OPEN ACCESS
OPUS AUGSBURG

Universität Augsburg
Universitätsbibliothek

Constraining self-organisation through corridors of correct behaviour: the Restore Invariant Approach

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

Angaben zur Veröffentlichung / Publication details:

Nafz, Florian, Hella Seebach, Jan-Philipp Steghöfer, Gerrit Anders, and Wolfgang Reif. 2011. "Constraining self-organisation through corridors of correct behaviour: the Restore Invariant Approach." In Organic computing — a paradigm shift for complex systems, edited by Christian Müller-Schloer, Hartmut Schmeck, and Theo Ungerer, 79–93. Basel: Birkhäuser.
https://doi.org/10.1007/978-3-0348-0130-0_5.

Nutzungsbedingungen / Terms of use: licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under these conditions:
Deutsches Urheberrecht
Weitere Informationen finden Sie unter: / For more information see:
https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publiz/

Chapter 1.5
Constraining Self-organisation Through Corridors of Correct Behaviour: The Restore Invariant Approach

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

Abstract Self-organisation aspects and the large number of entities in Organic Computing (OC) systems make them extremely hard to predict and analyse. However, the application of OC principles to, e.g., safety critical systems, is usually not conceivable without behavioural guarantees. In this article, a rigorous approach called the *Restore Invariant Approach* is presented, which provides a specification paradigm and a formal framework that allows to give guarantees for a system despite of self-organisation. The approach provides a method for specifying unwanted system states by constraining the system and defining a corridor of correct behaviour. Furthermore, a decentralised algorithm for monitoring and restoring the invariant based on coalition formation is presented.

Keywords Self-organisation · Formal verification · Decentralised algorithms

1 Introduction

Today, self-organisation is used to cope with the rising complexity of systems that often consist of a vast number of entities. These systems try to fulfil a global goal, often realised by a combination of local decisions and interactions with other entities. They are not influenced by an external control and they have the ability to find their system configuration on their own. This configuration is changed dynamically during runtime, depending on the current situation. These kinds of systems are desirable as self-organisation makes them highly resilient, adaptive, and robust. This makes them ideally suited for applications in which safety is a critical issue and in domains that have to deal with changing environments and unexpected disturbances. However, such domains require rigorous techniques for safety analysis or verification in order to get a system approved for deployment. Especially in automotive systems, aviation, railway, or production automation, proofs are required in order to show that the systems behave as intended.

Usually, the behaviour of a dynamic and autonomous OC-system is hard to predict and the concrete reaction to a particular situation is not necessarily deterministic. Therefore, techniques need to be developed that deal with the unforeseeable outcome of self-organisation processes. It is not feasible to specify the system by

explicitly listing all states that are acceptable or wanted. The *Restore Invariant Approach (RIA)*, developed by the project SAVE ORCA[1] and presented in this article, provides a different approach by specifying a corridor of correct behaviour. This is done by defining an invariant separating "good" system states from "bad" ones. The system tries to maintain this invariant as long as possible in order to stay within this corridor. The property that the system behaves correctly as long as it evolves within the corridor, is then subject to verification.

The verification techniques based on the specification of the corridor allow giving behavioural guarantees for an OC-system. However, the proof of these guarantees hinges on the correctness of the algorithms that are used for configuring the system in order to stay within the corridor. Such algorithms make use of the system's flexibility and its degrees of freedom in order to find new configurations to compensate system failures. Here, a decentralised algorithm for local reconfiguration of OC-systems based on coalition formation is presented. It uses local knowledge and does not need a central component for reconfiguration. The correctness of the reconfiguration algorithm is verified by the use of a result checker that assesses a configuration before it is enacted in the system.

This article is structured as follows: In Sect. 2 the *Restore Invariant Approach* is presented; it provides a method for the specification of OC-systems. Further, the framework for formal verification is sketched. An example scenario, which is used in the subsequent sections to illustrate the approach, is presented in Sect. 3. The definition of corridors of correct behaviour is described in Sect. 4. Then, a decentralised reconfiguration algorithm is detailed in Sect. 5. The article concludes with a summary of results and an outlook.

2 The Restore Invariant Approach

In this section the *Restore Invariant Approach (RIA)* is presented, which provides a specification technique for behavioural corridors and a core concept for treating OC-systems. The principle of the *RIA* is that the behaviour of an OC-system is specified by a predicate logic formula (called *invariant*), which defines the behavioural corridor of the system. The system's goal is to stay within this corridor as long as possible. Whenever the corridor is left, which corresponds to a violation of the invariant, the system starts a self-organisation phase and reconfigures itself in order to get back into the corridor and to make the invariant hold again. *RIA* also provides the basis for formal verification for ensuring functional correctness.

This idea of *RIA* leads to a two-layered view of an OC-system: one layer contains the functional part of the system and is often called the *system under observation and control* [6]; the other one contains the self-x intelligence and is responsible for monitoring the system and for reconfiguration in case of failures or disturbances. This is often realised with an Observer/Controller (O/C) architecture [12]. Figure 1 shows such an architecture.

[1] Formal Modelling, Safety Analysis, and Verification of Organic Computing Applications.

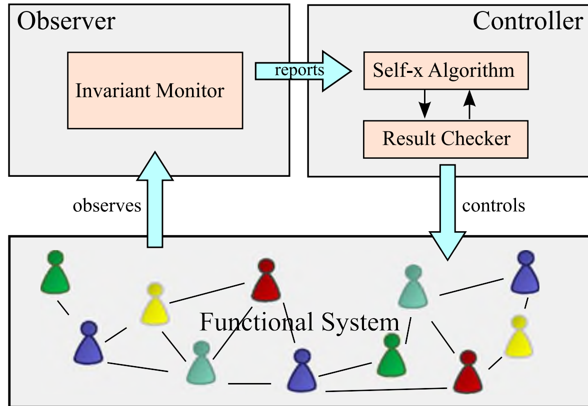

Fig. 1 Architectural view of an OC-system: An Oberserver is monitoring the functional system and reports invariant violations to the Controller. This is reconfiguring the system accordingly

The O/C layer can be either one O/C component monitoring the whole system or distributed Observer/Controllers on top of each agent, as described later in this article. The *Invariant Monitor* observes the system; whenever the specified corridor is left (i.e., the invariant is violated), it reports this and the current system state to the controller. The controller then tries to find a new configuration of the system that fulfils the invariant and leads back into the corridor. In OC-systems bio-inspired algorithms are often used. These algorithms are not necessarily sound nor complete. Therefore, a *Result Checker* component checks the output of the mechanism before forwarding it to the functional system, ensuring that only results that are consistent with the invariant are relayed.

In the following, a formal view of *RIA* is given. Further, the connection to system verification and the resulting proof obligations are presented.

2.1 A Formal View on the Restore Invariant Approach

As already mentioned above in the informal description of *RIA*, OC-systems have to deal with failures and environmental disturbances. These disturbances force the system into a state where it cannot fulfil its functionality. Therefore, two disjoint sets of states of an OC-system can be distinguished:

- a set $\mathcal{S}_{func}$ of *functional* states, in which the system can provide the desired functionality,
- a set $\mathcal{S}_{reconf}$ of *erroneous* or *reconfiguration* states in which the system cannot provide the functionality and a reconfiguration has to take place to get back into a state within $\mathcal{S}_{func}$.

Formally, the execution of an OC-system is the set of all possible traces. One trace out of this set describes a sequence of states and one possible temporal evolution of the system over time.

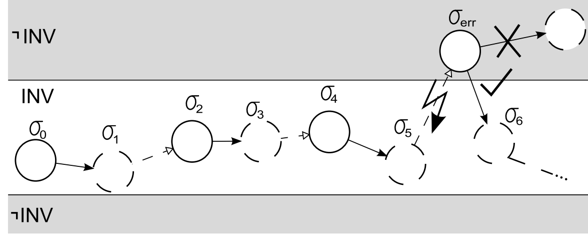

Fig. 2 Behavioural corridors of an OC-system: An invariant violation is compensated by system itself

The advantage of an OC-system is that it has the ability to compensate failures to get back into a functional state where it meets its requirements again. Traditional systems without self-x properties and any degree of freedom have traces in which the system gets into an error state, e.g., because of a component failure, and cannot recover from there.

Figure 2 shows a sample trace of a system. $\mathcal{S}_{func} = \{\sigma_0, \ldots, \sigma_6\}$ is the set of functional states that are within the behavioural corridor. All states that are not in the corridor belong to the set of reconfiguration states. Whenever a failure occurs, the system starts a self-reconfiguration and reaches a state within $\mathcal{S}_{func}$, which shows correct functional behaviour again.

This view is similar to the one proposed in Chap. 1.1. However, what is called *dead space* there is not explicitly considered. Here, states in which the system cannot successfully be reconfigured anymore are subsumed under $\mathcal{S}_{reconf}$. It is assumed that the system continuously tries to find a way to get back to the corridor, but fails to do so.

The *RIA* idea is to define the corridor of correct behaviour by defining a predicate *INV* called *invariant*, which holds in all functional states and does not hold otherwise. The predicate is called invariant as the system's goal is that this predicate holds on the entire trace. The set of states in the corridor can therefore be defined as $\mathcal{S}_{func} := \{\sigma \in \mathcal{S} | INV(\sigma)\}$. Traces consisting only of states in $\mathcal{S}_{func}$ are in some sense correct or "good" traces within the corridor and lead to expected behaviour. It is desired that an OC-system has only traces that consist of functional states or—whenever a failure occurs and it enters a state out of $\mathcal{S}_{reconf} := \mathcal{S} \setminus \mathcal{S}_{func}$—there will eventually be some state $\sigma \in S_{func}$ later in the trace. This property can be formalised as a temporal logic property $\Box INV \vee \Box(\neg INV \rightarrow \Diamond INV)$. In essence, this means that the system reconfigures. The temporal operator $\Box INV$ states "*INV* holds now and will *always* hold in the future", while $\Diamond$ *INV* means "*INV* holds now or will *eventually* hold in the future". For more information about the temporal logic used here and its operators, please refer to [4].

This formula can also be used for the specification of the self-reconfiguration ($Spec_{self\text{-}x}$) mechanism of the system. The formula specifies the effect of self-organisation and its properties. Again, only properties of the result are specified, not a detailed solution. This reflects exactly the idea of the behavioural corridor, where no exact configurations are specified, just required properties.

Self-organisation after a component failure decreases redundancy, as, e.g., broken hardware components cannot be replaced or resources of an agent are not usable

anymore. Nevertheless, the system's functionality can still be provided, e.g., because another component can take over. But as redundancy is not unlimited, it also means that there is some point at which restoring the invariant is not possible anymore. For a realistic (non-perfect) self-organisation, we need to modify the specification by adding a predicate Θ that states that no solution is possible anymore or some even weaker property, e.g., that no solution was found. This can be seen as the separation of *survival space* and *dead space* from Chap. 1.1. With some simplifications, the specification of the reconfiguration then looks as follows:

$$Spec_{self\text{-}x} := \Box(\neg INV \rightarrow \diamond(INV \vee \Theta))$$

The predicate Θ can be seen as a quality predicate. Depending on how it is formulated, the used algorithms can fulfil the specification or not. For example, the weakest Θ states "Algorithm result is `found-no-solution`" whereas the strongest is $\Theta = false$ ("Reconfiguration is always possible and successful"). Usually, properties in between, like "Enough redundancy is available" or "A functional state is reachable", are used for the specification.

2.2 Behavioural Guarantees

The separation of reconfiguration and functional parts of a system and the specification of behavioural corridors by invariants which are separating 'good' from 'bad' states build the foundation for verification of functional correctness and thus for behavioural guarantees for highly dynamic OC-systems. The underlying formal foundation [9, 10] allows a separated verification approach, in which verification of the used reconfiguration mechanism can be performed independently of the verification of the system's properties by using just the specification of the self-reconfiguration instead of the actual implementation. This leads to two steps in the verification process:

- Verification of expected properties P based on the formal model of the functional part of the system SYS_{func} relative to the specification of the self-reconfiguration $Spec_{self\text{-}x}$:

$$SYS_{func} \wedge Spec_{self\text{-}x} \models P$$

- Verification of the self-reconfiguration mechanism o/c_{impl} using the invariants described in Sect. 2.1 against its specification $Spec_{self\text{-}x}$:

$$o/c_{impl} \models Spec_{self\text{-}x}$$

Verification of the Functional System

Usually, the functional system consists of several components running in parallel. A component is specified as a transition system and has trace-based semantics which

is in line with the previous formal considerations. The complete functional system is the parallel composition of all components. More precisely, an interleaving semantics is used. Properties are formalised with Interval Temporal Logic (ITL), which explicitly includes the environment. This allows for a separation of system and environment. Further, an arbitrary environment can be considered without stipulating syntactic restrictions for the formulae describing the behaviour. More details on the formal semantics and the used logic can be found in [4].[2]

As OC-systems have a potentially vast number of components that are not predefined and can change during runtime, a monolithic verification of the whole system is not feasible. Therefore, a compositional method is used, in which parts of the systems are regarded separately before the analyses are combined to make statements for the complete system. This technique allows to give global guarantees by local reasoning about individual components. A compositionality theorem was proved that ensures correct lifting of the local properties to a global level and that defines the required proof obligations. The application to systems with self-x properties is presented in [10], which provides more details about compositional verification of OC-systems with *RIA*. With this approach, the verification of the functional system can be performed offline without the need of explicit consideration of the actual self-reconfiguration algorithm used.

Verification of Self-x Mechanisms by Verified Result Checking

The second step is to verify that the used self-reconfiguration mechanism is correct with respect to its specification. In OC-systems bio-inspired algorithms like genetic algorithms [7], decentralised algorithms like the one presented in Sect. 5, or learning techniques, e.g., neural networks or learning classifier systems, are often used. These algorithms do not necessarily return valid and correct results, nor are they always sound or complete. A technique to avoid direct verification of those complex algorithms and an alternative to online verification of the complete system is the use of result checking [5]. The idea is to add a short program *RC* (*Result Checker*) that checks the output of the self-reconfiguration algorithm.

Specification of **Result Checker** (*RC*)	
input	configuration (*conf*) of o/c_{impl}
output	'correct' if *Inv*(*conf*), 'incorrect', otherwise

If the calculated configuration (*conf*) of the algorithm restores the invariant, the checker just forwards the result. If *conf* violates the invariant, the result is blocked and feedback is provided to the self-reconfiguration algorithm. This reduces the

[2]Note that application of *RIA* and the general verification steps are independent of the logic used for system specification and formalisation of properties.

verification task of the O/C implementation to the formal verification of the result checker RC. This offline, a priori verification has to prove the following proof obligation:

$$RC \models Spec_{self\text{-}x}$$

If the result checker adheres to the specification of the reconfiguration process, i.e., to the properties that are required after a reconfiguration was performed, the result checker itself is formally correct and can thus detect faulty configurations.

3 Example Scenario

A running example to illustrate the application of the *RIA* is a set of self-organising resource-flow systems. Instances are, e.g., applications in production automation or logistics. The components of resource-flow systems can be described by the Organic Design Pattern (ODP), as depicted in Fig. 3. *Agents* are the main components in these systems, processing *resources* according to a given *task*. Every *agent* has several *capabilities*, which can be produce, process, or consume *capabilities*. Hence, the *task* is a sequence of *capabilities*, beginning with a *produce* capability and ending with a *consume* capability. Furthermore, each agent knows other agents it can exchange *resources* with. This is expressed in the *inputs* and *outputs* relations.

The *role* concept is introduced to define correct resource-flows through the system. Roles are assigned to agents and specify what the agent's part of the task is. A role is composed of a *precondition* telling the agent from which (*port*) it gets a resource and when the role can be applied, a sequence of capabilities (*capabilitiesToApply*) that should be applied if the role is executed, and a *postcondition* defining where the resource has to be handed over.

Agents can have multiple roles and select one when another agent wants to hand over a resource. The agent accepts resources only if the situation matches one of the role's precondition. This role is selected and the resource is processed according to

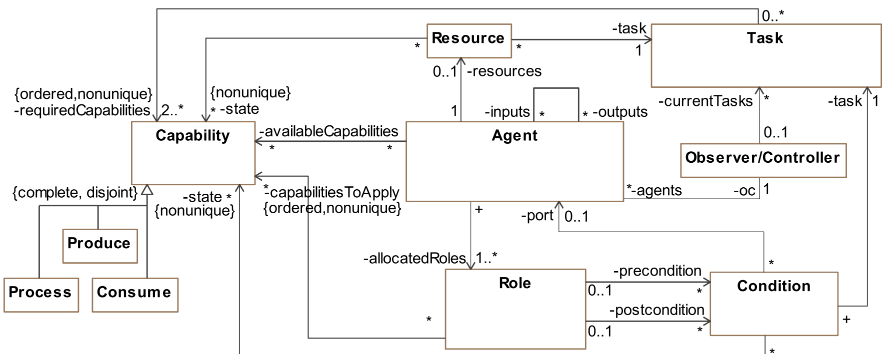

Fig. 3 Components of resource-flow systems

the capabilities defined in the role. At last, the agent tries to give the resource to the next agent, defined in the postcondition.

In this scenario, self-organisation is achieved by role allocation. In case of a failure, the system searches for a new valid role allocation. For a more elaborate model and further details on the software engineering aspects of self-organising resource-flow systems, please refer to Chap. 2.2.

4 Defining Corridors of Correct Behaviour

As described in Sect. 2, *RIA* proposes the implicit definition of correct states by specifying an invariant that defines the behavioural corridor, instead of explicitly listing all possible configurations a system may choose. The invariant can be regarded as a set of constraints on the system states and the system's configuration. In the example scenario, the invariant specifies valid role allocations with respect to the current system state. For instance, one predicate of the invariant may state that only capabilities are assigned that are available at the agent:

$$\begin{aligned} Inv_1(sys) :\Leftrightarrow\ & \forall ag \in Agents, \forall r \in ag.allocatedRoles: \\ & r.capabilitiesToApply \subseteq ag.availableCapabilities \end{aligned}$$

This predicate can be monitored locally by each agent's *Invariant Monitor*. In case an agent loses a capability, a reconfiguration is triggered. Another part is to specify the resource-flow property, which means that roles must be connected. So if one agent has a role that tells it to hand over a resource to another agent, the other agent must have a consistent role. This means that the agent is defined as one possible input.

$$\begin{aligned} Inv_2(sys) :\Leftrightarrow\ & \forall ag_{send} \in Agents, \forall r_{send} \in ag_{send}.allocatedRoles: \\ & ag_{rec} = r_{send}.postcondition.port \\ & \rightarrow \exists r_{rec} \in ag_{rec}.allocatedRoles : (r_{rec}.precondition.port = ag_{send}) \end{aligned}$$

The complete invariant is retrieved by the conjunction of all predicates. The predicates can be divided into two types. One type (called *'monitoring predicates'*) are invariants that have to be monitored during the system's execution (e.g., Inv_1), because they depend on variables that are also influenced by the environment and that can change their validity, e.g., when a failure occurs. The second type are invariants that are necessary for obtaining valid results, but they do not contain variables that can change during execution and are only assigned during reconfiguration (e.g., Inv_2 only uses roles). However, these constraints can be violated, e.g., when a complete agent breaks down. Such constraints are called *'consistency predicates'*.

The constraints are usually derived from OCL (Object Constraint Language) constraints annotated to the software engineering models. Transformation to a predicate logic formula is a straight-forward task and similar to the technique used, e.g., in [1].

The challenge in a self-reconfiguration phase is to find a valid assignment of roles with respect to the current system state. As the invariant constrains the assignments to the system variables, the reconfiguration problem can also be formalised as a constraint satisfaction problem [15]. Thus, a central reconfiguration mechanism that has global knowledge of the agents and their current state can be used to solve this problem. For an implementation, common off-the-shelf constraint solvers (e.g., Kodkod [9, 14]), genetic algorithms, or learning techniques can be used. However, a central control introduces a single point of failure as well as a bottleneck and often the agents have only local knowledge. In such situations, the problem has to be solved in a decentralised fashion. In the next section, a mechanism is presented which realises a fully decentralised Observer/Controller layer.

5 Decentralised Restoration of Invariants

The algorithm for reconfiguration presented here is characterised by three features. First, it is specialised and optimised for the class of self-organising resource-flow systems. While it is not limited to this class, it makes use of the property that roles establish a resource-flow in the system and, therefore, a topology the algorithm can exploit. Second, the algorithm is robust against failures, like partial failures of an agent (e.g., broken capabilities) or the breakdown of a complete agent. Finally and most importantly, the algorithm is decentralised and does not need global knowledge or a central control. One advantage of a local reconfiguration is that the rest of the system is not affected by a reconfiguration and therefore can continue working. Further, the algorithm is based on an asynchronous communication model.

5.1 Coalitions for Local Reconfiguration

The concept of the algorithm is to form a group of agents, called *coalitions*, that are able to restore the violated invariant locally. More detailed, the algorithm reacts to the breakdown of capabilities, the complete breakdown of an agent, and lost connections between agents. If one of those circumstances is detected, it reconfigures the system locally such that the invariant holds again for the coalition. In the following, the process of building the coalition is called *coalition formation*. The notion of coalitions and coalition formation originate in the field of multi-agent systems, where significant work on the topic was done. Particularly, this includes proposals for solving the set partitioning problem [13] or the coalition generation problem, where the challenge is to find suitable coalitions to perform a task together. In [11], a coalition is described as a goal-directed and short-lived organisation that internally coordinates its activities in order to achieve the coalition's goal. The structure of a coalition is flat, with an optional leader that represents the coalition. The coalitions presented in the following adhere to the same principles and have similar properties. The goal of a coalition is to locally restore the invariant by finding a set of agents that have every capability required for repairing the resource-flow.

5.2 Coalition Formation Strategy

As soon as an agent Ag_i recognises an invariant violation, it starts a reconfiguration by creating a new coalition C_i and becoming its leader. The objective of Ag_i is to find enough agents to be able to reconfigure the system in order to compensate the failure. In terms of the ODP, this means that processing of resources according to task τ can continue.

During the coalition formation, the algorithm exploits the existing system structure given by the current role allocation and the resource-flow defined by them, while being limited to local knowledge. Each coalition is led by one coalition member, which coordinates the coalition in order to find a set of agents that can restore the violated invariant locally. Every time a new agent is included into the coalition, the coalition's knowledge grows by the knowledge of the added agent, which consists of its allocated roles, available capabilities, and its possible inputs and outputs to other agents.

To be able to reconfigure, Ag_i must extend the coalition in order to get more degrees of freedom and increase flexibility. Therefore, the leader is asking other agents to join its coalition. If the coalition has enough members to reconfigure a continuous segment of τ, it stops acquiring further agents. This segment is called *connected task fragment(CTF)*. It is necessary to ensure that a consistent solution can be calculated and that the local solution leads to a global restoration of the invariant. Next, the agent identifies a subsequence of this segment (*task fragment to reconfigure*, *TFR*) that actually needs to be reconfigured, because not necessarily all agents that joined the coalition need new roles.

After that, leader Ag_i searches agents that are responsible for maintaining the resource-flow between the coalition and the parts of the system not involved in the reconfiguration. These agents are called *edge agents(EAg)* as they enclose the coalition with regard to the resource-flow. Edge agents are not necessarily new agents. Instead, agents that are already part of C_i can be edge agents (see Fig. 4). Agents that are not edge agents but are directly involved in the reconfiguration are called *core agents(CAg)*.

Subsequently, the resource-flow within the coalition is re-established. If this is not possible because the possible inputs or outputs do not allow a connection, new agents are recruited. These agents are called *resource-flow agents(RFAg)*. As soon as the resource-flow is established, leader Ag_i disbands the coalition. At a glance, the different agent types are defined as follows:

- *core agents(CAg)*: contains all agents of a coalition C_i that should apply at least one capability within *TFR* and all agents that were responsible for establishing the resource-flow within *TFR* before the reconfiguration started.
- *edge agents(EAg)*: consists of all agents that are responsible for a consistent interconnection to the rest of the system and whose precondition or postcondition must not change during reconfiguration.
- *resource-flow agents(RFAg)*: contains all agents that are needed to establish the resource-flow for the new role allocation and that are not already included in *CAg* or *EAg*.

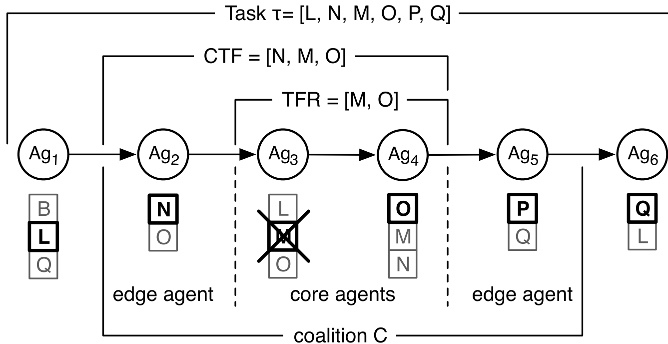

Fig. 4 Terminology and example of a coalition

The abstract example in Fig. 4 illustrates the terminology used. In the depicted situation, Ag_3 loses capability M that is needed to process resources with task τ. The resulting coalition then contains Ag_2, Ag_3, Ag_4, and Ag_5.

There are two types of reasons that trigger an agent to start a reconfiguration and form a coalition. One is an invariant violation that can be detected by one agent itself, e.g., a broken capability. The other type of invariant violation can only be detected by adjacent agents, e.g., when a complete agent breaks down. For both cases, the algorithm has a different strategy for recruiting new agents to its coalition, which is explained in the following.

5.3 Strategy for Local Variable Violation

One of the most important features for making a system more robust is reconfiguration caused by the loss of a capability, which is needed to execute an assigned role. Of course, loss of a capability diminishes redundancy and can only be compensated if there is at least one redundant capability in the system. This kind of failure violates the invariant, more precisely Inv_1 (see Sect. 4), which states that all the capabilities needed for an allocated role must be available. If a violation of the invariant on a local variable is detected, the agent starts inviting predecessors and successors with respect to the resource-flow defined by its roles.

In the situation depicted in Fig. 4, agent Ag_3 loses capability M. It creates a coalition C_3 with itself as leader. The *CTF* is $[M]$. Additionally, the agent has variable Cap_{nd} containing all needed capabilities to fulfil *CTF*, which only contains M at this moment. The goal of every leader is that the coalition's available capabilities Cap_{av} are a superset of Cap_{nd}.

In order to reach this goal and to compensate the failure, Ag_3 has to enlarge the coalition to a point where it finds an agent that has capability M. As the agents only have local knowledge, they cannot just pick the agent that has the missing capability. Instead, they have to iteratively extend their coalition until the agent that can apply M is finally found. Therefore, each leader alternately sends requests to join

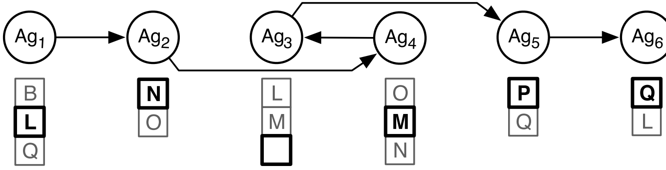

Fig. 5 Example reconfiguration by coalition formation

the coalition to agents contained in the inputs and outputs of coalition members, because these are the only agents known by the coalition. It first sends requests to its direct neighbours, with respect to the currently established resource-flow, which are in the sample scenario agents Ag_2 and Ag_4. So, first Ag_2 is asked and the coalition expands to $C_3 = \{Ag_2, Ag_3\}$. As Ag_2 has applied capability N until now, CTF changes to $[N, M]$. Further, Ag_2 transfers all its knowledge about inputs and outputs to Ag_3, which extends the amount of agents Ag_3 can request. Information about the available capabilities and the currently assigned roles of Ag_2 is also transferred. Because agent Ag_2 only has capabilities O and N, Ag_3 has to search further and asks Ag_4 next, which finally has the needed capability and $Cap_{nd} \subseteq Cap_{av}$ is satisfied. The situation now is: $C_3 = \{Ag_2, Ag_3, Ag_4\}$ and $CTF = [N, M, O]$.

Ag_3 now calculates a possible assignment of the capabilities and determines TFR, which is $[M, O]$ here. Ag_2 does not have to change its capability in order to solve the problem. The corresponding set of *core agents* is $CAg = \{Ag_3, Ag_4\}$. As the connection between reconfigured agents and non-reconfigured agents must be consistent, Ag_2 is an *edge agent* and Ag_5 is also requested to join the coalition: its role has to change because its input connection will be different afterwards. The set of *edge agents* is therefore defined as $EAg := \{Ag_2, Ag_5\}$. The coalition is now reconfigured and the new roles are allocated. Figure 5 shows the resulting capability assignment and the configured, new resource-flow. As soon as this is done, Ag_3, as the coalition's leader, disbands the coalition and processing continues.

5.4 Strategy for Complete Breakdown of an Agent

A complete breakdown of an agent violates the invariant, specifically predicate Inv_2. Because this violation cannot be detected locally by one agent, there is a slight difference in coalition formation and reconfiguration.

First of all, the breakdown must be detected, which is realised by "alive" messages exchanged by agents that are connected by the resource-flow. If, for example, agent Ag_3 in Fig. 5 completely breaks down, the agents Ag_2 and Ag_4 will detect this sooner or later as they no longer receive messages from the adjacent agent. Independently from each other, they both start a reconfiguration. As they do not know the broken agent's role and the capability it applied, they now have two objectives. Firstly, identify the capabilities that the agent has performed and, secondly, find agents that provide the missing ones. By utilising the system structure (i.e., the currently defined resource-flow) it is possible to recover the roles of the lost agent if

the knowledge of both coalitions is merged. Therefore, Ag_4 needs to find the predecessor (Ag_2) and Ag_2 the successor (Ag_4) of Ag_3.

Thus, Ag_2 searches for an agent that applies a role that is applied after its own one. This can be checked because the expected incoming resource state is part of each role. In contrast to a local violation, Ag_2 does not recruit its neighbours with respect to the resource-flow. That is because one neighbour is the faulty agent Ag_3, the other one is a predecessor (Ag_1), and further search in this direction will (most likely) not return a successor. For this reason, Ag_2 prefers agents that are not direct neighbours. As soon as Ag_2 finds a successor of Ag_3, it starts to follow the resource-flow backwards until it reaches Ag_4.

Since Ag_4 also detects the violation and starts a coalition on its own, there are two coalitions in the system. When one coalition tries to recruit an agent of the other one, the coalitions clash. If this happens, the coalitions are merged and one of the former leaders becomes the new leader of the united coalition. The selection is achieved by a standard leader election algorithm.

In addition, there are other situations where coalition clashes have to be resolved. This can be the case if, for example, two different failures occur at the same time, which results in two independent coalitions. If these two coalitions meet, they are merged as in the situation above. However, in larger systems, coalitions often do not meet, for example, as failures occur on opposite ends of the system. In such a case, there are two reconfigurations in parallel without recognising each other. As they do not interfere, they fix their failures independently and, therefore, restore the invariant of the complete system.

5.5 Discussion

The algorithm can reconfigure a system after an invariant violation locally without the need of a global Observer/Controller. Further, it can deal with strongly restricted knowledge of the individual agents because it distinguishes the type of invariant violation. These properties allow the application to systems of unlimited size. Nevertheless, the size and the runtime of the coalition is closely connected to the redundancy distribution within the system. If the next available capability is far away, the resulting coalition will be larger and reconfiguration will be more complicated than when it is available in the direct neighbourhood.

Under the assumption that there are no further failures in the non-reconfigured part of the system, it can be proved that, if the invariant is restored locally within the coalition, then the invariant also holds and is restored for the complete system.

The principle of this algorithm is also applicable to other system classes. The idea is to make use of domain knowledge and the topology of the system to be able to get by with local knowledge only. In this case, the knowledge that can be derived by the currently assigned roles and the knowledge gathered by the neighbours allows to solve the reconfiguration problem locally by forming a coalition of a few agents.

The correctness of the algorithm is ensured by the verified "Result Checker" component (see Sect. 2.2) which assesses the configuration before it is enacted in the

system. A more detailed view on the algorithm and discussion about some situations is presented in [2].

6 Summary and Outlook

In this article, the *Restore Invariant Approach* was presented. It provides a paradigm for the specification and analysis of OC-systems. *RIA* enables a separate treatment of self-x and functional behaviour and, therefore, allows to give behavioural guarantees despite of self-organisation. The behaviour of a system is not specified by giving explicit statements. Instead, behavioural corridors are specified to exclude unwanted behaviour. This is done by constraining the system states by so-called invariants, which the system then tries to keep satisfied.

To show the applicability of the presented theory, all results were applied to the system class of resource-flow systems and a decentralised algorithm based on local knowledge for the reconfiguration of such systems was introduced. A detailed view of the system class and a software engineering guideline based on *RIA* that allows to actually build real world systems in a repeatable and disciplined fashion is presented in Chap. 2.2.

The underlying formal framework allows to prove that a system behaves as intended and will not show undesirable behaviour without restricting its freedom to adapt to unforeseen situations. This is essential if applications in safety-critical domains should benefit from the advantages of Organic Computing techniques in the future. The framework provides methods and tools for formal specification and compositional verification of systems with self-x properties [10]. The presented technique allows the verification of systems with an unknown amount of agents and an explicit treatment of the environment. This is an important feature when considering OC-systems, which, by definition, interact with their environment. Traditional systems are developed with respect to the environment they will be deployed in, so the possible influences are known and considered during design time. OC-systems, in contrast, are able to deal with different, unforeseeable environments and therefore these considerations are shifted to run time. The complete theory is integrated into the interactive theorem prover KIV [3] and all proofs were performed with this tool.

The formal framework further allows to analyse a system with regard to the number of errors that can occur before a reconfiguration will no longer be successful. For this purpose, ADCCA [8] allows to formally analyse OC-systems for combinations of failures that can lead to a permanent violation of the invariant. Together with different metrics, the reconfiguration abilities of a system can be quantified and different systems can be compared.

References

1. Ackermann, J.: Formal description of OCL specification patterns for behavioral specification of software components. In: Workshop on Tool Support for OCL and Related Formalisms, Technical Report LGL-REPORT-2005-001, pp. 15–29, EPFL (2005)

2. Anders, G., Seebach, H., Nafz, F., Steghöfer, J.-P., Reif, W.: Decentralized reconfiguration for self-organizing resource-flow systems based on local knowledge. In: Proceedings of EASe 2011, Las Vegas, USA (2011, to appear)
3. Balser, M., Reif, W., Schellhorn, G., Stenzel, K.: KIV 3.0 for provably correct systems. In: Hutter, D., Stephan, W., Traverso, P., Ullmann, M. (eds.) Proc. Int. Wsh. Applied Formal Methods. LNCS, vol. 1641, pp. 330–337. Springer, Berlin (1999)
4. Bäumler, S., Balser, M., Nafz, F., Reif, W., Schellhorn, G.: Interactive verification of concurrent systems using symbolic execution. AI Commun. **23**(2–3), 285–307 (2010)
5. Blum, M., Kanna, S.: Designing programs that check their work. In: STOC '89: Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing, pp. 86–97. ACM, New York (1989)
6. Branke, J., Mnif, M., Müller-Schloer, C., Prothmann, H., Richter, U., Rochner, F., Schmeck, H.: Organic computing—addressing complexity by controlled self-organization. In: ISoLA, pp. 185–191 (2006)
7. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning, 1st edn. Addison-Wesley, Reading (1989)
8. Güdemann, M., Ortmeier, F., Reif, W.: Safety and dependability analysis of self-adaptive systems. In: Proceedings of ISoLA 2006. IEEE Comput. Soc., Los Alamitos (2006)
9. Nafz, F., Ortmeier, F., Seebach, H., Steghöfer, J.-P., Reif, W.: A universal self-organization mechanism for role-based organic computing systems. In: González Nieto, J., Reif, W., Wang, G., Indulska, J. (eds.) Autonomic and Trusted Computing. LNCS, vol. 5586, pp. 17–31. Springer, Berlin (2009)
10. Nafz, F., Seebach, H., Steghöfer, J.-P., Bäumler, S., Reif, W.: A formal framework for compositional verification of organic computing systems. In: Xie, B., Branke, J., Sadjadi, S., Zhang, D., Zhou, X. (eds.) Autonomic and Trusted Computing. LNCS, vol. 6407, pp. 17–31. Springer, Berlin (2010)
11. Rahwan, T., Ramchurn, S., Jennings, N., Giovannucci, A.: An anytime algorithm for optimal coalition structure generation. J. Artif. Intell. Res. **34**(1), 521–567 (2009)
12. Richter, U., Mnif, M., Branke, J., Müller-Schloer, C., Schmeck, H.: Towards a generic observer/controller architecture for Organic Computing. INFORMATIK 2006 – Informatik für Menschen! P-93, pp. 112–119 (2006)
13. Shehory, O., Kraus, S.: Methods for task allocation via agent coalition formation. Artif. Intell. **101**(1–2), 165–200 (1998)
14. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: Grumberg, O., Huth, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 4424, pp. 632–647. Springer, Berlin (2007).
15. Tsang, E.: A Glimpse of constraint satisfaction. Artif. Intell. Rev. **13**, 215–227 (1999)

F. Nafz (✉) · H. Seebach · J.-P. Steghöfer · G. Anders · W. Reif
Institute for Software & Systems Engineering, Universität Augsburg, Universitätsstr. 6a, 86159 Augsburg, Germany
e-mail: nafz@informatik.uni-augsburg.de

H. Seebach
e-mail: seebach@informatik.uni-augsburg.de

J.-P. Steghöfer
e-mail: steghoefer@informatik.uni-augsburg.de

G. Anders
e-mail: anders@informatik.uni-augsburg.de

W. Reif
e-mail: reif@informatik.uni-augsburg.de