] Denzinger J, Kidney J. Testing the limits of emergent behavior in MAS using learning of cooperative behavior. Proc. ECAI-06, Riva del Garda, 2006; 260–264.

[7] DeWolf T, Holvoet T. A Taxonomy for Self-* Properties in Decentralised Autonomic Computing. *Autonomic Computing: Concepts, Infrastructure, and Applications,* Parashar M, Hariri S (eds.). CRC Press, 2007; 101–120.

[8] Goldsby HJ, Cheng, BHC. Avida-MDE: a digital evolution approach to generating models of adaptive software behavior. Proc. GECCO'08, Atlanta, 2008; 1751–1758.

[9] Hartigan JA. *Clustering Algorithms*. John Wiley & Sons, 1975.

[10] Hewlett-Packard Development Company: HP Adaptive Infrastructure. http://www.hp.com/go/ai [13 April 2009].

[11] Kasinger H, Bauer B, Denzinger J. The Meaning of Semiochemicals to the Design of Self-Organizing Systems. Proc. SASO-08, Venice, 2008; 139–148.

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

[13] IBM: Autonomic computing initiative. http://www.research.ibm.com/autonomic/ [13 April 2009].

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

[15] Mamei M, Menezes R, Tolksdorf R, Zambonelli F. Case studies for self-organization in computer science. *Journal of Systems Architecture* 2006; **52**(8):443–460.

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

[17] McMinn P. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability* 2004; **14**(2):105–156.

[18] Mogul JC. Emergent (Mis)behavior vs. Complex Software Systems. Internal Report HPL-2006-2, HP Laboratories Palo Alto, 2005.

[19] Panait L, Luke S. Cooperative Multi-Agent Learning: The State of the Art. *JAAMAS* 2005; **11**(3):387–434.

[20] Rosenschein JS, Zlotkin G. Designing Conventions for Automated Negotiation. *Readings in Agents,* Huhns, Singh (eds.). Morgan Kaufmann, 1998; 353–370.

[21] Savelsbergh MWP, Sol M. The General Pickup and Delivery Problem. *Transporatation Science* 1995; **29**:17–29.

[22] SEBASE: Software Engineering By Automated SEarch Repository. http://www.sebase.org/sbse/publications/ [13 April 2009].

[23] Steghöfer JP, Denzinger J, Kasinger H, Bauer B. Learning Task Patterns to Improve Efficiency and Coordination in Decentralized Autonomic Computing Systems. Technical Report 2009-13, Institute for Computer Science, University of Augsburg, 2009, http://opus.bibliothek.uni-augsburg.de/volltexte/2009/1405/.

[24] Tennenhouse D. Proactive computing. *Communications of the ACM* 2000; **43**(5):43–40.