How to design and implement self-organising resource-flow systems

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

Angaben zur Veröffentlichung / Publication details:

Seebach, Hella, Florian Nafz, Jan-Philipp Steghöfer, and Wolfgang Reif. 2011. "How to design and implement self-organising resource-flow systems." In Organic computing — a paradigm shift for complex systems, edited by Christian Müller-Schloer, Hartmut Schmeck, and Theo Ungerer, 145–61. Basel: Birkhäuser. https://doi.org/10.1007/978-3-0348-0130-0_9.

Chapter 2.2
How to Design and Implement Self-organising Resource-Flow Systems

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

Abstract The construction of self-organising systems often leads to very ingenious and specific solutions to a concrete problem. These solutions cannot be easily transferred to other domains or systems. As the development of self-organising systems is a very time consuming and challenging task, instructions, methodologies, and tools to design and construct such systems in a generic and reproducible manner are required. This article presents a software engineering guideline along with a pattern for the class of resource-flow systems and details the steps that are required to implement systems designed according to the pattern. The guideline enables a software engineer to easily and reproducibly construct self-organising resource-flow systems. In addition, the presented concepts and techniques, i.e. the precise definition of the system structure and of behavioural corridors, observation of the corridors at runtime, and the verification of the system components' behaviour allow the engineer to guarantee correct system behaviour despite self-organisation.

Keywords Self-organisation · Software engineering · Multi-agent-systems · Resource-flow systems · Production automation

1 Introduction

In software engineering projects, a decisive virtue of the methods and tools employed is their reusability. The success rate of projects increases if a software engineering process, a modelling tool, a methodology, or an approach has been used before and there are experts in the team that are familiar and feel comfortable with these implements. Unfortunately, Organic Computing (OC) systems and self-organising systems in general do not yet have an established set of tools or even a real foothold within software research. To remedy this situation, we are proposing a software engineering guideline that is based on a pattern for self-organising resource-flow systems. Instances of this important class of systems were used to develop and demonstrate the viability of several techniques that can be used by software engineers within the framework of the guideline. Furthermore, we show how the implementation of such systems is aided by a runtime environment based on multi-agent technology and how a library of plug-in self-organisation mechanisms

enables an engineer to quickly choose and implement the best option for the specific system under construction.

These tools are applicable in two different situations: (1) a "classically" engineered system is already available; (2) the engineer has to build the system from scratch. Thus, they allow an engineer tasked with the re-design of a system to make it *OC-ready* by mapping the system to the pattern, applying the guideline, and using the runtime environment as an implementation basis. Due to a strict separation of functional and reconfiguration behaviour, the same possibilities are available if the system is newly designed. In both cases, the tools and techniques remain the same.

An important characteristic of self-organising systems is that, although they are able to reduce complexity by their autonomy, their behaviour is often hard to predict. To raise acceptance of self-organisation mechanisms in safety-critical systems, we propose a way to design and construct self-organising resource-flow systems which always behave in a correct manner even though the structure changes by self-organisation. Correct behaviour is ensured by the definition of behavioural corridors (see Chap. 1.5), their observation during runtime and the verification of the system components' functional behaviour. These techniques are made viable by a precise definition of the generic behaviour agents in resource-flow systems exhibit.

The article is structured as follows: First, the system class of self-organising resource-flow systems is defined and specified with the Organic Design Pattern (ODP). Section 3 presents a software engineering guideline that instructs an engineer how to systematically design and construct self-organising resource-flow systems. Then, Sect. 4 shows the implementation of an Observer/Controller architecture for ODP systems and defines the behaviour of the ODP's components. In Sect. 5 different self-organisation mechanisms are discussed and the use of the reference implementation "ORE" for a tool-supported construction of self-organising resource-flow systems is explained. Finally, Sect. 6 discusses future work and concludes the article.

2 Self-organising Resource-Flow Systems

In *self-organising resource-flow systems*, resources are processed according to a task by independent agents. Each agent has a number of capabilities it can apply to the resource. Agents can exchange resources with other agents as, e.g. given by the layout of a shop floor. To fulfil the task for a resource (i.e. to apply the correct capabilities in the correct order), a resource-flow is established that determines how the resource is moved through the system and processed on the way.

In traditional resource-flow systems, the entire processing comes to a standstill when one of the agents involved breaks or cannot apply one of its capabilities anymore. If the system contains redundancy with regard to the capabilities of the agents and flexibility of the resource transportation, a self-organising approach can change the resource-flow and make the system usable again with minimal delay. The agents are capable of locally detecting a failure by constantly monitoring an invariant INV which is usually a combination of several constraints (see Chap. 1.5). If the invariant

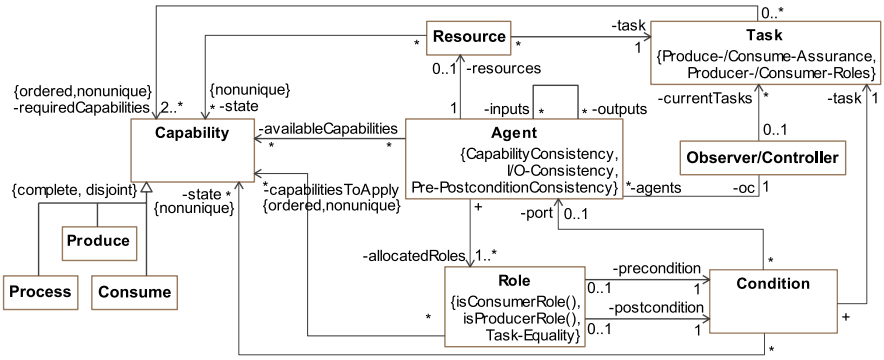

Fig. 1 Organic design pattern for resource-flow systems

does not hold anymore, they initiate a reconfiguration process during which the role that contains their configuration and the information of how to process resources are exchanged. The allocation of roles to agents determines the system setup and consequently the resource-flow. Thus, in self-organising resource-flow systems, self-organisation is a reconfiguration of the resource-flow, accomplished by calculating a new allocation of roles to agents.

The static structure for self-organising resource-flow systems is given by the Organic Design Pattern (ODP) (see Fig. 1). It contains a set of OCL-constraints (Object Constraint Language), e.g. the *CapabilityConsistency* constraint, which can be monitored locally by each agent and is expressed as:

$$self.availableCapabilities \rightarrow includesAll(self.allocatedRoles.capabilitiesToApply)$$

The constraint is defined in the context of the agent (*self*) and asserts that the role allocated to an agent only includes capabilities the agent can actually perform. A description of further constraints, how they can be expressed, and how they are used to specify a correct reconfiguration algorithm can be found in Chap. 1.5.

A new role allocation can be calculated as long as all capabilities required to fulfil the task are still available in the system and the agents that have these capabilities are linked by their potential *inputs* and *outputs* (see Fig. 1). Together with the *capabilities* still available at the agent, a new resource-flow can be computed that is enacted by the agents after they received a new set of *roles* they will have to apply. Both centralised and decentralised reconfiguration approaches are feasible. A brief discussion of different mechanisms is given in Sect. 5.

The nature of roles also enables such a system to be configured for different tasks in parallel. For each task, the role allocation determines a distinct resource-flow. If the system has to be reconfigured, only new roles for those resource-flows that require the agent to apply this capability are needed.

There are a number of examples for actual resource-flow systems that can be described with ODP: in *logistics applications*, the resources are packets that have to be transported from one place to another. The agents are distribution

centres that have different capabilities, e.g. accepting packages, delivering packages, or handling customs. Each centre that is able to accept or deliver a package has a unique denotation for these capabilities. The task is then a sequence like ($Produce_{Munich}$, *Customs*, $Consume_{Zurich}$) that describes that any package accepted at Munich has to go through customs before it is delivered at Zurich. Reconfiguration is used to determine the routes packages have to travel. In *production automation*, machines process workpieces by altering their state with redundant tools they have at their disposal. Autonomous carts transport workpieces between robots. A task could be (*Produce*, *Drill*, *Insert*, *Tighten*, *Consume*), describing the sequence of tool applications necessary. Reconfiguration defines which carts transport workpieces between which robots and which robot applies which tool.

The next section describes a guideline that helps software engineers to design and construct self-organising resource-flow systems that adhere to ODP's structure and thus gain the benefits self-organisation can provide for resource-flow systems.

3 Software Engineering Guideline

The guideline presented here is designed to tailor an existing SE process to be used for self-organising resource-flow systems. It allows to systematically enhance traditional systems with self-x properties or build new self-x systems. An important property of the guideline is that it separates design and construction of productive parts (responsible for the actual resource-flow, e.g. the control of machines or interactions with other physical systems) and self-x parts (responsible for the reconfiguration of the whole system). This is very important when a traditional system already exists and should only be enhanced by adding self-x properties. It also enables domain experts to design the productive part as usual and have an expert design the self-x part or use existing self-x mechanisms.

In this section, the guideline, which is described in detail in [4], and its activities are sketched. Figure 2 depicts a SPEM (Software and Systems Process Engineering Metamodel) model of the guideline. The highlighted activities are novel and will be shortly described in the following while the others are well known classical software

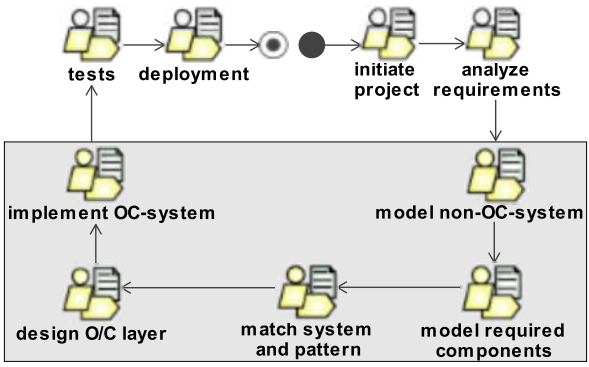

Fig. 2 Software Engineering Guideline in SPEM notation

engineering activities. The novel steps are supported by artefacts and techniques developed in the SAVE ORCA project.[1]

After initiating the project and analysing the requirements for the self-organising resource-flow system, the guideline instructs the engineer to *model the non-OC-system*. Thus, if a productive system already exists, this system must be investigated and the corresponding concepts have to be modelled. If not, the requirements are translated into classical SE models for the concepts needed in the system under construction. In both situations a conceptual model of the "traditional resource-flow system" is the result. This conceptual model is the basis for the activity *model required components*, which includes transforming the concepts into components. Self-organisation regarded as rearrangement of the system's structure needs some degree of freedom, which is added to the components in this task by adding redundant or additional properties. For example, a robot can have several tools multiple times or carts can reach additional robots compared to fixed conveyors. Then the activity *match system and pattern* follows, which encapsulates the mapping of traditional system components to components defined by the pattern presented in Sect. 2. After this, the designer determines the way the system self-organises in the activity *design O/C layer*. The activity also enables the engineer to make statements about the resilience of the system introduced by the self-organising behaviour with regard to the number and kind of failures that can occur. Finally, in *implement OC-system*, the self-organising resource-flow system is implemented in a runtime environment (see Sect. 5). The guideline ends with traditional tests and deployment activities.

To give an impression of what an activity of the guideline includes, the activity *implement OC-system* is described in detail in the following. One important part of the presented approach is the predefined behaviour of the agents in the context of the ODP. This enables a reference implementation (ODP Runtime Environment (ORE), see Sect. 5) where most of the dynamic agent behaviour and other crucial parts of the system are generically implemented. It is generic in a sense that it can be used for all systems modelled with the ODP and created with the help of the guideline, as the agent behaviour and interaction protocols as well as the reconfiguration mechanisms are usable for all instances. Of course, some parts still have to be adapted to the domain and application, as explained in step 12 to 14 of the guideline.

The activity *implement OC-system* describes all steps an engineer has to take to generate code out of the given artefacts (defined in the previous activities and steps 1 to 11) and to adapt the parts of the code which are domain dependent and cannot be automatically generated. In Fig. 3 all the steps and the artefacts needed and produced are depicted. Steps 12 to 14 use input artefacts and transform them into output artefacts as follows:

12. Generate Agent Definitions. The *domain models* (input artefact) and the *instantiation diagram* (input artefact) for classes derived from Agent provide the basis for the automatic generation of *agent definition files* (output artefact) which define the different kinds of agents in the ORE. These files contain clearly defined extension points that have to be manually implemented as follows.

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

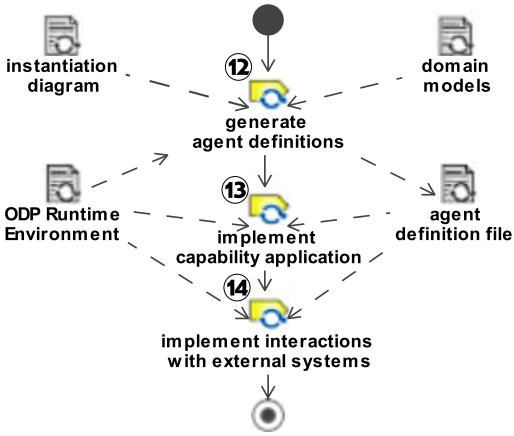

Fig. 3 Last activity of the Software Engineering Guideline: Implement OC-system

13. Implement Capability Application. Each domain processes the resources differently. In a production cell, for example, the robots add parts to a car body, thus changing its status. The engineer has to care about the correct physical change of the resource's status. Therefore, in many examples, additional processing and calculations have to be implemented as well.
14. Implement Interactions with External Systems. If a system has a physical part, e.g. robots, sensors or similar machines, the software agent must be aware of the state of its physical counterpart and able to manipulate it appropriately. As this is coupled to the application of capabilities and the interaction protocols of the agents, these steps are strongly related to the implementation of capability application. At this point it is, e.g. necessary to process sensory feedback, perform path planning and react to the physical environment of the agents.

By incorporating all tasks of the presented guideline into an existing software engineering process the software engineer is able to develop self-organising resource-flow systems systematically and reproducibly. One main benefit of the ODP and the guideline is, as already mentioned, the predefined behaviour of the agents, which is detailed in the next section. This includes functional behaviour as well as reconfiguration behaviour enabled by implementing an Observer/Controller architecture.

4 Functional and Reconfiguration Behaviour

The concept of an Observer/Controller architecture (O/C) can be found in many Organic Computing systems (see, e.g. Chap. 4.7, Chap. 5.1, Chap. 5.7). A generic O/C architecture for OC-systems, which is also the basis for the agent architecture in ODP systems, is given in [2]. The main components of this architecture are the system under observation and control (SuOC), an observer which continuously observes the state of the SuOC, and a controller which takes action if necessary, i.e. if the observer detects a situation that requires intervention by the controller. In the

following, it is demonstrated how this abstract architectural description can be implemented in an actual agent system for self-organising resource-flow systems.

4.1 An O/C Architecture with Base Agents and Reconfiguration Agents

There are two driving forces that guide the implementation of an O/C architecture. The first of them is cohesion [6], the grouping of functionality that operates on the same data or serves the same functionality. The second is separation of concerns, the definition of distinct aspects of a system whose functionalities do not overlap.

The abstract description of an observer assumes that it is possible to monitor a system with a separate system and continuously observe all relevant information that is required by the controller. In fact, however, the observer has to perform a more complex task as observation may include interactions with other components, e.g. to find out whether they are still alive. A true separation, even with techniques like aspects, would therefore be hard to realise and somewhat artificial. We thus argue that employing communicational cohesion by pairing the observation part of the O/C with the functional agent which holds all the data and the states that need to be observed (e.g. sensor data in a sensor node) is beneficial in actual implementations. The combination of observer and functional agent is called *base agent*. Thus, separation of concerns is achieved by introducing the base agent and the concept of a *reconfiguration agent* which acts as the controller and is only active when required. The formal counterpart of this separation is explained in Chap. 1.5 and relies on the partitioning of the behaviour into a formal model of an ODP-agent that relates to functionality and one that relates to correct reconfiguration.

Figure 4 shows the implementation of a decentralised Observer/Controller architecture according to above description. The functional part is enhanced by the observer part within the base agent. There are distinct controllers for different tasks

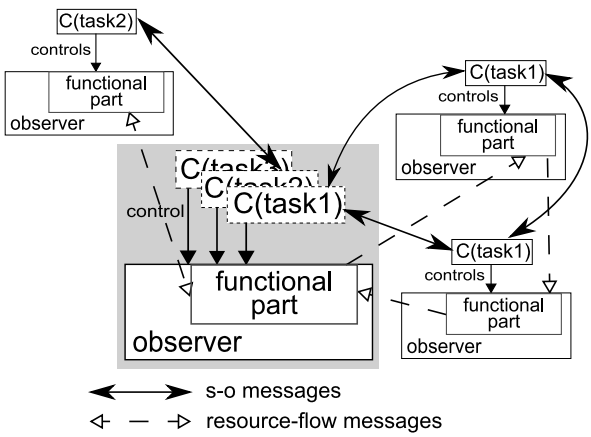

Fig. 4 Proposed implementation of an O/C architecture

that are performed by the base agents. In case a reconfiguration is necessary, the base agent starts a controller for this task, e.g. $C(task1)$. The agent in the centre of Fig. 4 has started several reconfiguration agents to reconfigure tasks 1, 2, and 3 which communicate with the controllers of the other base agents that participate in the reconfiguration. The messages (*s-o messages*) between the controllers are initiated by the respective reconfiguration mechanisms. As soon as the reconfiguration is finished, the controllers terminate and the base agents can resume processing resources according to the task. The way these two agent types interact and their functions are described in the following.

Base Agent

The base agent's main responsibility is to perform a function that helps to solve the problem the system was designed for. This can, e.g. be the control of a physical entity or machine, the application of an algorithm that alters data, or any other processing or control. This basic functionality, detailed in Sect. 4.2 and explained later on, should be modelled independently and separately from the self-x mechanisms and constitutes the *capabilities* of an agent. Several agents collaborate and employ their capabilities to resources to fulfil a common *task*.

The other responsibility of the base agent is the observation of its states and its data and the detection of undesirable behaviour. In resource-flow systems, where an agent is integrated into a resource-flow graph, the base agents also have to check the availability of their neighbours. In case of an internal error or a neighbour that is no longer reachable, the agent has to trigger the self-organisation mechanism. If another agent triggered the mechanism, it is also necessary to react to this event and participate in the reconfiguration.

The responsibilities that come with the observation and with the triggering of a reconfiguration are modelled in a state machine with five parallel states as seen in Fig. 6. The topmost state is the one which determines the reaction of the base agent in case of a failure that results in an invariant violation (INV! = true) or a *ReconfRequest* from another agent, indicating that help is required to reconfigure the system. *ReconfRequest*s are persistently collected as shown in the middle state.

It is not necessary to stop the entire agent in case one of its capabilities is no longer available or it determines that a neighbour is no longer reachable. Instead, only those configurations have to be excluded from processing that make use of the broken capability or the broken agent. Likewise, if another agent requested this

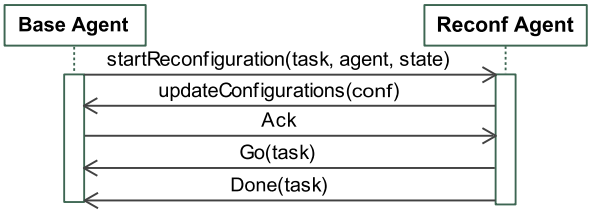

Fig. 5 Protocol for communication between base and reconfiguration agents

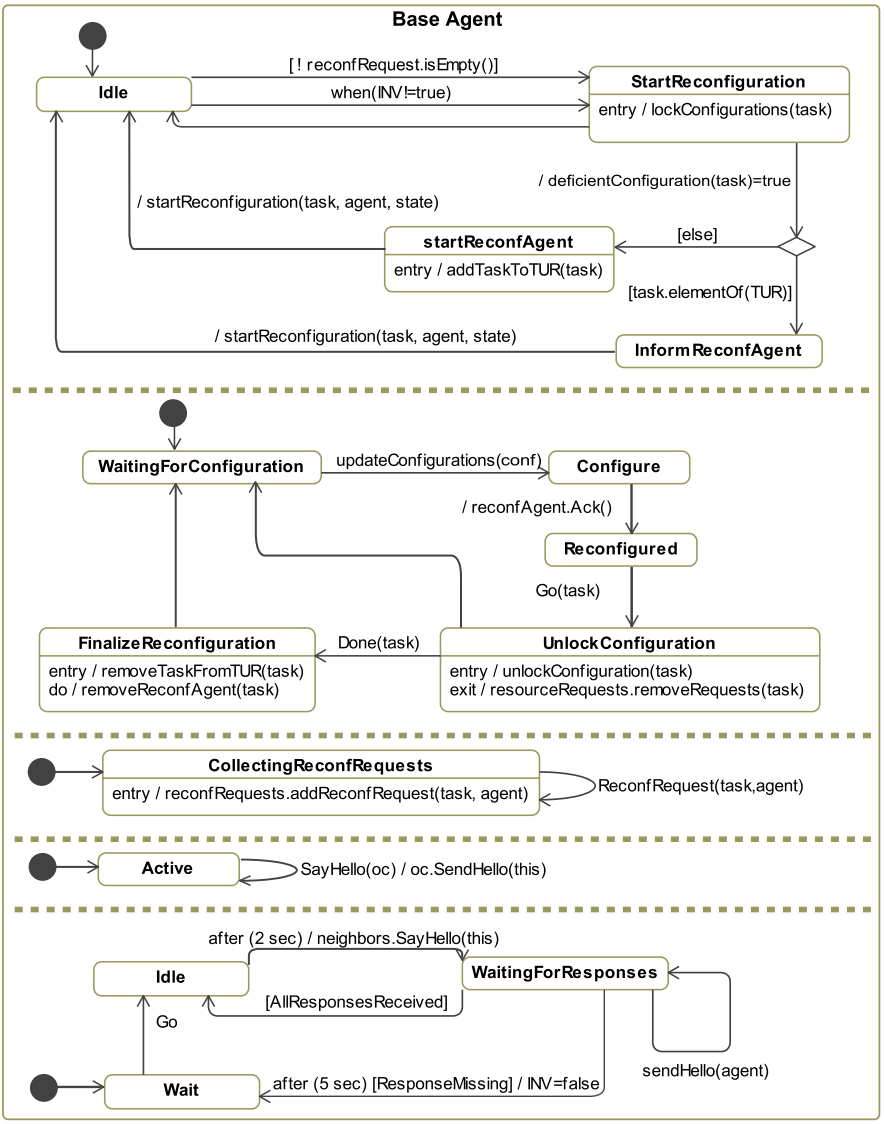

Fig. 6 Observer part of the Base Agent

agent to reconfigure, only the part of the configuration that is of concern for the task that is reconfigured has to be regarded. If the base agent currently performs a role for the task, that should be reconfigured, the base agent needs to be informed about this situation because the result of performing a role in context of a reconfigured task can not be foreseen. Therefore, the base agent always checks the flag *deficientConfigurations(task)* for the task in the role it currently applies, see Fig. 7. If the

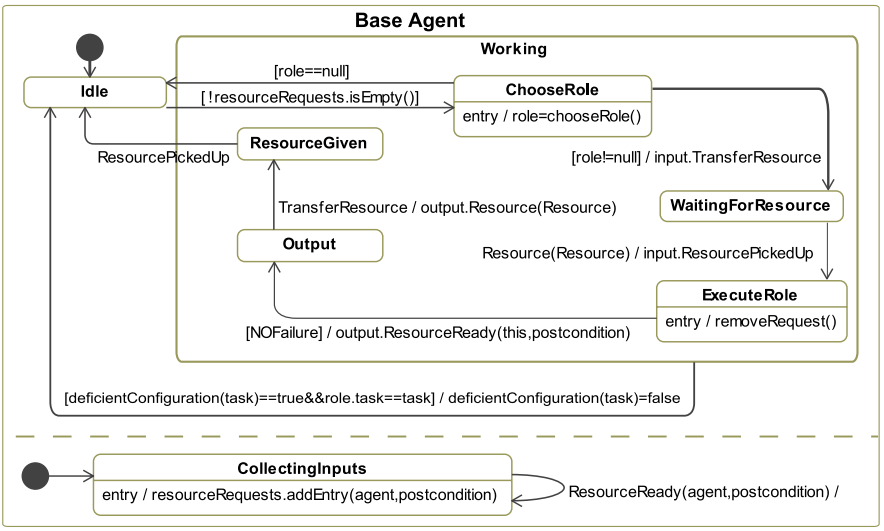

Fig. 7 Functional part of the Base Agent

flag is true, it returns to an idle state and restarts its functional behaviour without considering configurations for this task.

To accommodate this task-specific reconfiguration, each base agent has a hash map that contains all tasks the agent partakes in as keys and references to the corresponding reconfiguration agents as values. Whenever a reconfiguration has to take place to restore the invariant or react to a *ReconfRequest*, the base agent checks the map to see if a reconfiguration agent for the task has already been started. If none is available yet, a new one is started and the task is added to the list of *T*asks *U*nder *R*econfiguration (TUR). Then, this request is communicated to the new or the existing agent, respectively. The base agent does not need to know how the reconfiguration agent reacts to this information. It just waits for a new configuration for the task as depicted in the second state (*WaitingForConfiguration*).

If the base agent receives an *UpdateConfiguration(conf)* message, it sets the new configuration for the task. The new configuration is locked until the reconfiguration agent sends a *Go*. If the new configuration is compatible with the rest of the agent's configuration, the base agent acknowledges the new configuration. In the *Reconfigured* state the agent waits for the *Go(task)* message so that it can unlock the configuration. If the reconfiguration agent sends a *Done(task)* message, the reconfiguration is finished and the task for which the reconfiguration took place is removed from the tasks under reconfiguration (TUR). Otherwise, the agent waits for the next configuration that is calculated.

Apart from the failure of a capability of an agent, failure of an entire agent will also be detected by the neighbours of the failed agent and cause the violation of an invariant. Such failures are recognised by a standard heartbeat mechanism where the agents periodically "ping" their neighbours and if they do not react in time the

agent is assumed to be broken. This is modelled in the two states at the bottom of the state machine in Fig. 6.

Reconfiguration Agent

The controller part of the abstract Observer/Controller architecture is implemented in the reconfiguration agent. Each base agent starts a reconfiguration agent whenever an error has been detected or when it has been asked to participate in a reconfiguration triggered by another agent. Different self-organisation mechanisms are implemented with different reconfiguration agents that expose a generic protocol and interface to the base agents. Thus, different mechanisms can be easily exchanged by swapping different reconfiguration agents without any change to the base agent.

The reconfiguration agent communicates with the base agent with the protocol depicted in Fig. 5. A reconfiguration agent receives messages of type *startReconfiguration(task,agent,state)* and queues them internally. This is useful in situations where the reconfiguration agent cannot react immediately to the event because, e.g. it is already in the phase where configurations are being allocated. The event is handled after distributing the configurations to its base agent. By parametrising *startReconfiguration(task,agent,state)* and *Go(task)* with the task, a base agent can selectively reconfigure for specific tasks that are affected by an error.

After the reconfiguration has been started, a new configuration (conf) is calculated and transmitted to the base agent in an *updateConfigurations(conf)* message. The base agent checks the configuration and informs the reconfiguration agent (*Ack*) whether it is compatible with its allocated roles. The reconfiguration agent then sends *Go(task)* to indicate that the base agent can assume its productive state.

After the *Go* message has been sent, the reconfiguration agent checks if there are pending *ReconfRequest*s in its queue. If this is not the case, it will send a *Done(task)* message to the base agent and terminate itself afterwards. Otherwise, it calculates a new configuration and repeats the process. How the reconfiguration agent reacts if there are multiple ReconfRequests waiting depends on the concrete self-organisation mechanism. It can, e.g. discard the older ones and only use the latest one, combine them and handle them at once, or reconfigure for each of them sequentially.

4.2 Functional Behaviour in Self-organising Resource-Flow Systems

After detailing the Observer/Controller part of the agents, an instantiation of the functional part of the base agent is now described for the domain of self-organising resource-flow systems. An agent has a *state* which is instantiated with a tuple of its permissible inputs/outputs, currently allocated roles, and available capabilities. The *configuration* of an agent corresponds to a set of roles (Rs).

The monitoring behaviour of the base agent is extended by the functional part by adding two parallel states which model the characteristics of resource-flow systems: handle the processing of resources and govern the communication between the base agents. In Fig. 7, the state at the bottom depicts that the base agent always collects requests from other base agents that want to give resources to this agent. Whenever the base agent receives a *ResourceReady(agent, postcondition)* event, it puts an entry into the *request* map. This entry contains as key the base agent which sent the event and as values all roles which have a precondition according to the postcondition in the event. If the base agent is reconfigured, this request map is cleared for the according task.

The state at the top represents the agent in running mode. It starts with an *Idle* state which the agent only leaves if there are resources that have to be processed. When the agent is in the *Idle* state it always checks if the *request* map contains an entry. If so, it switches the state to *ChooseRole*. The entry action chooses one of the roles in the map if possible. A fairness function and a deadlock avoidance mechanism [5] determine the choice of a role. To avoid deadlocks, e.g. the agent at the entry of a loop in the resource-flow can decide that no role will be chosen (*role==null*). If that is the case, the agent goes back to the *Idle* state and tries again after a small delay. If a role is chosen (see Fig. 1 for details on the used concepts), the agent asks the agent (*input*) in the *request* map to transfer the resource (*TransferResource*). Then, the agent waits until it receives the resource (*Resource(res)*) and informs the sending agent about picking it up (*ResourcePickedUp*). In the *ExecuteRole* state the agent removes the corresponding entry from the *request* map and executes the capability defined in the chosen role. The agent informs the *output* agent defined in the postcondition of the role that the resource is ready and then waits in the *output* state until the *output* agent asks for the transfer of the resource (switch to *ResourceGiven* state). After the agent received the *ResourcePickedUp* event, the agent is in the *Idle* state again and waits for incoming requests. An important part of the resource-handshake between two base agents is that the sending agent has to hold on to the resource until the receiving agent is ready to receive the resource. On the one hand, this behaviour guarantees that an agent is not interrupted by incoming resources and on the other hand this is necessary as an agent does not have the ability to store a resource but can only accept a resource if it is able to process it right away. The main benefit of this design is that the receiving agent has the opportunity to decide which resource it will take next, thus enabling avoidance of deadlocks and starvation as described in [5].

Any failure that occurs in the *Working* state (e.g. the capability that is currently applied breaks) is detected by the observer part of the base agent. It sets *deficientConfiguration(task)* to true. If the flag becomes true for the currently processed task, the agent leaves *Working* and resets *deficientConfiguration(task)*. As the observer part already disabled all roles that belong to the task and the agent is no longer processing resources for it, the task can now be reconfigured while the agent can handle other tasks.

The behaviour of the base agent in self-organising resource-flow systems is now fixed and can be used in different instances. Thus, concrete instances like production automation or logistic scenarios do not have to redefine these concepts and

protocols. If they use the unaltered behaviour described here, all verifications and simulations that have confirmed correct functionality on the abstract level are still valid.

The behaviour described above is, as already mentioned, implemented in the reference implementation *ODP Runtime Environment* and thus can be used for the whole system class of self-organising resource-flow systems.

5 ODP Runtime Environment

The *ODP Runtime Environment* (ORE) offers an extensible and convenient platform for the implementation of systems based on the ODP and constructed according to the guideline. It provides the system dynamics and clearly defined extension points for application-specific behaviour as well as a plug-in mechanism for reconfiguration algorithms. A main advantage of using the ORE for the realisation of a system is that the artefacts created during the construction process are the input for a model-to-text-transformation that yields most of the source code required in an implementation. Therefore, the actual implementation effort is minimal and restricted mainly to the actual interactions between the agents and physical machines. The following paragraphs will introduce the architecture of the ODP Runtime Environment, the code transformation and the extension points, as well as different reconfiguration algorithms available for use in an application.

5.1 Architecture and Behaviour

The ORE is based on Jadex [1], a mature multi-agent system that uses the BDI paradigm for the modelling of agents. BDI stands for *Belief*, *Desire*, *Intention* and has originally been used to describe human cognitive processes. Jadex—as a procedural reasoning mechanism—maps this to *believes*, *goals*, and *plans*.

A belief is an arbitrary object that holds any kind of information and is stored in the agent's belief-base, a repository that can be queried and modified by the agent. Goals are the counterpart of BDI's desires and define the states the agent wants to achieve in an abstract way. Most importantly, a goal does not define how it can be achieved but rather only states a desired condition (i.e. retrieve a certain object, maintain a certain state, or perform a certain task). It is possible to define goals that are active all the time, that become active if a condition (no longer) holds, or ones which have to be dispatched manually. Whenever a goal becomes active one or more plans are activated. If there are several plans to achieve a goal, Jadex selects the most suitable one. In case the selected plan does not succeed, the other plans are executed until the goal is fulfilled or no more plans are available. Plans can also be triggered by the arrival of messages from other agents or by internal events. Whenever an agent executes a plan, it makes a commitment to pursue a goal or react to an event with this plan, thus stating its intention. A plan can instantiate goals, send

internal and external messages, and has the ability to synchronise with other agents by waiting for replies to messages or the occurrence of internal events.

Jadex also provides a complete communication and discovery infrastructure. Agents are registered at a directory facilitator which can be queried to receive the handles of other agents. Messages can be sent over a standard communication interface and are relayed to their respective recipient(s) by the framework. The reception of a message can be part of a protocol flow or cause an event at the receiver's end.

On top of the Jadex interfaces, an *OcInfrastructure* is provided that represents the Base Agent. This Jadex capability (the unit of reuse in Jadex) defines the generic parts of the base agent. The functional part of the base agent (see Fig. 7) like role selection and execution as well as resource handling and all involved messages are implemented generically. Additionally, the observer part of the base agent (see Fig. 6) is already in place. Monitoring constraints, checking availability of neighbours, and starting of as well as communication with the reconfiguration agents is fully implemented and available to the developer. Generic invariants, like the *CapabilityConsistency* constraint introduced in Sect. 2, are already included and observed.

5.2 Code Transformation and Extension Points

In order to use the predefined functionality, *OcInfrastructure* has to be included in an *agent definition file* (ADF). These files contain descriptions of application-specific beliefs, goals and plans as well as extension points that can be used to implement required behaviour that cannot be defined generically. Most of the information required to create these files is, however, already contained in the artefacts that are created during the software engineering process if the designer sticks to the guideline. Therefore, it is possible to automatically transform the models and thus create an ADF as proposed by step 12 of the guideline that only has to be augmented if specific requirements necessitate it. Furthermore, the transformation also yields skeleton implementations of plans for each of the capabilities defined in the models together with Jadex capability files that are automatically referenced in the ADFs.

Together, these files constitute a runnable implementation of the resource-flow system. The instance diagram created by the designer is transformed into a manager agent which can be started by Jadex and in turn bootstraps the system by starting all the agents defined in the instance. After the transformation, there are three pieces missing: the interactions with the physical machines, application-specific invariants that have to be monitored, and the reconfiguration algorithm. The different choices for the latter are described in the next section, while the former have to be added by augmenting the created files by the implementation of *extension points*.

In an actual resource-flow system, physical machines are represented by software agents. Agents control and structure the work flow on a very high level though and the steps needed to be performed by the machines are not modelled during the design process. Therefore, domain experts will have to design and implement the machinery and the interface between the agents and the machines. The plans,

generated for each of the capabilities, contain the appropriate extension points for the application of tools to the resources (step 13 of the guideline). Likewise, the predefined plans that handle the logical resource transmission contain extension points to implement handing over physical resources, according to step 14 of the guideline (see Sect. 3).

Some resource-flow systems also require additional invariants that the agents have to observe during runtime. The generic base agent contains an abstract goal which can be instantiated in an agent implementation. The goal becomes active whenever the application-specific invariants contained in its condition are violated and plans that start the reconfiguration process are automatically executed.

As a positive side effect, the automatic code transformation provides a way to check the consistency of the models. In an iterative process, the designer will refine the models in each iteration and can check the resulting models with the transformation. Also, a runnable system that can be used in simulations to check the viability of the system and evaluate the reconfiguration mechanism is available after each iteration. Testing and simulation are supported by a visual user interface, that shows the resource-flows, allows to add or remove agents, and to modify the capabilities and inputs/outputs available to each agent.

5.3 Plug-in Mechanism for Reconfiguration Algorithms

Just like *OcInfrastructure* represents a generic Base Agent, *OcReconfiguration* represents a generic Reconfiguration Agent. It implements the protocol depicted in Fig. 5 and can thus interact with the generic base agent. As this is the only link between base agents and reconfiguration agents and the base agents are thus agnostic with respect to the actual reconfiguration mechanism, different algorithms can be used with the same base agents, constituting a *plug-in mechanism* for reconfiguration algorithms. At the moment, there are three different algorithms available. Each of them is able to restore the invariant in the system in case it has been violated.

Weak self-organisation with central reconfiguration. The reconfiguration agent that was started after a failure was detected sends the information about its base agent to a central controller. The central controller starts a reconfiguration on all other agents and collects their configuration as well, thus creating a global model of the system. This model can then be used as the input for a solution algorithm, e.g. a constraint solver (see Chap. 1.5) or a genetic algorithm, that finds new configurations for all agents. After the solution has been found, the new roles are distributed to the agents and all of them are restarted.

Strong self-organisation with a wave-like reconfiguration. After a failure was detected, the reconfiguration agent contacts other agents in the vicinity to see if roles can be switched to restore the system. If an agent can provide assistance but in turn has to find a replacement for its own role, it sends a request to its neighbours. This constitutes a fully decentralised mechanism in which reconfigurations propagate through the system in the form of a wave [7].

Strong self-organisation with a coalition-formation algorithm. After a failure was detected, the reconfiguration agent forms a coalition of agents that, together, possess all capabilities required to restore the resource flow. For more information about this algorithm, see Chap. 1.5.

6 Conclusion and Future Work

This article presents a software engineering guideline that simplifies design and construction of self-organising resource-flow systems. The guideline accompanies an engineer towards a robust, flexible, self-organising system. It also enables offline verification of the agents' functional behaviour (see Chap. 1.5) and online adherence to a behavioural corridor, thus guaranteeing correct behaviour. This is achieved by three aspects: (1) a clearly defined system structure along with a predefined behaviour of the agents is provided by the Organic Design Pattern (ODP); (2) the behaviour is implemented as an Observer/Controller architecture to achieve clear separation of concerns; (3) the ODP contains a set of constraints for the domain of resource-flow systems, which specify the behavioural corridor.

The software engineering guideline is supported by the ODP Runtime Environment, an implementation of generic aspects of the ODP. Thus, the implementation can be used for any instance of resource-flow systems modelled with the pattern. In addition, the ORE provides a plug-in mechanism for different reconfiguration mechanisms, enabling reuse of self-organisation mechanisms for this system class.

Future work will include the analysis of other system classes as well as research on the feasibility to adapt the presented techniques to them. For data-flow systems, the structural adaptations of the Organic Design Pattern required were minimal [3].

References

1. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: A BDI reasoning engine. In: Multi-Agent Programming, pp. 149–174 (2005)
2. Richter, U., Mnif, M., Branke, J., Müller-Schloer, C., Schmeck, H.: Towards a generic observer/controller architecture for organic computing. In: INFORMATIK 2006 – Informatik für Menschen!, vol. P-93, pp. 112–119 (2006)
3. Seebach, H., Nafz, F., Holtmann, J., Meyer, J., Tichy, M., Reif, W., Schäfer, W.: Designing self-healing in automotive systems. In: Xie, B., Branke, J., Sadjadi, S., Zhang, D., Zhou, X. (eds.) Autonomic and Trusted Computing. Lecture Notes in Computer Science, vol. 6407, pp. 47–61. Springer, Berlin (2010)
4. Seebach, H., Nafz, F., Steghöfer, J.-P., Reif, W.: A software engineering guideline for self-organizing resource-flow systems. In: IEEE International Conference on Self-Adaptive and Self-Organizing System, pp. 194–203. IEEE Comput. Soc., Los Alamitos (2010)
5. Steghöfer, J.-P., Mandrekar, P., Nafz, F., Seebach, H., Reif, W.: On deadlocks and fairness in self-organizing resource-flow systems. In: Müller-Schloer, C., Karl, W., Yehia, S. (eds.) Architecture of Computing Systems – ARCS 2010. Lecture Notes in Computer Science, vol. 5974, pp. 87–100. Springer, Berlin (2010)
6. Stevens, W., Myers, G., Constantine, L.: Structured design. IBM Syst. J. **13**(2), 115–139 (1974)

7. Sudeikat, J., Steghöfer, J.-P., Seebach, H., Reif, W., Renz, W., Preisler, T., Salchow, P.: Design and simulation of a wave-like self-organization strategy for resource-flow systems. In: Proceedings of the 4th Workshop on Multi-Agent Systems and Simulation (2010)

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

F. Nafz
e-mail: nafz@informatik.uni-augsburg.de

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

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