Decentralized reconfiguration for self-organizing resource-flow systems based on local knowledge

Gerrit Anders, Hella Seebach, Florian Nafz, Jan-Philipp Steghöfer, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Anders, Gerrit, Hella Seebach, Florian Nafz, Jan-Philipp Steghöfer, and Wolfgang Reif. 2011. "Decentralized reconfiguration for self-organizing resource-flow systems based on local knowledge." In *2011 Eighth IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems, 27-29 April 2011, Las Vegas, NV, USA*, edited by Roy Sterritt, 20–31. Piscataway, NJ: IEEE. https://doi.org/10.1109/ease.2011.8.

Decentralized Reconfiguration for Self-Organizing Resource-Flow Systems Based on Local Knowledge

Gerrit Anders, Hella Seebach, Florian Nafz, Jan-Philipp Steghöfer, and Wolfgang Reif
Institute for Software & Systems Engineering
Augsburg University, Universitätsstr. 6a
86159 Augsburg, Germany
{*anders, seebach, nafz, steghoefer, reif*}*@informatik.uni-augsburg.de*

***Abstract*—The introduction of self-organization into a system promises, among other things, to reduce the system's complexity and to increase the system's robustness against failures and its adaptability to changes in its environment. An example for systems that profit from self-organization are resource-flow systems, e.g., production lines. Such systems are characterized by a number of independent agents that process resources by applying capabilities according to a given task.**

This paper introduces a decentralized reconfiguration mechanism that restructures a part of a resource-flow system in case of a failure. In order to do so, agents coordinate based on local knowledge and combine themselves into groups which are called coalitions. Each coalition then tries to restore the system's functionality, returning a previously consistent and correct system to a new consistent state, thus re-enabling correct processing of resources. As only local coalitions are formed, the parts of the system not involved in the reconfiguration process stay functional, meaning that the overall system does not come to a standstill.

***Keywords*-Coalition Formation; Decentralized Reconfiguration; Self-Organization; Multi-Agent Systems**

I. Introduction

Today, self-organization is explored as a means to cope with the complexity of systems that consist of several collaborating entities. Self-organization promises to make these systems more robust against failures, adaptive to new situations, and to increase their flexibility. With this in mind, a self-organizing system is characterized by the absence of external control structures and the capability to find its organizational structure on itself. Regarding the class of resource-flow systems that includes, among others, applications in the field of logistics or production automations, self-organization leads to many benefits. As described in [1], today's resource-flow systems are planned exhaustively and therefore are static and inflexible but optimized to their goal. For small series productions, it is desirable to have systems being able to cope with different changes in the environment. In [2], a software engineering guideline is presented which supports an engineer in designing self-organizing resource-flow systems. The guideline not only provides design artifacts describing the systems' structure and the components' behavior, but also techniques and formalisms to ensure the desired behavior of the whole systems despite self-organization. One step in the guideline is to choose the right reconfiguration mechanism for a given application or domain. In some situations, central reconfiguration with global knowledge is suitable and most efficient. But there are many environments which require decentralized mechanisms without external control. This paper presents a decentralized algorithm applicable in these environments which deals with local knowledge, meaning that there is no external or internal global knowledge base.

The presented algorithm is characterized by the following three features: first, it is specialized for a class of systems, i.e., self-organizing resource-flow systems. This provides the possibility to optimize with respect to specifics of this class of systems. Second, the algorithm is robust against diverse failures, such as breakdown of an agent, infrastructural problems, or partial failures of an agent. Last but not least, the algorithm is decentralized and gets by with pure local knowledge. One main benefit of local reconfiguration is that the bigger part of the system can perform as usual and is not interrupted by the reconfiguration. In addition, as the algorithm is embedded in the given guideline, it guarantees a "good behavior" of the system. This means, if the system is in a correct configuration before a failure occurs, the algorithm re-establishes the correct functionality if the prerequisites for a reconfiguration are given.

The rest of the paper is organized as follows: Section II introduces the class of systems this algorithm is designed for. An example and the basic concepts are given. Then, Section III presents the way the algorithm forms a coalition for solving an emerging problem. Several failures which can occur in self-organizing resource-flow systems plus the reactions of the algorithm are shown. Section IV shortly describes the implementation before Section V discusses the properties of the algorithm. Mechanisms for central or decentralized reconfiguration in the field of self-organizing systems and other related work is pointed out in Section VI. Section VII concludes the paper and presents future work.

II. Class of Resource-Flow Systems

The class of resource-flow systems contains many kinds of systems, for example, from the field of production au-

tomation or logistics. These systems are characterized by many independent units processing resources to achieve a global goal, e.g., produce a product with certain properties. As already mentioned, self-organization is very interesting for these kinds of systems because the ability for flexibility, redundancy, and different tasks is inherently given. Thus, it is possible to integrate self-organization concepts into the rigid systems of today. To introduce these concepts, an elaborated software engineering guideline presented in [2] is used which delivers a pattern (Organic Design Pattern, short ODP) providing an architecture for self-organizing resource-flow systems. In addition, it determines the behavior and interactions of the independent units called agents.

In the following, the main concepts and terms of the ODP are introduced which are necessary for understanding the reconfiguration algorithm. In systems designed with the help of the ODP, an *agent* has *capabilities* to process *resources* according to a given *task*. The task is a sequence of capabilities and the *state* of the resource is always a prefix of the task that has to be done for this resource. Which capabilities to apply when is determined by the *roles* of the agent, each related to exactly one task. For this purpose, a role's *precondition* (prec) and *postcondition* (postc) tell the agent, among other things, from which other agent it receives resources and to which agent it has to hand over processed resources. These conditions are 3-tuples of a target agent (*port*) from which, respectively to which, the resource is taken, respectively given, the current state of the resource and the task that needs to be done. An agent can have several allocated roles and can also participate in several resource-flows. This means if there are several resources in the system with different tasks (processing steps), there must also be several resource-flows, one for each task. Corresponding to the task and the state of the resource, the agent is able to choose one of its allocated roles and process the resource accordingly.

Our running example is an adaptive production cell for small series car production. Figure 1 shows one part of an initial system configuration. Here, you can see two types of agents, robots and carts. The agents' capabilities are shown on the right side of each agent. The carts do not have any capabilities in this example. The robots have some out of the following capabilities: weld the body (B), insert the motor (M), attach the exhaust (E), insert the lights (L), and attach the wheels (W). The bold letter indicates the capability the robot is currently applying. For example, robot one (R_1) has the capabilities B, L, W and is currently welding car bodies (B). The arrows indicate the resource-flow of the car body (the resource). Table I not only represents the capability allocation, but also the *input* and *output* relations of the agents. This information is necessary since an agent is not capable of exchanging resources with arbitrary agents. The possible resource-flows are defined by the *Input-/Output-graph (I/O-graph)*. In the example, cart C_2 is able to receive

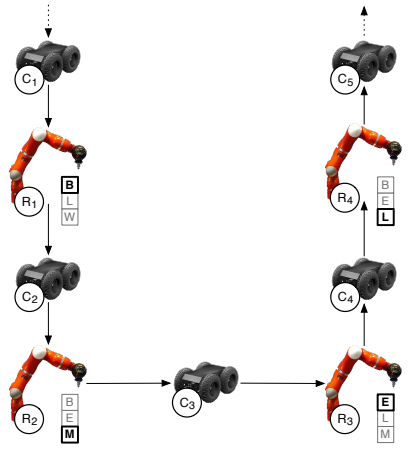

Figure 1. One part of the initial resource-flow graph showing the agents' initial role allocation

Table I
PART OF THE INITIAL SYSTEM CONFIGURATION

Relation	Elements
$R_1.availableCapabilities$	$\{B, L, W\}$
$R_2.availableCapabilities$	$\{B, E, M\}$
$R_3.availableCapabilities$	$\{E, L, M\}$
$R_4.availableCapabilities$	$\{B, E, L\}$
$R_x.inputs$	$\{C_1, C_2, C_3, C_4, C_5\}$
$R_x.outputs$	$\{C_1, C_2, C_3, C_4, C_5\}$
$R_2.inputs$	$\{C_1, C_2, C_3, C_4, C_5\}$
$R_2.outputs$	$\{C_1, C_3, C_4, C_5\}$
$C_y.availableCapabilities$	$\{\ \}$
$C_z.inputs$	$\{R_1, R_2, R_3, R_4\}$
$C_z.outputs$	$\{R_1, R_2, R_3, R_4\}$
$C_2.inputs$	$\{R_1, R_3, R_4\}$
$C_2.outputs$	$\{R_1, R_2, R_3, R_4\}$
$C_3.inputs$	$\{R_1, R_2, R_3, R_4, C_4\}$
$C_3.outputs$	$\{R_1, R_2, R_3, R_4, C_4\}$
$C_4.inputs$	$\{R_1, R_2, R_3, R_4, C_3\}$
$C_4.outputs$	$\{R_1, R_2, R_3, R_4, C_3\}$
$Tasks$	$\{[\ldots, B, M, E, L, \ldots], [\ldots], \ldots\}$

with $x \in \{1, 3, 4\}$, $y \in \{1, 2, 3, 4, 5\}$, and $z \in \{1, 5\}$

resources from all robots except robot R_2.

The system configuration is given by the allocation of roles to the agents. In the example, initially, R_2 has the following role:

```
Precondition: (C2,[...,B],[...,B,M,E,L,...])
Capabilities to Apply: [M]
Postcondition: (C3,[...,B,M],[...,B,M,E,L,...])
```

According to the role's precondition, the resource is taken from C_2 with *state* $[\ldots, B]$ and *task* $[\ldots, B, M, E, L, \ldots]$ and the capability "insert motor" (M) is applied. Subsequently, as stated in the role's postcondition, the resource is given to C_3 with the new *state* $[\ldots, B, M]$ and *task* $[\ldots, B, M, E, L, \ldots]$. As the example only considers a part of the whole system, the dots in *task* and *state* indicate the remaining capabilities relative to the full task of the resource.

The *resource-flow graph (RF-graph)* for a task is defined

by the connections between the agents as they are determined by the roles for the task. If the ports in the pre- and postconditions are combined, they yield a path through the I/O-graph. The communication infrastructure is independent from the I/O and RF-graph. As soon as an agent is aware of another agent, it can communicate with this agent. Usually, an agent only knows agents which are in its input or output relations.

Accurate system configurations are ensured by several constraints that can be monitored at runtime by the agents themselves. Some of these constraints are listed in [3]. The constraints define a correct role allocation and thus form the specification of the reconfiguration algorithm. An example is the **I/O-Consistency** for an agent (self):

$$\forall a \in agents : \forall r \in a.allocatedRoles :$$
$$r.prec.port \in a.inputs \wedge r.postc.port \in a.outputs$$

This constraint states that all agents that are assigned as the port in the pre- or postcondition of any of the allocated roles have to be part of the I/O-relation as well. It is violated in case an agent is no longer responsive.

Summarizing, the constraints describe "valid system configurations" and are monitored by the agents. In case of violations, they are restored by a reconfiguration mechanism. This idea is called "Restore Invariant Approach". A detailed description of this mechanism can be found in [4]. It is obvious that a central reconfiguration mechanism which knows all agents, all capabilities, inputs, outputs, and so on is able to find a solution for this constraint satisfaction problem. But if the problem should be solved decentralized and most notably if the agents have only their local knowledge, the situation gets really difficult. This paper now introduces an algorithm which re-establishes a "valid system configuration". It starts the coalition formation only with the knowledge of the agent that identifies a violation of a constraint.

III. Coalition Formation Strategy

The main idea of the algorithm proposed in this paper is to form groups of agents – coalitions – that are able to restore locally violated constraints. More precisely, the algorithm reacts to broken capabilities, inputs, or outputs as well as to the breakdown of entire agents and reconfigures the system so that, after reconfiguration, all locally monitored constraints hold again. This is achieved by updating the sets of allocated roles of agents that participate in a coalition. In the following, the process of building the coalitions is called *coalition formation*.

In the course of coalition formation, the algorithm makes use of the existing system structure while it is restricted to local knowledge. Whenever an agent is added to a coalition, the coalition's knowledge of the system grows as each agent has a set of capabilities, allocated roles, and usually several inputs and outputs to other agents.

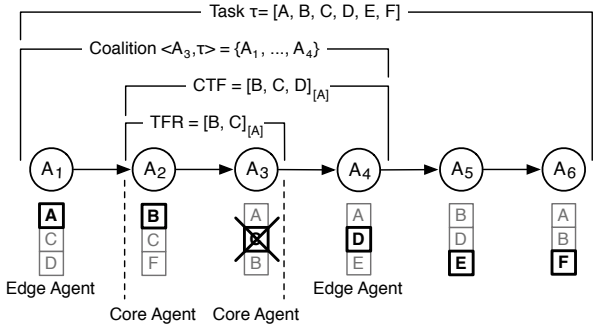

Figure 2. A_3 loses its capability C that is needed to process resources in task τ. First, it recruits A_4 which cannot help. Afterwards, it recruits A_2 resulting in a coalition with members $\{A_2, A_3, A_4\}$ and $CTF = [B, C, D]_{[A]}$. Since A_2 has capability C and A_3 capability B, they can "swap" roles which results in in $TFR = [B, C]_{[A]}$ and $CA = \{A_2, A_3\}$. Edge agents A_1 and A_4 connect the reconfigured part (TFR) with the parts of the system that are not reconfigured. Therefore, A_1 becomes a member of the coalition.

Each coalition is able to restore multiple violated constraints at once and is led by one of its coalition members. This agent is called *leader*. It coordinates its coalition in order to restore locally violated constraints of a specific task. As soon as an agent A_i notices the violation of a locally monitored constraint, it initiates the reconfiguration for the affected task τ by creating a new coalition in case it is not yet included in a coalition reconfiguring τ. Otherwise, it waits until the current reconfiguration is finished and starts a new coalition afterwards if necessary. Since the initiator A_i becomes the leader of this coalition and reconfigures τ, the coalition is called $\langle A_i, \tau \rangle$. At this point, $\langle A_i, \tau \rangle$ has only one member – its leader A_i. The objective of A_i is to reconfigure the members of $\langle A_i, \tau \rangle$ for task τ. However, in order to bring the reconfiguration to completion, a certain level of flexibility is needed. Therefore, the leader A_i increases its coalition's flexibility by inviting other agents to participate in $\langle A_i, \tau \rangle$ for τ. While $\langle A_i, \tau \rangle$ grows, the most important information is about input and output relations of newly added coalition members. The reason for that is that this is the only way a coalition becomes acquainted with other agents of the system. If $\langle A_i, \tau \rangle$ has enough flexibility to reconfigure a continuous segment of τ, the so-called *connected task fragment* (CTF), there is no need to expand it any more. After this, $\langle A_i, \tau \rangle$ determines the part of CTF that is actually reconfigured in terms of calculating new roles. This part is called *task fragment to reconfigure* (TFR). Thereafter, leader A_i recruits agents that are responsible for maintaining the interconnections, i.e., the resource-flow, between the reconfigured part of the system and the parts of the system that are not reconfigured. These agents form the set of *edge agents* (EA) as they enclose the reconfigured part with regard to the resource-flow. Edge agents are not necessarily new agents; instead,

they may already be part of the coalition. Furthermore, $\langle A_i, \tau \rangle$ specifies the agents that should apply capabilities within TFR. These agents and those that were responsible for TFR before the reconfiguration was initiated form the set of *core agents* (CA). Next, the resource-flow between the agents is re-established. If a resource-flow between two agents cannot be established (e.g., due to insufficient inputs and outputs), A_i recruits additional agents in order to increase the coalition's flexibility. Those that are useful form the set of *resource-flow agents* (RFA). As soon as the resource-flow is rebuilt, leader A_i distributes the updated role allocations to all coalition members. Finally, the leader gives all coalition members a signal to resume operation and $\langle A_i, \tau \rangle$ dissolves itself. The small example in Figure 2 illustrates the terminology introduced in this section.

In the following, more precise definitions of the sets CA, EA, RFA, and the task fragments CTF and TFR are given:

Core Agents (CA): Contains all agents of a coalition $\langle A_i, \tau \rangle$ that should apply at least one capability within TFR. Additionally, the set contains all agents that were responsible for processing resources or maintaining the resource-flow between the first (r_f) and the last role (r_l) that have to be replaced due to a broken capability, port, agent, or due to a change in the application of capabilities. In other words, this set contains agents that lose allocated roles or get completely new ones.

Edge Agents (EA): Contains all agents with roles r_j that formerly sent/received resources that were received/sent by r_f/r_l. r_j's post-/precondition port is updated during reconfiguration.

Resource-Flow Agents (RFA): Contains all agents that are needed to generate the resource-flow and that are not included in CA or EA. These agents get additional roles for τ. Other roles are not modified or replaced.

Connected Task Fragment (CTF): The segment of task τ that is reconfigured by a coalition. CTF starts at the first capability that is applied by the role having the shortest precondition state α, and ends at the last capability that is applied by the role having the longest postcondition state β (only those roles are considered that are allocated to members of $\langle A_i, \tau \rangle$ for task τ). CTF is defined as

$$CTF = [\tau_a, \ldots, \tau_b]_\alpha,$$

where $a, b \in \mathbb{N}$ and $1 \leq a \leq b \leq \tau.length$. τ_k denotes the k-th capability in τ. α and β are defined as follows:

$$\begin{aligned} \alpha &= min\{r_{A_k}(\tau, l).prec.state \mid A_k \in \langle A_i, \tau \rangle \wedge l \in \mathbb{N}\} \\ \beta &= max\{r_{A_k}(\tau, l).postc.state \mid A_k \in \langle A_i, \tau \rangle \wedge l \in \mathbb{N}\} \\ &= \alpha + [\tau_a, \ldots, \tau_b] \end{aligned}$$

$r_{A_k}(\tau, l)$ is the l-th allocated role of agent A_k for task τ. Furthermore, here, $+$ is the operator for list concatenation and min/max are operators that return the shortest/longest state that is included in a set of states. α is a resource's state $[\tau_1, \ldots, \tau_{a-1}]$ after the application of the $(a-1)$-th capability in τ or the empty list $[\,]$ for $a = 1$. Consequently, immediately before a capability of CTF is applied to a resource, its state is α. For this reason, α unambiguously defines the task fragment's location in τ. This information is important as a task can consist of a recurrent sequence of capabilities. β is the state of a resource that leaves CTF.

Task Fragment to Reconfigure (TFR): The part of $CTF = [\tau_a, \ldots, \tau_b]_\alpha$ a coalition actually reconfigures in terms of calculating and distributing an updated role allocation. Let α be a prefix of a state γ, then the task fragment to reconfigure is denoted as

$$TFR = [\tau_m, \ldots, \tau_n]_\gamma,$$

where $m, n \in \mathbb{N}$ and $a \leq m \leq n \leq b$. Analogously to CTF, γ defines the location of TFR in τ. TFR starts at the first and ends at the last capability in CTF that should be applied by an agent that does not hold an appropriate role.

In the following subsections, the algorithm's behavior and actions are explained by different reconfiguration scenarios. Additionally, the utilization of core, edge, and resource-flow agents in the course of a reconfiguration is clarified. Starting with the basic reconfiguration procedure in Section III-A, Section III-B shows reconfiguration due to I/O failures, whereas Section III-C highlights reconfiguration in case of a breakdown of an entire agent.

A. Reconfiguration Due to a Broken Capability

The reconfiguration due to a defective capability that is needed to correctly perform an allocated role is an important feature when dealing with reconfiguration in resource-flow systems.

Regarding the system presented in Section II, assume that R_2 loses its capability M. As R_2 needs M to perform its role $r_{R_2}(\tau, 1)$, which applies M to the resource-flow, R_2 creates a new coalition $\langle R_2, \tau \rangle$ with leader R_2 that reconfigures task $\tau = [\ldots, B, M, E, L, \ldots]$.

In order to repair the system, R_2 has to enlarge $\langle R_2, \tau \rangle$ to a point at which it includes an agent that is capable of applying M to the resource-flow. However, R_2 has limited knowledge and therefore it cannot send a request to an arbitrary agent of the system to join the coalition. Furthermore, R_2 does not know which agent has capability M. Initially, R_2 knows agents listed in its inputs and outputs as well as those agents that are stated in pre- and postcondition ports of its allocated roles. Hence, R_2 knows $\{C_1, \ldots, C_5\}$ (see Table I).

Denote K as the set of known agents that are not members of the coalition, then, currently, K is $\{C_1, \ldots, C_5\}$. C_{Av}, the set of available capabilities of agents participating in

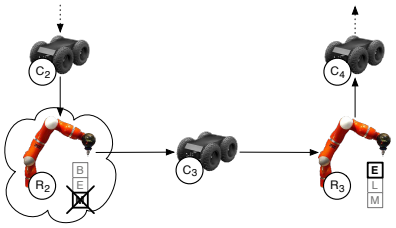

Figure 3. R_2 starts a reconfiguration for task τ due to its broken capability M and becomes the leader of coalition $\langle R_2, \tau \rangle$

$\langle R_2, \tau \rangle$, is $R_2.availableCapabilities = \{B, E\}$. Additionally, CTF is $[M]_{[\ldots,B]}$ because of:

$$r_{R_2}(\tau, 1).prec.state = [\ldots, B]$$
$$r_{R_2}(\tau, 1).postc.state = [\ldots, B, M]$$

Therefore, C_{Nd}, the set of capabilities that are needed to fulfill CTF, is $\{M\}$. Figure 3 shows the situation in which R_2 started a new coalition with members $\{R_2\}$. The defective capability M is marked with a cross.

The first goal of every leader is to satisfy

$$C_{Nd} \subseteq C_{Av}, \tag{1}$$

which means that the union of all capabilities of coalition members C_{Av} has to contain all capabilities C_{Nd} that are needed to fulfill CTF.

To satisfy Equation 1, R_2 asks other agents to participate in $\langle R_2, \tau \rangle$. R_2 puts its requests in an order that is given by two queues Ω_{in} and Ω_{out} with $\Omega_{in} \cup \Omega_{out} = K$. Ω_{in} contains all agents that are listed in the inputs of coalition members. Analogously, Ω_{out} contains the outputs. Whenever the coalition has to be enlarged, the first agent of a queue is dequeued, removed from the other queue, and requested to join the coalition. Starting at Ω_{out}, the algorithm alternates between both queues. With each agent that enters a coalition, its leader's knowledge grows and Ω_{in} as well as Ω_{out} are updated so that $\Omega_{in} \cup \Omega_{out} = K$ holds at any time. Whenever a queue is updated, the agents, which are known to participate in the task that is to be reconfigured, are moved to the front of the queue. An agent A_i is known to participate in task τ if the coalition contains an agent that has at least one role whose pre- or postcondition port points to A_i. Among others, one reason for preferring agents that are known to participate in the task to reconfigure is that it is beneficial to restructure a system without influencing the processing of other tasks. For instance, an agent applying different capabilities in different tasks has to switch capabilities which needs time and therefore slows down the system.

Initially, the queues are initialized as

$$\Omega_{in} = [\widetilde{C_2}, \widetilde{C_3}, C_1, C_5, C_4]$$
$$\Omega_{out} = [\widetilde{C_3}, C_1, C_4, C_5]$$

(agents that are marked with a tilde are known to participate in τ) because of $R_2.inputs = \{C_1, \ldots, C_5\}$ and $R_2.outputs = \{C_1, C_3, C_4, C_5\}$. Further, it is known that C_2 and C_3 participate in τ since $r_{R_2}(\tau, 1)$ takes resources from C_2 and gives them to C_3.

Consequently, R_2 removes the first entry of Ω_{out}, in this case C_3, and requests this agent to participate in $\langle R_2, \tau \rangle$ for τ. As a result, C_3 starts reconfiguration for τ, joins $\langle R_2, \tau \rangle$, and transmits its knowledge in the form of inputs, outputs, capabilities, and allocated roles to R_2, whereupon R_2 updates the coalition's information as well as $\Omega_{in} = [\widetilde{R_3}, \widetilde{C_2}, \ldots]$ and $\Omega_{out} = [\widetilde{R_3}, C_1, \ldots]$. CTF stays unchanged as $r_{C_3}(\tau, 1)$ does not apply any capabilities to the resource-flow.

As Equation 1 is not satisfied, R_2 invites the first agent of Ω_{in}, that is R_3, to join $\langle R_2, \tau \rangle$. As a result, CTF expands to $[M, E]_{[\ldots,B]}$ because of:

$$r_{R_3}(\tau, 1).postc.state = [\ldots, B, M, E]$$

However, since R_3 has capabilities $\{E, L, M\}$, R_2 finally found an agent that is able to apply capability M to the resource-flow. Equation 1 holds.

R_2 now calculates the combinations in which the coalition members can apply the capabilities listed in task fragment CTF. Since capabilities M and E are needed to fulfill CTF and R_2 has capabilities $\{B, E\}$, whereas C_3 has no capabilities at all and R_3 can apply $\{E, L, M\}$, there are two possible combinations: *Combination 1* plans that R_3 applies M and R_2 capability E to the resource-flow, which means that R_2 as well as R_3 would be involved in τ after reconfiguration. In *Combination 2*, R_3 should perform both capabilities and R_2 none. The algorithm makes use of a number of heuristics to estimate the quality of combinations and to sort the combinations according to different criteria. The most important one is to optimize a system's throughput by involving as many agents as possible in the application of capabilities specified by the task. Therefore, the algorithm chooses *Combination 1*.

In the next step, TFR is determined. Here, TFR equals $CTF = [M, E]_{[\ldots,B]}$ as capability M should be applied by R_3 in *Combination 1*, which was formerly done by R_2, and capability E should be performed by R_2, which was formerly done by R_3. Consequently, the set of *core agents*, $CA = \{R_2, C_3, R_3\}$, is identified as stated in Section III with $r_f = r_{R_2}(\tau, 1)$ and $r_l = r_{R_3}(\tau, 1)$. The set includes C_3 because this cart was responsible for conveying resources from R_2 to R_3.

Since each coalition reconfigures a specific task fragment and therefore changes a part of the structure of the associated resource-flow, it is of great importance to ensure that the reconfigured part will correctly interact with the parts of the system that are not reconfigured. Hence, there is a need for connections between agents with updated role allocations and agents that are not reconfigured. These connections are

Table II
INFORMATION USED FOR RECONFIGURATION

Information	Elements
$\langle R_2, \tau \rangle$	$\{C_2, R_2, C_3, R_3, C_4\}$
CA	$\{R_2, C_3, R_3\}$
EA	$\{C_2, C_4\}$
RFA	$\emptyset$
CTF	$[M, E]_{[\ldots,B]}$
TFR	$[M, E]_{[\ldots,B]}$
C_{Nd}	$\{M, E\}$
C_{Av}	$\{B, E, L, M\}$

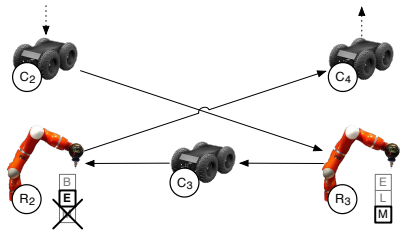

Figure 4. R_2 and R_3 "swapped" roles to compensate the loss of capability M in R_2

identified with the help of r_f and r_l. On the one hand, the agent that formerly performed r_f (R_2) took resources from another agent (C_2) that now has to send its resources to the agent that should apply the first capability in TFR (R_3). On the other hand, the agent that formerly performed r_l (R_3) gave resources to another agent (C_4) that now has to receive resources from the agent that should apply the last capability in TFR (R_2). If TFR was an empty sequence, the first edge agent (C_2) would have to send resources to the second one (C_4). Thus, the postcondition port of $r_{C_2}(\tau, 1)$ and the precondition port of $r_{C_4}(\tau, 1)$ have to be updated accordingly. Because of that, both agents, C_2 and C_4, are requested to join the coalition. As C_2 and C_4 are "special" in the sense that either only the pre- or postcondition ports of their roles may be updated, C_2 and C_4 form the coalition's set of *edge agents* (EA). The final sets of edge and core agents are depicted in Table II.

Finally, the algorithm checks if the resource-flow can be re-established within the coalition. For this purpose, *Dijkstra's Algorithm* for calculating shortest paths is applied to a weighted, directed I/O-graph (see Section II). The weight of a directed edge is 1 if its head is element of $CA \cup EA \cup RFA$. Otherwise, the edge has weight $2|\langle R_2, \tau \rangle|$, where $|\langle R_2, \tau \rangle|$ is the size of the coalition. The weighting guarantees that a path will not include an agent $A_i \notin CA \cup EA \cup RFA$ if there is a path that consists only of agents $A_j \in CA \cup EA \cup RFA$ because of $|CA| + |EA| + |RFA| \leq 2|\langle R_2, \tau \rangle|$ (factor 2 is needed as sets CA and EA are not disjoint in all cases, whereas the sets RFA and CA as well as RFA and EA are always disjoint). This ensures that the algorithm prefers agents whose role allocations will be updated anyway. Therefore, the set of agents with updated roles is kept as small as possible. If a path includes such an agent A_i, A_i is added to the set of *resource-flow agents* (RFA) and the weights are updated as stated above. For each capability τ_i in TFR, a path is calculated from the agent that should apply τ_i to the agent that should apply the next capability τ_{i+1} in TFR if there is one. Additionally, a path is calculated from the edge agent that gives resources to the coalition to the agent that applies the first capability in TFR because it is not guaranteed that these agents are able to directly exchange resources. For the same reason, a path is calculated from the agent that applies the last capability in TFR to the edge agent that receives resources from the coalition. If TFR was an empty sequence, the resource-flow would be re-established between the edge agents.

In this case, edge agent C_2 is able to directly give resources to R_3. After applying M to the resource-flow, R_3 sends its resources via C_3 to R_2 since R_2 is not included in R_3's outputs. Subsequently, R_2 gives the resources to edge agent C_4.

A part of the leader's knowledge and the reconfigured system are depicted in Table II and Figure 4.

B. Reconfiguration Due to a Broken I/O-Port

This section outlines the reconfiguration of the initial system shown in Figure 1 in a situation in which R_2 loses its output port C_3 and C_3 its input port R_2. Reconfiguration is necessary, as otherwise the resource-flow would come to a standstill. As R_2 notices that it is not able to send resources to C_3, which is needed to perform $r_{R_2}(\tau, 1)$, and C_3 detects that it is not able to receive resources from R_2, which is needed to perform $r_{C_3}(\tau, 1)$, both agents independently start new coalitions $\langle R_2, \tau \rangle$ and $\langle C_3, \tau \rangle$ for task τ. On the one hand, $\langle R_2, \tau \rangle$ knows that C_3 has to be reconfigured because C_3 must have a role that receives resources which are sent by $r_{R_2}(\tau, 1)$. On the other hand, C_3 knows that R_2 has to be reconfigured because R_2 must have a role that sends resources which are then received by $r_{C_3}(\tau, 1)$. As a result, the first objective of $\langle R_2, \tau \rangle$ is to recruit C_3 and, analogously, $\langle C_3, \tau \rangle$ wants to recruit R_2.

However, an important property of coalitions is that agents must not participate in more than one coalition for a specific task at the same time. This is necessary since multiple, simultaneous memberships in different coalitions reconfiguring the same task can lead to inconsistent role allocations. Regarding a situation in which several reconfigurations for the same task exist in parallel, an agent would update its role allocation with roles it received most recently. In conjunction with roles other agents receive, these roles could violate constraints that specify the resource-flow as a chain of interlocking roles (see *con3* and *con4* in [3]). This leads to the necessity of merging coalitions that collide; a situation in which at least one coalition wants to recruit

a member of another coalition. This situation is called a *clash*. Whenever two coalitions are merged, a leader election mechanism determines the agent A_i that should lead the merged coalitions by utilizing the lexicographic order of unique agent identifiers; the other leader A_j is degraded to a common member of A_i's coalition. All members of A_j's coalition are assigned to A_i's coalition and all knowledge of A_j's coalition is transferred to A_i.

While R_2 waits for a response from C_3 indicating that the agent joins the coalition and vice versa, both leaders realize that they wait for each other and that the clashing coalitions have to be merged. In order to avoid deadlocks, the agents unambiguously determine the leader that stops waiting and initiates the merging of both coalitions.

Assume that C_3 won the leader election and that the merging was successfully completed. Because of $CTF = [M]_{[\ldots,B]}$, $C_{Nd} = \{M\}$, and $C_{Av} = \{B, E, M\}$, Equation 1 holds.

TFR equals CTF as CTF is of length one. Because C_3 does not have any capabilities, there is only one combination that fulfills TFR. In this combination, R_3 retains the responsibility for applying M to the resource-flow.

Finally, the set of core agents is $\{R_2, C_3\}$ because R_2's and C_3's roles have to be replaced by a new one. Therefore, the adjacent agents C_2 and R_3 become edge agents of $\langle C_3, \tau \rangle$.

While calculating the resource-flow between core and edge agents, the algorithm cannot determine a path from core agent R_2, which has to apply capability M in TFR, to edge agent R_3. The reason for that is that R_2 lost its output to C_3 and it is not able to directly send resources to C_2 or R_3.

Since an agent A_j is not able to receive resources from agents that are not included in its inputs, the leader requests agents $A_i \in A_j.inputs$ to join its coalition until the path exists. If the coalition includes all agents in $A_j.inputs$, the leader asks the inputs of the inputs and so on until the path exists or all such agents have been requested. As before, a leader prefers those agents A_i that are known to participate in τ. In case there is no such path, the leader switches the selected TFR or enlarges the coalition by an arbitrary agent while preferring agents that participate in τ. However, if such a path exists, all agents that form the path, but are not included in the sets CA or EA, are added to the set RFA.

In this case, $R_3.inputs$ is $\{C_1, \ldots, C_5\}$ and C_2, C_3, as well as C_4 are known to participate in τ. Since C_2 and C_3 are already a coalition members, C_4 is added to $\langle C_3, \tau \rangle$. Furthermore, C_4 becomes a resource-flow agent because resources can be sent from R_2 to R_3 via C_4, and C_4 is not included in the sets CA or EA.

A part of C_3's knowledge and the reconfigured system are shown in Table III and Figure 5. C_4 now has two roles – one role to convey resources from R_2 to R_3 and another to transport them from R_3 to R_4 – whereas C_3's role is not replaced by a new one.

Table III
INFORMATION USED FOR RECONFIGURATION

Information	Elements
$\langle C_3, \tau \rangle$	$\{C_2, R_2, C_3, R_3, C_4\}$
CA	$\{R_2, C_3\}$
EA	$\{C_2, R_3\}$
RFA	$\{C_4\}$
CTF	$[M]_{[\ldots,B]}$
TFR	$[M]_{[\ldots,B]}$
C_{Nd}	$\{M\}$
C_{Av}	$\{B, E, M\}$

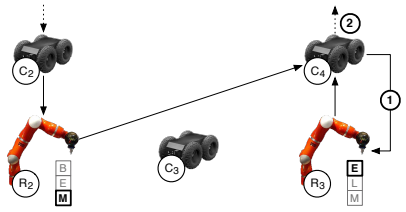

Figure 5. After reconfiguration, C_4 has two roles, whereas C_3 has no roles at all

C. Reconfiguration Due to the Loss of an Agent

This section shows reconfiguration of the initial system depicted in Figure 1 due to the breakdown of an entire agent[1], using the example of agent C_3. In order to detect malfunctioning agents, all agents use a *ping mechanism* to monitor the liveness of agents they cooperate with. An agent cooperates with another agent if they exchange resources. By using this mechanism, R_2 and R_3 observe that C_3 is not available and independently start new coalitions $\langle R_2, \tau \rangle$ and $\langle R_3, \tau \rangle$. Initially, both coalitions have two objectives: first, to identify the capabilities that were performed by the malfunctioning agent and second, to restore the resource-flow. The problem is that, owing to the restriction on local knowledge, R_2 as well as R_3 do not know C_3's allocated roles and the way resources take through the system.

However, by utilizing the system structure, i.e., the structure of the resource-flow which is defined by the system's role allocation, it is possible to reconstruct the roles of a malfunctioning agent if enough knowledge is available. For this purpose, on the one hand, if a coalition member's role r_a has to be replaced because of a broken agent A_i that represents r_a's postcondition port, a coalition's leader searches a role r_b that is applied after r_a to the resource-flow. On the other hand, if a coalition member's role r_c has to be replaced because r_c's precondition port equals A_i, a leader searches a role r_d that is applied before r_c to the resource-flow. Without using information provided by the

[1]In case this agent is currently a leader of a coalition, the coalition members realize the breakdown and form new coalitions in order to reconfigure the system.

pre- and postcondition ports of a role, a role r_b is known to be applied after role r_a to the resource-flow if

$$\begin{aligned} & r_a.postc.task = r_b.postc.task \\ & \wedge \ (r_a.postc.state \sqsubset r_b.postc.state \\ & \vee \ (r_a.capabilitiesToApply \neq [\,] \\ & \wedge \ r_a.postc.state = r_b.postc.state)) \end{aligned} \tag{2}$$

evaluates to $true$, where $\sqsubset$ is the proper list prefix operator.

In order to reconstruct the roles of the broken agent C_3, $\langle R_2, \tau \rangle$ searches a role that is applied after $r_{R_2}(\tau, 1)$ and $\langle R_3, \tau \rangle$ looks for a role that is applied before $r_{R_3}(\tau, 1)$. Therefore, the leaders recruit new agents by making use of the queues Ω_{in} and Ω_{out}. However, in this case, it is not useful to recruit agents that are known to participate in task τ because the leaders would prefer agents that do not fulfill Equation 2: leader R_3 would recruit C_4 and then R_4, and leader R_2 would recruit C_2 and then R_1. For this reason, whenever a leader tries to satisfy Equation 2, it prefers agents that are not known to participate in the task by always recruiting the last agent of a queue. In this example, assume that $R_2.outputs$ contains R_4 and $R_3.outputs$ contains R_2. Hence, $\langle R_2, \tau \rangle$ recruits R_4. $r_{R_4}(\tau, 1)$ fulfills Equation 2 with regard to $r_{R_2}(\tau, 1)$.

As soon as Equation 2 is satisfied with the help of a role r_h, a leader tries to gather all agents that are involved in the processing of CTF. This can be done by either simulating the resources-flow in its usual or reverse direction. This is important because it is necessary to gather all functioning agents that are involved in the processing of CTF in order to bring the system back to a consistent state with respect to the constraints that define an accurate system configuration.

Here, CTF is $[M, E, L]_{[\ldots, B]}$ because of:

$$\begin{aligned} r_{R_2}(\tau, 1).prec.state &= [\ldots, B] \\ r_{R_4}(\tau, 1).postc.state &= [\ldots, B, M, E, L] \end{aligned}$$

In order to contain all agents that are involved in the processing of CTF, R_2 simulates the resource-flow, starting at $r_{R_2}(\tau, 1)$.

Whenever the simulation performs a role of a broken agent, it proceeds with a role r_i that fulfills Equation 2 with a shortest postcondition state. However, before the usual resource-flow simulation continues, the resource-flow is simulated backwards, starting at r_i. When the inverse resource-flow simulation comes across a broken agent, the usual resource-flow simulation proceeds at r_i. This procedure ensures that the simulation gathers as much agents as possible that are involved in the processing of CTF.

Consequently, since C_3, the agent that represents the postcondition port of $r_{R_2}(\tau, 1)$, is broken, $\langle R_2, \tau \rangle$'s resource-flow simulation continues at $r_{R_4}(\tau, 1)$ in reverse direction. Hence, C_4 becomes a member of $\langle R_2, \tau \rangle$.

In the meantime, $\langle R_3, \tau \rangle$ looks for an agent whose roles satisfy Equation 2 and asks R_2 to join the coalition. Subsequently, R_2 wants to recruit R_3, but it notices the clash. As a result, the coalitions are merged as shown in Section III-B; $\langle R_3, \tau \rangle$ is integrated in $\langle R_2, \tau \rangle$.

Table IV
INFORMATION USED FOR RECONFIGURATION

Information	Elements
$\langle R_2, \tau \rangle$	$\{C_2, R_2, R_3, C_4, R_4\}$
CA	$\{R_2, R_3\}$
EA	$\{C_2, C_4\}$
RFA	$\emptyset$
CTF	$[M, E, L]_{[\ldots, B]}$
TFR	$[M, E]_{[\ldots, B]}$
C_{Nd}	$\{M, E, L\}$
C_{Av}	$\{B, E, L, M\}$

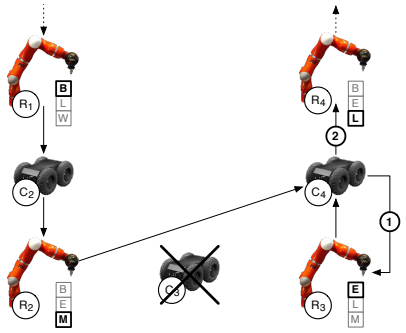

Figure 6. Since C_3 is broken, C_4 assumes the role which transports resources from R_2 to R_3

Now, as CTF stays unchanged, the coalition contains all agents that are involved in processing CTF. The sets C_{Nd} and C_{Av} are $\{M, E, L\}$ and $\{B, E, L, M\}$. As a result, Equation 1 is satisfied.

The reconfiguration proceeds as outlined in Section III-A. In this case, the leader selects an agent combination which defines that $TFR = [M, E]_{[\ldots, B]}$ is the task fragment being reconfigured. As a consequence, r_f equals $r_{R_2}(\tau, 1)$ and r_l equals $r_{R_3}(\tau, 1)$. Thus, CA is $\{R_2, R_3\}$ and EA is $\{C_2, C_4\}$. Although R_4 is a member of $\langle R_2, \tau \rangle$, it is not included in the sets CA, EA, or RFA, which means that R_4's role allocation is not altered during reconfiguration. Table IV shows a part of the leader's knowledge and Figure 6 depicts the reconfigured system.

The next two sections describe in which context the algorithm is implemented (Section IV) and evaluate its qualities with respect to completeness, correctness, and scalability (Section V).

IV. Implementation

The generic concepts of the ODP are implemented in the ODP Runtime Environment (ORE) [5] using the multi-agent system *Jadex* [6]. The ORE distinguishes between two types of Jadex agents: *base agents* that provide the functionality to process resources and to establish the resource-flow, and *reconfiguration agents* that implement the self-organization

mechanism as, for example, the presented coalition formation. Each base agent continuously monitors its state and local constraints which must hold for an overall correct system behavior. Reconfiguration agents are created whenever a reconfiguration is necessary, triggered by their base agents or another reconfiguration agent, and communicate with base agents through a clearly defined interface.

Currently, the algorithm does not calculate the new roles itself. It utilizes a constraint solver such as *Alloy* [7] or *Kodkod* [8] to determine new roles for a specific task fragment. For this purpose, the task fragment to reconfigure as well as the agent sets CA, EA, and RFA are used as input for the calculation. Once the constraint solver's calculation is completed, the coalition formation algorithm combines the reconfigured and untouched roles to a new role allocation. Subsequently, the roles are distributed to the coalition members.

V. Discussion

This section analyses the different qualities of the coalition formation algorithm regarding completeness, correctness, and scalability.

A. Completeness

As it would be the case for every algorithm dealing with local knowledge, the coalition formation algorithm is not complete because there can be a global solution that cannot be found locally. However, the following paragraphs specify situations in which the algorithm comes up with a solution.

Let R be the binary I/O-relation on the set of agents in the system:

$$xRy :\Leftrightarrow y \in x.inputs \vee y \in x.outputs \vee x = y$$

Further, let R^* be the reflexive transitive closure of R and x the leader of a coalition that reconfigures a task τ. Then the coalition formation algorithm always finds an existing solution if there is a set of agents S that is able to reconfigure a sufficiently large task fragment τ_f of τ, where τ_f equals S's CTF:

$$S = \{y \mid xR^*y\}$$

That is because coalitions grow over time by utilizing I/O-relations and, eventually, TFR equals τ_f. Consequently, in the worst case, the coalition equals the *Grand Coalition* [9], which contains every agent in the system, and reconfigures a task fragment $\tau_f = \tau$ that covers the whole task.

In case there is no such set S because of unfavorable I/O-relations (i.e., x cannot recruit additional agents that are necessary for reconfiguration), define the agent set S' that is able to reconfigure a sufficiently large task fragment τ_f of τ and S' as

$$S' = \{y \mid zR^*y \wedge z \in L\},$$

where L is a set of agents. Then the algorithm comes up with a solution if x's coalition is merged with other coalitions being led by agents that form the set L. This can happen if x's coalition has no input or output to an agent outside the coalition, but coalitions that are led by agents contained in L do have inputs or outputs to members of x's coalition and, furthermore, try to recruit some of these agents.

The statement holds as a coalition that is not able to complete reconfiguration waits for clashes with other coalitions.

Otherwise, reconfiguration fails. For example, this can be the case if at the time of reconfiguration the I/O-graph is disconnected, i.e., it is divided into two or more disconnected subgraphs G_i and if a reconfiguration, which takes place in subgraph G_j, needs an agent of subgraph G_k with $j \neq k$.

B. Correctness and Complexity

The coalition formation algorithm is developed in the context of the SAVE ORCA project. The main goal of this project is to give correctness guarantees about the system's behavior despite self-organization mechanisms. As stated in Section IV, currently, a coalition utilizes a constraint solver to generate new roles that fulfill the coalition's TFR. Under the assumption that a system was in a consistent state before a failure occurred, that there are no further failures in the non-reconfigured part of the system, and that the constraint solver works correctly, it can be proved that a global consistent state is restored if invariants are restored locally within the coalition. That is because the coalition is clearly separated from the rest of the system by edge agents, the reconfiguration of the coalition is assumed to be correct, the configuration of the rest of the system is not touched, and the resource-flow is re-established by the use of edge agents.

The algorithm is very complex since for a constant set of coalition members, in the worst case, it has to check the resource-flow on all possible combinations in which the coalition members can apply the capabilities listed in the coalition's CTF. That is why the algorithm tries to keep the coalition as well as CTF small. However, a typical self-organizing resource-flow system provides sufficient degrees of freedom in the form of redundant agents, capabilities, and alternative resource-flows, thus enabling various "valid system configurations". As Section V-C shows, the algorithm's behavior on such a system is promising, especially as it is restricted to local knowledge.

C. Scalability

In this section, the coalition formation algorithm is empirically evaluated. All tests were performed on randomly generated systems in the ORE. The first part of this section shows the connection between redundancy and reconfiguration properties, such as coalition size and runtime[2]. The second part exposes the algorithm's behavior in a situation

[2]Runtime measurements were restricted to coalition formation and therefore did not include the runtime of the constraint solver.

Table V
REDUNDANCY LEVELS FOR SYSTEMS OF SIZE 24

Level	$\#caps/agent$	$\#ins/agent = \#outs/agent$
very low	2	3
low	4	6
medium	6	9
high	8	12
very high	12	18

in which a system is enlarged without changing the number of capabilities, inputs, and outputs per agent.

In order to figure out how redundancy influences the performance of the algorithm, we observed the algorithm's behavior on systems of equal size, i.e., they contained the same number of agents but varying redundancy. Therefore, we randomly generated ten systems. Each system contained 24 agents. Initially, each agent applied one capability within a task of length 24. There were two systems for each of the five redundancy levels that are listed in Table V. The redundancy levels define different degrees of redundancy in a system of size 24, for example, redundancy level *medium* states that each agent had 6 different capabilities ($caps$), 9 inputs (ins), and 9 outputs ($outs$). Additionally, the number of agents having a certain capability equaled the number of capabilities per agent. Thus, for *medium* redundancy level, each capability was assigned to 6 agents. Furthermore, whenever an agent A_i contained an agent A_j in its outputs/inputs, A_i was contained in A_j's inputs/outputs. For each redundancy level, we measured average runtime, coalition size, and number of reconfigured agents that were necessary to complete reconfiguration due to a broken capability of an arbitrary agent. Coalition size may differ from the number of reconfigured agents since a coalition may contain agents that are not useful for reconfiguration.

As can be seen in Diagram 7, in general and unsurprisingly, the coalition formation algorithm benefits from increasing redundancy. As the graphs show, it is particularly advantageous to increase redundancy if it is rare. Furthermore, the runtime curve reveals that runtime is more or less inversely proportional to the level of redundancy. Additionally, the number of reconfigured agents approaches the number of coalition members with increasing redundancy. These numbers almost coincide at *very high* redundancy level. The reason is that the higher redundancy, the more likely a coalition recruits agents suitable for reconfiguration. Further, it can be noticed that coalition size approximates 4 at *very high* redundancy level. Regarding the fact that nearly every coalition that is initiated due to a broken capability has to have a minimum size of 4 (two core and edge agents), the algorithm forms almost minimum coalition sizes. Consequently, there is no need to increase redundancy further since coalition formation and runtime would not behave significantly better.

The second part of the tests was performed on six systems,

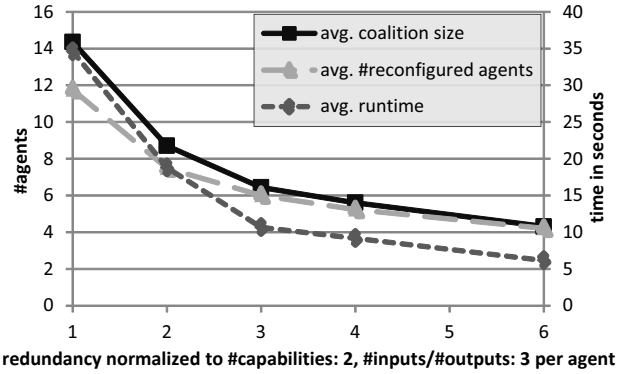

Figure 7. Reconfiguration of systems consisting of 24 agents for different redundancy levels

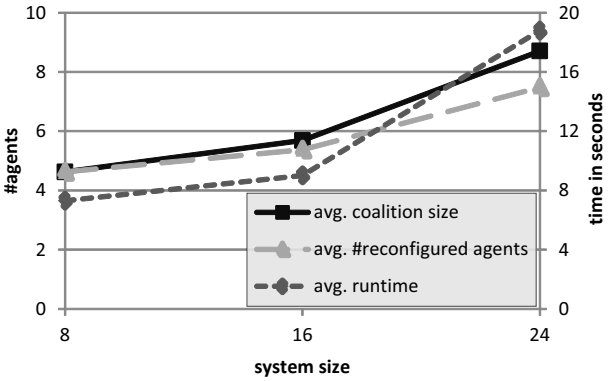

Figure 8. Reconfiguration of systems of different sizes and constant redundancy level *low* ($\frac{\#capabilities}{agent} = 4$, $\frac{\#inputs}{agent} = \frac{\#outputs}{agent} = 6$)

each with redundancy level *low* of Table V. The systems contained either 8, 16, or 24 agents. There were two systems for each system size. The tests were performed in the same manner as for Diagram 7.

Diagram 8 demonstrates that the coalition formation algorithm is able to reconfigure large systems in acceptable time in comparison to reconfiguration runtime for smaller systems, even if the redundancy level is left unchanged and therefore low. However, the runtime curve shows that self-organizing systems consisting of more than 24 agents should feature more flexibility, i.e., redundancy. The diagram also illustrates that coalition size and number of reconfigured agents increases with the size of the system if the number of capabilities, inputs, and outputs per agent stays constant. The larger such a system, the more the number of reconfigured agents differs from the actual size of the coalition. As above, that is because the probability of recruiting unsuitable agents increases with larger systems.

VI. RELATED WORK

There has been much work in the field of coalition formation algorithms in conjunction with multi-agent systems,

particularly for solving the set covering, the set partitioning [9], or the coalition generation problem [10], where the goal is to find suitable coalitions whose members cooperate in the performance of a set of tasks.

[10] describes a coalition as a goal-directed and short-lived organization that internally coordinates its activities in order to achieve the coalition's goals. Additionally, the structure within each coalition is described as flat, optionally featuring a leader that represents the coalition. These characteristics pretty well match the properties of the coalitions which are presented in this paper. However, our coalitions are used for reconfiguration and therefore the typical activities of the coalition formation process, which can be found in [10], as coalition value calculation, coalition structure generation, and pay-off distribution do not take place here.

In [11], multiple agents with simple skills can form coalitions to provide complex skills in a self-organizing evolvable assembly system environment. If a component fails or the task changes, coalitions are formed and rearranged with the help of a so-called ontology agent. This agent holds information about all registered agents which includes information about the agents' skills. Thus, the coalition formation process is not restricted to local knowledge.

Coalition formation in the context of service composition is presented in [12]. The coalition formation process is simplified by using a blackboard architecture that provides a common information space in the form of a blackboard.

By contrast, our algorithm uses completely local knowledge which is possible by utilizing information extracted from the underlying system structure.

A decentralized coordination mechanism for self-organizing, anticipatory vehicle routing is proposed in [13]. However, the system architecture differs from the one used by the coalition formation algorithm as it features a distributed software entity that reflects the real environment and can be explored by software agents on behalf of others.

[14] introduces a design pattern for the engineering of self-organizing emergent systems whose decentralized coordination is based on digital infochemicals. In such systems, agents emit infochemicals into the environment in which they diffuse to other locations and thus enable exchange of information.

The ideas proposed in [13], [14] are also decentralized self-organization mechanisms which make use of the environment as communication medium. The main difference to the algorithm given here is that these mechanisms run permanently to coordinate the systems' components, whereas the coalition formation is only used when deficient parts of the whole system need to be reconfigured.

An algorithm for automatic configuration of applications in the domain of pervasive computing is shown [15]. These are applications that consist of multiple components running on heterogeneous, often resource-limited, specialized, or mobile devices. Each component has structural and resource requirements that specify valid compositions of components in terms of functionalities and local resource requirements of a component's instantiation. The approach for finding valid configurations is based on Distributed Constraint Satisfaction and, therefore, translates requirements into constraints. More precisely, it uses an algorithm for Asynchronous Backtracking presented in [16] as its basis. Because pervasive systems are highly dynamic, the algorithm is able to deal with fluctuations that occur during the execution of the algorithm, e.g., situations in which resources or devices become unavailable. However, the algorithm is currently not capable of reconfiguring local parts of a configuration if a failure is detected during the execution of an application. In contrast to the algorithm presented in this paper, it has to recalculate a complete configuration.

[17] presents a distributed algorithm for generic role assignment in wireless sensor networks. Each sensor node has a local cache table that contains information the algorithm needs for the assignment of roles to nodes. This includes information about properties and roles of the corresponding sensor node as well as of its neighbors. Furthermore, each sensor node holds a role specification that defines roles and rules for how to assign roles to sensor nodes. The algorithm runs on each sensor node and uses information from the node's cache table and role specification. Because information available at sensor nodes is not restricted to local knowledge, whenever a relevant property changes, a sensor node informs its neighbors about this change by broadcasting update messages. A sensor node that receives an update message determines its role and, if necessary, forwards the message to other sensor nodes. Unlike the coalition formation algorithm, which is triggered in case of an invalid role allocation and which guarantees the assignment of valid role allocations, the algorithm in [17] is continuously executed and tolerates invalid role assignments.

Another completely decentralized and ODP-based reconfiguration mechanism is introduced in [18], where reconfiguration propagates through the system like a wave until a new role allocation is found. This mechanism differs from coalition formation in that reconfiguration is done by "swapping" roles between the agents until a new valid configuration is found without forming groups or gathering local knowledge. However, until now, pending points are reconfiguration due to broken inputs or outputs and the breakdown of an entire agent. This approach will be compared to the coalition formation algorithm in more detail in a forthcoming paper.

VII. Conclusions and Future Work

This paper introduced a coalition formation algorithm as a means of decentralized reconfiguration of self-organizing resource-flow systems. In the course of reconfiguration, the algorithm is entirely restricted to local knowledge. The discussion in Section V showed that the algorithm scales very well in systems which are prepared for self-organization,

that is, which have a medium degree of redundancy in relation to their system size. The algorithm combined with the techniques developed in the project SAVE ORCA, in particular the runtime environment, enables self-reconfiguration in self-organizing resource-flow systems with a guaranteed correct behavior. Furthermore, the principle of this algorithm, i.e., being able to cope with local knowledge by making use of domain knowledge and the topology of the system that is to be reconfigured, is applicable to other system classes, too.

Future work includes a further improved version of the coalition formation algorithm. More precisely, it will be able to determine new role allocations itself, instead of relying on a constraint solver. In this scenario, a result checker will validate the correctness of the results produced by the coalition formation algorithm so that the behavioral guarantees still hold. Scalability is improved due to reducing the solution space and therefore overcoming the limitations of the constraint solver. In addition, the future version of the algorithm provides potential for optimizing the role allocation, given that a constraint solver only calculates one correct, but not always optimal, solution.

ACKNOWLEDGMENTS

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

REFERENCES

[1] A. Hoffmann, F. Nafz, H. Seebach, A. Schierl, and W. Reif, "Developing Self-Organizing Robotic Cells using Organic Computing Principles," in *Workshop on Bio-Inspired Self-Organizing Robotic Systems, Proceedings of the International Conference on Robotics and Automation (ICRA 2010)*, 2010.

[2] 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.

[3] 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)*, 2009.

[4] M. Güdemann, F. Nafz, F. Ortmeier, H. Seebach, and W. Reif, "A specification and construction paradigm for Organic Computing systems," in *Proceedings of the Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE Computer Society Press (2008), 2008, pp. 233–242.

[5] 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*. IEEE/ACM Digital Library, 2009.

[6] L. Braubach, A. Pokahr, and W. Lamersdorf, "Jadex: A BDI Agent System Combining Middleware and Reasoning," in *Software Agent-Based Applications, Platforms and Development Kits*, September 2005, pp. 143–168.

[7] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[8] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *In Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. Wiley, 2007, pp. 632–647.

[9] O. Shehory and S. Kraus, "Methods for task allocation via agent coalition formation," *Artificial Intelligence*, vol. 101, no. 1-2, pp. 165–200, May 1998.

[10] T. Rahwan, S. Ramchurn, N. Jennings, and A. Giovannucci, "An Anytime Algorithm for Optimal Coalition Structure Generation," *Journal of Artificial Intelligence Research*, vol. 34, no. 1, pp. 521–567, 2009.

[11] M. Pechoucek, V. Marik, and O. Stepankova, "Coalition Formation in Manufacturing Multi-Agent Systems," in *Proceedings of the 11th International Workshop on Database and Expert Systems Applications (DEXA)*, September 2000, p. 241.

[12] I. Müller, R. Kowalczyk, and P. Braun, "Towards Agent-based Coalition Formation for Service Composition," in *Proceedings of the IEEE/WIC/ACM international conference on Intelligent Agent Technology (IAT)*, December 2006, pp. 73–80.

[13] D. Weyns, T. Holvoet, and A. Helleboogh, "Anticipatory Vehicle Routing using Delegate Multi-Agent Systems," in *Intelligent Transportation Systems Conference*, 2007, pp. 87–93.

[14] H. Kasinger, B. Bauer, and J. Denzinger, "Design Pattern for Self-Organizing Emergent Systems Based on Digital Infochemicals," in *EASE '09: Proceedings of the 2009 Sixth IEEE Conference and Workshops on Engineering of Autonomic and Autonomous Systems*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 45–55.

[15] M. Handte, C. Becker, and K. Rothermel, "Peer-based Automatic Configuration of Pervasive Applications," *International Journal of Pervasive Computing and Communications*, vol. 1, no. 4, pp. 251–264, 2005.

[16] M. Yokoo, E. Durfee, T. Ishida, and K. Kuwabara, "The Distributed Constraint Satisfaction Problem: Formalization and Algorithms," *IEEE Transactions on Knowledge and Data Engineering*, vol. 10, no. 5, pp. 673–685, 1998.

[17] C. Frank and K. Römer, "Algorithms for Generic Role Assignment in Wireless Sensor Networks," in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems (SenSys '05)*. New York, NY, USA: ACM, 2005, pp. 230–242.

[18] J. Sudeikat, J.-P. Steghöfer, H. Seebach, W. Reif, W. Renz, T. Preisler, and P. Salchow, "Design and Simulation of a Wave-like Self-Organization Strategy for Resource-Flow Systems," in *4th International Workshop on Multi-Agent Systems and Simulation (MAS&S 2010)*, August 2010.