A genetic algorithm for self-optimization in safety-critical resource-flow systems

Florian Siefert, Florian Nafz, Hella Seebach, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Siefert, Florian, Florian Nafz, Hella Seebach, and Wolfgang Reif. 2011. "A genetic algorithm for self-optimization in safety-critical resource-flow systems." In *2011 IEEE Workshop on Evolving and Adaptive Intelligent Systems (EAIS), 11-15 April 2011, Paris, France*, edited by Plamen Angelov, Dimitar Filev, and Nikola Kasabov, 77–84. Piscataway, NJ: IEEE. https://doi.org/10.1109/eais.2011.5945915.

A Genetic Algorithm for Self-Optimization in Safety-Critical Resource-Flow Systems

Florian Siefert, Florian Nafz, Hella Seebach, Wolfgang Reif
Institute for Software & Systems Engineering
Augsburg University, Germany
E-Mail: {siefert, nafz, seebach, reif}@informatik.uni-augsburg.de

***Abstract*—Organic Computing tries to tackle the rising complexity of systems by developing mechanisms and techniques that allow a system to self-organize and possess life-like behavior. The introduction of self-x properties also brings uncertainty and makes the systems unpredictable. Therefore, these systems are hardly used in safety-critical domains and their acceptance is low. If those systems should also profit from the benefits of self-x properties, behavioral guarantees must be provided. In this paper, a genetic algorithm for the self-optimization of resource-flow systems is presented. Further, its integration into an architecture which allows to provide behavioral guarantees is shown.**

I. Introduction

During the last decades the complexity and requirements of industrial applications, like automated production processes, are steadily increasing. Nevertheless, many technical systems are tailored very rigidly to the originally intended behavior and the specific environment they will work in. If not foreseen during design time, these systems can hardly react to failures and changes in the environment. Organic Computing (OC) [1] is trying to tackle these challenging aspects. The idea is to build systems that can autonomously adapt to a changing environment and optimize themselves to the current situation at runtime. The benefits of those systems are that they can compensate failures or provide a better performance compared to conventional systems. These abilities are often referred to as self-organizing, self-optimizing or self-x properties in general.

However, especially in safety-critical domains, as, for example, production automation and avionics, one wants to have behavioral guarantees, despite uncertainty of self-organization. Here, the challenge is to allow the system to adapt itself, but still to be able to guarantee correct behavior.

A large class of industrial systems is the class of resource-flow systems. In resource-flow systems, agents handle resources by receiving them from another agent, processing them according to a given task, and handing them over to another agent that performs further steps of the task. An instance of this are flexible manufacturing systems or logistic systems. A self-organizing resource-flow system is a system that finds the routes and assigns the different production steps on its own and further is able to self-organize in case of a failure or changing requirements to continue working.

Previous work included design and construction of self-organizing resource-flow systems [2]. A separation of self-x and functional behavior was proposed. This allows to specify behavioral corridors by constraints and to give guarantees. The problem that should be solved by self-organization was therefore specified as a constraint satisfaction problem (CSP) [3]. It constrains the possible configurations of the system to correct ones, which lead to the wanted behavior. A constraint solver [4] was used to calculate new valid configurations [5]. This approach allows to give guarantees as only valid configurations are forwarded to the functional system. Nevertheless, in a CSP, all solutions are treated equally, and the solver just returns one solution which fulfills the constraints no matter how good it is. But usually some configurations are better than others. They need less resources or have a higher performance, for example. Therefore, it is interesting to find not only one but an optimal or at least a "good" configuration. Further, constraint solvers are usually slow, due to the fact that they are systematically and exhaustively exploring the solution space. In this paper, a genetic algorithm is presented which tackles these issues and allows to find optimal solutions.

The paper is structured as follows: Sect. II gives a short overview of the class of self-organizing resource-flow systems, their design, and their ability to self-optimize. Moreover, a short introduction to genetic algorithms is given. Afterwards, Sect. III presents the model of the genetic algorithm that is used to find new configurations. In Sect. IV the genetic algorithm is evaluated and some results are presented. Finally, Sect. V concludes the paper.

II. Self-Optimizing Resource-Flow Systems

In this section, an introduction to the class of self-organizing resource-flow systems is given. First, an architectural view on these systems and an extension is presented, which allows the use of heuristic or even incorrect self-reconfiguration algorithms. The components of a resource-flow system are shortly described with the help of a design pattern afterwards. Then the reconfiguration and optimization problem is illustrated, and the use of genetic algorithms for an optimized reconfiguration is explained.

A. Architecture

Most OC systems, just as the class of resource-flow systems presented here, consist of two parts (see Fig. 1). One functional part providing the basic functionality, and another part which incorporates the organic intelligence, often in form of a observer/controller (o/c) component [6]. While the functional part of the system is working as a traditional system, the o/c

part is responsible for monitoring the system and, in case of a failure, for reconfiguring the system in such a way that it can continue working. This view allows the separation of the self-x behavior and the functional behavior which has several advantages for the analysis and verification of the system as now both parts can be treated separately.

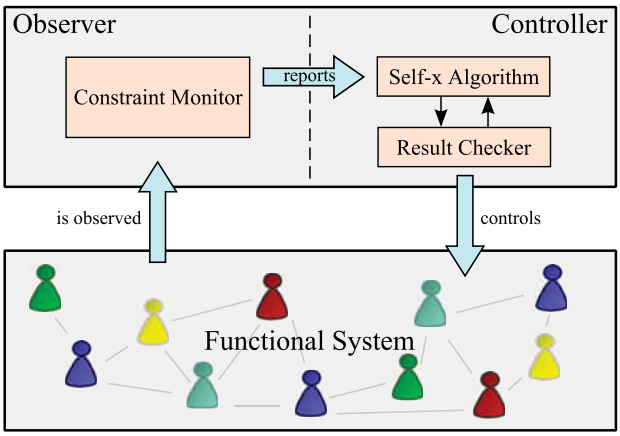

Fig. 1. Architecture of Organic Computing Systems

The idea is to specify a valid system configuration by constraints on the system variables of the functional part. Configurations not violating the constraints imply the intended behavior. This allows the definition of a behavioral corridor without the need to exactly define how the system should look like. The o/c is then monitoring these constraints, and, in case of a violation, it is reconfiguring the system such that the constraints hold again. This approach is called the *Restore Invariant Approach*, and is described in detail in [7].

To ensure correct behavior of the system, one must ensure that the result of the self-reconfiguration algorithm is correct. Nevertheless, in OC systems often genetic algorithms [8], learning classifier systems [9], or other learning techniques (e.g., neural networks) are used for realization of self-x features. Those do not necessarily return valid and correct results. Therefore, we added a result checker (RC) component, which is integrated into the controller. Its input is the result of the self-x algorithm, i.e., the configuration the o/c wants to forward to the system. The RC then checks if this is in-line with the defined constraints, and in case it is, relays it to the system. In case it is not, the configuration is rejected. This allows the use of arbitrary self-x mechanisms, even incorrect ones. Another advantage is that only the RC has to be verified to guarantee correctness, which is usually less complex compared to the verification of the complete self-x algorithm.

B. Organic Design Pattern

The components of resource-flow systems can be described by a pattern called Organic Design Pattern (ODP) as depicted in Fig. 2.

Agents are the main components in these systems, processing *resources* according to a given *task*. Every *agent* has several *capabilities*, divided into producing, processing, and consuming *capabilities* (*produce*, *process*, and *consume*). Consequently, the *task* is a sequence of *capabilities* beginning with a producing *capability* and ending with a consuming *capability*. Furthermore, the *agent* knows a couple of *agents* it can interact with and hand over *resources*. This is encapsulated in the *inputs* and *outputs* relation. The *role* concept is introduced to define correct resource-flows through the system. This means an *agent* has *roles* allocated telling it from which *agent* it receives the *resource* (*precondition/port*), which *capabilities* to apply, and then to which *agent* to hand over the *resource* (*postcondition/port*). Thus, the *roles* establish the connections between the *agents* and the combination of all *roles* forms the resource-flow. For more details on software engineering and modeling of self-organizing resource-flow systems and the design pattern see [10].

C. Self-Optimization

The OCL constraints in Fig. 2 specify valid allocations of *roles* to *agents* which leads to correct system behavior. They define the behavioral corridor. Each role allocation which fulfills the constraints is a valid one and leads to correct processing of resources. Typical constraints for the class of resource-flow systems are, for example, “only capabilities are assigned via roles that are available at that agent” or “all needed capabilities (to fulfill the task) must be at least assigned once”. More constraints can be found in [5], where the reconfiguration problem is formalized as a constraint satisfaction problem (CSP), which then can be solved by a constraint solver. Nevertheless, standard constraint solvers usually return the first solution they found, no matter how good it is as all solutions are equally good. This is sufficient for functional correctness, but usually one wants to find optimal solutions, which basically involves comparing all solutions of the CSP according to a given cost function f. In resource-flow systems, for instance, the o/c should reconfigure in a way that the load is balanced between the agents and the throughput is maximized, i.e., that a minimum of roles are assigned to the agents, and a minimum of capabilities need to be applied within one role. This leads to a constraint satisfaction optimization problem (CSOP) as defined by Tsang in [3]. A CSOP is basically a CSP together with an optimization function f which maps every solution to a numerical value. The task is to find a solution with the optimal f-value. Two important techniques for tackling CSOPs are branch and bound algorithms [11] which use heuristics to prune the search space, and genetic algorithms with a stochastic approach [12]. The latter one is used here for the implementation of the self-x algorithm in the o/c to allow an optimized reconfiguration of the resource-flow system.

D. Genetic Algorithms

Before the genetic algorithm for self-organizing resource-flow systems is explained in detail, this section gives a short overview of genetic algorithms. For more details see, for example, [8].

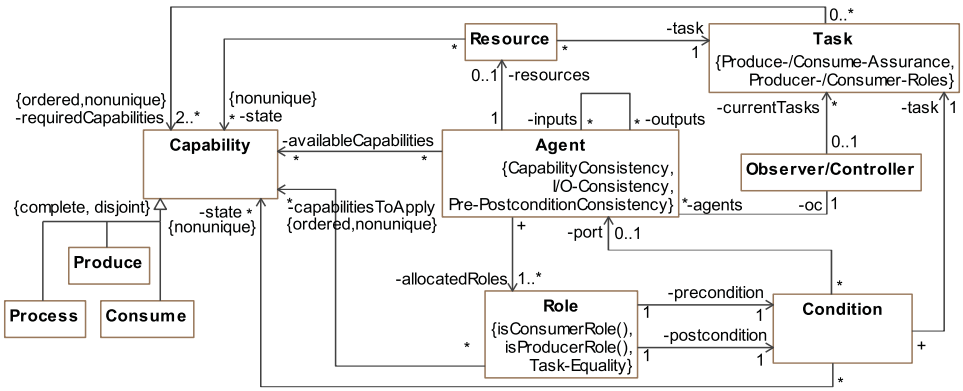

Fig. 2. Components of Resource-Flow Systems

Genetic algorithms are used to find solutions in search or optimization problems. The aim of genetic algorithms is to find a solution for the given problem. Genetic algorithms are inspired by natural evolution and try to imitate it.

The basic concept of a genetic algorithm are the *individuals*. An individual represents a solution for the given problem, and several individuals form the *population* of a *generation*. Each individual has a *fitness*, characterizing how good the individual solves the problem. The fitter the individual, the better the solution. The individual's fitness is crucial for its survival and reproduction, since fitter individuals are more likely to be chosen to breed new offspring ("survival of the fittest"). After offspring was created in the *crossover*, every offspring is *mutated* to a certain mutation probability. While the offspring forms the population of the next generation, often supplemented by (the best) individuals of the former generation, the former generation "dies".

This whole process is repeated several times. Usually, the algorithm terminates if it reached a predefined number of generations or found a solution that is good enough.

III. Model of the Genetic Algorithm

This section shows the application of a genetic algorithm to the self-organizing resource-flow systems presented in Sect. II to allow self-optimization for these systems. After the system monitors a violation of a constraint and the task cannot be fulfilled anymore, it triggers a reconfiguration to restore the system's invariants again. In order to be able to use the genetic algorithm to solve the reconfiguration problem, the system has to tell the algorithm the current system state, i.e., the current task, all agents, and every agent's inputs, outputs, and available capabilities.

There are two possibilities for a genetic algorithm to fill its population: either it can only allow correct solutions and throws away all incorrect solutions, or it can also allow incorrect solutions. Our algorithm allows incorrect solutions in the population, because a concentration on correct solutions would immensely restrict the search space of the algorithm. Furthermore, a lot of work would have to be done in designing crossover, mutation, etc., which then would rather be a systematic manipulation than a random mutation. When allowing incorrect solutions in the population, they have to be less fit than correct solutions, of course.

Because the genetic algorithm's operations have the aim to improve incorrect solutions, they are mostly not maintaining the correctness of a solution. However, correct solutions are not lost, since the best solutions are stored (more details later).

Next, we want to present the most important parts and operations in detail.

A. Individuals

Each individual of a genetic algorithm represents a solution for the search problem, here the reconfiguration problem. Most of the system's components are already fix and thus are no solution of the problem and not usable as individuals. These fix components are the task and the agents with their inputs, outputs, and available capabilities. Not being able to change the task and the agents, the change of single roles or the entire assignment of roles to agents are the only degrees of freedom we have. Choosing a single role as individual is rather awkward, since it is no solution for the reconfiguration problem and the fitness of a single role cannot be judged: a single role might be correct itself and might not violate a role-bounded constraint, but could be incorrect within the interaction of all other roles in the system, which would violate constraints that monitor the interaction of roles. Thus, only the assignment of roles to all agents, a *role allocation*, comes into consideration as individual of our algorithm. A role allocation is a list of agents with each agent knowing its inputs, outputs, capabilities, and its allocated roles. The aim of the algorithm is to find a correct role allocation, i.e., a correct allocation of roles to each agent in the system that fulfills all constraints and satisfies the system's task.

Let us introduce a formal definition of role allocations. Let RA be the set of all possible role allocations. Then the role allocation $ra \in RA$ is defined as follows:

$$ra := (ra_{a_1}, \ldots, ra_{a_n})$$

with being ra_{a_i} the agent a_i ($i \in \{1, \ldots, n\}$) in the role allocation ra, who has knowledge about its assigned roles $r_1, \ldots, r_j$:

$$ra_{a_i} := (a_i, \{r_1, \ldots, r_j\})$$

Each role allocation has the same ordered list of agents $a_1, \ldots, a_n$.

A role has several attributes (see Sect. II-B):

$$r := (\ (task, state, port),\\ capabilitiesToApply,\\ (task', state', port')\ )$$

The first tuple is the role's precondition, the second tuple is the role's postcondition. A condition always contains the task, the resource's current state, and the port, i.e., where to get the resource from or where to give the resource to next.

Every role knows about the system's task, $task = task' = [t_1, \ldots, t_z]$[1]. The role's precondition defines where the resource comes from ($port$) and what state it then has, denoted by $state = [s_1, \ldots, s_p]$, $0 \leq p \leq z$. If $p = 0$, the state would be empty. Furthermore, a role determines which capabilities have to be applied to the given resource. This is done in $capabilitiesToApply = [c_1, \ldots, c_q]$, $0 \leq q \leq z$. If $q = 0$, no capability is applied. The postcondition's state informs about the resource's state after the application of the role's capabilities to be applied, i.e., $state' = [s_1, \ldots, s_p, c_1, \ldots, c_q]$, $0 \leq p + q \leq z$. Finally, the postcondition tells where the resource is handed next ($port'$).

In a role that violates no role-bounded constraints $state$ is a prefix of $task$ ($state \sqsubseteq task$), as well as $state' \sqsubseteq task'$. Moreover, $state + capabilitiesToApply = state'$, with "+" meaning the concatenation of lists.

B. Fitness of Individuals

In order to measure the quality of the algorithm's individuals, i.e., the quality of its found solutions for the reconfiguration problem, we have to introduce a fitness function that assigns a fitness value to each individual – the fitter an individual the better the solution. The fitness of an individual directly influences its chance to reproduce and survive, as can be seen later on in Sect. III-E.

In [3], E. Tsang suggests to tackle the application of genetic algorithms to tightly constrained CSOPs, as is the reconfiguration problem of resource-flow systems, with a penalty function. A penalty function assigns a penalty, i.e., a low fitness value, to individuals that violate constraints. Our fitness function uses the same principle.

When using a penalty function, there exist several alternatives to punish an incorrect role allocation. Either a role allocation that violates constraints is punished once, no matter how many constraints are violated; or it gets a punishment for every constraint that is violated by the role allocation, as used here. This option allows us to distinguish between really bad solutions that violate a lot of constraints, and almost correct solutions that violate only a few constraints. It is even possible for us (and actually realized that way) to use different heights of penalties for every constraint.

But we do not only want to find correct solutions; our aim is to find optimal solutions. Thus, correct but not optimal role allocations have to be slightly punished, too. In our case, an optimal role allocation is a role allocation, in which every agent does not have more than one allocated role and does not apply more than one capability in a role. A suboptimal role allocation slows down the processing of a resource, e.g., a role allocation with an agent that applies more than one capability. A role allocation with an agent that applies more than one capability takes more time than a load-balanced role allocation in which every agent applies one or no capability, since changing the agent's capabilities takes a lot of time. The penalty, however, must not be that hard like a violation of a constraint, so that correct but suboptimal role allocations have a better fitness than incorrect role allocations. Consequently, optimal (thus correct) role allocations do not get any penalties and, therefore, have the best fitness.

[1]Here, $[\ldots,\ldots]$ is to be understood as the notation of an ordered list of capabilities.

C. Crossover

In the crossover, offspring role allocations are bred by parent role allocations. Here, we use a simple *one-point-crossover* that randomly chooses a cutting point, divides each of the two parent role allocations in two parts at the (same) cutting point, and recombines the front part of the one parent with the rear part of the other parent and vice versa. This process is shown in Fig. 3, where two parent role allocations ra^k and ra^j (left side) are divided between agent a_i and agent a_{i+1}, and are recombined to two new offspring role allocations (right side).

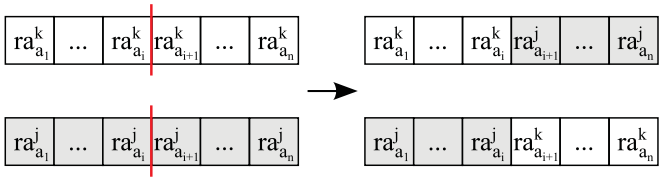

Fig. 3. Crossover of two Role Allocations

D. Mutation of Individuals

After crossover, the offspring is mutated, i.e., every role in a role allocation is mutated with a certain probability. This implies that every role in a role allocation might be mutated, but possibly no role in a role allocation is mutated, too.

As already explained, mutation modifies roles. This could be a single part of a role that is mutated, like a port, or the whole role. We have invented several mutation rules that mutate different parts of a role. If a role has to be mutated, one of the rules is chosen randomly and applied to the role. The rules are weighted differently, i.e., some rules are more likely to be chosen than other rules.

Each rule has the motivation to improve either the role itself or the whole role allocation. Thus, an incorrect role allocation might get repaired by applying a mutation rule. However, a role allocation might of course get worse after mutation took place. In the following, all mutation rules are introduced.

1) Mutate Ports: The first mutation rule we want to explain is the mutation of ports. As the name already suggests, the ports of a role are mutated, i.e., the information from what agent the resource comes from, and where it is handed afterwards. Previously, it is randomly decided whether to mutate only one of the two ports (and which one) or to mutate both ports. However, the assignment of new ports is not totally loose; actually, the ports are randomly chosen out of the agent's set of inputs or outputs, so that no "wrong" port is assigned that does not match the agent's inputs or outputs.

The idea of this rule is the possible improvement in the role allocation's resource-flow. Ports that did not fit together previously, e.g., agent a_i wanted to give the resource to agent a_j, but a_j expected to get the resource from another agent a_k, could match after the application of this mutation rule.

2) Delete Capability: This rule deletes the last capability of the capabilities to be applied and the last capability of the postcondition's state. If no capability is applied, no mutation is done.

So suboptimal roles, in which the agent has to apply two capabilities, get better, because the agent only has to apply one capability afterwards. The whole role allocation could also improve, since a task's step that is applied twice by mistake then possibly is only applied once.

The application of this rule is shown below:

$$\begin{aligned} capabilitiesToApply = [c_1, \ldots, c_q] &\overset{mutation}{\longrightarrow} \\ capabilitiesToApply = [c_1, \ldots, c_{q-1}], & \\ state' = [s_1, \ldots, s_p, c_1, \ldots, c_q] &\overset{mutation}{\longrightarrow} \\ state' = [s_1, \ldots, s_p, c_1, \ldots, c_{q-1}] & \end{aligned}$$

As you can see, the last capability c_q is deleted from $capabilitiesToApply$ and from $state'$.

3) Left Shift Capability: If this mutation rule is chosen, the first capability of the capabilities to be applied is moved to the end of the precondition's state. Again, no mutation is done if no capability is applied.

Just like in the rule above, this rule helps suboptimal roles getting better by reducing the applied capabilities by one, and helps incorrect role allocations getting correct.

$$\begin{aligned} state = [s_1, \ldots, s_p] &\overset{mutation}{\longrightarrow} \\ state = [s_1, \ldots, s_p, c_1], & \\ capabilitiesToApply = [c_1, \ldots, c_q] &\overset{mutation}{\longrightarrow} \\ capabilitiesToApply = [c_2, \ldots, c_q] & \end{aligned}$$

The above example presents the rule of left shifting a capability. The first element of $capabilitiesToApply$, c_1, is removed and put at the end of $state$.

4) Right Shift Capability: Analogously to the rule above, the last capability of the precondition's state can be shifted to the first position of the capabilities to be applied.

By right shifting a capability from the precondition's state to the capabilities to be applied, roles that apply no capability improve, because they apply one capability subsequently. Moreover, if a task's step is missing in the role allocation, the application of this rule might solve this problem.

The rule is demonstrated in the following example, in which s_p, the last element of $state$, is moved to the front of $capabilitiesToApply$:

$$\begin{aligned} state = [s_1, \ldots, s_p] &\overset{mutation}{\longrightarrow} \\ state = [s_1, \ldots, s_{p-1}], & \\ capabilitiesToApply = [c_1, \ldots, c_q] &\overset{mutation}{\longrightarrow} \\ capabilitiesToApply = [s_p, c_1, \ldots, c_q], & \end{aligned}$$

5) Add parts of the Task: In addition to deleting or shifting capabilities, parts of the task could be added to the end of the capabilities to be applied and the postcondition's state. More precisely, the first task's step that is not contained in the postcondition's state is added.

As before, this rule helps to improve suboptimal roles with no applying capability by applying one capability afterwards, and it helps to improve role allocations where a task's step is missing.

For $t_1, \ldots, t_{i-1} \in state'$ and $t_i \notin state'$, the following example shows $capabilitiesToApply$ and $state'$ before and after the mutation, when t_i was added to both:

$$\begin{aligned} capabilitiesToApply = [c_1, \ldots, c_q] &\overset{mutation}{\longrightarrow} \\ capabilitiesToApply = [c_1, \ldots, c_q, t_i], & \\ state' = [s_1, \ldots, s_p, c_1, \ldots, c_q] &\overset{mutation}{\longrightarrow} \\ state' = [s_1, \ldots, s_p, c_1, \ldots, c_q, t_i] & \end{aligned}$$

6) Mutate whole Role: Mutate whole role replaces the old role with a completely new generated role that is self-consistent, i.e., it violates no role-bounded constraint, e.g., the resource's state when it is given to the agent, the applied capability, and the resource's state when it is given to the next agent do not disagree. Moreover, the assigned ports and the applied capabilities suit the agent's input, output, and available capabilities.

The replacement of an old and probably bad role with a completely new role could improve the role itself and the entire role allocation.

Let us give a short example:

$$\begin{aligned} state = [s_1, \ldots, s_p] &\overset{mutation}{\longrightarrow} \\ state = [t_1, \ldots, t_i], & \\ capabilitiesToApply = [c_1, \ldots, c_q] &\overset{mutation}{\longrightarrow} \\ capabilitiesToApply = [t_{i+1}, \ldots, t_j], & \\ state' = [s_1, \ldots, s_p, c_1, \ldots, c_q] &\overset{mutation}{\longrightarrow} \\ state' = [t_1, \ldots, t_j], & \end{aligned}$$

where $i \leq j \leq z$. $state$, $capabilitiesToApply$, and $state'$ got completely replaced such that $state \sqsubseteq task$, $state' \sqsubseteq task'$, and $state + capabilitiesToApply = state'$.

7) Mutate Resource-Flow: The last rule is the only one that does not only mutate one single role, but looks at all roles in the role allocation. It tries to improve the role allocation's resource-flow, i.e., the route of the resources in the system. This is done by going through the list of agents and, for every agent a_i, looking at its postcondition's port, i.e., the agent a_j where to hand the resource next. Now a_j is searched, and its precondition's port, i.e., the agent where the resource comes from, is examined: if the precondition's port is unequal to a_i, the port is set to a_i. Hence, a_i gives the resource to a_j, and a_j now gets the resource from a_i.

E. Other Decisions

This section gives a short overview of other decisions that had to be made in order to run the genetic algorithm.

1) Initialization: In the initialization, the genetic algorithm creates the initial population. Thus, the algorithm has to create several random role allocations. This is done by assigning each agent one randomly generated role that is self-consistent, like in the mutation rule "Mutate whole Role".

As a possible extension in the future, the system's last configuration could be added in the starting population, too, in order to have an already quite good individual to start the search from at the beginning.

2) Selection: To select parent role allocations for crossover, we use the common *roulette-wheel selection*, which selects individuals directly proportional to their fitness. That means that a fitter individual is more likely to be selected for crossover than a less fit individual ("survival of the fittest"). This increases the algorithm's chance to improve the quality of the population. Nevertheless, role allocations with a low fitness can be selected with a lower probability, too, trying to avoid a too soon convergence in a wrong direction.

3) Replacement Scheme: The generated offspring and the best k individuals of the parent generation are the starting population for the next generation. This replacement scheme is called *elitism*. We use it to maintain best solutions, because the search space is huge. A "proof" can be seen in Sect. IV.

4) Termination: If a termination condition is satisfied, the algorithm stops and verifies if the solution is correct by asking the result checker. If so, the solution is returned to the system subsequently. One termination condition is the number of generations: the algorithm stops if it has processed a predefined number of generations. Of course, the algorithm might not yet have found a correct solution at that time.

Alternatively, the algorithm could terminate at once if it found a solution that satisfies minimum criteria, i.e., the solution is correct (but not necessarily optimal). This decreases the runtime indeed, but – in case of a suboptimal solution – also avoids the chance to find a better or optimal solution. However, we abort the algorithm if a solution satisfies minimum criteria, because our results showed that the ongoing search for a better solution took too much time, especially in larger systems.

Additionally, the algorithm could be restarted after it has reached the end and has not yet found a solution. In order to prevent an endless computation, it may run at most a predefined number of times.

IV. Evaluation

In order to evaluate the genetic algorithm, we implemented it in Java. This section now shows the tests we made to optimize the algorithm's parameters, to study the algorithm's scalability, and to compare the results with the former used constraint solver.

A genetic algorithm has numerous possibilities for scaling, adjusting, and optimizing. Apart from the fact that the algorithm could be implemented with, e.g., other mutation rules or an n-point-crossover, there exist several parameters that can be fine-tuned, e.g., the population size, the mutation rate, the number of generations, the number of elitism elements, or the crossover probability. Moreover, all mutation rules can be weighted differently. The fitness function can be modified as well, for example, by varying and weighting the penalties for violating constraints.

In the chosen system setting, each agent had the full set of inputs, outputs, and available capabilities. This means that the search space had the full size. By constraining the agents' sets we would probably get better results, because the search space gets smaller. Furthermore, in order to see how much time it takes the algorithm to find a solution if it is not disturbed, it did not restart after a certain amount of generations, although this would improve the average runtime, since, in some cases, it runs in the wrong direction, what takes a lot of time.

First of all, the tests showed that if the algorithm does not maintain the best individual of the parent generation, it is not usable. The runtime rises tremendously while the found solutions get worse. Anyway, the number of individuals that are taken from the parent to the offspring generation makes not that big a difference. However, we got slightly better results when taking three individuals from one generation to the other instead of one, two, or more than three.

Fig. 4 shows the results for the optimization of the mutation rate in a system of 17 agents, the standard system size in one of our case studies. For this purpose, we varied the mutation rate while the other parameters were kept constant, i.e., the population size was 50, the algorithm should terminate at the latest after 50,000 generations, and the three best individuals were taken from the former generation to the next generation. For each mutation rate we made 100 tests. The mutation rate varied from about 1.5 % to 17.5 %.

As can be seen, we got the best results with a mutation rate around 9-11 %, where the algorithm's found solutions were in approximately 90 % of all tests correct and in about 60 % of all tests optimal.

After finding a good setting for the mutation rate, we optimized the population size. Again, the system had 17 agents, and all other parameters were kept constant, i.e., the mutation rate was around 9 %, termination was set to 50,000 generations, and we kept the three best individuals. The population size varied from 50 individuals to 1,000 individuals, each tested 100 times. The results can be seen in Fig. 5.

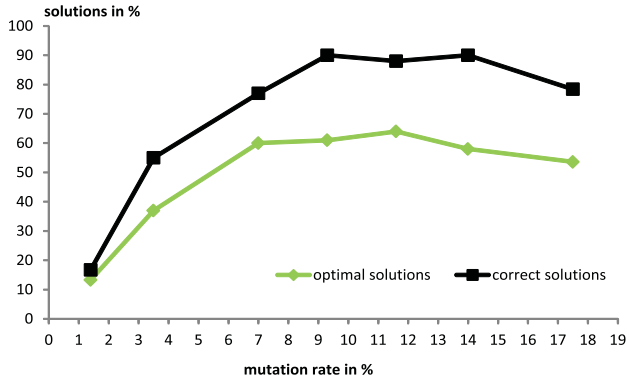

Fig. 4. Optimizing the Mutation Rate for 17 Agents

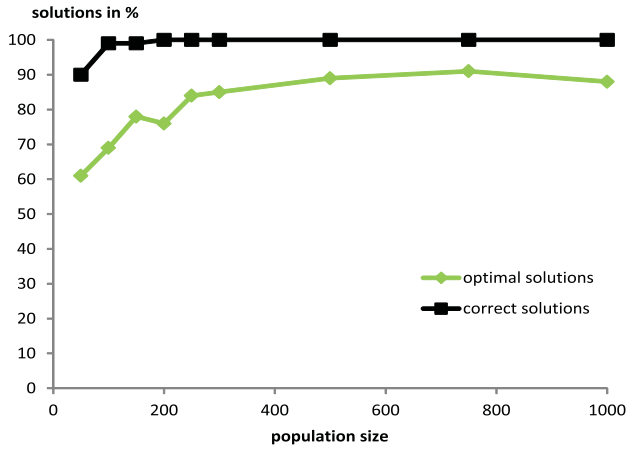

Fig. 5. Optimizing the Population Size for 17 Agents

Increasing the population size continuously improved the quality of the algorithm's found solutions. The number of correct solutions improved from 90 % at 50 individuals to 100 % at 200 individuals, and stayed from then on constant at 100 %. The number of optimal solutions increased from about 60 % to 91 % at 750 individuals. A further increase of the population size did not significantly improve the quality of the found solutions, but considerably enlarged the runtime.

In our case study, we mostly operate with a system size of 9 or 17 agents. In order to test the algorithm whether it can also be applied for bigger systems, we made some tests with different numbers of agents, which should give a rough idea about scalability. For every system size we made 100 tests where all parameters were fixed, and measured the number of optimal and correct solutions, the runtime, and the number of generations the genetic algorithm had to search. The mutation rate and the population size were indeed constant for a single system size, but were different compared to other system sizes, since, for example, a system with 9 agents has no need for 750 individuals – this only would slow down the algorithm. However, we have not yet searched for an optimal parameter set for systems with sizes of 21 and 25 agents, and guessed in each case a parameter set that should be quite good. With an optimal parameter set, the algorithm would further improve. The results are presented in Tab. I.

The table shows that the algorithm found a correct solution in every case, even in bigger systems, and in most cases it found an optimal solution. However, there is certainly a remaining possibility that the algorithm does not find a correct solution. What cannot be seen in the table is the fact that the non-optimal solutions in almost every case were nearly optimal, i.e., only one agent applied two capabilities instead of one capability. The average runtime is quite low at the beginning and gets higher with an increasing number of agents. Since the search space increases immensely with every additional agent, the average runtime of course rises, too. You should not be too confused about the quite constant average number of generations compared to the increase of the average runtime, as the runtime increases due to the higher number of agents and individuals.

We also examined whether the algorithm is able to deal with relatively large systems. For this purpose, we did one test run with 51 agents, without an optimal parameter setting. The genetic algorithm found a nearly optimal solution (one agent applied two capabilities) after less than 5.5 hours, which is of course too long to be used in a real world application. Usually, failures occur locally at one agent and, therefore, not necessarily the whole system needs to be involved for reconfiguration. Hence, the idea is to localize the failure and apply the genetic algorithm to a small part of the system, with fewer agents.

The use of a genetic algorithm tailored to self-organizing resource-flow systems was a big step forward compared to the use of a constraint solver. The constraint solver we used before to reconfigure the system was by far slower, but it certainly was not optimized for this specific problem domain. For example, the calculation for a solution in a system with 9 agents took in average about 6 seconds, the calculation in a system with 13 agents around 75 seconds. In larger systems with more than 20 agents it even took hours or did not complete at all.

V. Conclusion and Future Work

The use of genetic algorithms for solving classical CSOPs is not new (e.g., [13], [14]). For example, job-shop scheduling problems are solved with genetic algorithms in many cases (see [15] and [16]). In the field of Organic Computing systems, genetic algorithms are often used for implementation of self-optimization. In [17], evolutionary algorithms and learning classifier techniques are used to optimize traffic lights and therefore traffic flow. But usually the optimization problem is not a CSOP; rather, the genetic algorithm is used to find an optimal parameter set for the underlying system.

In this paper, we presented a genetic algorithm for the reconfiguration of safety-critical systems. The used architecture is a variant of the observer/controller architecture proposed in [6]. As the evaluation shows, the systems must not be excessively large. Current work is to try to localize the reconfiguration problem, such that a subset of the agents can reconfigure without comprising the complete system. This is promising

# Agents	Population Size	Mutation Rate	Optimal Solutions	Correct Solutions	Avg. # Generations	Avg. Runtime
9	75	23.33 %	98 %	100 %	79	0.3 s
13	200	9.33 %	91 %	100 %	201	6.5 s
17	750	9.33 %	91 %	100 %	211	53.7 s
21	1500	9.33 %	83 %	100 %	286	244.4 s
25	3000	9.33 %	77 %	100 %	314	855.7 s

TABLE I
SCALABILITY TEST DATA

as failures occur locally. For reconfiguration, the system must be in a so-called *quiescent state* [18], such that no harm occurs during the reconfiguration. To allow the use in safety-critical domains, a result checker component was added. In the future, the result checker could provide valuable feedback for the genetic algorithm in case of a non-valid solution. It knows which constraint was violated and that probably allows to draw conclusions, e.g., about which mutation rules are better than others.

The genetic algorithm is used for self-optimization of self-organizing resource-flow systems. It is integrated into the reference implementation presented in [19]. The reconfiguration problem can be formulated as a CSP. While standard CSP solvers only return one solution, not necessarily the best, the presented algorithm allows to add optimization criteria to the problem and comes up with a better or even optimal solution compared to the CSP solver. The results showed that the algorithm is applicable to these class of systems for solving the CSOP and finding optimal configurations. Together with the result checking component it can also be used in safety-critical domains, where a valid configuration has to be provided.

The next steps are to further optimize the parameters and improve the convergence and scalability by doing further work on the mutation rules. Another interesting extension that is planned is the introduction of coordinates for the agents, which allows to include distance or other locality criteria in the fitness function. Further the effect of the use of two-point- or n-point-crossover could be interesting to investigate. The effect of different distributions of the capabilities, e.g., sparse or dense redundancy, as well as the degree of interconnection for the agents on the convergence and quality is another issue worth investigation.

ACKNOWLEDGMENT

This work is partly sponsored by the priority program "Organic Computing" (SPP 1183) of the German Research Foundation (DFG) in the project SAVE ORCA.

The authors would like to thank Matthias Sommer for implementing most parts of the genetic algorithm.

REFERENCES

[1] C. Müller-Schloer, "Organic Computing – On the Feasibility of Controlled Emergence," in *CODES+ISSS '04: Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 2–5.

[2] H. Seebach, F. Ortmeier, and W. Reif, "Design and Construction of Organic Computing Systems," in *Proceedings of the IEEE Congress on Evolutionary Computation 2007*. IEEE Computer Society Press, 2007.

[3] E. Tsang, "Foundations of Constraint Satisfaction," London and San Diego, 1993.

[4] E. Torlak and D. Jackson, "Kodkod: A Relational Model Finder," in *Tools and Algorithms for Construction and Analysis of Systems*. Wiley, 2007, pp. 632–647.

[5] F. Nafz, F. Ortmeier, H. Seebach, J.-P. Steghöfer, and W. Reif, "A universal self-organization mechanism for role-based Organic Computing Systems," in *Proceedings of the Sixth International Conference on Autonomic and Trusted Computing (ATC-09)*, vol. 5586. Springer, 2009.

[6] U. Richter, M. Mnif, J. Branke, C. Müller-Schloer, and H. Schmeck, "Towards a Generic Observer/Controller Architecture for Organic Computing," *INFORMATIK 2006 – Informatik für Menschen!*, vol. P-93, pp. 112–119, 2006.

[7] F. Nafz, H. Seebach, J.-P. Steghöfer, S. Bäumler, and W. Reif, "A Formal Framework for Compositional Verification of Organic Computing Systems," in *Proceedings of the 7th International Conference on Autonomic and Trusted Computing (ATC 2010)*. Springer, 2010.

[8] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Professional, 1989.

[9] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor: University of Michigan Press, 1975.

[10] H. Seebach, F. Nafz, J.-P. Steghöfer, and W. Reif, "A Software Engineering Guideline for Self-organizing Resource-Flow Systems," in *Proceedings of the Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2010)*, 2010.

[11] E. L. Lawler and D. E. Wood, "Branch-And-Bound Methods: A Survey," *Operations Research*, vol. 14, no. 4, pp. 699–719, 1966.

[12] E. Tsang and T. Warwick, "Applying genetic algorithms to constraint satisfaction optimization problems," in *ECAI*, 1990, pp. 649–654.

[13] D. E. Brown, C. L. Huntley, and A. R. Spillane, "A Parallel Genetic Heuristic for the Quadratic Assignment Problem," in *Proceedings of the 3rd International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 406–415.

[14] H. Mühlenbein, "Parallel Genetic Algorithms Population Genetics and Combinatorial Optimization," in *Proceedings of the 3rd International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 416–421.

[15] S. Kobayashi, I. Ono, and M. Yamamura, "An Efficient Genetic Algorithm for Job Shop Scheduling Problems," in *Proceedings of the 6th International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 506–511.

[16] T. Yamada and R. Nakano, "Genetic Algorithms for Job-Shop Scheduling Problems," in *In Proceedings of Modern Heuristic for Decision Support*. Morgan Kaufmann, 1997, pp. 474–479.

[17] H. Prothmann, F. Rochner, S. Tomforde, J. Branke, C. Müller-Schloer, and H. Schmeck, "Organic Control of Traffic Lights," in *Proceedings of the 5th International Conference on Autonomic and Trusted Computing*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 219–233.

[18] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Trans. Softw. Eng.*, vol. 16, pp. 1293–1306, November 1990.

[19] F. Nafz, F. Ortmeier, H. Seebach, J.-P. Steghöfer, and W. Reif, "A generic software framework for role-based Organic Computing systems," in *SEAMS 2009: ICSE 2009 Workshop Software Engineering for Adaptive and Self-Managing Systems*, 2009.