Efficiency Testing of Self-adapting Systems by Learning of Event Sequences

Jonathan Hudson, Jörg Denzinger
Department of Computer Science, University of Calgary
Calgary, Canada
Email: {hudsonj, denzinge}@cpsc.ucalgary.ca

Holger Kasinger, Bernhard Bauer
Department of Computer Science, University of Augsburg
Augsburg, Germany
Email: {kasinger, bauer}@informatik.uni-augsburg.de

***Abstract*—Adding self-adaptation as a property to systems aims at improving the efficiency of this system. But there is always the possibility for adaptations going too far, which is very detrimental to the trust of users into a self-adapting system. In this paper, we present a method for testing the efficiency of a self-adapting system, more precisely the potential for inefficiencies after adaptation has taken place. Our approach is based on learning sequences of events that set the system up so that a second following learned sequence of events is reacted to very inefficiently by the system. We used this approach to evaluate a self-adapting system for solving dynamic pickup and delivery problems and our experiments show that the potential inefficiencies due to self-adaptation are smaller than the inefficiencies that the non-adapting base variant of the system is creating.**

***Keywords*-testing; learning; dynamic optimization**

I. Introduction

Quality and thoroughness of testing are important factors for the trust of users into any kind of system, be it physical systems like power plants or cars or pure software systems. Naturally, what constitutes thorough and high quality testing depends very much on the system that is tested and the properties it is tested for. While testing "conventional" systems for their intended behavior essentially boils down to having the time to go through all these behaviors, testing them for "negative" properties, like under no circumstances showing a certain behavior, already is a difficult task and the quality of the testing heavily depends on the human tester and his abilities. For self-adapting systems, this becomes an even more difficult task, since unwanted or bad behavior might only emerge after a series of adaptations of the system. And if such a system consists of several interacting components (*agents*), then a tester faces additionally the danger of unwanted emergent behavior, also called *emergent misbehavior* (see [6]). Expecting every human tester to find crucial problems with self-adapting systems with the potential for emergent behavior is a highly doubtful assumption and does not instill trust in such systems in general.

In this paper, we present an automated approach to test a self-adapting, self-organizing multi-agent system that solves dynamic pickup and delivery problems. A key property for such a system is the efficiency of the delivery behavior and, with regard to its self-adaptation, the potential for (temporary) loss of efficiency due to self-adaptation and change of the environment. Our testing approach is based on learning sequences of events for the tested system to encounter and react to. More precisely, we extend the evolutionary learning approach of [2] to search for two sequences of events that represent tasks and when they are announced to the tested system. One sequence, the set-up, is aimed at having the system adapt itself to it by exposing the system repeatedly to this sequence. After the system is optimally adapted, the break sequence is given to it and the aim of our learning test system is to find two such sequences so that the efficiency achieved by the tested system for the break sequence is much worse after adaptation than without adaptation, providing users with a practical example of how bad a temporary loss of efficiency can get.

We applied our approach to the self-adapting improvement of a system for dynamic pickup and delivery problems based on digital infochemical coordination (see [4]). The self-adaptation is achieved by a so-called advisor that identifies recurring task sequences that are not well handled by the base system, determines how the tasks should be handled and creates exception rules for the transportation agents that achieve the intended solution (see [9]). We used our testing approach also to find problem instances for which the base system of [4] is not very efficient. Our test system found event sequences where the base system's efficiency was on average 3.5 times worse than the optimal solution, whereas the self-adapting variant was only two times worse for the break sequence than without self-adaptation. This is, in our opinion, a rather good result for the self-adapting variant, since for many instances this variant improves the behavior substantially (as documented in [4]).

After this introduction, in Section II we present the general idea of learning of event sequences for testing. Section III introduces our application area and Section IV the system to be tested. Section V instantiates the general idea for the system and Section VI reports on our experimental evaluation. After a view on the related work in Section VII, Section VIII concludes with some ideas for future work.

II. Testing using Learning of Event Sequences

In this section, we present our general scheme for testing a self-adapting system for potential loss of efficiency due to self-adaptation using learning of event sequences. While

 ISBN: 978-1-61208-109-0

a self-adapting system naturally does not have to consist of several agents, the system we present in Section IV for instantiating our approach does, so that we assume that our self-adapting system to be tested is a set $A_{tested} = \{Ag_{tested,1}, ..., Ag_{tested,m}\}$ of agents. Our system A_{tested} will work within an environment Env. There might be other systems acting in Env, either single agents or groups, that interact with A_{tested} and the environment itself might also change. We see the actions of other systems as well as all environmental changes as events that may or may not influence A_{tested}. In order to allow for events caused by other systems, for our testing we add a new group of agents $A_{evgen} = \{Ag_{evgen,1}, ..., Ag_{evgen,n}\}$ that are controlled by a learner and that are generating events in the environment for A_{tested} to react to. This general setting is depicted in Figure 1.

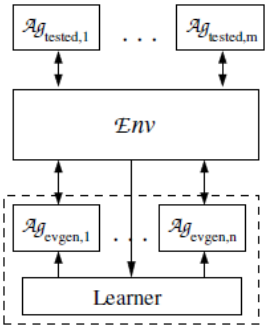

Figure 1. General setting of our approach

Formally, each agent $Ag_{evgen,i}$ creates a sequence of events $((ev_1^i, t_1^i), (ev_2^i, t_2^i), ..., (ev_l^i, t_l^i))$, which along with the reaction of the agents of A_{tested} produce a sequence of environmental states $e_0, e_1, ..., e_x$. This sequence of environmental states is utilized by the machine learner to evaluate the associated sequence of events to find better event sequences.

While several different machine learning techniques can be used to learn event sequences, we will use an evolutionary learning approach as suggested in [2]. Starting with a set of randomly created event sequences for each $Ag_{evgen,i}$, each element (*individual*) is evaluated by running the events in the environment and analyzing the resulting sequence of environmental states using a *fitness function*. Then the best individuals are used to create new individuals using *genetic operators*. The best individuals together with the new ones form a new set (generation) and this process is repeated for a given number of rounds.

III. Dynamic Pickup and Delivery Problems

The general pickup and delivery problem (PDP, see [7]) is a well-known problem class. Many of its instantiations require solving dynamic instances of this problem and many of those have many instances that allow for a self-adapting system to improve efficiency. There are also several variants of the PDP, for example PDP with time windows, that add additional constraints to the problem. While our testing method can be used for systems for all of these variants, in our experiments we used a system solving the following variant.

As the name suggests, a PDP consists of a sequence of pickup and delivery tasks and in a dynamic PDP these tasks are announced not all at the beginning but over a period of time. The tasks are performed by one or several transportation agents. The variant of PDP we are interested in requires repeatedly solving such sequences of tasks, where a sequence is called a *run instance*, and, in order to have a chance for self-adaptation, we require that run instances have at least several recurring tasks.

More formally, a run instance for a dynamic PDP is a sequence of task-time pairs $((ta_1, t_1), (ta_2, t_2), ..., (ta_k, t_k))$, where each ta_i consists of a pickup location $l_{pick,i}$, a delivery location $l_{del,i}$, and a required capacity $ncap_i$ and $t_i \in Time$, $t_i \leq t_{i+1}$, with $Time$ being the time interval in which the run instance is to be performed. t_i is the time at which ta_i is announced to the transportation agents. A transportation agent Ag has a transport capacity cap_{Ag} and has to perform both the pickup and the subsequent delivery to complete a task.

The solution sol produced by a set of transportation agents $\{Ag_1,...,Ag_m\}$ is represented as

$$sol = ((ta'_1, Ag'_1, t'_1), (ta'_2, Ag'_2, t'_2), ..., (ta'_k, Ag'_k, t'_k))$$

where $ta'_i \in \{ta_1, ..., ta_k\}$, $ta'_i \neq ta'_j$ for all $i \neq j$, $Ag'_i \in \{Ag_1, ..., Ag_m\}$, $t'_i \leq t'_{i+1}$, $t'_i \in Time$. A tuple (ta'_i, Ag'_i, t'_i) means that the pickup of ta'_i was done by Ag'_i at time t'_i.

There are many possibilities how to measure the efficiency $eff(sol)$ of the agents when producing sol. Examples are distance covered by the agents, time needed to fulfill all tasks, balance among the agents and many more, especially all kinds of combinations. For our variant, we use as efficiency measure the total distance traveled by all agents. There are also many different environments in which agents can work on the tasks. In our variant, we use a simple grid, where each node represents a possible location.

IV. Solving PDP using Digital Infochemicals and an Advisor

Solving all kinds of dynamic PDP variants is rather difficult, simply because of the dynamic nature that requires to make decisions without really knowing what additional tasks might come up later. Usually, the time frames for a run instance do not allow to create an optimal plan every time a new task is announced (the static PDP problem is already in NP for most variants). As a consequence, many of the systems used to solve dynamic problems mostly ignore the efficiency aspect and concentrate on other useful system

 ISBN: 978-1-61208-109-0

properties, like robustness against failure, graceful degradation, easy extendibility and general openness of the system. Digital infochemical coordination (see [4]) is a coordination concept for self-organizing multi-agent systems that shows these properties and it has been used as the coordination concept for a system solving the variant of PDP presented in the last section. Essentially, each location in each task is represented by an agent emitting infochemicals that lead the transportation agents to them when a task is announced. The transportation agents also emit infochemicals to coordinate and avoid having all of them moving to the same pickup location. If a location agent has been served, it, again, uses specific infochemicals to inform the transports that they are not needed by it anymore. Due to lack of space and because a deeper understanding of this base system is not necessary for understanding our instantiation of the testing idea from Section II, we will not go into any more detail of this base system. The interested reader should consult [4].

From a practical application perspective, efficiency of a pickup and delivery system cannot be ignored! Therefore, [9] extended the system from [4] to improve efficiency using an additional agent, an *efficiency improvement advisor* Ag_{EIA}, that essentially adds to the base system the ability to reflect on the behavior shown by the transportation agents and to adapt this behavior after having seen really inefficient behavior that is likely to happen again in future run instances. Since Ag_{EIA} is part of the system, the system as a whole is self-adapting.

The advisor works as follows: when the agents have dealt with a run instance, they go back to their depot at which the advisor is located and report their local history to it. The advisor uses this data to compile a global history for the whole system over a given number of run instances. Using this global history, Ag_{EIA} identifies a sequence $(ta_1^{rec}, ..., ta_l^{rec})$ of recurring tasks by clustering all tasks from the run instances using Sequential Leader Clustering (see [3]). Every cluster with a size slightly smaller or equal to the number of run instances indicates a recurring task. The clustering makes use of a similarity measure for tasks, but for our testing this measure is not of interest. Then Ag_{EIA} computes an optimal (or at least very good) solution for the sequence of recurring tasks according to the efficiency measure eff using an optimization algorithm for the static variant of our PDP (in our case, we used a genetic algorithm that uses or-trees as basis for the genetic operators). If the optimal solution is much better than what the agents generated in the last run instance, Ag_{EIA} starts creating advice for the agents.

The advice consists of so-called *ignore rules* that are iteratively produced by Ag_{EIA}. It compares the produced solution with the optimal one and identifies the first position where the two solutions are not identical (with regard to task or agent). It then creates an ignore rule for the agent that performed the task in the produced solution that essentially has as condition the task (its pickup location and the needed capacity) and as action the advice to the agent to *not* do this task. Since the base system is self-organizing, another agent will pick up the task now ignored by the previous (and from the perspective of efficiency wrong) agent. After an ignore rule was given to an agent, the advisor waits until after the next run instance to see if now the produced solution is near enough to the efficiency of the optimal one. If not, the production step for an ignore rule is repeated.

By repeating the clustering every time new data is available allows for dealing with a change of recurring tasks over time. Theoretically a big change of tasks between two run instances could result in a big (temporary) loss of efficiency for the second run instance. Getting some practical idea how bad this potential loss can get can improve a user's trust into the system dramatically, which is what our testing system in the next section is aimed at producing.

V. Instantiating our Testing Approach

There are several potential sources for a system as described in the last section to produce rather inefficient solutions to run instances and thus several sources for distrust in the system. We are interested in the effects of the advisor and especially the potential for overadaptation and the consequent loss of efficiency when the system encounters a run instance with a unique set of tasks after adapting to a different recurring set of tasks.

When instantiating the general idea from Section II to create a test system for the self-adapting system presented in Section IV, we followed the usual approach of human testers to concentrate on the test goal and to eliminate influences that are not in the test goal. So, in order to speed up adaptation in our test system, there are no non-recurring tasks in the setup run instance. The test system concentrates of finding a setup sequence of tasks (events), that the systems repeatedly encounters and adapts to, along with a second sequence, which is encountered after the setup adaptations. And the test goal is to find such a pair of sequences for which the efficiency for the second sequence after adaptation is much worse than without adaptation taking place. The transportation agents (and the advisor) form the set A_{tested}. In [4] and [9], agents at the appropriate locations for a task announce the tasks to the tested agents, so that we have indeed a set of event generating agents. But for the number of tasks we use in our experiments, we have a lot of locations that are not used by them. Therefore we use run instances in our individuals and create them centrally in our learner.

More precisely, an individual of our evolutionary learner is a pair (es_{setup}, es_{break}), where each of the two run instances is a sequence of task-time pairs. As evolutionary operators we use the usual single point mutation and crossover on lists by first selecting a position o in either es_{setup} or es_{break}. A mutation then takes the run instance $((ta_1, t_1), ..., (ta_l, t_l))$ and creates

$$((ta_1, t_1), ..., (ta_o^{new}, t_o^{new}), ..., (ta_l, t_l))$$

where either ta_o^{new} is a different task than ta_o or $t_o^{new} \neq t_o$.

Crossover takes two run instances $((ta_1^1, t_1^1), ..., (ta_l^1, t_l^1))$ and $((ta_1^2, t_1^2), ..., (ta_l^2, t_l^2))$, and creates a new instance

$$((ta_1^{new}, t_1^{new}), ..., (ta_l^{new}, t_l^{new}))$$

where each (ta_i^{new}, t_i^{new}) is either equal to (ta_i^1, t_i^1) or to (ta_i^2, t_i^2). Then the new individual copies the not selected run instance from the first parent pair and adds the newly created run instance as the other element of the new pair.

We also added a so-called targeted operator, which is a twin point mutation operator that attempts to create similar tasks between the two run instances of the individual to allow the ignore rules created for adaptation to the setup instance to negatively impact the break instance. This is achieved by aligning a task-time pair between the instances. If the individual (es_{setup}, es_{break}) is

$$(((ta_{1,1}, t_{1,1}), ..., (ta_{1,l}, t_{1,l})), ((ta_{2,1}, t_{2,1}), ..., (ta_{2,l}, t_{2,l})))$$

then the targeted twin point mutation operator selects a position i and creates the new individual

$$(((ta_{1,1}, t_{1,1}), ..., (ta_{1,i}^{new}, t_{1,i}^{new}), ..., (ta_{1,l}, t_{1,l})),$$
$$((ta_{2,1}, t_{2,1}), ..., (ta_{2,i}^{new}, t_{2,i}^{new}), ..., (ta_{2,l}, t_{2,l})))$$

where $ta_{1,i}^{new}$ and $ta_{2,i}^{new}$ are identical tasks and $t_{1,i}^{new} = t + \epsilon_1$ and $t_{2,i}^{new} = t + \epsilon_2$ for randomly chosen $t \in Time$ and small numbers ϵ_1 and ϵ_2.

While an obvious fitness measure for an individual (es_{setup}, es_{break}) would be to simply compute the difference in efficiency between the created solution for es_{break} with adaptation ($eff(sol_{+ad}(es_{break}))$) and without adaptation (($eff(sol_{-ad}(es_{break}))$), our initial experiments showed that the learner needed some more "advice" to create the individuals we wanted quickly. More precisely, the learner had problems due to many early individuals where no adaptation took place and due to individuals where $sol_{+ad}(es_{break})$ was too near to the optimal solution. Both types of individuals clearly are not of interest for our test goal and therefore we created a fitness measure punishing them:

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

with

$$pract(es_{break}) = max[(eff(sol_{+ad}(es_{break})) - eff(sol_{-ad}(es_{break}))) * w_{pract}, 0]$$
$$theo(es_{break}) = max[(eff(sol_{+ad}(es_{break})) - eff(sol_{opt}(es_{break}))) * w_{theo}, 0]$$
$$adapt(es_{setup}) = max[(eff(sol_{-ad}(es_{setup})) - eff(sol_{+ad}(es_{setup}))) * w_{adapt}, 0]$$

and $sol_{opt}(es_{break})$ being the optimal solution for the break run instance.

We account for the difference of the solution quality produced for es_{break} before and after adaption weighted by a parameter w_{pract}. We also take account of the difference between the emergent solution for es_{break} after adaptation and the optimal solution weighted by a parameter w_{theo} measuring how far the solution is from the theoretical optimum, along with taking account of the difference between the solution produced for es_{setup} before and after advice, using w_{adapt} as parameter for its importance.

The learning testing system for the self-adapting system for PDP as described above can be easily modified for other testing goals around efficiency of a system. For example, the efficiency of the underlying self-organizing base system can be tested using only one run instance as individual (essentially just having es_{break}), not using the targeted twin point mutation operator and using as a fitness function

$$fit_{base}(es_{break}) = \frac{eff(sol_{-ad}(es_{break}))}{eff(sol_{opt}(es_{break}))}.$$

In the next section, we will not only report on our experiments testing the self-adapting system for PDP, we will also provide results on how bad the efficiency of the base system can be compared to an optimal solution, to put our results for the self-adapting system into perspective.

VI. Experimental Results

In this section, we describe several experiments performed using our test system to evaluate the self-adapting system for PDP from Section IV. First we present the general settings for the experiments and then the results of our test system for the settings. To put these results in perspective, we also used the variant of the test system described in the last section for the base system for PDP.

A. Experimental Settings

Each experiment used a 10×10 grid with a depot in the middle. We used the settings for the various parameters of the base system and advised variant reported in [9]. Our testing used the following percentages for a new population to be created by the evolutionary operators: 10 percent best survived, 30 percent generated using crossover and 60 percent via mutation, 30 for each kind of mutation.

Every experimental series consisted of 5 runs of the testing system due to the random effects of the evolutionary learning process. In all experiments we used two transportation agents, Ag_1 and Ag_2, and 2, 4, 6, 8 and 10 tasks. This limitation to two agents is because within the chosen range of task sizes the addition of other agents was unnecessary so that also self-adaptation would not be necessary. Also, from a testing perspective we are interested in small examples that show a problem. $Time$ was an integer interval of $[0, 50]$ for 2 tasks, $[0, 100]$ for 4 tasks, $[0, 150]$ for 6 tasks, $[0, 200]$ for 8 tasks and $[0, 250]$ for 10 tasks. For each number of tasks we performed one run to get an idea after what generation no improvement seemed to occur anymore and we used this number to limit the run length of the other test runs. For the provided results we indicate both the average efficiency loss and the maximal loss among the 5 runs.

 ISBN: 978-1-61208-109-0

B. Quantitative Results

Table I summarizes our experimental results for testing the self-adapting system for the PDP as set up in the last subsection. For the fitness function we used weights of $w_{pract} = 25$, $w_{adapt} = 5$, and $w_{theo} = 1$. This means that the primary component is how much worse the break run instance is solved after adaptation, compared to the efficiency the system shows for the break instance without the adaptation. The other components are there to make sure that the setup instance really leads to an adaptation and that the break instances are solved badly, as described in the last section.

Tasks	Generations	Average	Maximum
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

Table I
EFFICIENCY LOSS RESULTS FOR SELF-ADAPTING SYSTEM

Table I shows the average efficiency loss due to the adaptation is around a factor of 2 and the worst found examples are also very near to that factor[1]. 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 result is only two times worse than without the advisor. With regard to the efficiency of our test system itself, the 5 runs accounted for in the 10 task entry of the table took 19.4 hours to complete.

Before we look more closely at one of the runs from Table I to see what causes the loss of efficiency, we will first look at the results of our testing system when modified to evaluate the efficiency of the base self-organizing system. Table II presents the results of our test system for the same numbers of tasks and the same grid setting as for Table I. As can be seen, the worst event sequences found by our test system are clearly worse than the efficiency loss potential found for the self-adapting system. One of the goals of the particular self-adapting system we are testing was to keep the strengths of the base system, especially the self-organization ability, so that big changes in events from one run instance to the next would have no big impact. Tables I and II show that this goal was indeed achieved.

C. A "bad" problem instance

Figures 2 and 3 visualize one of the examples with 4 tasks. The setup run instance is

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

[1]As already stated, the efficiency loss is computed as the efficiency of the solution for es_{break} after adaptation divided by the efficiency of the base system without adaptation for es_{break}.

Tasks	Generations	Average	Maximum
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

Table II
EFFICIENCY PROBLEMS OF BASE SYSTEM (fit_{base})

and the break instance is

((D,E),18),((F,I),38),((B,D),55),((J,F),78).

Without the advisor, the system solves the setup instance by having Ag_1 (indicated by the dashed lines) fulfill task ((F,K),19), starting to respond to task ((C,G),74) but then switching to task ((J,A),75). Ag_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 Ag_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 2 shows, this does not result in a perfect solution after the advice, since Ag_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). This shows a limitation of the existing ignore exception rules provided by the advisor since it cannot enforce a time for the agents to complete tasks in.

Without having the advisor, the break run instance is solved by the system by having Ag_1 fulfill (((D,E),18), then ((F,I),38) and then ((J,F),78). Ag_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. After having adapted to the setup instance, the system solves the break instance in the following manner. Ag_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),78) due to the other exception rule. This means that Ag_2 fulfills first ((F,I),38), then does ((B,D),55), and then performs ((J,F),78).

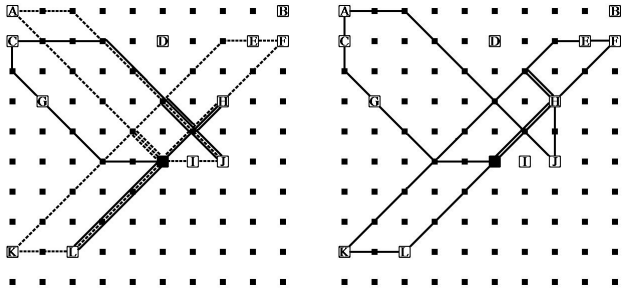

Figure 2. Setup without Advice (Right) vs. Setup with Advice (Left)

This example shows the importance of having targeted operators to bring knowledge about the tested system into the

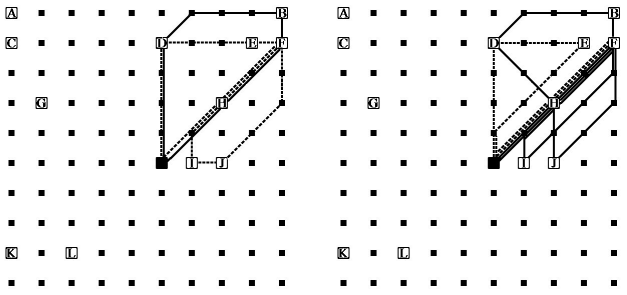

Figure 3. Break without Advice (Right) vs. Break with Advice (Left)

learning process. The twin point mutation operator connects tasks in the setup and the break instance to create events that trigger the exception rules, which may then result in unwanted behavior. These unintended consequences are the danger of an advisor and our test system gives us an idea of the potential inefficiencies produced.

VII. Related work

The use of learning/self-adapting systems to test systems for certain properties/test goals has become a very active research area over the past years, although not under the terms "learning" or "self-adaptation" (many use "search based" instead). Many of the evolutionary approaches for testing reported at [8] are, in fact, learning systems. For example, [1] evolves a schedule of given events for a scheduler (resp. an executable model of it) with the goal to find times for the announcement of the given events that lead to infeasible schedules. [1] comes nearest to our approach, but the tasks have to be given, not learned, and the tested system is a single agent, not a group.

With regard to testing self-adapting or even just self-organizing systems, this topic has not drawn a lot of attention, so far. Naturally, for testing the wanted behavior of such systems, standard testing methods can be and are used. But the added difficulty to "negative" testing has not been the focus of research beyond the already mentioned works.

VIII. Conclusion and Future Work

We presented a method to test self-adapting systems for adaptations that result in a loss of efficiency and for the tasks that are solved less efficiently. By learning sequences of events that set up the system so that a follow-up sequence is badly solved, our method provides users of self-adapting systems with an idea what can go wrong and allows them to evaluate this risk compared to the gains the self-adapting system is providing. This aims at increasing the trust into the system. Naturally, the developers of the system also can use the found sequences to improve their system.

We used a system based on our method to test a self-adapting, self-organizing multi-agent system for one variant of dynamic pickup and delivery problems. Our experiments showed that while our system was only able to find event sequences for the self-adapting system that were around 2 times worse than what the system without the self-adaptation would have achieved, for this base system without adaptation a variant of our test system was able to find on average event sequences that it solved around 3.5 times worse than would be the optimum solution. In our opinion, this strengthens the claim of the self-adapting system developers that their adaptation approach is very targeted to situations in which the base system is bad and represents only a minimal intrusion into the base mechanism.

Naturally, like all testing, our method cannot guarantee to find the worst event sequence there is for the tested system. But, compared to a human tester, it has no bad days and works always on a consistent level, especially if a test consists of several runs of the system as in our experiments. And our system does not retire and leave a company with novice testers.

There are several directions for future research. The developers of the self-adapting system we used have ideas for additional types of exception rules that are more invasive into the base system than the ignore rules (see [5]). Testing these new rules should provide an idea if this increases the potential for inefficiency. Along the development and integration of these new exception rules, our test system can also be used to realize some kind of test-driven development for self-adapting systems. Finally, we plan to use our method to develop test systems for other self-adapting systems for other applications.

References

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

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

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

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

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

[6] J.C. Mogul: Emergent (mis)behavior vs. complex software systems, SIGOPS Operating Systems Review 40(4), 2006, pp. 293–304.

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

[8] SEBASE: Software Engineering By Automated SEarch Repository, http://www.sebase.org/sbse/publications/, as seen on Jun. 18, 2010.

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

 ISBN: 978-1-61208-109-0